THIRD-PARTY LIBRARY SECURITY MANAGEMENT FOR MOBILE APPLICATIONS

A Dissertation Document in
Computer Science and Engineering
by
Wenhui Hu

© 2016 Wenhui Hu

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2016
The dissertation document of Wenhui Hu was reviewed and approved* by the following:

Patrick D. McDaniel  
Distinguished Professor of Computer Science and Engineering  
Dissertation Co-Advisor, Co-Chair of Committee

Peng Liu  
Professor of Information Sciences and Technology  
Dissertation Co-Advisor, Co-Chair of Committee

Trent R. Jaeger  
Professor of Computer Science and Engineering

Constantino Lagoa  
Professor of Electrical Engineering

Damien Octeau  
Postdoctoral Scholar, University of Wisconsin and Pennsylvania State University  
Special Member

Chita R. Das  
Distinguished Professor of Computer Science and Engineering  
Interim Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.
Abstract

In recent years, the Android operating system has had an explosive growth in the number of applications containing third-party libraries for different purposes. In this thesis, we identify three library-centric threats in the real-world Android application markets: (i) the library modification threat, (ii) the masquerading threat and (iii) the aggressive library threat. These three threats cannot effectively be fully addressed by existing defense mechanisms such as software analysis, anti-virus software and anti-repackaging techniques. The limitation of existing mechanisms is that the existing mechanisms ignore the fact the library and applications coexist, but are from different stakeholder in the ecosystem.

For existing libraries, related security problems in applications, we propose a library integrity verification tool for Android applications at application stores. This is non-trivial because the Android application build process merges library code and application-specific logic into a single binary file. Our approach uses reverse-engineering to achieve integrity verification. The evaluation results also provide empirical insight into the library integrity situation in the wild.

In addition, third-party libraries are detected to contain potentially malicious behaviors. With the Android security model, the core logic and the third-party library share permissions. Solutions are provided to enforce library isolation. Unfortunately, libraries in the isolation still share some permissions with the core logic. In order to generate distinct policies for third-party libraries, we need to know which part of libraries works for the core logic.

Existing analysis tools cannot differentiate libraries' behaviors assisting for the application's main functionality from libraries' behaviors for itself because of two limitations: 1) the analysis cannot do backwards analysis; 2) the analysis cannot distinguish different stakeholders within one application. This shortcoming leads to the difficulty of creating a precise policy for library isolation solutions. The missing part is the intra-app cross-stakeholder data-flow analysis. To address it, we propose a callsite-aware bidirectional static analysis tool. The tool enables to classify the Android framework APIs in libraries into six categories. The classification information can be used to generate polices. The dissertation also has an enforcement of the library isolation.
# Table of Contents

List of Figures viii
List of Tables ix
Acknowledgments xi

Chapter 1 Introduction 1
  1.1 Library-Centric Security Threats 2
  1.2 Library based Isolation 3
  1.3 Thesis Statement 4
  1.4 Threat Models 5
    1.4.1 The Threat Model of the Library Integrity Verification 5
    1.4.2 The Threat Model of the Policy Generation 6
  1.5 Contributions 7

Chapter 2 Preliminary Concepts 9
  2.1 Libraries in Android 9
    2.1.1 Android Application Compilation 11
    2.1.2 Library Post-Processing 11
  2.2 Limitation of Android Security Model 12
    2.2.1 Android Security Model 12
    2.2.2 Malware and application developers 12
  2.3 Relationship between Library and Application Logic 13

Chapter 3 Related Work 14
  3.1 Smartphone Platform Security 14
  3.2 Application Behavior Modification 15
  3.3 Android Application Library 16
    3.3.1 Library Detection 16
    3.3.2 Advertising Library 16
List of Figures

2.1 Compilation process for Android applications. ................. 10
4.1 Store Side Library integrity verification: system overview. .......... 29
4.2 Calculating the canonical library digest and canonical file digest for original libraries. ........................................ 33
4.3 Calculating library digest and file digest for libraries in Android applications. .................................................. 34
4.4 Popularity of the Top 20 Libraries in Our Dataset. ................. 37
4.5 Number of Libraries Contained by Each Application. ............. 38
5.1 System Overview. .................................................. 62
6.1 LibGuardian: System Overview. ........................................ 86
6.2 Application Retrofit Process. ........................................... 87
6.3 The Snap of Original Application. ..................................... 92
6.4 The Snap of Retrofit Application. ..................................... 94
## List of Tables

4.1 Categories of the top 100 detected libraries, and their source code available situation. ........................................... 38
4.2 Usage information of the top 100 detected libraries. ................................. 39
4.3 Information of the Reference Database. ........................................... 41
4.4 In-the-wild Library Integrity Verification Results. (All Known Versions) .. 42
4.5 In-the-wild Library Integrity Verification Results. (Some Versions) ....... 43
4.6 In-depth Analysis of In-the-wild Verification Not Passing Samples. ....... 45

5.1 Meanings of different read operation and write operation in Application Logic. ■: need further analysis; ▲: may be potentially malicious behaviors; ▼ other module relies on it; ♦ helps to understand the relationship among modules. ........................................... 59
5.2 Meanings of different read operation and write operation in libraries. ■: need further analysis; ▲: may be potentially malicious behaviors; ▼ other module relies on it; ♦ helps to understand the relationship among modules. 60
5.3 Permissions used by top 20 libraries. ........................................... 68
5.4 DroidBench results ............................................................................ 68
5.5 Comparison of forward and backwards analysis. .................................. 69
5.6 Classification results for advertising libraries. ...................................... 72
5.7 Classification results for social network libraries. .................................. 73
5.8 Classification results for platform libraries. ........................................ 74
5.9 Classification results for bug tracking libraries. ..................................... 74
5.10 Classification results for audio/graphics engine libraries. .................... 75
5.11 Classification results for libraries containing potentially malicious behaviors. 75
5.12 Classification results for functionality libraries. .................................. 76

6.1 Retrofit Latency. .............................................................................. 92
6.2 Retrofit Successful Rate and Size Overhead. ..................................... 93

A.1 The Top 100 Libraries (Part 1). ...................................................... 101
A.2 The Top 100 Libraries (Part 2). ...................................................... 102
A.3 The Top 100 Libraries (Part 3). ...................................................... 103
A.4 The Top 100 Libraries (Part 4)
Acknowledgments

I have been in Penn State for over 9 years. During this period, my life has been changed several times. I have so many affecting stories that I cannot make a complete list. Here, I would like to thank all those without whom this dissertation cannot be competed.

First, I would like to thank my advisor Dr. Patrick McDaniel. He has always given me the valuable guidance. The guidance helped me cognize and overcome my limitations in research. He taught me how to read research paper. He also taught me how to write research papers. He got me speedup without pressuring me. He encouraged me when I was passive. He also gave me the space to cooperate with others. The only pity is that I got him too late.

I would also like to thank my advisor Dr. Peng Liu. He gave me the opportunity to continue my study when I was facing a big difficulty. Whenever he found my shortcoming, he pointed it out kindly, and gave me time to improve. I am lucky to have these two amazing advisors. One would spend more time with me, when the other was busy. Hence, I improved a lot during the past three years.

I would also like to thank the members of the SIIS lab. The idea of the library integrity paper was not admitted at the beginning. That was one of the most difficult times in my life. At that time, the discussions with Stephen McLaughlin and Hayawardh Vijayakumar helped the idea survive. Damien Octeau became the coauthor of that paper, and he also served as my committee member later. The dataset from Google play was contributed by Matthew Dering. I still remember that many discussions happened in the middle night at that golden age. Stephen visited State College for the research discussion after my comprehensive exam. I also thank Yuqiong Sun, Devin Pohly and Xinyang Ge for their support.

I would also like to thank the members of cyber security lab, especially for the friendly atmosphere. In particular, I thank Jun Wang, Bin Zhao, Mingyi Zhao, Ping Chen and Le Guan for their constructive discussions for research.

Finally, I am grateful for the support from my friends and family in China and in the United States. The love from my parents is endless. They always encouraged me for almost everything. Friendship is the other source of the power. I would to thank all friends to be there during the past years.
Dedication

To my parents, for their love and support.
Chapter 1

Introduction

With the improvement of hardware, mobile computing rises. Smartphones now take the biggest market share in the world [1]. Smartphones not only have powerful computation ability, but also have multiple sensors that enable mobile sensing. With these advantages, third-party developers can develop innovative applications under the application store model to collect revenue. Unfortunately, besides the convenience from these innovational applications, aggressive behaviors from applications become a big concern. Samples of aggressive behaviors include collecting users’ privacy information and annoying notifications. Please see Chapter 3 for measurement results and solutions of these problems.

The Android operating system holds the biggest market share in the world of smartphones [2]. For such a success, third-party applications play an important role in the whole ecosystem [3]. These application developers integrate libraries into their applications for different purposes. For example, advertising-supported free applications have become popular in the world of Android. An advertising network distributes advertising libraries. Third-party application developers collect revenue by adding these advertising libraries into their applications. Pearce et al. found that 49% of applications in their dataset are supported by advertising libraries [4].

Besides advertising libraries, Android application developers also include other libraries into their applications. For example, Zebra crossing (ZXing) [5] is a library that provides functionality of 1D/2D barcode image processing. Another example is the license verification library (LVL) [6] by Google. With LVL, applications can query Google Play to obtain their license status at runtime.
1.1 Library-Centric Security Threats

The Android platform provides a security model to guarantee isolation among applications [7]. This model is built based on the assumption that applications from different developers should be isolated. The atomic unit of protection is one application. However, in the real world, within many Android applications, application logic and libraries coexist. Application logic and libraries have different intentions because they are from different stakeholders in the ecosystem. The security model of Android platform ignores different intentions from various stakeholders within the same application. This shortcoming leads to threats in the real world.

Three Library-Centric Security Threats: Unfortunately, third-party libraries that come with applications can be modified to be malicious. For example, AntiLVL [8] is a free tool available online that modifies third-party libraries in applications in order to subvert standard license protection methods such as Amazon Appstore DRM and Verizon DRM. Such attacks require reverse-engineering and a build process, which is called a repackaging process.

Besides modifying existing libraries, attackers could also create new libraries that masquerade as existing libraries. To make people believe this is a good library, attackers usually would do masquerading. For example, use the same namespace that is used by good libraries. For instance, as reported in [9], DroidKungFu, a famous malware, uses names such as com.google.ssearch and com.google.update to pretend to be published by Google for legitimate and benign purposes.

In addition to the modification threat and the masquerading threat mentioned above, it has also been proven that some legitimate third-party libraries have aggressive behaviors such as collecting device owner’s email address. Based on the report from FireEye Blog [10], one popular advertising library that has aggressive behaviors has been used in over 1.8% of the applications in their dataset and these affected applications have been downloaded more than 200 million times in total. As reported in the follow-up news [11], benign application developers remove problematic third-party libraries after being notified.

Limitation of Existing Techniques: The three library-centric security threats mentioned above cause very serious consequences in the real world. However, we find they cannot be effectively fully addressed by existing defense mechanisms. Although software analysis [12, 13], anti-virus software [14, 15] and anti-repackaging techniques [16, 17] can detect some malicious behaviors in library code, it is hard to tell whether library providers or application developers are the offenders because the application
developers can modify library code during the development process.

In the case of the modification threat, the application developer could blame the library provider for malicious library behaviors, even though the application developer is the real attacker and the library provider is a victim of the attack. Regarding the masquerading threat, the application developer could also blame the library provider for malicious library behaviors, even though the real attacker is the application developer and the victim is the library provider. Regarding the aggressive library threat, the library provider could blame the application developers for aggressive library behaviors because the developer can modify the library. In this case, the real attacker is the library provider and the victim is the application developer.

However, it is important to figure out the offenders in case of library-centric security threats in order to protect the reputation of benign stakeholders. A good reputation is important to both benign application developers and benign library providers in the ecosystem. In most cases, the application developers do not know all behaviors of the library they use while library providers have no control over who will use their library. There is a need for a technique to protect the reputation of the benign party when library-centric security threats happen.

1.2 Library based Isolation

Third-party libraries of Android applications have been detected to contain potentially malicious behaviors. Recent research works isolate the libraries. Good isolation mechanisms are in literature as shown in Chapter 3.

However, a successful defense requires the other important fact: the policy of the isolation. There are policies used in the evaluation part of isolation mechanisms. Normally, the policy works in that specific situation. In general, we find these policies all have some shortcoming.

- **Policy 1:** *When a library is using a sensitive API, the policy blocks it.* This policy is generated based on the assumption that if a library tries to do something sensitively, we should block it. However, most libraries are chosen for some functionality. To achieve those functionality, the library needs to invoke sensitive APIs. For instance, ZXing requires permissions to setup Wi-Fi for the application.

- **Policy 2:** *When a library is asking for the same set of permissions of its function, the policy allows it.* The policy is generated based on the assumption that if a
developer chooses a library, he should know the function of the library. In reality, this assumption is a bit strong, but still can be accepted. However, we find some libraries use the same set of permissions for its legitimate functions and its malicious functions. For instance, SMSSealingLib [18] requires permission related to SMS because it does the in-app purchase via SMS. However, this library also collects end users’ SMSes and send them to on-line server.

• **Policy 3:** *When a library is asking for the same set of permissions of its function, the policy allows it except a few permissions.* This policy is generated against the above attacks. However, this will break the application’s main functionality. ZXing uses the camera permission because it needs to scan barcode; blocking this permission blocks the legitimate functions.

The reality is that application developers do not really understand the functions of libraries well. Even if the developers understand the libraries’ functions, it is still difficult to select a suitable policy for the isolation.

### 1.3 Thesis Statement

The focus of this work is on those security problems caused by the coexistence of application logic and third-party libraries in mobile platforms. To understand how third-party libraries are used in applications, we reverse-engineer applications to collect library files for measurement of libraries usage situations in the wild.

For existing library-related security problems in applications, we propose a library integrity verification tool for Android applications at application stores. This is non-trivial because the Android application build process merges library code and application-specific logic into a single binary file. Our approach uses reverse-engineering to achieve integrity verification. The evaluation results also provide empirical insight into the library integrity situation in the wild.

To prevent the potential library-related security problems, we not only develop a system to enforce these policies, but also make an enforcement. The tool classifies the Android framework APIs in libraries into six categories via the bidirectional callsite-aware static taint analysis. The classification information can be then used to generate policies. Policies can be then used by the enforcement, that intercepts Android framework APIs through a repackaging process.
We now formulate our Thesis statement.

\textit{Despite attempts to improve mobile application security models, permissions are often still shared between applications and their third-party libraries. To make stronger guarantees about mobile application security, core logic and third-party libraries must be governed by distinct policies.}

1.4 Threat Models

This dissertation contains two major parts: the library integrity verification and the policy generation. The threat model of the library integrity verification and the threat model of the policy generation are not identical.

1.4.1 The Threat Model of the Library Integrity Verification

Trust Model: We assume application stores and security companies that use Duet are trustworthy. We also trust the library provider to provide libraries without modifications. We assume that library providers provide all versions of their libraries.

For those libraries that require the application developers to do post-processing, Duet cannot perform integrity verification. Google In-App Billing [19] is such a library. It is published as source code. Application developers are guided to modify the code and perform post-processing on it. This is a special category of libraries. Most libraries are published in bytecode and Duet can therefore be used to verify their integrity.

Reflection does not have any influence on the correctness of Duet. The integrity verification decision for a library is based on whether the library has been modified. Thus, cases where reflection is used as part of a library or to invoke library functions from the application do not affect the verification process. If the library code is modified by the application developer in a way that replaces API calls with reflection (e.g., for obfuscation), then the library cannot pass the integrity verification.

In the library integrity verification, we only support the official Android SDK and only support the existing compiler options. Duet can be extended to support customized SDK and customized compilers.

Threat Model (Assumptions): This work will focus on the three library-centric security threats mentioned in Chapter 1, namely the library modification threat, the masquerading threat, and the aggressive library threat. We assume libraries used for Android applications are public. Other stakeholders besides application developers can also access those libraries directly from the providers. To the best of our knowledge,
most libraries for Android applications are freely available online; a small number of libraries are available online with a license fee. See Section 4.5.1 for demographic study.

For those libraries that are not published online, we consider them to be proprietary libraries. Duet cannot perform integrity verification for these libraries because Duet cannot collect the original files for them. In such situations, application developers should take the responsibility for library-centric threats. If these application developers and library providers would like to verify the library integrity in Android applications, they can either use Duet themselves, or provide the libraries to third-party organizations for verification.

Applications can also load libraries at runtime. Duet cannot be used to verify library integrity in these cases because this happens dynamically. Application stores could use other techniques such as [20] for analyzing unsafe and malicious dynamic code.

Currently, Duet does not support native libraries. Native code is only used in about 6.3% of Android applications [13]. In order to verify the integrity of native libraries, it is possible to simply compare the original library files with the files within the .apk file directly because the native library will not be processed by the dx compiler.

Note that the presence of a modified library does not necessarily imply that an attack has been performed. For example, developers sometimes perform obfuscation on library code. We argue that developers have incentives not to perform such modifications. It is usually possible to avoid modifying library code even in the case of obfuscation, since obfuscating tools can be set up not to modify certain parts of the code of an application. In cases where application developers choose to modify the library for benign purposes, they prevent integrity verification. As a result, they may be deemed responsible if a library exhibits malicious or aggressive behaviors.

1.4.2 The Threat Model of the Policy Generation

Trust Model: We assume the library files can be identified with some methods. In this dissertation, the libraries are identified by their namespaces. In other cases, library files could be separated by techniques such as machine learning. If the library files cannot be separated from application logic for reasons such as application developers’ obfuscation, the libraries cannot be isolated because the boundary of the isolation is ambiguous. In that situation, the policy for the isolation is not useful.

We assume we have an Android framework API map to the behaviors that the policy would like to control. In this dissertation, the evaluation uses the permission map from existing work PScout. This map has its own limitations, and our technique inherits these
Threat Model (Assumptions): The policy generation work focuses on the generation of a policy for the library isolation. However, we don’t trust the library. The library could contain potentially malicious behaviors. These potentially malicious behaviors could not only be from the library provider in the official release, but also be inserted by other attackers. Our technique neither answers whether one behavior in the library is malicious, nor how this behavior is inserted into the library. Our focus is how to classify the Android framework APIs into six categories for the policy generation.

The policy generation does not trust the code of application logic, either. This means that the application logic could be malicious. Other techniques could be used to detect the malicious behaviors from the application logic. But, our technique focuses on the classification for the policy generation.

However, the correctness of the classification is threatened by a few facts.

- The native libraries are not well supported by the tool yet. Hence, the tool cannot be used to do the classification of native libraries at the moment.

- Java reflection is another problem. Although our technique could determine the data dependencies of the parameters and return values of the reflection, but it does not know what is reflected in the runtime if the value is only known in the runtime. Therefore, the tool cannot do the classification and generate policy for the java reflection.

- For those libraries are loaded at runtime, our technique cannot generate the policies because the tool uses static taint analysis.

- Because the classification is based on the data dependency analysis, the tool also has limitations that the data dependency analysis tool has, including inaccurately modelling the arrays for the performance, inaccurately modelling the Android application runtime environment by missing callbacks, and so on.

- Multithreading is not supported by the tool.

1.5 Contributions

In this dissertation, we make the following contributions.

- To the best of our knowledge, Duet is the first technique/tool for application stores that can verify the integrity of libraries used in Android applications.
• We ran *Duet* on 100,000 applications to test integrity of 15 different libraries. The results indicate that *Duet* is an effective solution for library integrity verification in Android applications.

• We present in-depth analysis that illustrates the library modification by application developers in the real world.

• To the best of our knowledge, we propose the first classification framework to generate the policies for the library isolation.

• We have the first implementation of such a classification framework, which includes an implementation of backwards taint analysis.

• We apply the tool to analyze 24 libraries in 7 different categories for the classification of Android framework APIs used by libraries. Compare to Policy 1, Policy 2 and Policy 3 in Section 1.2, the results indicate the generated policy has advantages.
2.1 Libraries in Android

The libraries on a smartphone can be broken down into two categories. Some libraries enable execution of the operating system. The second type of libraries is for execution of applications. Android relies on about a hundred dynamically-loaded libraries for the execution of the operating system. Some of the libraries are in fact other open source projects, such as Bionic [21]. The other libraries are generated within Android Open Source Project (AOSP) [22]. For instance, libbinder.so is the Binder library for Android inter-process communication. All these libraries are merged together within AOSP, and are made available by the Android software stack. Since these libraries are part of the Android framework, the operating system providers should verify the integrity of these libraries in various stages of the build process. In this dissertation, we do not discuss attacks on these libraries or the verification of their integrity.

This dissertation focuses on the second type of libraries, namely libraries for Android applications. In practice, these libraries are published by their creators in `.jar` or `.class` files. These libraries normally contain one or several packages that are collections of `.class` files, with each package defining a namespace for the `.class` files it contains. The Android application developers then use these libraries to build applications. The three library-centric security threats introduced in Chapter 1 target these libraries. In this dissertation, we propose a new integrity verification technique for these libraries, and this new technique can address these three threats.
Figure 2.1. Compilation process for Android applications.
2.1.1 Android Application Compilation

Android applications are developed in Java but compiled to Dalvik bytecode [23]. This bytecode runs in a platform-specific Dalvik Virtual Machine (DVM), which is optimized for devices with (relatively) low computing resources (e.g., smartphones and tablets). The compilation is generally a two-step process, as shown in Figure 2.1. In step one Java source code (.java files) are compiled into .class files. The libraries are already in the format of .class files coming from the library providers. Here, the developer could do some post-processing. In step two, all .class files are compiled into one .dex file. During the compilation process, the Java .class files composing the application are converted to a single .dex file. The main differences between .class files and the .dex file are as follows. The constant pools containing the constants used by each class are merged into a single .dex constant pool, thereby avoiding a lot of constant replication. Other changes include: register architecture, control flow structure, ambiguous primitive types, null references, and comparison of object references [13, 24].

During the above compilation process, .class files are either generated based on application source code or directly from imported libraries. Then the Dalvik dx compiler consumes .class files, and recompiles them to Dalvik bytecode, which is a .dex file. During the compilation process, even the unused .class files (for both application logic and libraries) are compiled into the .dex file.

The .dex file, and other files required by the application, such as resources, assets, certificates, and manifest file, are then put into a ZIP file formatted package based on the JAR file format. This package is called Android application package (APK) file.

2.1.2 Library Post-Processing

In the real world, libraries are often not directly compiled to Dalvik bytecode. Instead, some post-processing is done before the libraries are compiled into the .class file. Post-processing for Java .class files include shrinkage, optimization, and obfuscation. Some libraries are post-processed by library providers before release. In this dissertation, we call this type of post-processing Provider’s Post-Processing on the library. Some application developers perform post-processing on libraries before compilation of Android applications. In this dissertation, we call this type of post-processing Developer’s Post-Processing on the library. Provider’s Post-Processing helps the library providers protect source code against reversing engineering. Developer’s post-processing leads to not passing the library integrity verification as discussed in Section 4.6.
2.2 Limitation of Android Security Model

The Android platform provides a security model to guarantee isolation among applications [7]. This model is built based on the assumption that applications from different developers should be isolated. The atomic unit of protection is one application. However, in the real world, within many Android applications, application logic and libraries coexist. Application logic and libraries have different intensions because they are from different stakeholders in the ecosystem. The security model of Android platform ignores different intensions from various stakeholders within the same application. This shortcoming leads to threats and users’ requirements in the real world.

2.2.1 Android Security Model

Android is designed with the consideration of security because it is an open smartphone operating system that allows third-party applications [7]. Applications have their own Linux user and group IDs. By default, applications have only limited access to the operating system and applications cannot interact with each other. For inter-application communications and further access to the operating system, Application developers define usages by using the permission system. End users grant these permissions to applications before installation.

The Android platform security architecture prevents third-party applications disturb operating system and third-party applications from other application developers. Therefore, the atomic protection domain is one application. This model works perfectly if all application developers are responsible for the whole code in applications. However, in the real world, application developers have to use some third-party libraries without known all their behaviors. With the current security architecture, using third-party libraries bring the problem in the real world as we explained in Section 2.2.2

2.2.2 Malware and application developers

For Android applications, many application developers collect revenue by supporting advertising in the applications. To enable advertising in Android applications, advertising networks release advertising libraries. Then, application developers integrate advertising libraries during the application development process.

Advertising libraries are released in bytecode. Therefore, application developers don’t know whether the libraries have the hidden behaviors besides the advertising. In the real world, many advertising libraries are detected to have aggressive behaviors such as
collecting users’ email address [25]. These aggressive behaviors could annoy end users and make users to choose applications from competitors. So, many application developers update applications to remove madware after the detection of madware [11] [10].

The scenario above can also happen on other type of third-party libraries for Android applications, such as analytics libraries that help developers understand the end users. After the damage by the libraries, application developers could rely on Duet [26] to clear their name. However, there is need for both application developers and end users to prevent potential aggressive behaviors from third-party libraries.

### 2.3 Relationship between Library and Application Logic

Third-party libraries for Android Application have different types of relationships with the application logic based on whether and how a library provide necessary functionalities to the key functionality of the application. There are three different situations.

- All code of a library provides necessary functionalities to the key functionality of the application.
- None of a library provides necessary functionalities to the key functionality of the application.
- Parts of a library provide necessary functionalities to the key functionality of the application.

Besides the above three type relationships, there also exists the dependency among libraries in the real world. These relationships are critical for a security solution because the ignorance of these relationships will lead to mistakes in solutions.
Chapter 3

Related Work

Security and privacy in Android have become popular topics in the research community. Instead of targeting to provide solutions for traditional security problems such as malware detections, repackaging detection and so on, this dissertation aims to provide solutions to the library related security problems in the mobile platform. There also exist works about third-party libraries of Android applications. Compared to these work, the unique of this dissertation is that it provides solutions to allow application logic and libraries coexist healthy.

3.1 Smartphone Platform Security

Kirin [27] is addressing permission combinations of Android applications. Whenever third-party applications are installed, the security requirements will be checked. The limitation of this approach is that applications might collaborate to bypass the protection of Kirin. As the extension of Kirin, Saint [28] does runtime inspection on the permission state.

DeepDroid [29] is an dynamically enforcement for enterprise policy on Android devices.

For application security, dynamic taint analysis is used in both Android OS [30] and iOS [31]. The results from both operating systems are similar. End users’ private data might be leaked to servers on the Internet.

Static analysis is widely used in application security on Android OS. Enck et al. [13] and Chan et al. [32] obtain the application source code by decompiler and use existing tools to do analysis. Other works [33, 13, 34, 35, 36, 37, 12, 17, 38] do the analysis on
either the class bytecode or intermediate code. These works have detected malware and give the direction of future research to improve the security on the Android platform. Static analysis is also used to analyze the security problem caused by inter component communications [39].

All these works treat one Android application as an unit. Libraries and application logic are treated as the same. Those techniques that use static analysis and dynamic analysis can detect malicious behaviors, even if the malicious behaviors happen in the library code. However, none of these techniques can check the integrity of library code in Android applications. Hence, they can neither help the library providers clear their names for the modification threat and the masquerading threat, nor help the application developers clear their names for the aggressive library threat.

One problem in the Android platform is the confused deputy problem and the application collusion. Bugiel et al. proposed XManDroid [40] as a solution for privilege escalation and collusion. Dietz et al proposed Quire [41] to prevent the confused deputy problem.

### 3.2 Application Behavior Modification

Third-party applications are created by various developers in the world. A few developers are malicious. In order to get malicious code be installed into the end users’ phone, these malicious developers publish third-party applications which contain aggressive behaviors. Existing works [42, 43, 44, 45] attempt to solve this problem by application behavior modifications during a repackaging process.

Aurasium [42] uses repackaging technology to insert codes into the Android applications. The inserted code enables the sandbox of the application for selected behaviors. Retrofit [43] is an enforcement to modify the selected application behaviors by modification on the Dalvik byte code. Airbag [44] modifies the scope of the sandbox of Android applications to include the native Android runtime into the sandbox. However, all these solutions treat the whole application as a whole, and ignore the various stakeholders within one application.
3.3 Android Application Library

3.3.1 Library Detection

Library detection has been well studied. IDA Pro’s Fast Library Identification and Recognition Technology (FLIRT) [46] is a popular library identification technique using byte pattern matching algorithms. In another work [47], Griffin et al. detect libraries based on the heuristic that a library function cannot statically call any user-written function. Both of these techniques can be extended to detect third-party libraries for Android applications. Advertising library detection has also been studied [48]. This work uses machines learning to detect advertising libraries. The major limitation of this work as library detection is that there is no ground truth for the detected over 500 libraries. However, none of them can verify the library integrity.

3.3.2 Advertising Library

Advertising libraries have been the focus of recent works. Grace et al. [36] and Steven et al. [49] have analyzed advertising libraries in the real world. They found over one hundred types of different advertising libraries using static analysis. They found that advertising networks sometimes collect information from end users, e.g., collecting contacts in the phone. These works focus on the analysis of the behavior of advertising libraries.

Android Library Related Works Threats introduced by third-party libraries for Android applications have been analyzed by the community in different directions. The first family is to analyze the potentially malicious behaviors in the libraries. Grace et al. [36] and Steven et al. [49] have analyzed advertising libraries in the real world. They found over one hundred types of different advertising libraries using static analysis. They found that advertising networks sometimes collect information from end users, e.g., collecting contacts in the phone. [50] has done the library isolation in the .class file level.

Researchers proposed solutions to use different applications for application logic and for the advertising library [51, 52]. In this approach, the advertising library and the remaining application logic have their own protection domain. AdDroid [4] uses another approach to separate advertisements from application by supporting advertisements in a system service. Both of these solutions solve the problem that the application logic and the library share the same protection. However, for the modification threat and the masquerading threat, discussed in this dissertation, attacks still work when the library is in another protection domain. For the aggressive library threat, the problem is still
the same while the victims could be the application developers or the operating system provider based on which solution is chosen.

In addition, the android OS has to be modified to support sharing the screen with multiple applications. The limitation of these solutions is that end users might uninstall the advertising applications. To prevent the uninstallation, the Android platform has to be extended to support application dependencies. The Android platform has to add extra permissions to support advertising applications. The Application store has to verify the advertising application very carefully to prevent impostor advertising networks. One limitation of this solution is that AdDroid requires the android development teams to add advertising libraries and provides new APIs and permissions. According the result in [9], there are over one hundred advertising libraries. The implementation tasks will be non-trivial. Another limitation is that the updates from advertising networks might require the update of the whole platform.

In a recent work, Chen et al. [53] does an incomplete study of third-party libraries using in Android and iOS. Their results show that libraries containing potentially malicious behaviors widely exist. All these works focus on the analysis of the behavior of libraries and do not analyze the intra-app cross-stakeholder relationships.

### 3.3.3 Library Removal

In [54], El-Harake et al. remove the advertising libraries from Android applications through a repackaging process. The approach is based the knowledge of advertising libraries. The major limitation of the approach is that it will hurt the benefits of the application developers.

#### 3.3.3.1 Native Library

Android applications can also include the native library. According to the measurement study by Viennot et al. [55], popular applications contain more native libraries. NativeGuard [56] isolates the native library from other components of the application by putting the native library into another application. This work is different from our work because it does not modify the library behaviors. Therefore, aggressive behaviors can still function within the separated library protection domain.
3.3.3.2 Other Approaches

There are also several new approaches to support advertising in Android applications [57, 58, 59]. However, these approaches have not been widely accepted in the real market yet. It is hard to tell which type of approaches could be used as a replacement of current library usage model. Furthermore, the three library-centric security threats can happen to any type of library, not just advertising libraries. Our library integrity verification technique is a tool for any category of Android library and is able to be used in the wild.

3.4 Information Flow Analysis Tools

Information flow analysis technique has been used to analyze Android application. FlowDroid [60] is an open-source forward static information flow analysis for Android applications. FlowDroid is an context, flow, field-sensitive analysis, and thus, is precise. FlowDroid also uses an on-demand backwards taint analysis for the alias analysis to make the analysis fast. FlowDroid also models the lifecycle of Android applications to solve the problem introduced by the multiple entry points of Android applications. However, FlowDroid cannot be used to analyze the intra-app cross-stakeholder relationship efficiently with trial extension.

Epicc [61] and COAL [62] are tools to analyze the inter-component communications of Android applications. These analysers are on-demand and flow-sensitive. The inter-component communications could happen both intra-app and inter-app. Then, IccTA [63] combines FlowDroid with Epicc to detect the inter-component and inter-application information leakages. It is clear that these tools and the policy generation target different, but related research goals.

Another important work is DroidSafe [64]. DroidSafe did a less precise analysis because its analysis is not flow-sensitive. The advantages of DroidSafe is that it models the running environments more precise than other works. These modeling includes lifecycle including callbacks, inter-component communications and so on. DroidSafe even manually creates a list of sources and sinks for the analysis. All these analysis is expensive. But, DroidSafe requires about 7-8 times more time when compared to FlowDroid. Again, DroidSafe and the policy generation target different, but related research goals.

FlowTwist [65] is an interesting tool which combining the backwards analysis and forward analysis together to analyze confused deputy problems. With such a sophisticated design, the analysis scales. The limitation of this work is that there is no points-to analysis. This tool cannot be used to analyze intra-app cross-stakeholder relationship.
3.5 Isolation

The stakeholder-oriented isolation is also a hot topic recently. Researchers proposed solutions to use different applications for application logic and for the advertising library [51, 52]. In this approach, the advertising library and the remaining application logic have their own protection domain. AdDroid [4] uses another approach to separate advertisements from application by supporting advertisements in a system service. Both of these solutions solve the problem that the application logic and the library share the same protection. PEDAL [48] modifies the advertising libraries for selected four behaviors. These work do not touch how applications’ main functionality use library. All of them only works in the special case: advertising library which has no library usage at all.

All these works do the intra-app isolation based on different stakeholders. None of them formally analyzed the relationship among the application’s main logic and libraries. Such solutions may break the application. A few works focus on the advertising libraries with the assumptions that advertising libraries are unnecessary for the main application functionality.

3.6 Other Security Solutions to Android Applications

3.6.1 Repackaging

Researchers have noticed repackaged Android applications. Zhou et al. [9] found 86% of malware is repackaged. Researchers analyze either intermediate codes (smali) or java classes (from reverse engineering) of the applications to generate CFGs and program dependence graphs (PDGs) [17, 16, 66]. The CFG and PDG are then used to detect the repackaged applications. These solutions are targeting to application plagiarism. They require removing library code for making decisions on potential repackaging. Therefore, these techniques cannot be used to address the three library-centric security threats.

For repackaging detection techniques, the purpose is to detect the repackaged applications. Applications have been modified during the repackaging process are still considered as repackaged. A few methods say that they can even detect the obfuscated repackaging, because they calculate the similarity based on things that will not be modified during the obfuscation. For example, a public activity cannot be modified during the obfuscation; otherwise, the application will not work.
3.6.2 Code Clones

Besides repackaging detection for Android applications, both the detection of similar software applications and the detection of code clones have been well studied in [67] and [68] respectively. These techniques detect similarity between applications by comparing strings, tokens, trees, or semantics. None of them can be used to verify the software integrity.

3.6.3 Inline reference monitors for Java applications

Reference monitor concepts have been widely used in different application. The idea has been used for Java applications [69] [70]. The different between our work and the existing work is: 1) our work focuses on the library; 2) our work targets the policy generation.

3.7 Malware Detection

Malware detection is an important topic for security. MIGDroid [71] uses a signature-based approach to detect the malware. It first extracts API call sequences and then constructs the method invocation graph. Then, it will generate the subgraphs and calculate a threat score according to the sensitive APIs invoke in the subgraph. If the score exceeds a threshold, the module in the application corresponding to the application is considered as malware.
Chapter 4

Duet: Library Integrity Verification

4.1 Introduction

The Android operating system holds the biggest market share in the world of smartphones [2]. For such a success, third-party applications play an important role in the whole ecosystem [3]. These application developers integrate libraries into their applications for different purposes. For example, advertising-supported free applications have become popular in the world of Android. An advertising network distributes advertising libraries. Third-party application developers collect revenue by adding these advertising libraries into their applications. Pearce et al. found that 49% of applications in their dataset is supported by advertising libraries [4].

Besides advertising libraries, Android application developers also include other libraries into their applications. For example, Zebra crossing (ZXing) [5] is a library that provides functionality of 1D/2D barcode image processing. Another example is the license verification library (LVL) [6] by Google. With LVL, applications can query Google Play to obtain their license status at runtime.

Three Library-Centric Security Threats: Unfortunately, third-party libraries that come with applications can be modified to be malicious. For example, AntiLVL [8] is a free tool available online that modifies third-party libraries in applications in order to subvert standard license protection methods such as Amazon Appstore DRM and Verizon DRM. Such attacks require reverse-engineering and a build process, which is called a repackaging process.

Besides modifying existing libraries, attackers could also create a new malicious library. To make people believe this is a good library, attackers usually would do
masquerading. For example, use the same namespace that is used by good libraries. For instance, as reported in [9], DroidKungFu, a famous malware, uses names such as `com.google.ssearch` and `com.google.update` to pretend to be published by Google for legitimate and benign purposes.

In addition to the modification threat and the masquerading threat mentioned above, it has also been proven that some legitimate third-party libraries have aggressive behaviors such as collecting device owner’s email address. Based on the report from FireEye Blog [10], one popular advertising library that has aggressive behaviors has been used in over 1.8% of the applications in their dataset and these affected applications have been downloaded more than 200 million times in total. As reported in the follow-up news [11], benign application developers remove problematic third-party libraries after being notified.

**Limitation of Existing Techniques:** The three library-centric security threats mentioned above cause very serious consequences in the real world. However, we find they cannot be effectively fully addressed by existing defense mechanisms. Although software analysis [12, 13], anti-virus software [14, 15] and anti-repackaging techniques [16, 17] can detect some malicious behaviors in library code, it is hard to tell whether library providers or application developers are the offenders because the application developers can modify library code during the development process.

In the case of the modification threat, the application developer could blame the library provider for malicious library behaviors, even though the application developer is the real attacker and the library provider is a victim of the attack. Regarding the masquerading threat, the application developer could also blame the library provider for malicious library behaviors, even though the real attacker is the application developer and the victim is the library provider. Regarding the aggressive library threat, the library provider could blame the application developers for aggressive library behaviors because the developer can modify the library. In this case, the real attacker is the library provider and the victim is the application developer.

However, it is important to figure out the offenders in case of library-centric security threats in order to protect the reputation of benign stakeholders. A good reputation is important to both benign application developers and benign library providers in the ecosystem. In most cases, the application developers do not know all behaviors of the library they use while library providers have no control over who will use their library. There is a need for a technique to protect the reputation of the benign party when library-centric security threats happen.
Research Objective: To address the limitation of existing techniques, we develop an integrity verification technique. When Android applications are submitted to Android application stores, testing the integrity of third-party libraries in applications can effectively address library-centric threats. Third-party libraries that become malicious after modifications and masquerading libraries cannot pass library integrity verification. This guarantees that the library provider is not the attacker. However, if the malicious behaviors come from aforementioned legitimate and problematic third-party libraries, benign application developers should be protected. If these problematic libraries pass library integrity verification, it proves that the malicious behaviors in these libraries are from the library providers, instead of application developers.

Verifiers for library integrity: How to verify the integrity of libraries relies heavily on who is going to do the verifications. It is clear that we have at least two candidates: application developers and application stores. Obviously, it is straightforward for application developers to verify the library integrity during the build process. For example, developers can verify library integrity by comparing libraries’ checksums with checksums from the library providers. However, letting the developers do verification cannot effectively fully address the three library-centric security threats because developers can modify the library. As a result, the limitations of existing techniques are still not addressed.

Realizing that letting developers do verification is not enough, we look into the reason behind it: application developers and third-party library providers are different stakeholders in the ecosystem. Their code has different intentions while they do form a symbiotic relationship. Therefore, a suitable verifier for library integrity cannot be its symbiont, the application developer. We find if stores can do the verification successfully, the three library-centric security threats can be very well addressed, due to the same reasons in Research Objective.

Challenges for store side verification: There are three major challenges for the integrity verification of Android application libraries by application stores. First, in the Android application build process, library code and application-specific logic are “blended into pieces”, mixed and merged into a single binary file. The application store cannot tell whether the library has been modified before compilation by reading the application binary directly. Second, in most cases, the application store cannot get the source code of Android applications to repeat the Android application compilation process for integrity verification. Finally, library files reverse-engineered from the application binary are different from original library files collected from their provider, so
that the application store cannot just use reverse engineering for integrity verification.

**Our approach:** To overcome these challenges, we propose Duet: a library integrity verification tool for Android applications at application stores. Duet first collects the original library files from their providers. With the observation that reverse-engineered library files go through a build process and a reverse-engineering process, Duet takes a novel mirroring approach in which original files also go through a build process and a reverse-engineering process in order to create reference files. Library files reverse-engineered from applications that use unmodified libraries are exactly the same as reference files. Duet builds the reference database that stores all these reference files and their digests (checksums). In particular, we use Dare [24] as our reverse-engineering tool. The reverse-engineering is also called retargeting in this Chapter.

Duet needs to use original library files to build the reference database. This assumes that third-party libraries used for Android applications are public. Considering that application developers can access libraries, it is reasonable to assume that other stakeholders in the ecosystem can also access those library files directly from the providers. With the reference database, application stores can then directly verify library integrity in applications.

**Our Main Contributions:** Our main contributions are as follows:

- To the best of our knowledge, Duet is the first technique/tool for application stores that can verify the integrity of libraries used in Android applications.
- We ran Duet on 100,000 applications to test integrity of 15 different libraries. The results indicate that Duet is an effective solution for library integrity verification in Android applications.
- We present in-depth analysis that illustrates the library modification by application developers in the real world.

### 4.2 Security Model and Problem Statement

#### 4.2.1 Security Model

**Trust Model:** We assume application stores and security companies that use Duet are trustworthy. We also trust the library provider to provide libraries without modifications. We assume that library providers provide all versions of their libraries.

For those libraries that require the application developers to do post-processing, Duet cannot perform integrity verification. Google In-App Billing [19] is such a library. It
is published as source code. Application developers are guided to modify the code and
perform post-processing on it. This is a special category of libraries. Most libraries are
published in bytecode and Duet can therefore be used to verify their integrity.

Reflection does not have any influence on the correctness of Duet. The integrity
verification decision for a library is based on whether the library has been modified.
Thus, cases where reflection is used as part of a library or to invoke library functions
from the application do not affect the verification process. If the library code is modified
by the application developer in a way that replaces API calls with reflection (e.g., for
obfuscation), then the library cannot pass the integrity verification.

In this Chapter, we only support the official Android SDK and only support the exist-
ing compiler options. Duet can be extended to support customized SDK and customized
compilers.

Threat Model (Assumptions): This work will focus on the three library-centric
security threats mentioned in Chapter 1, namely the library modification threat, the
masquerading threat, and the aggressive library threat. We assume libraries used for
Android applications are public. Other stakeholders besides application developers can
also access those libraries directly from the providers. To the best of our knowledge,
most libraries for Android applications are freely available online; a small portion of
libraries are available online with a license fee. See Section 4.5.1 for demographic study.

For those libraries that are not published online, we consider them as proprietary
libraries. Duet cannot perform integrity verification for these libraries because Duet
cannot collect the original files for them. In such situations, application developers
should take the responsibility for library-centric threats. If these application developers
and library providers would like to verify the library integrity in Android applications,
they can either use Duet themselves, or provide the libraries to third party organizations
for verification.

Applications can also load libraries during runtime. Duet cannot be used to verify
library integrity in these cases because this happens dynamically. Application stores
could use other techniques such as [20] for analyzing unsafe and malicious dynamic code.

Currently, Duet does not support native libraries. Native code is only used in about
6.3% of Android applications [13]. In order to verify the integrity of native libraries, it
is possible to simply compare the original library files with the files within the .apk file
directly because the native library will not be processed by the dx compiler.

Note that the presence of a modified library does not necessarily imply that an attack
has been performed. For example, developers sometimes perform obfuscation on library
code. We argue that developers have incentives not to perform such modifications. It is usually possible to avoid modifying library code even in the case of obfuscation, since obfuscating tools can be set up not to modify certain parts of the code of an application. In cases where application developers choose to modify the library for benign purposes, they prevent integrity verification. As a result, they may be deemed responsible if a library exhibits malicious or aggressive behaviors.

4.2.2 Problem Statement

Although developers can verify the library integrity, letting developers perform integrity verification cannot effectively address the three library-centric security threats. It is clear that application store side verification is necessary and critical.

Problem Statement: How to enable application stores to do library integrity verification without cooperating with application developers (without knowing the source code of applications).

4.2.3 Use Cases

In the Smartphone ecosystem, different stakeholders would like to see library integrity verification results for various purposes. Library providers care about the library integrity because the library modification threat hurts the providers’ benefits. Library providers also care about the masquerading threat because malware in their libraries could hurt their reputations. Because the library providers have no control of the application development process, they need store side library verification technique. In special cases, the application store is also the library provider, for instance Amazon and Google. In fact, Amazon is the victim of the library modification threat [8, 16] and Google is the victim of the masquerading threat [9]. Hence, they have motivations to perform store side library integrity verification to protect themselves.

We find that application developers also need store side library verification. Although application developers can verify the library integrity during the build process, the store still has no information about the library integrity status. In the case of the aggressive library threat [10], benign application developers need library integrity verification to prove their innocence [11]. Of course, developers can submit all source code and build configuration for repeating the build process as evidence. However, it is not only a bad idea to give source code to others, but also an extra burden for developers to keep source code and configuration for each version of applications. The store side verification can solve all these concerns.
4.3 System Overview

4.3.1 Naive Solutions and Challenges for Store Side Verification

We find store side library integrity verification is non-trivial because the store has no control about the Android application build process. Here, we discuss several naive solutions and explain why they cannot work. One simple method to verify library integrity is to collect library files from the application package, and compare these files with the library files that are from the library provider. However, this method cannot work for Android applications. During the compilation of Android applications, the .class files of both libraries and application logic are merged together into one single .dex file. Fields and methods from each .class file are separated, and stored in different locations of the .dex file. Because each application has various .class files as application logic, the compilation process generates different .dex files for different applications even if they use the same library. To verify one must locate every piece of the library and put together each .class file. However, this is a very complicated thing to do. That is without reverse engineering, there seems to be no way to collect the library file.

The second method to achieve library integrity verification is to repeat the compilation process of applications. If we have the java source code or the .class files of one Android application, we can repeat the Android application compilation process to generate the .dex file. Then, we can verify the library integrity by comparing the generated .dex file with the .dex file in the application package. However, this method requires source code or .class files of this application. In the real world, we cannot collect these data for millions of applications. In practice, it is very difficult to convince most application developers to release source code or .class files. Therefore, this method can only be used in special cases when source code or .class files are available.

Therefore, we have to use reverse engineering technology to get the .class files of libraries from .dex files for library integrity verification. Another method is to compare the .class files after reverse engineering with the original library files. The reverse-engineered (retargeted) files are functionally equivalent to the original files, however they are not exactly the same. For example, original files may include debugging information, which is not necessary for their normal execution. This debugging information is not recovered by the retargeting process. As a consequence, the retargeted files cannot directly be compared to the original library files from the library providers.

Challenges: To address this problem, we need to handle the following challenges:

- C1: The Android application compilation process mixes .class files of library and
application logic together. The application store cannot verify the integrity of the libraries by reading the binaries of the `.dex` files.

- **C2:** The application store cannot get the source code of the application logic, and the application store has no knowledge of development configuration for each application. Therefore, repeating the compilation process of applications to achieve integrity verification of libraries is not feasible.

- **C3:** Library files from reverse engineering are different from original library files collected from their provider. Therefore, comparing `.class` files after reverse engineering and `.class` files before compilation to achieve integrity verification is not feasible.

### 4.3.2 Our Idea (Double Reverse-Engineering)

To overcome these challenges, we have to collect library files from the applications by using reverse-engineering. After that, we also need a correct method to compare the reverse-engineered library files with the original files from library providers. With the observation that reverse-engineered library files go through a build process and a reverse-engineering process, *Duet* takes a novel mirroring approach in which original files also go through a build process and a reverse-engineering process to create reference files. Library files reverse-engineered from applications that use unmodified libraries are exactly the same as reference files. *Duet* builds the reference database that stores all these reference files and their digests (checksums).

### 4.3.3 Architecture of Duet

As shown in Figure 4.1, *Duet* has three major modules. Module A builds the reference database; Module B processes applications; Module C compares results from Module B with the reference database for integrity verifications.

In Module A, *Duet* first downloads original libraries from the library providers. Then, *Duet* compiles original libraries to Dalvik bytecode and retargets them to get the retargeted `.class` files, which are the reference files. *Duet* merges the content of all `.class` files of a given library into a single file and calculates the hash value of this file. This value is called *canonical library digest*. *Duet* also calculates a hash value for each `.class` file of this library. These values are called *canonical file digests*. We explain how *Duet* uses these digests when we explain Module C. The canonical library digest, canonical file digests, and reference files are all stored into the reference database.
For an Android application, Module B collects retargeted .class files of its libraries after retargeting. For a given library, retargeted .class files are used to calculate its library digest after being merged together. The library digest is a hash value. For every .class file of the library, Module B calculates its file digest that is also a hash value.

Then, Module C first compares the library digest with canonical library digests in the reference database. Once a match is found, the library passes the integrity verification. Otherwise, Module C compares all file digests with canonical library digests in the reference database. If every file digest can match, the library also passes the integrity verification.

Compared to file digests, calculation of library digest requires only one hash calculation. Hence, it is fast. Finding a match with canonical library digest in the reference database means that the library has not been modified. It requires that there are neither extra .class files nor missing .class files in libraries.

However, the above ideal situation does not always happen in the real world. One possible situation is that some application developers use a shrinker to remove the unused .class files before Android application compilation in order to reduce the size of the application. Missing some .class files cannot lead to meaningful security attacks. Hence, Duet uses file digest comparison to tolerate it.

Another situation in the real world is that several libraries from the same provider
might share the same namespace. For example, Android support library [72] has three different libraries in the directory “/android/support/”. One application might contain all these three libraries. We find this is happening in the real world. Duet also uses file digest comparison to tolerate this situation.

4.3.4 Why this solution works?

The original library files (.class files) are first compiled into Dalvik format, and then during the retargeting process the library files (.class files) are generated again. These two types of libraries files are semantically equivalent because changes happened in the Dalvik compilation have been recovered during the retargeting process. In particular, the changes and the recoveries are as follows.

• During the Dalvik compilation, all .class files are merged together as a single one .dex file. In the retargeting process, parsing class and method data and fields is straightforward because the Dalvik format containing the information to recover this change.

• During the Dalvik compilation, type information has been changed. In the retargeting process, a strong type inference algorithm is used to recover the type information. Here, the Dare tool first generates constraints on types based on definitions and uses. These constraints are then solved to infer unknown types.

• During the Dalvik compilation, the Java bytecode is translated into Dalvik opcodes. During the retargeting process, 9 rules are generated to translate all 257 Dalvik opcode into Java bytecode.

Although it is not straightforward, the Dalvik file contains enough information to recover the semantically equivalent .class files. Dare [24] is such a tool to achieve this goal. Our integrity verification tool is built upon Dare.

4.4 Design and Implementation

4.4.1 Reverse-Engineering Requirements

Our double reverse-engineering idea requires the following two properties which enable our design to work very well.
Property 1: Distinctiveness. Different libraries compiled into Android applications should get retargeted to different code. In other words, it should be possible to distinguish the code from different libraries after the retargeting process.

Property 2: Identity. If the same library is compiled into different applications, retargeting these applications should yield bytecode for the library that is identical across all applications (Identity guarantee). In other words, the retargeting process enables us to recognize when a library has been integrated into different applications.

We select the Dare tool [24] as our reverse-engineering tool because we find that it provides these two guarantees. The first one is trivial. Different libraries A and B have semantically different Java bytecode, which gets compiled to semantically different Dalvik bytecode. The Dare retargeting process is formally defined and ensures that code semantics is preserved from Dalvik to Java bytecode. Thus, retargeting the Dalvik bytecode of A and B results in different Java bytecode.

Further, Dare provides identity guarantees. When a given library is compiled into different applications, it get compiled to very similar bytecode. The class names, field names, method names and the structure of the method code are identical. The only difference occurs when an instruction references a constant (e.g., integer or string constant) by using an index to a constant pool element. Since the Dalvik compilation process merges all Java constant pools together, constant indices for the library are different between applications. The retargeting process is such that, despite the differences in constant pool indices in the Dalvik bytecode, the indices in the retargeted code of the library are the same. The reason why this is true is that Dare uses Jasmin [73] for bytecode assembly. In order to use a constant in Jasmin code, the value of the constant has to be textually “described”. For example, an integer constant is described by its value. Also, a method reference is described by the signature of the method and the name of its declaring class. This description only depends on the value of the constant and not on its original index in the .dex constant pool. Thus, the Jasmin code for a given library is the same after retargeting different applications that contain the library. This in turn results in identical Java bytecode after assembling the Jasmin code.

4.4.1.1 Potential Attack Surface

Evading Library Integrity Verification by Duet (False Negative): It is fatal for an integrity verification tool if the tool/technique can be evaded by attackers. In our
problem, this means that a library is detected as unmodified, even though it was in reality modified. In order for a library to be detected as unmodified, its retargeted files have to be strictly identical to the reference retargeted files. Thus, false negatives can only occur if different Dalvik code map to the same retargeted files. The only parts of the .class files that have an influence on runtime behavior (and are therefore potential targets of attacks) are the fields and methods. Field declarations are simply composed of a type, a name, and in some case an initial value. Thus any modification would be detected by Duet. Method code is more complex, with 257 possible kinds of instructions. However, as described in [24], the mapping between Dalvik and Java bytecode is unambiguously defined. There are rare cases where different Dalvik bytecode map to the same Java bytecode. However, in these cases the different Dalvik bytecode structures are semantically equivalent. For example, there are two ways to fill an array with data in Dalvik bytecode. One is to add data to the array one element at a time. The second one consists in using a single fill-array-data instruction. Both cases are translated to Java bytecode using the same pattern. That is because they are semantically equivalent. In addition to this example, we have considered all other cases where different Dalvik instructions patterns map to similar Java bytecode patterns and in all cases the Dalvik semantics are the same. As a result, while it is possible that our approach may miss some code modifications, the modified code would not be semantically different. In particular, malicious modifications cannot go undetected.

4.4.2 Building the Reference Database

When building the reference database, Duet first downloads the original libraries from library providers. Then, Duet calculates the canonical library digests and canonical file digests for libraries.

4.4.2.1 Canonical Library Digest

As shown in Figure 4.2, Duet takes five steps to calculated the canonical library digest: compilation, retargeting, library directory information collecting, library encoding, and digest calculation.

Compilation: Duet uses the Dalvik dx compiler to generate the .dex file. Different versions of dx generate different .class files. When we build the reference database, we should consider all possibilities. We go through all versions of Android application SDK and we find that there are four different versions of dx so far. We also find two of them generate the same result.
Figure 4.2. Calculating the canonical library digest and canonical file digest for original libraries.
Figure 4.3. Calculating library digest and file digest for libraries in Android applications.
**Retargeting:** We use Dare [24] to retarget the .dex to Java .class files. Dare offers a feature, which consists in rewriting unverifiable (i.e., malformed) Dalvik bytecode to generate verifiable Java bytecode. This feature could potentially lead to application-specific modifications of a library. Indeed, one of the main causes of unverifiability is that sometimes Dalvik bytecode refers to missing classes [24]. If such a class is included in one application $A$ but is excluded in application $B$, then Dare would rewrite the code in application $B$ but not in application $A$ (since $A$ does not have a missing class). Therefore, we deactivate the rewriting feature in Dare.

**Library Directory Information Collecting:** After retargeting, the retargeted .class files are organized according to their package name. For example, .class files from AdMob [74] are located in directory “/com/google/ads/”. Duet needs this information in order to separate the .class files of the library from other parts of applications in the future.

**Library Encoding:** A single library commonly has hundreds of .class files. During the file encoding, we merge the content of all retargeted .class files of the library into one file.

**Digest calculation:** Once Duet gets the encoded file from the library encoding step, Duet calculates a cryptographic hash as the canonical library digest of the particular library.

After the above process, Duet stores the following information into the reference database: the library provider, version, canonical library digest, and library directory information.

### 4.4.2.2 Canonical File Digest

For each original library, Duet calculates a cryptographic hash for each retargeted .class file, which is a canonical file digest. These canonical file digests and the names of .class files are also stored into the reference database.

### 4.4.2.3 Issues

For the reference database, it is critical to store all legitimate canonical library digests and legitimate canonical file digests. Otherwise, Duet will make wrong conclusions for the integrity verification. It requires Duet not only supports legitimate behaviors of library providers but also supports all legitimate settings in the developers’ build processes. In particular, Duet needs to solve the following issues: (1) Collect all history versions of original libraries from their providers; (2) Support all versions of dx compilers; (3)
Support all possible options of $dx$ compilers. We explain how $Duet$ solves these three issues in Section 4.5.3.

4.4.3 Library Integrity Verification in Applications

As shown in Figure 4.3, the integrity verification in applications also needs five steps: decompression, retargeting, library separating, library encoding, and digest calculation. Retargeting, library encoding, and digest calculation are the same as the steps when building the reference database. The other two steps, decompression and library separating, are explained as follows.

- **Decompression:** In the library integrity verification, $Duet$ gets the $\text{classes.dex}$ from the $\text{.apk}$ files of Android applications. The $\text{.apk}$ files are in zip format. We use “unzip” to get the $\text{classes.dex}$. We also use this step to verify that the application does not be damaged during network transmissions.

- **Separating library and application logic:** After retargeting, the retargeted $\text{.class}$ files are organized according to their package names. With the library directory information in the reference database, we can easily separate libraries from other application logic.

4.5 Evaluation

We have two main evaluation goals. First, we want to do library integrity verification against 100,000 Android applications in the wild to assess the extent to which $Duet$ can help address the three library-centric security threats in the real world. Our measurements directly help the potential victims of the aggressive library threat to clear their names. The measurements also estimate an upper bound on how many library usages in the wild could suffer from the modification threat and the masquerading threat. The second goal is to validate the decisions made by $Duet$. In particular, $Duet$ makes two types of decisions: (1) one library passes the integrity verification, and (2) it does not pass the integrity verification. We want to evaluate whether the decisions are trustworthy. So we will do in-depth analysis regarding whether $Duet$ is making any incorrect decisions.

Our dataset has 100,000 applications downloaded from Google Play between February 2012 and September 2013. For applications that have multiple versions, only the latest version is included in the dataset. In Section 4.5.1, we detect the top 100 libraries used in these applications, and analyze the library usage. Then, we evaluate $Duet$ by both
in-lab testing and measurements on this dataset.

### 4.5.1 Libraries in Android Applications

For all 100,000 Android applications, we use Dare to do retargeting to get all `.class` files. After that, we scan Java namespaces, and count how many times each particular namespace is used. With namespace list sorted by frequency, we map namespaces to libraries with a manual online search. This process is repeated until we collect the top 100 Android application libraries. Considering that a popular library is usually reused in various applications, we get a list of most popular libraries in our dataset by this method. Figure 4.4 shows the top 20 detected libraries.

During our manual library mapping, we also identify the category of each library and its source code availability information. As shown in Table 4.1, there are 39 utility libraries meant to facilitate the application development process. For instance, the Android support library from Google can simplify the process of targeting different hardware. Many of these libraries are based on open source projects or are open source projects themselves. 87.18% of libraries in this category have source code available.
Figure 4.5. Number of Libraries Contained by Each Application.

Table 4.1. Categories of the top 100 detected libraries, and their source code available situation.

<table>
<thead>
<tr>
<th>Category</th>
<th># of Libraries</th>
<th># of Source Available</th>
<th>Source Available Percent (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>App-Dev</td>
<td>39</td>
<td>34</td>
<td>87.18%</td>
</tr>
<tr>
<td>Advertising</td>
<td>34</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Service</td>
<td>10</td>
<td>2</td>
<td>20.00%</td>
</tr>
<tr>
<td>Analytics</td>
<td>9</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Game</td>
<td>8</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td><strong>36</strong></td>
<td><strong>36.00%</strong></td>
</tr>
</tbody>
</table>
Table 4.2. Usage information of the top 100 detected libraries.

<table>
<thead>
<tr>
<th># of Category</th>
<th>Usage (%)</th>
<th>Percent in Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>App-Dev</td>
<td>149,291</td>
<td>54.03%</td>
</tr>
<tr>
<td>Advertising</td>
<td>84,617</td>
<td>30.62%</td>
</tr>
<tr>
<td>Service</td>
<td>16,762</td>
<td>6.07%</td>
</tr>
<tr>
<td>Analytics</td>
<td>17,458</td>
<td>6.32%</td>
</tr>
<tr>
<td>Game</td>
<td>8,189</td>
<td>2.96%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>276,317</strong></td>
<td><strong>100.00%</strong></td>
</tr>
</tbody>
</table>

The next popular category is advertising. We detected 34 advertising libraries. Other 10 libraries are providing important services to the Android applications. We name these libraries as the service libraries. One example is Google Play Billing library, which provides the service of in-application purchase. Another library category is the analytics library. An analytics library helps developers know how and when users use their applications. 9 libraries in the top 100 libraries fell into this category. The last category of libraries is the game library. There are 8 libraries in this category. The source code of most of these libraries is not available, most likely because they aim at monetizing applications.

With the top 100 library list, we then check how widely libraries are used in Android applications. As shown in Figure 4.5, one application that uses 22 libraries as the extreme case. In our dataset, 83,044 (83.04%) applications use at least one library in the top 100 list.

As shown in Table 4.2, we detect 276,317 library usage cases in total. 149,291 occurrences (54.04%) happen in the App-Dev category. These usages happened in 66,035 (66.04%) different applications. The remaining 127,026 (45.97%) usages happen in the following categories: advertising, service, analytics, and game. Correspondingly, these usages happen in 60,009 (60.01%) different applications.

Generally, the above results indicate that libraries are widely used in the Android applications. For those libraries having business behind, the modification threat could cause damages. 60.01% of applications in our dataset contain these libraries. For library providers, the masquerading threat could hurt library providers’ reputation. For benign application developers, the aggressive library threat could lead to being excluded from application stores. In other words, a tool that can address the three library-centric secu-
4.5.2 In-lab testing of the correctness of Duet

To ensure that we know 100% of the ground truth, instead of using a real-world library, we create a library by ourselves and denote it as the “original library”. Then, we modify it manually with two different methods: (1) we modify the java source code of the library, get the modified .class files using a Java compiler, and compile these .class files to an Android application; (2) we first build an application that uses the “original library”; then, we use reverse engineering tools [75] to repackage this application and modify the library during the process. In the modifications, we make changes on different targets including APIs, fields, and .class files. Finally, we use Duet to perform integrity verification between the original library and the modified ones. As shown in the following table, Duet detects all these manual modifications.

<table>
<thead>
<tr>
<th>Attack Methods</th>
<th>Class Modification</th>
<th>Repackaging</th>
</tr>
</thead>
<tbody>
<tr>
<td>API Removal</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>API Addition</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>API Modification</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>Field Removal</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>Field Addition</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>Field Modification</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>.class File Removal</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
<tr>
<td>.class File Addition</td>
<td>Not Pass</td>
<td>Not Pass</td>
</tr>
</tbody>
</table>

4.5.3 In-the-wild Library Integrity Verification Results

After in-lab testing, we then perform in-the-wild library integrity verification. We need to build the reference database with the real-world libraries before integrity verification.

**Building the Reference Database:** As we explained in Section 4.4.2.3, it is critical but non-trivial to build the reference database. The first requirement is to collect all history versions of third-party libraries. It is not difficult for application stores to request all versions from library providers by requesting them. But, it is difficult for us because we cannot afford the communication costs with a large number of library providers. Further, our demand could be ignored for various reasons. For instance, some old versions have vulnerabilities such that the provider does not want to provide them.
Table 4.3. Information of the Reference Database.

<table>
<thead>
<tr>
<th>Rank</th>
<th># of Version</th>
<th># of Canonical Library Digest</th>
<th># of Canonical File Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdMob</td>
<td>1</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>Android Support</td>
<td>2</td>
<td>41</td>
<td>656</td>
</tr>
<tr>
<td>AppBrain</td>
<td>40</td>
<td>89</td>
<td>1,424</td>
</tr>
<tr>
<td>AdFonic</td>
<td>57</td>
<td>9</td>
<td>144</td>
</tr>
<tr>
<td>Others</td>
<td>63</td>
<td></td>
<td>252</td>
</tr>
<tr>
<td>Total</td>
<td>218</td>
<td>2,732</td>
<td>252,404</td>
</tr>
</tbody>
</table>

However, we still manage to collect all known versions for 4 libraries. For example, we download all known versions of AdMob based on its release note [74]. Besides the above 4 libraries, we collect some versions for another 11 libraries.

For these 4 libraries with all known versions, the total number of usage cases is 70,275. These libraries cover both the most popular closed source library and the most popular open source library, as well as contain libraries from both well-known library providers and relatively small library providers. Analysis on them provides empirical insight into library integrity situation in the wild.

For the dx compiler, we go through all versions of the Android application SDK and find that there are four different versions of dx so far. We also find two of them generate the same result. In addition, dx compiler has two working optimization options, Duet supports all combinations of these options.

Overall, the reference database takes 6.28GB. As shown in Table 4.3, the reference database contains 218 original libraries, 2,732 canonical library digests, and 252,404 file digests. The rank in the table is the rank of the library in the top 100 list.

Library integrity verification results: Now, we are ready to use real-world libraries and applications to evaluate the effectiveness of Duet. Table 4.4 shows the integrity verification pass ratio of 4 libraries with all known versions. The highest passing rate is achieved with AdFonic with 85.46%; the lowest passing rate is 72.82%. These numbers indicate that libraries are not modified after release in 80.50% of cases on average in the wild.

The remaining 19.50% of cases do not pass the integrity verifications. These not
Table 4.4. In-the-wild Library Integrity Verification Results. (All Known Versions)

<table>
<thead>
<tr>
<th>Rank</th>
<th># of Detection</th>
<th>Library Digest</th>
<th>File Digest</th>
<th>Compilation with Customized Options</th>
<th># of Passings</th>
<th># of not Passings</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdMob</td>
<td>1</td>
<td>35,726</td>
<td>28,264(79.11%)</td>
<td>40(0.11%)</td>
<td>10(0.03%)</td>
<td>28,314(79.25%)</td>
</tr>
<tr>
<td>Android Support</td>
<td>2</td>
<td>32,002</td>
<td>23,243(72.63%)</td>
<td>37(0.12%)</td>
<td>24(0.07%)</td>
<td>23,304(72.82%)</td>
</tr>
<tr>
<td>AppBrain</td>
<td>40</td>
<td>1,522</td>
<td>1,050(68.99%)</td>
<td>235(15.44%)</td>
<td>0(0.00%)</td>
<td>1,285(84.43%)</td>
</tr>
<tr>
<td>AdFonic</td>
<td>57</td>
<td>1,025</td>
<td>876(85.46%)</td>
<td>0(0.00%)</td>
<td>0(0.00%)</td>
<td>876(85.46%)</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>(76.55%)</td>
<td>(3.92%)</td>
<td>0(0.03%)</td>
<td>(80.50%)</td>
</tr>
</tbody>
</table>
passing, however, do not always mean malicious library modification or masquerading library. In Section 4.6, we will look into these not passing cases and do in-depth analysis.

We perform the integrity verification on 11 libraries with some versions in order to check whether Duet can be used for other Android libraries. Table 4.5 shows the integrity verification pass ratio of these libraries. The highest passing rate is achieved with OldAdMob\(^1\) with 69.32%; the lowest passing rate is 2.71%. Every library has some samples that can pass. This indicates that Duet is a tool that can handle integrity verification of any Android library.

In both Table 4.4 and Table 4.5, some samples pass the integrity verification by using file digest. This fact indicates that Duet does tolerate two real-world issues aforementioned in Section 4.3.3 by introducing the file digest as the supplement of the library digest.

In Table 4.4, 80.47% of samples pass the integrity verification by matching the canonical digests generated with the default dx options. At the same time, 0.03% of samples pass the integrity verification with the canonical digests generated with customized dx options. This indicates that (1) Duet does tolerate different dx compiler options; (2) Very few application developers customize the dx compiler options.

\(^1\)OldAdMob was the library released by AdMob when it was an independent company.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Library</th>
<th># of Detection</th>
<th>Library Digest</th>
<th>File Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>BugSense</td>
<td>8,156</td>
<td>221(2.71%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>7</td>
<td>Flurry</td>
<td>6,741</td>
<td>2,107(31.26%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>12</td>
<td>Millennial Media</td>
<td>4,296</td>
<td>1,864(43.39%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>15</td>
<td>InMobi</td>
<td>3,344</td>
<td>201(6.01%)</td>
<td>401(11.99%)</td>
</tr>
<tr>
<td>18</td>
<td>OldAdMob</td>
<td>3,126</td>
<td>2,167(69.32%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>24</td>
<td>AdWhirl</td>
<td>2,458</td>
<td>1,183(48.13%)</td>
<td>2(0.08%)</td>
</tr>
<tr>
<td>25</td>
<td>Mobclix</td>
<td>2,403</td>
<td>647(26.92%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>31</td>
<td>RevMob</td>
<td>1,864</td>
<td>225(12.07%)</td>
<td>1(0.05%)</td>
</tr>
<tr>
<td>37</td>
<td>MobFox</td>
<td>1,679</td>
<td>188(11.20%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>86</td>
<td>ZestAdZ</td>
<td>521</td>
<td>356(68.33%)</td>
<td>0(0.00%)</td>
</tr>
<tr>
<td>88</td>
<td>Cauly</td>
<td>504</td>
<td>66(13.10%)</td>
<td>0(0.00%)</td>
</tr>
</tbody>
</table>
4.5.4 Implications of Passing Rates

As we mentioned at the beginning of Section 4.1, if we can fully trust the passing conclusions made by Duet, the three library-centric threats can be effectively addressed. Third-party libraries that become malicious after modifications and masquerading libraries cannot pass library integrity verification. This guarantees that library provider is not the attacker. The measurements estimate 19.50% library usages in the wild have been modified in the development process. They are the upper bound of how many library usages in the wild could suffer from these two threats. However, if the malicious behaviors come from aforementioned legitimate and problematic third-party libraries, benign application developers should be protected. If these problematic libraries pass library integrity verification, it proves that the malicious behaviors in these libraries are from the library providers, instead of application developers. Our measurements directly help 80.50% of library usages, which are the potential victims of the aggressive library threat to clear developers’ names.

Whether we can fully trust passing conclusions depends on whether Duet generate any false negatives and/or false positives. By false negative we mean that a library is detected as unmodified, even though it was in reality modified. In Section 4.4.1.1, we have discussed whether false negative could exist. By false positive, we mean that an unmodified library fails to pass verification. Because the false positive issue is critical, we will use dedicated one section to do in-depth analysis. In particular, we will manually check whether false positive exists in Section 4.6.

4.5.5 Performance

Since Duet is designed to be used in the real world, the performance is an important factor. Retargeting takes 5,259ms on average as the majority of the time of the library integrity verification. All other processing for the verification of one library takes 27.9ms on average that is much less than the time for retargeting. For an application, the retargeting happens just once while the other processing repeats for each detected library. In our dataset, each application has three libraries on average. Therefore, it takes less than 7 seconds on average for an application to do the library integrity verifications for all libraries that this application uses.
Table 4.6. In-depth Analysis of In-the-wild Verification Not Passing Samples.

<table>
<thead>
<tr>
<th></th>
<th>Rank</th>
<th># of not Passing</th>
<th>Code Insertion</th>
<th>Obfuscation</th>
<th>Optimization</th>
<th>Missing Original Library</th>
<th>Remain</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdMob</td>
<td>1</td>
<td>7,412 (20.75%)</td>
<td>63 (0.18%)</td>
<td>5,772 (16.16%)</td>
<td>509 (1.42%)</td>
<td>451 (1.26%)</td>
<td>617 (1.73%)</td>
</tr>
<tr>
<td>Android Support</td>
<td>2</td>
<td>8,698 (27.18%)</td>
<td>5,607 (17.52%)</td>
<td>2,150 (6.72%)</td>
<td>356 (1.12%)</td>
<td>0 (0.00%)</td>
<td>585 (1.83%)</td>
</tr>
<tr>
<td>AppBrain</td>
<td>40</td>
<td>237 (15.57%)</td>
<td>4 (0.26%)</td>
<td>213 (13.99%)</td>
<td>16 (1.05%)</td>
<td>0 (0.00%)</td>
<td>4 (0.26%)</td>
</tr>
<tr>
<td>AdFonic</td>
<td>57</td>
<td>149 (14.54%)</td>
<td>2 (0.20%)</td>
<td>144 (14.05%)</td>
<td>3 (0.30%)</td>
<td>0 (0.00%)</td>
<td>0 (0.00%)</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>(19.50%)</td>
<td>(4.54%)</td>
<td>(12.73%)</td>
<td>0 (0.97%)</td>
<td>(0.32%)</td>
<td>(0.96%)</td>
</tr>
</tbody>
</table>
4.6 In-depth Analysis of Applications that do not Pass Verification

In our Dataset, 19.50% of libraries do not pass the integrity verification. We collect names of .class files in these libraries and perform manual analysis on a randomly selected set of applications in order to find the reasons of not passing integrity verification. In some situations, we also compare the decompiled code to check the not passing reasons.

As shown in Table 4.6, we find the major reasons of not passing are code insertion, obfuscation, optimization, and missing original libraries.

**Code Insertion:** We find some libraries have code that is not from the library provider. We find these inserted .class files by checking if there are any .class files not a member of the set of .class files in the reference database.

We find there are several different ways to insert code into libraries: (1) Developers add their own code such as MyViewActivity.class into the namespace of libraries. (2) Other libraries lodge into the namespace of original libraries. For instance, Watson [76] lodges in the Android Support library. (3) The library provider may allow other libraries to use its namespace. For instance, AdMob allows other advertising libraries that use its mediation service to use “/com/google/ads/mediation/”.

For developers, it is not a good idea to use the namespace that is used by a library provider. This allows the inserted code to access the “package” methods and fields, which might cause problems. For instance, a “package” field could have the same name as another filed in another package. Inserted code by developers could use the wrong filed because of carelessness. For the library provider, allowing others’ libraries to use its namespace may bring conveniences in its management. Duet can be extended to support this, as long as these admisive libraries are also collected.

For the four libraries, we manually checked every .class file. The results are as follows. 63(0.18%) applications among 35,726 applications using the AdMob library have code inserted. For AppBrain and AdFonic libraries, we observed similar percentages, 0.26% and 0.29%, respectively (see Table 4.6). However, we observed a significant high percentage of the Android Support library. 5605 applications out of 32,002 applications have code insertion. This is because Android Support library is the only open source library in these four libraries. Therefore, developers could modify its source code and add their own code as well.

Having code insertion does not really mean that the code inserted has malicious intent. A lot of code insertion is actually done by benign developers. Separating good
code inserted from malicious one is out of the scope of this work.

**Obfuscation:** When we analyze names of the `.class` files to detect the code insertion, we notice that some libraries have been obfuscated by application developers from the fact that some `.class` names have been modified. For instance, `android/support/a/a/A.class` is never used in the original libraries, but is detected in the dataset. These obfuscated `.class` file names sometimes even contain special characters from Chinese or Japanese. We manually build a list of `.class` names for these obfuscated cases. As shown in Table 4.6, on average 12.73% of libraries are detected to have been obfuscated by application developers.

Our analysis on libraries indicates one interesting thing: many libraries have been obfuscated by the library provider before release. In these cases, another obfuscation by the application developers does not benefit developers that much. We suggest the application developer should not perform obfuscation to libraries that have already been obfuscated. ProGuard [77] does have options to achieve this function.

**Optimization:** Optimization on libraries by developers is another reason for not passing integrity verification. For example, we use Soot [78] to decompile the `.class` files of one AdMob library to get source code and compare with decompiled source code from the original library. In this case, optimization is found as one reason for not passing verification. For instance, in `com/google/ads/AdActivity.class`, we figure out that the code inlining optimization has changed `.class` files. As another example, we find annotations in samples of Android Support library have been removed. There are several ways developers can remove annotations in a library. For instance, developers can use ProGuard to perform the removal. As shown in Table 4.6, we find 0.97% of libraries do not pass the integrity verification because of optimizations.

**Missing Original Libraries:** Another reason of not passing integrity verification is missing the original libraries. Although we manage to collect all known versions for 4 libraries, it is still possible that we missed some very old versions. For example, we download all known versions of AdMob based on its official release note [74]. The release note contains version information after March 15, 2011. According to our knowledge, Google also released advertising libraries that were called GoogleAdView.jar before that date. In the reference database, we only have two versions of GoogleAdView.jar.

The GoogleAdView.jar contains GoogleAdView.class that is not used in subsequent AdMob. With this particular `.class` name, we detect 459 samples of GoogleAdView.jar. However, 451 of these 459 samples do not pass the integrity verification (see Table 4.6). The passing ratio for GoogleAdView.jar is 1.74%(8 of 459) that is much less than 80.50%.
Therefore, we must miss some versions of *GoogleAdView.jar* when we build the reference database.

This fact indicates that a reference database with all versions of libraries is critical for *Duet*. Library providers can provide old versions to the application store directly. They have an incentive to do so, as it protects their reputation in the event of a library-centric attack. Library providers are willing to provide this information because it will protect their reputation in case of the library-centric attacks. In this process, library providers do not share extra information with application stores because libraries are publicly available.
Chapter 5

Policy Generation for Library Isolation

5.1 Introduction

With over one million applications available in the Google Play store [79], Android now has the largest number of applications in the world [80]. Being part of this success, third-party libraries play an important role because application developers integrate various libraries into their applications for different purposes. For example, an advertising network distributes advertising libraries. Developers integrate advertising libraries into the applications they develop to collect revenue. As another example, other library providers release libraries to provide functionality. Zebra crossing (ZXing) [5] is such a library that provides functionality of 1D/2D barcode image processing. In the Android ecosystem, the application developers and the various kinds of library providers are different stakeholders and they form a symbiotic relationship.

Intra-App Cross-Stakeholder Security Threats: The Android platform provides a security model to guarantee isolation between applications [7]. This model is built based on the assumption that applications from different developers should be isolated. The unit of protection is one application in the Android security model; however, when the application itself contains code from multiple stakeholders, the application is found using third-party libraries with potentially malicious behaviors. These libraries are also called potentially harmful libraries. Potentially harmful libraries are widely used in the real world. According to the study conducted in [53], 6.84% of applications from Google Play are found to use potentially harmful libraries. The potentially harmful
behaviors include sending SMS messages, making voice calls, tracking users, collecting private data, and taking pictures and sending them out. From the permission point of view, these potentially malicious behaviors can be broken into two classes:

- Potentially malicious behaviors requiring extra permissions: This problem is also denoted as over privileged, which means that the potentially malicious behaviors in a library require permissions besides what the library functionality really needs. In reality, advertising libraries are found to have potentially malicious behaviors besides supporting advertisements. Such so-called madware wildly exists and is becoming more and more popular (infected 23.8% of applications on Google Play) [25]. As one example, mobiSage, one advertising library, is found to require the permission to access the camera on a smartphone.

- Potentially malicious behaviors requiring no extra permissions: Other potentially malicious behaviors do not require extra permissions beyond what are needed for the library functionality. One in-app purchases library [18] is in this category. In this particular example, the library makes user-requested SMS-based in-app purchases, which makes its requests of accessing both SMS and networks look legitimate. However, the library collects all SMS messages in the smartphone and send them to a remote server without user consent. This single library is found to be used by over 18,000 applications.

Limitations of Existing Techniques: The two intra-app cross-stakeholder security threats mentioned above cause very serious consequences in the real world. However, we find they cannot be fully addressed by existing defenses or library isolation policies. Software analysis [60, 64] can detect some potentially malicious behaviors in applications. Between any chosen source (e.g., contacts, camera) and any chosen sink (e.g., a network interface), FlowDroid can do context, flow, field-sensitive information flow analysis. However, FlowDroid cannot do two things:

- Capability 1: Given an API call in a library that changes a state of the smartphone, conduct scalable backward data flow analysis and provide complete knowledge about which stakeholders could ask the library to change the state.

- Capability 2: Given several stakeholders in an application, the analysis tool is aware when the data-flow drifts from the code of one stakeholder to the code of another stakeholder.
Without Capability 1 and Capability 2, it is hard to use FlowDroid to analyze the potentially malicious behaviors in libraries because: 1) potentially malicious behaviors and the legitimate functionality of a library do not have to have both sources and sinks. For instance, sending SMS messages stealthily in a library does not need a meaningful source as defined by FlowDroid; using a library to get the current GPS location does not need a meaningful sink as defined by FlowDroid. Missing sources or sinks will make the analysis impossible. 2) missing sources will also make forward taint analysis not scale. Without sources, forward taint analysis has to set all possible API calls as sources in a brute force way, forcing the analysis to be repeated again and again. Hence, the existing tool cannot distinguish whether Android framework API calls in libraries assist another stakeholder or not (Q1).

If the above two capabilities are not developed, it would be very difficult to properly isolate untrusted libraries. In principle, a sound isolation policy cannot be developed without answering Q1 correctly. And without a sound isolation policy, isolation mechanisms in many cases would either over-constrain or under-constrain an untrusted library. Over-constraining leads to denial of service. Under-constraining leads to like private data leakage and other bad consequences.

**Insight: Intra-App Cross-Stakeholder Data Dependency Analysis is Missing.** For a general third-party library, it contains two types of behaviors. The first type of behaviors serve a purpose for the application’s main functionality. The second type of behaviors are used only by the library itself. A solution handling the intra-app cross-stakeholder threats needs to distinguish these two types of behavior.

If we could correctly classify behaviors as either the first type or the second type, we know the potentially malicious behaviors within libraries will belong to the second type. In that case, an isolation solution could intercept and check the Android framework API which belong to the second type to prevent private data leakage.

Hence, understanding how the application logic uses libraries will help us to deal with the intra-app cross-stakeholder threats. Unfortunately, intra-app cross-stakeholder data dependency analysis is missing from the existing software analysis tools. Without such a tool, existing isolation solutions can only handle the threats in an ad hoc manner. For instance, **PEDAL** [48] can isolate advertising libraries from application logic with the assumption that the application logic does not rely on advertising libraries. This assumption has, however, not been be formally verified so far. If this assumption is not correct, the legitimate application logic can be affected by the isolation mechanism, and as a consequence some essential app functionality can no longer be provided to the user.
**Our Approach: Callsite-Aware Bidirectional Static Taint Analysis.** To address the limitations of existing defenses and policies, we first define the code from one stakeholder as one module in the application. Then, we model the intra-app cross-stakeholder relationships as certain types of bidirectional data-flow between modules. Finally, we develop a tool to analyze these data-flow statically. With the results from the static taint analysis, we could generate policies more effective for library isolation.

When modeling the intra-app module relationships, two type of library operations are defined. The read operations mean the library assists the application logic in collecting a state of the smartphone. In contrast, the write operations mean the library assists the application logic in changing a state of the phone. The read operations can be analyzed efficiently by the existing forward taint analysis techniques. However, forward taint analysis does not scale when analyzing the write operations. Forward taint analysis starts from a source and goes forwards following the control flow graph until reaching a sink. When analyzing the write operations, we need to know how the parameters are set (the data dependency of the parameters). This means we have the sinks, but don’t know which methods are sources. To use all methods outside of libraries as sources will force forward taint analysis to be repeatedly executed a lot of times. Using forward taint analysis for write operations does not scale. Therefore, we implement an extension [65] of the IFDS algorithm as a backward taint analysis tool for the write operations. Backward taint analysis starts from the APIs of write operations and goes reversely in the control flow graph. It scales because it only needs to be executed once.

Besides supporting bidirectional taint analysis, another important factor is to distinguish different modules within one application based on the callsites of statements.

In the evaluation, we find that 15 of the top 20 libraries from [26] could be threatened by the intra-app cross-stakeholder security threats. The potentially malicious behaviors include: tracking users, collecting private data, taking pictures and sending them to a remote server, eavesdropping, sending SMS messages, and making voice calls. The evaluation also shows backwards taint analysis analyzes the write operations efficiently while forward taint analysis does not scale. In measurements obtained on DroidBench [81], the backwards analysis achieves 86% recall and 83% precision. Finally, bidirectional callsite-aware static taint analysis is used in our experiments to analyze 24 different libraries in 7 library categories to classify the Android Framework APIs in libraries.

**Our main contributions:**

- To the best of our knowledge, we propose the first classification framework to generate the policy for library isolation.
• We have the first implementation of such a classification framework, which includes an implementation of backwards taint analysis.

• We apply the tool to analyze 24 libraries in 7 different categories for the classification of Android framework APIs used by libraries.

5.2 Background and Potential Applications

5.2.1 Libraries for Android applications

Android applications require different libraries to function. These libraries can be broken down into two categories. Some libraries are chosen by the operating system providers. This library is generated within Android Open Source Project (AOSP) [22]. All these libraries are merged together within AOSP, and are made available by the Android software stack. Since this type of libraries are parts of the Android framework, the operating system providers should take the responsibility if these libraries have malicious behaviors or vulnerabilities. For instance, a vulnerability in Stagefright which is a multimedia library is considered as the vulnerability of mobile operating system [82].

The second type of libraries are chosen by application developers. This chapter focuses on the second type of libraries, namely libraries chosen by application developers. In practice, these libraries are published by their creators in .jar or .class files to provide some functionality for applications. Application developers choose libraries based on their own requirements. After the application build process, the code from this type of libraries and the code of application logic coexist in applications.

In practice, some libraries are well maintained by the providers on the official websites. However, even these libraries are found to contain potentially malicious behaviors [10]. Application developers may use libraries from public code repositories (e.g., GitHub). These libraries may be modified and potentially malicious behaviors exist. Further, the application developers may use an unofficial programming environment, which could insert potentially malicious behaviors into libraries like the XCodeGhost attack [83].

5.2.2 Dilemma for application developers

Application developers need to use third-party libraries for the functionality provided by the library. However, application developers do not know all behaviors of third-party libraries because libraries are released in binary code. Use libraries containing
potentially malicious behaviors can hurt the reputation of the application developers. In reality, developers updated applications to remove problematic libraries to save their reputation [11].

Application developers may read the documents of libraries to know the permissions required by libraries. But, the library could use a permission probe to enjoy all permissions of the application without explaining in the documents. Advanced developers could analyze the binary code of libraries to know the set of permissions used by the library. But, code analysis is a big burden for the developer. Only knowing the permission set of libraries cannot prevent those potentially malicious behaviors requiring no extra permissions.

The headache comes from the Android ecosystem. Without changing of current develop model of Android applications, these problems cannot be avoided. With current settings, the community needs a comprehensive understanding of the intra-app cross-stakeholder relationships in order to develop a healthy ecosystem.

5.2.3 Applications of Intra-App Cross-Stakeholder Analysis

Existing data-flow analysers for Android applications either treat the whole application as an atomic unit [60], or target to the inter component communications [61]. Intra-app cross-stakeholder analyser presented in this chapter can be used for numerous important analysis, for example:

**Detecting the data dependency among different stakeholders:** The first application of the analyser is to find how the application logic relies on libraries for functionality. Further, the analyser also tells how one library relies on another library for functionality.

**Distinguishing the necessary part and the extra part within code of one stakeholder:** The analysis could distinguish parts of the library are necessary for the application logic if the data dependency exists; other parts of the library are extra (unnecessary) for the application if there is no data dependency. This information could be used for 1) Extra part may be potentially malicious behaviors, and are needed be analyzed. 2) This information could guide the stakeholder-oriented isolation to avoid breaking the application’s main functionality. 3) Library providers could also use this tool to analyze the library to identity the useless part of the library during the library development process. 4) The application developers could get a deep understanding of the library to select libraries with low risks.
```java
public class AppLogic {
    public void register() {
        LibFoo lib = new LibFoo();
        String unique_id = lib.getUniquePhoneID();
        String msg = "User: " + this.getUserName + " | UniqueID: " + unique_id;
        lib.sendSMS("+1 800 0000 0000", msg);
    }
}

// Library with malicious behaviors.
public class LibFoo {
    public String getUniquePhoneID() {
        TelephonyManager tm = (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);
        String device_id = tm.getDeviceId();
        String obf_dev_id = "";
        for (char c : device_id.toCharArray())
            obf_dev_id += c + "_";
        return obf_dev_id;
    }

    public void sendSMS(String number, String msg) {
        SmsManager sms = SmsManager.getDefault();
        sms.sendTextMessage(number, null, msg, null, null);
        sms.sendTextMessage("+1 999 8765 4321", null, msg, null, null);
    }
}

// Advertising Lib.
public class AdvertisingLib {
    ...
}
```

Listing 5.1. Example: Application logic and libraries
5.3 Reducing Intra-App Cross-Stakeholder Module Relationship Analysis to a Data-Flow Analysis Problem

Listing 5.1 shows a representative example of application logic and a library. The library has two functionalities: 1) getting a unique ID of the phone; 2) sending a SMS message. The application logic uses both in its registration process. However, besides assisting the application logic, the library also contains other behaviors. For instance, it collects content of SMS messages sent by the application logic by sending them to another phone number. Another malicious behavior is that the library will collect the device ID when application logic uses the library to send a SMS message. We will use this example to demonstrate how to model the intra-application module relationship in this section.

5.3.1 Definition of Intra-Application Modules

Android applications contain code from different stakeholders. For a given application, one part of code is from the applications’ developer and we call it as the application logic. Application developer chooses a few third-party libraries to be used; each library is from its own provider. Application developer and those library providers are different stakeholders. Within one application, code from one provider is considered as one module. In other words, application logic is considered as a module; each third-party library is also considered as a module. For instance, in Listing 5.1, we have three modules: application logic, libFoo, and advertisingLib.

5.3.2 Definition of Operations

Our analysis aims at understanding the behaviors in each module and the relationship between different modules. Behaviors are invoked by one or several API calls. Each API call is considered as one operation. One operation belongs to a particular module if the callsite of that operation belongs to that module.

Besides callsite awareness, operations are further divided into the read operations and the write operations.

The Read Operation: One operation is called as a Read Operation if the API collects a state of the phone.

The Write Operation: One operation is called as a Write Operation if the API changes a state of the phone.

\footnote{The Write and Read Operation: It is possible that one API changes a state of the phone and then collects another state of the phone as its return value. In this chapter, we simply this by considering}
In summary, we use the $<API, CallSite, R|W>$ to describe the operation. In the LibFoo of Listing 5.1, we have the following operations.

- $<\text{TelephonyManager getDeviceId}, \text{line 17}, R>$
- $<\text{TelephonyManager getDeviceId}, \text{line 32}, R>$
- $<\text{SmsManager sendTextMessage}, \text{line 28}, W>$
- $<\text{SmsManager sendTextMessage}, \text{line 29}, W>$
- $<\text{SmsManager sendTextMessage}, \text{line 33}, W>$

### 5.3.3 Data-flow Analysis for Read and Write Operations

Next, we analyze the data-flow. For the read operations, our analysis focused on how the return value is used. Hence, the analysis tracks the return value and generate the data-flow. If a data-flow reaches another module, this data-flow is considered to work for that module. If a data-flow never reaches other modules, the data-flow is considered to work for the module where the read operations lives.

Further, if all data-flow of a read operation work for other modules, this read operation is considered to work for other modules. If none data-flow of a read operation works for other modules, this read operation is considered to work for its own module. If some data-flow work for other modules and the other data-flow work for self, the read operation is considered as a hybrid case.

In the LibFoo of Listing 5.1, the read operation on line 17 works for other modules; the read operation on line 32 works for its own module; there is no hybrid case in this example.

For the write operations, our analysis focused on how the values of parameters are set. Hence, the analysis computes all data-flow reaching parameters. If a data-flow comes from another module, this data-flow is considered to work for that module. If a data-flow never reaches other modules, the data-flow is considered to work for the module where the write operations lives.

Further, if all data-flow of a write operation work for other modules, this write operation is considered to work for other modules. If none data-flow of a write operation works for other modules, this write operation is considered to work for its own module. If some data-flow work for other modules and the other data-flow work for self, the write operation is considered as a hybrid case.

In the LibFoo of Listing 5.1, the write operation on line 28 works for other modules; the write operation on line 33 works for its own module; the write operation on line 29 it as a write operation.
is a hybrid case.

5.3.4 Understanding the Data-flow Analysis Results

After the data-flow analysis, both the read operations and the write operations are divided into three different categories: work for other modules, work for self, and hybrid. Table 5.1 summaries the meanings of different categories when operations live in the application logic.

Work for other modules: The read operations mean the application logic sets values of libraries. It means the library relies on the configure from the application logic. However, the write operations in this category are strange because it means that the library uses the application logic to change a state of the phone. Further analysis is required.

Work for self: For application logic, this category means the application logic does its own work. Hence, both the read operations and the write operations are considered normal.

Hybrid: The read operations mean containing two behaviors 1) application logic’s own behavior; 2) the application logic sets values of libraries. The first one is normal, and the second one indicates the library relies on the application logic. However, the write operations in this category are still strange because it means when the application logic changes a state of the phone, parts of the parameters come from libraries. Obviously, further analysis is required.

Table 5.2 summaries the meanings of different categories when operations live in libraries.

Work for other modules: For libraries, this category means the library assists other modules to achieve some functionality. Hence, both the read operations and the write operations are considered normal. This category also indicates the dependency among modules.

Work for self: For libraries, this could be caused because parts of libraries are not used; however, it could also be caused because libraries contain potentially malicious behaviors. Therefore, these operations need to be further analyzed.

Hybrid: For libraries, this category means containing two behaviors 1), the library assists another module; 2) extra behaviors that may contain potentially malicious behaviors. The first one introduces the dependency among modules. The second one needs to be further analyzed.

In the LibFoo of Listing 5.1, the read operation on line 17 indicates that the application
Table 5.1. Meanings of different read operation and write operation in Application Logic. ■: need further analysis; ▲: may be potentially malicious behaviors; ▼ other module relies on it; ◆ helps to understand the relationship among modules.

<table>
<thead>
<tr>
<th></th>
<th>The Read Operation</th>
<th>The Write Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work for other modules</td>
<td>◆ App sets values for libraries.</td>
<td>■ Libraries use App to change phone state.</td>
</tr>
<tr>
<td>Work for self</td>
<td>App’s own behavior.</td>
<td>App’s own behavior.</td>
</tr>
<tr>
<td>Hybrid</td>
<td>App’s own behavior.</td>
<td>■ Libraries use App to change phone state.</td>
</tr>
<tr>
<td></td>
<td>◆ App sets values for libraries.</td>
<td></td>
</tr>
</tbody>
</table>
Table 5.2. Meanings of different read operation and write operation in libraries. ■: need further analysis; ▲: may be potentially malicious behaviors; ▼ other module relies on it; ♦ helps to understand the relationship among modules.

<table>
<thead>
<tr>
<th></th>
<th>The Read Operation</th>
<th>The Write Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work for other modules</td>
<td>♦ Library assists another module.</td>
<td>♦ Library assists another module.</td>
</tr>
<tr>
<td>Hybrid</td>
<td>■ ♦ Either library assists another module;</td>
<td>■ ♦ Either library assists another module;</td>
</tr>
<tr>
<td></td>
<td>▲ or extra behaviors.</td>
<td>▲ or extra behaviors.</td>
</tr>
</tbody>
</table>
logic relies on this read operation. However, the read operation on line 32 is extra behavior and further analysis could find that it will leads to privacy leakage. In the LibFoo of Listing 5.1, the write operation on line 28 indicates that the application logic relies on this write operation; the write operation on line 33 is extra behavior and further analysis could find that it will leads to privacy leakage; further analysis of the write operation on line 29 indicates that it is an extra behavior and will cause privacy leakage (For this case, further analysis is based on knowing parameters’ functionality).

5.3.5 API Classification and Policy Generation

Based on data dependency, we could classify Android framework APIs in libraries into six categories.

Writing data dependency APIs (WDD) : All parameters are set by core logic.

Reading data dependency APIs (RDD) : All return values reach core logic.

Library’s own writing APIs (WL) : All parameters are set by the library.

Library’s own reading APIs (RL) : All return values never leave the library.

Writing data dependency APIs (WDD) : All parameters are set by core logic.

WDD + WL (hybrid) : The data dependencies involved in an API call are a mix of WDD and WL.

RDD + RL (hybrid) : The data dependencies involved in an API call are a mix of RDD and RL.

Based on the six categories, the policies for isolating sensitive Android framework APIs in libraries can use the following principles.

Principle 1 : WDD and RDD indicate the library is assisting the app, and the app has intention to get assistance. WDD and RDD mean no policies for isolating sensitive

Principle 2 : For WL and RL API calls, the default policy should Deny.

Principle 3 : For WDD + WL (hybrid) and RDD + RL (hybrid), the policy is Deny in this work; but one could do context-based isolation (for instance, ask users’ decisions).

With these principles, we could generate distinct policies after we classify Android framework APIs in libraries based on data dependency analysis.
5.4 System Overview

Figure 5.1 shows an overview of our intra-application module relationship analysis process. Given an application, we start by parsing the application into class files to extract third-party libraries used in it (1). Application logic and every library are modules in the remaining analysis. For each module, we scan all class files to detect Android framework API calls (2). For each module, a list of read operations and a list of write operations are generated. Then, we use the application and detected operations to perform data-flow analysis (3). The results will classify the operations into six categories: WDD, WL, WDD + WL (hybrid), RDD, RL, and RDD + RL (hybrid). The classification could then be used to generate policy.

**Library Separation:** Android application code is distributed in a platform-specific Dalvik bytecode format that contain both the application logic and third-party libraries. Therefore, we use Dare [24] to retarget Dalvik bytecode to Java bytecode.

Then, we use namespaces to detect libraries, similar with [26, 84]. These approaches work for all categories of third-party libraries, while the limitation is that the namespaces of libraries may be modified during the build process. According to the result in [26], over 80% libraries in their dataset are not modified. Hence, the majority could be covered during the analysis.

Other approaches [48, 85] identify advertising libraries, a particular category of libraries, through semantics (e.g., the usage of Android components, or specific APIs) and then performs a machine classification for the detection. These approaches could tolerate modification on libraries, but can only work for advertising libraries. Our analysis framework could support other library separation technique.
Operation Detection: Detecting operations is trivial. During the retargeting process for library separation, besides bytecode Dare also generates files in Jasmin [73], a low-level intermediate representation. Each class file has a homologous text file in Jasmin format. In these text files, Android framework APIs are human readable. We can detect read/write operations by scanning these text files.

The list of Android framework APIs for analysis is another important input. Android has a permission system. Those framework APIs protected by permissions are suitable to be analyzed. Because the mapping between APIs and permissions are incomplete in the official document, we choose the permission map from PScout [86]. We manually go through detected APIs and assign each API as “write” or “read” based on whether it changes the phone’s state or not. If the analysis requires other framework APIs not in the map, we just add them into the list. If a framework API is hard to be decided as “write” or “read”, we consider both cases.

Data-flow Analysis: We use static taint analysis for the data-flow analysis. Classical static taint analysis tools, such as [60] require predefined sources and sinks. To track sensitive “tainted” information through the application, these approaches start at a predefined source (e.g., an API method returning unique PhoneID) and then follow the data flow until they reach a given sink (e.g., a method sending the information through SMS messages).

The above approach is similar with our goal for the analysis of the read operations. Our tool starts at the read operations, and then follows the data flow until it reaches another module or given sinks within the same module. Hence, we need to extend the static taint analysis tool to allow it support callsite-aware sinks that means any methods in other modules should be considered as sinks.

However, the approach is quite different with our goal for the analysis of the write operations. With the above approach, the write operations could be defined as the sinks. Unfortunately, it is hard to define the sources for the analysis. If all potential sources are used in the analysis, the analysis does not scale any more (See Section 5.6.3).

Our solution to this problem is to do backwards taint analysis for the write operations. To track sensitive “tainted” information through the application, the tool start at a write operation, (e.g., a method sending the information through SMS messages). and then follows the data flow reversely in the control flow graph until it reaches other modules.
5.4.1 Limitations

We assume there is no dynamic code loading in the application. We do not detect the data dependency by side channels or implicit flows. We do not support multiple threads. We assume the Linux kernel and libraries, the Android framework, and the Dalvik VM are all trusted.

Our analysis is based on the sources and sinks as Android framework APIs. But, build a complete list of sources and sinks is out of the scope of this work. An attack cannot be detected if it uses Android framework APIs not in the list.

Our analysis tool does not handle java reflection soundly. We leave this to other works that focus on reflection.

5.5 Bidirectional Callsite-aware Static Taint Analysis

To analyze the data-flow among different modules within one Android application, we need to achieve a few things: 1) the lifecycle should be modelled precisely; 2) both forward taint analysis and backwards taint analysis should be supported; 3) callsite-aware sources and sinks should be supported.

5.5.1 Precise Modelling of Lifecycle

Compared to Java applications, Android applications have their unique characteristics: multiple entry points, asynchronously executing components, and callbacks. These characteristics should be modelled precisely for the data-flow analysis. We use a solution from the existing tool, FlowDroid [60]. FlowDroid analyzes these three factors and generates a \textit{dummyMain.class} file. The \textit{dummyMain.class} file is used with the application to generate the control flow graph, which is then used in the data-flow analysis.

5.5.2 Precise Bidirectional Taint Analysis

As explained in Section 5.4, our analysis requires both forward taint analysis and backward taint analysis. FlowDroid [60] is a precise context, flow, filed, object-sensitive forward taint analysis tool. It is the state of the art and open source. Our forward taint analysis part is an extension of FlowDroid supporting callsite-aware sources and sinks. Our explanation focuses on the backwards taint analysis.
5.5.2.1 IFDS Extension to Support Unbalanced Return Flows

The backwards taint analysis built on top of the existing Interprocedural Finite Distributive Subset (IFDS) framework [87] within the Soot [88]. IFDS algorithm addresses data-flow problems with distributive flow functions over finite domains. If the data-flow problem is modelled in this fashion, the analysis problem can be reduced to a graph reachability problem. The graph is called exploded super graph, in which for each node $(s, d)$ is reachable from a start node if a data-flow fact $d$ holds at a statement $s$. The extended version of IFDS [89] also supports on-demand construction of the supergraph to provide better scalability.

The backwards analysis requires the support of unbalanced returns. Unbalanced return flow occurs when processing a return of a method, while no matching previous call was processed. In a forward analysis, calls are always processed before returns, making unbalanced returns impossible. But the backwards analysis tracks the tainted data reversely, which naturally requires supporting such unbalanced return flows. To support the unbalanced returns, we use an existing extension of the IFDS algorithm from FlowTwist [65].

5.5.2.2 On-demand Forward Alias Analysis

Inspired by the on-demand backwards alias analysis of FlowDroid, we also choose a demand-driven approach for the alias analysis, which executes within the same context-sensitive IFDS framework as the taint analysis. Using analyses like points-to sets is too costly because this category of analysis needs to computes alias information for all variables. The on-demand analysis is triggered at assignments to heap variables, for instance, statements of the form $x = y.f$. The alias analysis then walks forward through the control-flow graph. Whenever an alias is found, it triggers the backwards analysis, propagating an aliased taint from the current GPS location at which the alias was found.

5.5.2.3 Transfer Functions of Backwards Taint Analysis

The backwards taint analysis starts directly at each of the identified and reachable write operation. The set $T$ is a set of tainted access path. At the beginning, it holds that $T = \emptyset$. Once the analysis reaches a call to a source, tainted access paths for parameters are added to the set. When processing any given statement, the set $T$ of incoming taints are transformed into a set of outgoing taint according to the transfer functions. The analysis terminates when the data-flow reaches other modules.
The supergraph of the IFDS framework has four types of edges: normal, call, return and call-to-return. During the backwards analysis, called methods are entered by traversing the method’s return edges. On the other hand, the return edges are entered by the calls. The normal edges and the call-to-return edges have no changes.

The flow functions of backwards taint are different from forward taint analysis. Use normal flow function as an example. For an assignment statement $x.f^n = y.f^m$ with $n, m \in \mathbb{N}_0$, the backwards normal flow function does the following tasks: T1) add new access paths of the right side if the left side is tainted; T2) remove access paths related the left side if the left side has been redefined and is not a part of array. The forward normal flow function has the same task T2, but it has a different task T1: add new access paths of the left side if the right side is tainted.

### 5.5.3 Callsite-Aware Extension

Besides supporting bidirectional taint analysis, another important factor for the intra-application module analysis is to distinguish different modules within one application. We use a callsite extension to enable it.

Our analysis starts with the read operations for forward taint analysis; and it starts with the write operations for backwards taint analysis. Both the read operations and the write operations are detected for a given module. The sources of the taint analysis are from the same module deciding by the callsite of APIs. Please note, when operations in another module need to be analyzed, we need to run the tool with the corresponding configure.

On the other hand, every method in other modules are considered as the sinks of the analysis. Whenever the data-flow reaches sinks, the analysis terminates and the data-flow is detected. This design will not bring performance penalty. Analysis starts with the sources (operations) and terminates at the sinks. Therefore, adding extra sinks will not bring extra workload for the analysis.

### 5.6 Evaluation

The bidirectional static taint analysis tool contains forward analysis part and the backwards analysis part. Forward analysis is an extension of FlowDroid [60]. It is about 200 lines of code to support callsite-aware sources and sinks. In contrast, for backwards analysis we create the new implementation. It introduces 36 new java files and is about 11,000 lines of code in total.
Our evaluation addresses the following research questions.

**RQ1** How serious is the intra-app cross-stakeholder security threats?

**RQ2** What precision is achieved by the tool?

**RQ3** Do we need a bidirectional static taint analysis tool or do we just need a forward taint analysis tool?

**RQ4** How can this tool to classify the Android framework APIs in libraries of real Android applications?

The next sections address each question in details.

### 5.6.1 RQ1: How serious is the intra-app cross-stakeholder security threats?

The intra-app cross-stakeholder security threats could be caused by the library provider if the provider releases libraries with potentially malicious behaviors. However, there are other possibilities: libraries have been modified. In the real world, application developers could get the third-party libraries from a variety of sources like public code repositories (e.g., GitHub), and online forums. Libraries could be modified with potentially malicious behaviors in any steps from the library providers to the application developers.

Existing research results have shown that potentially malicious behaviors are trying to avoid user’s attention by either choosing the operations (API) that don’t need the user’s consent [53], or permission probe [36]. It means that the potentially malicious behaviors in libraries leverage the permission set the application already gets (for its legitimate functionality) to execute potentially harmful actions. Therefore, we would know what behaviors the permission set required by the library could cause.

We use a top 20 third-party library list sorted by the usage frequency from [26]. We downloaded these libraries directly from their providers to get the namespace of each library. Then, we use a dataset with 10,000 free applications from Google Play. We use smali to transfer Android applications into smali code, which can be readable by human beings. Next, namespace is used to identify library code. With the permission map from PScout [86], we get the permission set for the top 20 libraries in Table 5.3.

Table 5.3 shows that the permission set of 15 libraries could be used by potentially malicious behaviors without user consent like: tracking users, collecting private data, taking pictures and sending them to a remote server, and eavesdropping. This means 75% (15 of 20) of libraries could be suffered by the intra-app cross-stakeholder security threats.
Table 5.3. Permissions used by top 20 libraries.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Library Name</th>
<th>INTERNET</th>
<th>LOCATION</th>
<th>READ_PHONE_STATE</th>
<th>WRITE_CONTACTS</th>
<th>SEND_SMS</th>
<th>CAMERA</th>
<th>RECORD_AUDIO</th>
<th>WRITE_EXTERNAL_STORAGE</th>
<th>CHANGE_WIFI_STATE</th>
<th>READ_LOGS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdMob</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Android Support Libraries</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>AndroidAnnotations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Facebook Android SDK</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>bugsense</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Google Play Billing Library</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>millennialmedia</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>gson</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Twitter4J</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>actionbarsherlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Google Analytics</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>millennialmedia</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>phonegap</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>apperhand</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>InMobi</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>16</td>
<td>Leadbolt</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>ZXing</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>AdMob (old version)</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>acra</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>codehars</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4. DroidBench results

<table>
<thead>
<tr>
<th></th>
<th>Missed Flows /Caught Flows</th>
<th>Positive</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>2/26</td>
<td>4</td>
<td>86%</td>
<td>93%</td>
</tr>
<tr>
<td>Backwards</td>
<td>4/24</td>
<td>5</td>
<td>83%</td>
<td>86%</td>
</tr>
</tbody>
</table>

5.6.2 RQ2: What precision is achieved by the tool?

The precision of the tool is in fact about whether the tool can detect the data dependency without FP and FN. The support of the callsite-awareness allows us to distinguish the code from different stakeholders and does not affect the data dependency. Therefore, we do not consider the callsite-awareness when we answer RQ3. We choose the DroidBench [81] as our benchmark.

5.6.3 RQ3: Do we need a bidirectional static taint analysis tool?

We create a third-party library, which helps the developer to send SMS messages. With this library, we then create a very simple application, that just sends the SMS messages
by using the library. On the other side, we randomly download 10 applications from Google Play. We repackaged these applications and add our library inside. Then, we enforce both forward and backwards taint analysis.

<table>
<thead>
<tr>
<th></th>
<th>Latency (Backwards)</th>
<th>Latency (Forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our App</td>
<td>20s</td>
<td>305s</td>
</tr>
<tr>
<td>App 1</td>
<td>25s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 2</td>
<td>30s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 3</td>
<td>60s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 4</td>
<td>65s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 5</td>
<td>32s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 6</td>
<td>58s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 7</td>
<td>70s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 8</td>
<td>40s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 9</td>
<td>41s</td>
<td>timeout</td>
</tr>
<tr>
<td>App 10</td>
<td>32s</td>
<td>timeout</td>
</tr>
</tbody>
</table>

Backwards analysis starts from the Android framework API that sending SMS messages. A flow is detected when it reaches outside of the library. The analysis also stops if nothing is detected. As shown in Table 5.5, our application and 10 repackaged applications all finished within 70 seconds. Two data flows are found in our application: one is to set the number and the other is to set the message data. In other 10 applications, no data flow is found because the library is not used.

Then, we do forward taint analysis. We set all methods not in the library as the possible sources. We also set a timeout value for the analysis which is 100 times of backwards analysis latency. Only our application could finish the analysis before timeout. It is because our application logic is simple. Duplicated data flows are produced because all methods on the same path are added. In experiment, several repackaged applications have the “out of memory” problem. In that case, we just replaced it with another application from Google Play.

5.6.4 RQ4: How can this tool to classify the Android framework APIs in libraries of real Android applications?

Experiment setting: We create a dataset containing 7 library categories and 24 different libraries. They represent various libraries not only in the top 100 libraries in this
dissertation, but also important cases from related works. For each library, 10 applications in the real world are selected randomly. In this study, we only consider those libraries that have not been obfuscated by the application developers. For the read operations and write operations, we use the permission map of Android 4.1.1 from PSout [86]. All Android framework APIs protected by permissions at the dangerous level are set as the potential read/write operations. We also add about 20 Android framework APIs those are reported as the top sinks of the privacy leakage as the potential write operations. These added APIs are either sending data to network or doing logging.

Manual analysis is done for these applications. When needed, Android applications are de-compiled for the analysis. For each library, we only show results of 2 applications. Please note there are several factors that have impact on the results: 1) the version of a library; 2) how does the application logic use a library; 3) post-processing by the application developers such as removing unused .class files. Hence, applications using the same library could have different results. Although, logging is a kind of write operations, the analysis just lists the number of logging due to the time limitation.

5.6.4.1 Summary of Classification Results

Recall that for RL and WL API calls, the policy should be set as “deny”; for RDD and WDD API calls, the policy should be set as “allow”; for the two hybrid classes, context-based access control may be needed. According to the results in Table 5.6, Table 5.7, Table 5.8, Table 5.9, Table 5.10, Table 5.11, and Table 5.12, we obtain several interesting findings.

Finding 1: Except for social network libraries, platform libraries, and functionality libraries, RL API calls exist in the other four categories of libraries. This means that for these four categories of libraries, our classification can unambiguously confirm that the execution of certain sensitive API calls indeed should be denied. The generated policy should be set as “deny”. The way the policy is generated ensures that the policy will not affect the application logic.

Finding 2: Except for social network libraries and audio/graphics engine libraries, WL API calls exist in the other five categories of libraries. For these five categories of libraries, the generated policy for WL API calls should be set as “deny”. The way the policy is generated ensures that it will not affect the application logic because these write operations are the libraries’ own behaviors.
Finding 3: RDD API calls exist in the platform libraries. For RDD API calls, the generated policy should be set as “allow”. It will allow the application logic to work correctly. This finding is particularly interesting when considered together with Finding 1 and Finding 2: the three findings indeed confirm our earlier hypothesis that two sensitive API calls in the same library may need to be treated very differently in the isolation policy. The capabilities to handle such needs are largely missing in the literature.

Finding 4: WDD API calls exist in platform libraries, bug tracking libraries, and functionality libraries. The classification results indicate that the generated policy for these write operations should be set as “allow”. Similar to Finding 3, this finding is particularly interesting when considered together with Finding 1 and Finding 2: the three findings again confirm our earlier hypothesis that two sensitive API calls in the same library may need to be treated very differently in the isolation policy.

Finding 5: WDD + WL (hybrid) API calls do not exist in advertising libraries, bug tracking libraries, and audio/graphics engine libraries. For these three categories of libraries, the generated policy for write operations (no hybrid) do not need be context-based. However, for the other four categories of libraries, context-based access control may be needed. For the RDD + RL (hybrid) API calls, we need more results before judging because in the current experiments, some important read operations, including read operations of received data from networks, are not considered.

These findings indicate that the generated policies should be better than Policy 1 in Section 1.2. The experiment results confirm that the existence of RDD and WDD API calls. According to Policy 1, all sensitive APIs should be blocked. This will affect the application’s functionality when RDD or WDD API calls exist. In contrast, the generated policies in our approach will allow these sensitive API calls.

Compare to Policy 2 and Policy 3 in Section 1.2, the generated policies also have advantages. For Policy 2 and Policy 3, one difficulty is how to describe the library’s functionality and assign the right set of permissions. The existence of RDD and WDD API calls indicates that the library’s functionality could be analyzed by data dependency. Policy 2 and policy 3 have one shortcoming, the permissions will be assigned to the whole library, which could contain both legitimate functions and malicious functions. The generated policies could designate a separate rule to each operation based on the classification results.
### 5.6.4.2 Results for each library Category

Table 5.6 shows the results for the Advertising libraries. For **AdMob**, there are no read operations in both applications. Application 1 has 7 write operations, and the application 2 has 10 write operations. All these write operations are logging.

**Flurry**: Application 1 has 3 read operations for collecting GPS location information, 1 read operation for collecting device information, 75 write operations for logging, and 70 write operations for sending data to networks. Application 2 has 3 read operations for collecting GPS location information, 101 write operations for logging, and 71 write operations for sending data to networks. All these read operations and write operations (except logging) are found as the library own behaviors.

**Google Analytics**: Both application 1 and application 2 have no read operations, 5 write operations for logging, and 1 write operations for sending data to networks. All these read operations and write operations (except logging) are found as the library own behaviors.

**millennialmedia**: Application 1 has 2 read operations for collecting device information, 113 write operations for logging, and 24 write operations for sending data to networks. Application 2 has 1 read operations for collecting device information, 123 write operations for logging, and 26 write operations for sending data to networks. All these read operations and write operations (except logging) are found as the library own behaviors.

**InMobi**: Application 1 has 6 read operations for collecting GPS location information, 1 read operation for collect device information, 18 write operations for logging, and 14 write operations for sending data to networks. Application 2 has 6 read operations for collecting GPS location information, 1 read operation for collect device information, 18 write operations for logging, and 30 write operations for sending data to networks.

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AdMob</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>AdMob</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>AdMob</td>
<td>0 + 7 (logging)</td>
<td>0 + 10 (logging)</td>
<td>0 + 75 (logging)</td>
<td>0 + 75 (logging)</td>
<td>0 + 71 (logging)</td>
<td>0 + 71 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Flurry</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Flurry</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Flurry</td>
<td>0 + 7 (logging)</td>
<td>0 + 10 (logging)</td>
<td>0 + 75 (logging)</td>
<td>0 + 75 (logging)</td>
<td>0 + 71 (logging)</td>
<td>0 + 71 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Google Analytics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>0 + 5 (logging)</td>
<td>0 + 5 (logging)</td>
<td>0 + 5 (logging)</td>
<td>0 + 5 (logging)</td>
<td>0 + 5 (logging)</td>
<td>0 + 5 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>millennialmedia</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>millennialmedia</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>millennialmedia</td>
<td>24 + 113 (logging)</td>
<td>0 + 113 (logging)</td>
<td>26 + 123 (logging)</td>
<td>0 + 123 (logging)</td>
<td>0 + 123 (logging)</td>
<td>0 + 123 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InMobi</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>InMobi</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>InMobi</td>
<td>14 + 18 (logging)</td>
<td>0 + 18 (logging)</td>
<td>30 + 18 (logging)</td>
<td>0 + 18 (logging)</td>
<td>30 + 18 (logging)</td>
<td>0 + 18 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Leadbolt</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Leadbolt</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Write Operation</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
<td>RDD</td>
<td>RL</td>
<td>Hybrid</td>
</tr>
<tr>
<td>Leadbolt</td>
<td>20 + 69 (logging)</td>
<td>0 + 69 (logging)</td>
<td>20 + 69 (logging)</td>
<td>0 + 69 (logging)</td>
<td>20 + 69 (logging)</td>
<td>0 + 69 (logging)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5.7. Classification results for social network libraries.

<table>
<thead>
<tr>
<th>Library</th>
<th>App</th>
<th>Read Operation</th>
<th>RDD</th>
<th>RL</th>
<th>Hybrid</th>
<th>Write Operation</th>
<th>WDD</th>
<th>WL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook</td>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7 + 5 (logging)</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Facebook</td>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7 + 5 (logging)</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Android SDK</td>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20 + 3 (logging)</td>
<td>0</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Android SDK</td>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4 + 9 (logging)</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

All these read operations and write operations (except logging) are found as the library own behaviors.

**Leadbolt:** Both application 1 and application 2 have 5 read operations for collecting GPS location information, 2 read operations for collecting device information, 69 write operations for logging, and 20 write operations for sending data to networks. All these read operations and write operations (except logging) are found as the library own behaviors.

**Overall:** Advertising libraries are supporting advertisement independently. Some libraries have insider helper functions for data sending, logging, and so on. This reduces the number of read/write operations. However, it does not reduce the number of data flows.

The results also indicate that the current list of potential read/write operation is not complete. For example, AdMob does have the behavior of sending data to networks. This is not detected due to that particular API is not in the list. That API is not protected by the permission. In fact, the permission protects the API to create the connection. Due to the same reason, receiving data from network (read operation) is not detected either.

Table 5.7 shows the results for the social network libraries. For **Facebook Android SDK**, there are no read operations in both applications. Both application 1 and application 2 have 7 write operations for sending data to networks, and 5 write operations for logging. All write operations for sending data are hybrid, meaning that parts of parameters are from libraries, while the other parameters are from the application logic.

**Twitter4J:** There are no read operations in both applications. Application 1 has 20 write operations for sending data to networks, and 3 write operations for logging. Application 2 has 4 write operations for sending data to networks, and 9 write operations for logging. All write operations for sending data are hybrid, meaning that parts of parameters are from libraries, while the other parameters are from the application logic.

**Overall:** The manual analysis of these write operations show that they are in fact assisting the application logic (WDD). But the application logic only provides part of data that is sending out. For instance, the library should format the data according to
Table 5.8. Classification results for platform libraries.

<table>
<thead>
<tr>
<th></th>
<th>Read Operation</th>
<th>RDD</th>
<th>RL</th>
<th>Hybrid</th>
<th>Write Operation</th>
<th>WDD</th>
<th>WL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>phonegap</td>
<td>App 1</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>16 + 89 (logging)</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>App 2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>4 + 32 (logging)</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Titanium</td>
<td>App 1</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>12 + 33 (logging)</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>App 2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>13 + 25 (logging)</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.9. Classification results for bug tracking libraries.

<table>
<thead>
<tr>
<th></th>
<th>Read Operation</th>
<th>RDD</th>
<th>RL</th>
<th>Hybrid</th>
<th>Write Operation</th>
<th>WDD</th>
<th>WL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>bugsense</td>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + 30 (logging)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + 30 (logging)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>acra</td>
<td>App 1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5 + 74 (logging)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>App 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>5 + 74 (logging)</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

forms of the http messages. This indicates that our classification rules have limitations.

Table 5.8 shows the results for the platform libraries. For phonegap Android SDK, application 1 has 4 read operations for collecting GPS location information, 16 write operations for sending data to networks, and 89 write operations for logging. Application 2 has 4 read operations for collecting GPS location information, 4 write operations for sending data to networks, and 32 write operations for logging. All read operations are assisting the application logic (RDD). 16 write operations of application 1 and 4 write operations of application 2 are assisting the application logic (WDD). The remaining write operations are using data from both the library and the application logic.

Titanium: application 1 has 3 read operations for collecting GPS location information, 12 write operations for sending data to networks, and 33 write operations for logging. Application 2 has 3 read operations for collecting GPS location information, 13 write operations for sending data to networks, and 25 write operations for logging. All read operations are assisting the application logic (RDD). 10 write operations of application 1 and 11 write operations of application 2 are assisting the application logic (WDD). The remaining write operations are using data from both the library and the application logic.

Overall: This category of libraries provides a platform to application developers. For example, the phonegap allows application developers to create application with HTML, CSS and JavaScript. Therefore, application will depend on the library for all kinds of behaviors. However, the boundary between application logic and the is not very clear for this category of libraries. We need further manual analysis to get strong results.

Table 5.9 shows the results for the bug tracking libraries. For bugsense Android SDK, both application 1 and application 2 have 0 read operations, and 30 write operations for logging.

acra: Both application 1 and application 2 have 1 read operations for collecting
Table 5.10. Classification results for audio/graphics engine libraries.

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>Hybrid</th>
<th>Write Operation</th>
<th>WDD</th>
<th>WL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmod</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + 2 (logging)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AndEngine</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0 + 8 (logging)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cocos2d-x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 + 9 (logging)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.11. Classification results for libraries containing potentially malicious behaviors.

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>Hybrid</th>
<th>Write Operation</th>
<th>WDD</th>
<th>WL</th>
<th>Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>admogo</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td>7 + 284 (logging)</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>adwo</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>5 + 62 (logging)</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>wanpu</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1 + 7 (logging)</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

GPS location information, 5 write operations for sending data to networks, and 74 write operations for logging. The GPS location data is sent to the online server. The read operation and one write operation are the library’s own behaviors. Other write operations for sending data assist the application logic (WDD).

**Overall:** BugSense does have the behaviors of sending data to networks. The missing write operation of BugSense indicates that our API list is not complete.

Table 5.10 shows the results for the audio/graphics engine libraries. Sometimes, libraries belong to this category are also called as the game library. These libraries contain native libraries. This analysis is based on the Dalvik part of the library, and is quite limited. Overall, all libraries have write operations for logging, but only the AndEngine has 2 read operations for the GPS location information. The analysis shows these read operations are library’s own behaviors. However, the data flows could be broken in the native part due to the tool’s limitation.

Table 5.11 shows the results for libraries containing potentially malicious behaviors. For admogo, application 1 has 6 read operations for collecting device information, 2 read operations for collecting GPS location information, 1 read operation for collecting running applications, 7 write operations for sending data to networks, and 284 write operations for logging. Application 1 has 1 read operations for collecting device information, 4 read operations for collecting GPS location information, and 155 write operations for logging. All these read operations and write operations (except logging) are found as the library own behaviors.

adwo: Both application 1 and application 2 have 6 read operations for collecting device information, 5 write operations for sending data to networks, and 62 write oper-
Table 5.12: Classification results for functionality libraries.

<table>
<thead>
<tr>
<th>Library</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
<th>App 1</th>
<th>App 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Asynchronous Http Client</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Universal Image Loader</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Base64Coder</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLF4J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ViewpagerIndicator</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ZXing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>App 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>App 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For **wanpu**: Application 1 has 3 read operations for collecting device information, 1 write operation for sending data to networks, and 7 write operations for logging. Application 2 has 3 read operations for collecting device information, 1 write operation for sending data to networks, and 11 write operations for logging. All these read operations are found as the library own behaviors. The write operations for data sending are found as the hybrid. The main functionality of this library is on-line backup; thus, the network operations assist the application logic sometimes.

Table 5.12 shows the results for the functionality libraries. For **Android Asynchronous Http Client** and **Android Universal Image Loader**, all applications have one write operation for sending data to networks. All these write operations are hybrid because the data are from both application logic and the library.

For **base64coder**, **SLF4J**, and **viewpagerIndicator**, neither read operations nor write operations are detected. This indicates that these libraries do not contain behaviors that need to be controlled. However, this statement should be verified again after we have a complete list of Android framework APIs map of behaviors.

For **ZXing**: The application 1 has 0 read operation, 2 write operations for sending data to networks, 2 write operations for Wi-Fi setup, and 47 write operations for logging. The write operations for sending data are hybrid because the data comes from both application logic and the library. The write operations for Wi-Fi setup assist the application logic (WDD); in fact, this is a function provided by ZXing.

The application 2 has 0 read operations, 2 write operations for sending data to networks, 2 write operations for Wi-Fi setup, and 74 write operations for logging. The write operations for sending data are hybrid because the data comes from both application logic and the library. However, the write operations for Wi-Fi setup are detected as the
library’s own behaviors. The manual analysis shows that the library has this function, but the application does not use it.

However, the results for ZXing seems strange because it requires the permission to access camera. After the analysis, Android framework APIs about the camera and the analysis of de-compiled source code, we find the permission protect the API `Camera.open()`. But, the permission does not protect other related APIs. In this case, The parameters are set by APIs such as `Camera.setPreviewDisplay(SurfaceHolder)`, and the write operation should be `Camera.startPreview()`. But, none is protected by the permission. The tool does not know the connections between these APIs, either. Thus, the results do not contain them.

Another point is that even if the knowledge of camera APIs is modelled and supported. It will invoke a read operation through the callback. The analysis tool does support callbacks, but it requires us to provide a complete list of callbacks. However, we find the ZXing library creates its own callbacks. Missing these callbacks created by the library will cause incomplete results.

5.6.4.3 Overall Results

The evaluation results show that our classification framework could classify the Android framework APIs in third-party libraries into six categories. The software structure of libraries has impact on the classification results. A helper function in the library may be always classified as the hybrid case. The evaluation results also indicate a few problems that we should improve.

- A complete map between Android framework APIs and behaviors. We find the permission map is not enough for the analysis. We don’t need the whole list. Instead, we should define a few behaviors that our tool is targeting. For each behavior, we need a complete list of APIs.

- We need more understanding about the APIs for different behaviors. The tool does a good job when parameters are set and return value is returned in the same API. In real world, many behaviors use such a single APIs. However, we do find that other behaviors, such as taking a picture, require a series of APIs. We need to model these behaviors and make the tool to support them.

- We need to model the library more precisely. In particular, we find some libraries create their own callbacks. We need to detect these callbacks similar as EdgeMiner
[90], that detects the callbacks in the framework. Then, we need to add detected callbacks into the analysis for a complete result.

From the evaluation results, we need to adjust our story to narrow down the scope to make it in a timely matter. With the story, we should have a list of behaviors. Then, for each behavior, we need a list of APIs and models for the API series. We also need to expand the tool according to the story.
Chapter 6

LibGuardian: Library Isolation

In recent years, the Android operating system has had an explosive growth in the number of applications containing third-party libraries for different purposes. Application developers need to use third-party library, but third-party library may contain malicious behaviors. This has caused problem in the real world. For example, madware is the advertising library with aggressive behaviors. Existing work notices this problem, but they ignore the fact that the application’s main functionality may rely on the library. Therefore, existing solution only works when there are no library usages in application logic. In this chapter, we provide LibGuardian to check the library usage information to detect the unnecessary part of third-party libraries. LibGuardian also intercept sensitive Android framework API calls in the unnecessary functionality to prevent the potential malicious behaviors. Evaluation results show that LibGuardian can prevent these malicious behaviors effectively without side effect.

6.1 Introduction

The Android platform provides a security model to guarantee isolation among applications [7]. This model is built based on the assumption that applications from different developers should be isolated. The atomic unit of protection is one application. However, in the real world, within many Android applications, application logic and libraries coexist. Application logic and libraries have different intentions because they are from different stakeholders in the ecosystem. The security model of Android platform ignores different intentions from various stakeholders within the same application. This shortcoming leads to threats in the real world [25].
Advertising libraries are found to have aggressive behaviors, for example, collecting the device owner’s email address, creating icons on the screen, and so on. This so-called madware wildly exists and is becoming more and more popular [25]. Developers update applications to fix problems caused by madware to save their reputation [11]. However, developers do not know whether an advertising library is madware or not when they choose an advertising library.

Existing work has noticed the problem caused by the usage of third-party advertising libraries. They focus on the problem of over privilege. Android applications should request permissions in order to invoke sensitive Android framework APIs. Over privilege means applications are forced to request permissions that they do not need because library, which they use, need those permissions. Those solutions separate libraries into other protection domains [4, 51]. Therefore, applications do not need to request permissions for used libraries. However, aggressive behaviors of madware still function in its own protection domain because all these solutions do not shepherd the behaviors of libraries. Another type of existing works [43, 42] customizes the application behaviors to improve the application security, for instance, providing flexible fine-grained network access control. However, these solutions do not distinguish between application logic and libraries. For example, in an application, reading SMS is used for online backup, while it could be used by an aggressive library to collect personal information. In such situations, these existing works of customizing behaviors cannot solve the problems caused by libraries.

We assume that benign application developers do not get the benefits from aggressive behaviors from third-party libraries. The developers integrate the third-party libraries in order to use the functionality provided by the libraries. For instance, the advertising library helps developers to collect revenue. Those aggressive behaviors from the libraries are in fact not necessary for the main functionality of the application. Furthermore, the functionality of particular categories of libraries are not necessary for the main functionality of the application, either. For all these unnecessary functionalities in the library, we should have an access control mechanism to protect end users from the aggressive behaviors.

In short, third-party libraries for Android applications have two type of behaviors. The first type of behaviors serves a purpose for the other parts in applications; whereas the second type is used only by the library itself. We observed that many libraries contain the second type behaviors. We call this second type of behaviors as unnecessary behaviors of third-party libraries. Potential malicious behaviors from third-party libraries mainly
In this chapter, we propose LibGuardian, which is designed to detect those unnecessary functionalities and enforce an access control mechanism for unnecessary functionalities in order to prevent potential malicious behaviors in these unnecessary functionalities. We first use bidirectional callsite-aware static taint analysis from Chapter 5 to identify the necessary functionalities and unnecessary functionalities in libraries. For those unnecessary functionalities inside libraries, we intercept their Android framework API calls by a repackaging process on applications. This enables end users to mediate access to sensitive resources for the purpose of unnecessary functionality. Therefore, end users are protected from the potential malicious behaviors by third-party libraries.

For those unnecessary functionalities in libraries, we do not aim to just remove them. We believe that in a healthy ecosystem, each honest stakeholder should have its place. For instance, advertising is very important for many application developers to make a living. Hence, we believe an access control mechanism is better than the removal of third-party libraries with potential aggressive/malicious behaviors.

Our experiment is based on the first 10,000 applications from a dataset of 100,000 free applications from the Google play. We do the sensitive APIs detection and library usage analysis on the top 100 libraries from [26]. Then, we do the application retrofit on 750 popular applications. Effectiveness evaluation shows that LibGuardian can be used to prevent malicious behaviors from unnecessary behaviors of third-party libraries without side effect.

Our main contributions are as follows:

- We introduce the technique/tool to provide fine-grained access control to third-party libraries.
- We are the first to check library usage information to avoid side effect caused by application rewriting approaches.

## 6.2 Background

### 6.2.1 Stakeholders within the Ecosystem

In the smartphone ecosystem, there are several different stakeholders: cellular network providers, handset manufacturers, OS manufacturers, third-party application developers, end users, application stores and so on. Each of these stakeholders has its own intention. At the same time, it has its own concerns in terms of security or privacy. In this section,
we focus on those stakeholders related to third-party libraries of Android applications. Particularly, they are third-party applications developers, library providers, and end users.

Benign application developers create applications to provide services to end users. Based on different services, the application developers collect revenues in different ways. For instance, creating interesting advertising-supported free gaming application to get more usage time. Another example is to create a client application for existing services to get paid by the service provider, for instance to make a client application for a credit card company. For those benign developers, their major tasks are providing functionalities to end users in order to make money. They do not want to harm end users with aggressive and/or malicious behaviors. When bad behaviors occur, end users may choose applications from competitors.

Library providers have different intentions. For example, advertising libraries are provided by the advertising networks in order to enable their service: advertising. Other library providers are making money by providing services to the developers, for example, bug tracking. Besides these functionalities, the library providers could do aggressive/malicious behaviors in order to make money for library providers themselves. For instance, an aggressive advertising network may force end users to listen to an audio advertisement before making a call. The aggressive library providers do not care about their reputation that much because 1) end users normally consider the aggressive behaviors are coming from the application instead of a particular library; 2) aggressive library providers do not care the long-term reputation because they target to the short money. For benign library providers, they would like to maintain their reputation in order to get more market share and make more money.

In reality, application developers have to use third-party libraries in order to get the development process fast enough to survive in the market. However, application developers do not know all behaviors of third-party libraries because libraries are released in binary code. Hence, developers do not want aggressive/malicious behaviors from the libraries, but they have limited understanding of the libraries they use.

End users need the application’s main functionality, so that they install and use applications. End users have their own concerns about security and privacy. In particular, they hope to protect data such as the address book, SMS messages, and passwords; they pay attention to the usage of invasive hardware such as the microphone; they are also worried about eavesdropping voice conversations and location tracking on their phones. When aggressive or malicious behaviors occur, it is difficult for end users to judge who
is the offender, either application developers or library providers. The more important thing is that the damage has been caused. Hence, a protection mechanism of preventing aggressive/malicious libraries is much more useful.

End users also have diverse concerns. In many situations, they have different options for the same behaviors. For instance, an advertising library may suggest a good choice because it has information such as gender and location. On the contrast, losing money by calling premium number will be considered as malicious by almost every user.

6.2.1.1 Case Study: Advertising Library and Madware

In the real world, advertising libraries are found to have aggressive behaviors, for example, collecting the device owner’s email address, creating icon on the screen, and so on. This so-called madware wildly exists and is becoming more and more popular [25]. Some advertising libraries insert code to enable remotely execution during run time. This enables the man-in-the-middle attacks in the real world [91]. In reality, application developers are worrying about their reputation. Hence, developers updated applications to fix problems caused by madware to save their reputation [11]. However, Developers do not know whether an advertising library is madware or not when they choose an advertising library. Even worse, the damage has happened when developers update applications.

6.2.2 Problem Statement

Instead of removing all unnecessary functionalities for the application’s main functionality from third-party libraries, we provide an access control mechanism to prevent the potential malicious behaviors from third-party libraries. Thus, we could provide a healthy ecosystem containing various benign stakeholders.

Problem Statement: How to protect end users against potential behaviors of unnecessary parts of libraries used in the Android applications via an access control mechanism?

6.2.3 Use Cases

Our tool benefits several different stakeholders.

- Application developers could use this tool to avoid the damage to end users in the case of Madware. Even developers are not familiar with some advertising networks, developers can try to cooperate with advertising networks without worrying about their reputation by using LibGuardian.
• Application stores can use this tool to achieve access control to the advertising library and then release the remade library to developers. In the real world, application stores attempt to make the market healthier. This tool allows application stores to have a kind of control to third-party libraries.

• A trustworthy third-party organization can also use this tool to generate the remade libraries for developers.

6.3 System Overview

6.3.1 Design Choices

Library providers have different intentions from the application developer. Hence, library should be assigned a different access control policy. There are several ways to achieve this.

• Integrate the library into the Android operation system. The library becomes a service. The shortcomings of this solution are as follows. 1) there are so many different libraries in the real world, so it is difficult to integrate all into the operation system. 2) Android operating system will become too complex if it contains all kinds of services. 3) Android operating system updates and library updates will not be independent anymore.

• Separate the library into an independent application. The major limitations of this solution are as follows. 1) This solution introduces the application dependency into Android applications. 2) Because one application might use multiple libraries. Based on measurement results of library usages, in the extreme case, one application should use over 20 libraries. Therefore, one application will be divided into over 20 applications.

Therefore, under current conditions separating third-party libraries into a different protection domain is not a good idea. At this time, there are huge numbers of third-party libraries in the real world. The number of library providers is much less than that of application developers. In estimation, it is at least over 1,000. A general solution to shepherd these libraries should not introduce one service or application per library. In this chapter, a real solution should use Application retrofit to control the behaviors of libraries.
Besides the application developer, the smartphone ecosystem also enables third-party library providers different ways to make money. In the black market, a library provider could sell the privacy information he collects. This is one reason why unnecessary functionalities exist. However, the library should have functionalities to make application developers to choose it. Therefore, application’s main functionality could also rely on the library. A solution simply modifies the behaviors of the whole library may bring side effect to the application’s main functionality.

6.3.2 Our Idea

To overcome these challenges, we have to analyze how application logic use libraries by static analysis. After that, we could identify unnecessary functionalities in the libraries. Then, LibGuardian intercept the Android framework API calls in the unnecessary functionalities during an application retrofit process. After this process, end users could set their own policies for these unnecessary functionalities without worrying breaking the main functionality of applications.

6.3.3 Architecture of LibGuardian

As shown in Figure 6.1, LibGuardian has four major modules. The first three modules are used to analyze how applications use libraries to get the unnecessary functionalities of libraries; LibGuardian then intercepts these functionalities through a repackaging process. The fourth module is an application in the phone for end users to configure the policies.

- In the first module, LibGuardian uses Android applications and the permission map to analyze what set of sensitive APIs are used by libraries. Here, sensitive APIs mean those APIs protected by permissions on dangerous level.

- Then the LibGuardian uses the bidirectional callsite-aware static taint analysis tool from Chapter 5 to analysis how library uses the sensitive APIs that were detected in the first module. In particular, we analyze whether the applications’ main functionality requires the libraries by checking whether library usage exists.

- In the third module, the LibGuardian intercepts the sensitive API calls in the library. Here, LibGuardian also generates a handler for each interception. In the runtime, these handlers will allow/deny the API calls or provide a mocked data based on the policy.
• The last module is an application in the phone. Users can use this application to manage policy for libraries’ unnecessary functionalities.

6.4 Design and Implementation

6.4.1 Library Usage Analysis

Application developers have various reasons to include third-party libraries into applications. For instance, advertising libraries help developers to earn revenue; billing libraries help developers to achieve billing functionality. In the first sample, the main functionality of the application does not rely on the libraries. While in the second sample, the main functionality of the application relies on the library. In other words, applications cannot function well without libraries if they rely on libraries for their main functionality.

When the application relies on the library for their main functionality, we call this relationship as the library usage in this chapter. Because our approach focuses on control the Android framework APIs protected by permissions at dangerous level, we only analysis whether these Android framework API calls within libraries belong to library usage or not. If a particular API call belongs to library usage, our system will not intercept this particular API call in order to avoid the side effect to the application’s main functionality. Otherwise, it is safe to intercept the API call to control the unnecessary part of the libraries.

This part analysis uses the bidirectional callsite-aware static taint analysis tool from Chapter 5. We first detect the usage instance. Then we check the reading dependency and the writing dependency. Finally, we get the necessary part and the unnecessary part.
Figure 6.2. Application Retrofit Process.
6.4.2 Application Retrofit

After the library usage analysis, we now have a list of Android framework APIs which are working for the unnecessary functionalities in libraries. The next task is to intercept these APIs in order to achieve protection through an application retrofit process. As shown in Figure 6.2, LibGuardian uses .apk file as the input for application repacking. In the implementation, we use smali/baksmali [92] to do application retrofit. It uses smali to transfer the binary code into human readable assembler code. Then, LibGuardian intercepts the sensitive APIs in the library code. At the same time, LibGuardian generates the intercepted method handler and other supporting files. Finally, LibGuardian rebuilds the application by using baksmali. In practice, we also consider inheritance relationship.

6.4.2.1 Android framework API Interception

There exist two types of the API interception. According to the java specification [93] [94], there exist two types to invoke Java APIs. The first type is invoking statically. This type of invoking is used for those static methods; therefore, the reference of the methods is known. For intercepting this type of APIs, LibGuardian just replace the APIs with the method names of intercepted method handler.

The second type is invoking dynamically. For example, when an instance is created for a class, the reference of the method is unknown until runtime. In these cases, invoking dynamically is required. When invoking dynamically happens, the top of the stack is in fact the reference to the method, so we could replace the invoking dynamically with invoking statically with an extra parameter.

6.4.2.2 Intercepted Method Handler Generation

LibGuardian generates a handler for each intercepted method. Within the method, if users allow the behavior, the handler can call the original Android framework API and return its result; if users deny the behavior, the handler will not call the original Android framework APIs, but generate an exception. According to the setting of users, the handler could also return a mocked result. In many cases, a mocked data could avoid the application crash.

We generate the java source code automatically. Then, we use java compiler and the dx compiler to generate the .dex file. Finally, we generate smali code by using smali tool.
6.4.2.3 Other Support Files Generation

We also write java source code for applications to communicate with the library policy management application which will be introduced later in this section. We use the same approach in Section 6.4.2.2 to transfer these java source code into smali code.

6.4.3 Library Policy Management Application

The *LibGuardian* also contains an application in the phone for end users to manage their policies to unnecessary functionalities of libraries. The library policy management application will automatically start after the operating system boot. In each phone, *LibGuardian* has a database that stores policies of library behaviors. Only the policy management application can read/write to this database. The policy management application provides GUIs for end users to manage policies for library behaviors. For a given behavior, users can select allow or deny. The management application has a service, which will communicate with the retrofit applications. The service will fetch the policy for library and let application know via inter component communication, which is the standard IPC mechanism in Android.

The application contains a GUI. End users can use this GUI to modify the policy for libraries. Once the policy has been modified, the library policy management application notifies the application by sending intents to them.

For each application with the *LibGuardian* access control, the application will fetch the policy via communication with the service of management application whenever the application starts. If the service is not available, the application uses the default settings. When application knows a change of the policy by receiving the notice from the library policy management application, it will fetch the new policy.

6.5 Evaluation

In this section, we first analyze how libraries use Android framework APIs protected by permissions on the dangerous level. Then, we use the bidirectional callsite-aware static taint analysis tool to check the read usages and write usages, so that we can detect the unnecessary part of the library. Then, we perform the application retrofit for top 20 libraries. We also do the resource measurement. Finally, we perform effectiveness evaluation.

Our analysis is based on this setting. We have a database with 100,000 free applications from Google Play. For third-party libraries, we use the top 100 library list by usage.
frequency from [26]. This list is chosen because it has manually verified the existence of these libraries.

6.5.1 Sensitive APIs Detection

For Android applications, potential malicious behaviors are executed via invoking sensitive Android framework APIs. These sensitive Android framework APIs are protected by the permission system. Before any analysis and protection, LibGuardian need know which set of sensitive Android framework APIs are used by third-party libraries. Because permission map from the official document is incomplete, we choose the permission map from PScout [86]. PScout uses static analysis to build the map between Android framework APIs and permissions.

We use a top 100 third-party library list based on the usage frequency from [26]. We collect those library files from their providers directly. With the analysis on these library files, we know the package names for all libraries in the list. Then, we use smali to transfer Android applications into smali code, which can be readable by human beings. Now, we can use package name to identify library code. LibGuardian scan all identified smali code to detect those APIs protected by the permissions on dangerous level. In this part, we also consider the inheritance relationship.

6.5.2 Sensitive API Usage Analysis

Android framework defines four different protection levels for Android permission APIs. Permissions on the normal level are granted automatically whenever requested. Third-party applications normally do not use the permissions on signature level or signature-OrSystem level. Hence, in this analysis, we focus on those Android application APIs protected by permissions on dangerous level.

We select the first 10,000 applications from our dataset and perform sensitive API usage analysis on the top 100 libraries. Because the space limitation, we only show the results of the top 20 in Table 5.3. This result is the same as what we have in Chapter 5 because the setting is the same. Based on the results and intentions of libraries, we can divide the libraries into two types:

- The first type of library does not use any sensitive Android framework APIs for example Gson.
- The second type of library does use sensitive Android framework APIs.
In this chapter, we only need to give further attention to the second type of library. Library do need access some sensitive APIs in order to achieve functionality. But, some sensitive API usage seems strange. For example, we find that APIs related to \texttt{INTERNET}, \texttt{LOCATION}, and \texttt{READ_PHONE_STATE} are widely used by advertising libraries. The combinations of these APIs allow the advertising library to achieve the targeting advertising. However, advertising libraries also use APIs protected by other permissions such as \texttt{WRITE_CONTACTS}, \texttt{SEND_SMS}, \texttt{RECORD_AUDIO}, and \texttt{CAMERA}. The usage of these APIs could lead to the potentially malicious behaviors.

Then, we use the backwards taint analysis to analyze the write usages and use the forward taint analysis to analyze the read usages.

6.5.3 Resource Measurement

After the library usage check, we now get a list of APIs for each libraries. Every APIs in the list can be intercepted without breaking the application’s main functionalities.

We also develop a tool based on Monkey Runner. The tool can install, and execute the applications automatically. The tool also helps us to record the runtime latency.

We first do the application retrofit on the first 10,000 applications. However, we find a lot of retrofit applications crashed in run-time. We then run the original applications with the same UI inputs. We find that original applications also crashed in many cases. This indicated that we need a list of well-developed applications for resource measurement.

Therefore, we choose the top 25 applications from 30 different categories in Google play. The top 20 libraries have 2112 usage instances in these 750 applications. We have done retrofit for all these library usage instances. For each library, we intercept all sensitive Android framework APIs in the list after the library usage check.

At the beginning, we set the policy as “by default deny” for all intercepted APIs. In the implementation, this means an exception will be thrown. Then, we execute applications. Whenever, we find a crash because of un-handled exceptions, we tried to replace the exception with a null return or mocked data. Eventually, 84.67% retrofit application can be executed without crashing. The crashing of remaining application is caused by un-handled exceptions. The runtime overhead is less than 1%.

The retrofit successful rate is shown in Table 6.2. The major reason of repackaging failure is the error of aapt (a tool for application resource management), not from our technique. Table 6.2 also shows the size overhead caused by our technique. Please note that some applications become smaller in terms of size because the applications use .dex files generated by old version of dx compilers. However, the new versions of dx compilers
Table 6.1. Retrofit Latency.

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>569101</td>
</tr>
<tr>
<td>Min</td>
<td>1042</td>
</tr>
<tr>
<td>Avg</td>
<td>32282</td>
</tr>
</tbody>
</table>

Figure 6.3. The Snap of Original Application.

have optimizations in terms of the size of .dex file. Our technique generates the .dex files in new format, so the total size is reduced. This table also indicates that our technique introduces reasonable size overhead for Android applications.

Table 6.1 shows the latency for the application retrofit latency. The largest latency is 569101 ms; the smallest latency is 1042 ms. On average, it takes 32282 ms to retrofit one application. These numbers indicate that our technique can be used in the real world.

6.5.4 Effectiveness

We use advertising library and set the policy as “deny for all”. Then, we check snapshots for both original applications and retrofit applications. We find that the advertisement has been blocked. Please see Figure 6.3 and Figure 6.4 as an example.
<table>
<thead>
<tr>
<th>Rank</th>
<th>Library Name</th>
<th># of Detected</th>
<th>Successful Rate</th>
<th>Size after Retrofit (K)</th>
<th>Size Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AdMob</td>
<td>417</td>
<td>100.00%</td>
<td>6416</td>
<td>99.22%</td>
</tr>
<tr>
<td>2</td>
<td>Android Support Libraries</td>
<td>283</td>
<td>99.29%</td>
<td>7979</td>
<td>98.61%</td>
</tr>
<tr>
<td>3</td>
<td>AndroidAnnotations</td>
<td>286</td>
<td>99.30%</td>
<td>7950</td>
<td>99.52%</td>
</tr>
<tr>
<td>4</td>
<td>Facebook Android SDK</td>
<td>147</td>
<td>99.32%</td>
<td>11715</td>
<td>98.49%</td>
</tr>
<tr>
<td>5</td>
<td>bugsense</td>
<td>69</td>
<td>98.55%</td>
<td>5039</td>
<td>101.73%</td>
</tr>
<tr>
<td>6</td>
<td>Google Play Billing Library</td>
<td>153</td>
<td>98.69%</td>
<td>15117</td>
<td>98.54%</td>
</tr>
<tr>
<td>7</td>
<td>flurry</td>
<td>163</td>
<td>99.39%</td>
<td>10801</td>
<td>97.52%</td>
</tr>
<tr>
<td>8</td>
<td>gson</td>
<td>73</td>
<td>97.26%</td>
<td>8246</td>
<td>98.02%</td>
</tr>
<tr>
<td>9</td>
<td>Twitter4J</td>
<td>39</td>
<td>100.00%</td>
<td>9010</td>
<td>98.63%</td>
</tr>
<tr>
<td>10</td>
<td>actionbarsherlock</td>
<td>58</td>
<td>98.28%</td>
<td>6432</td>
<td>99.95%</td>
</tr>
<tr>
<td>11</td>
<td>Google Analytics</td>
<td>83</td>
<td>100.00%</td>
<td>7815</td>
<td>100.17%</td>
</tr>
<tr>
<td>12</td>
<td>millennialmedia</td>
<td>91</td>
<td>100.00%</td>
<td>9423</td>
<td>100.16%</td>
</tr>
<tr>
<td>13</td>
<td>phonegap</td>
<td>9</td>
<td>100.00%</td>
<td>3398</td>
<td>101.98%</td>
</tr>
<tr>
<td>14</td>
<td>apperhand</td>
<td>45</td>
<td>100.00%</td>
<td>4847</td>
<td>101.84%</td>
</tr>
<tr>
<td>15</td>
<td>InMobi</td>
<td>72</td>
<td>100.00%</td>
<td>12283</td>
<td>98.84%</td>
</tr>
<tr>
<td>16</td>
<td>Leadbolt</td>
<td>21</td>
<td>100.00%</td>
<td>4561</td>
<td>100.04%</td>
</tr>
<tr>
<td>17</td>
<td>ZXing</td>
<td>14</td>
<td>100.00%</td>
<td>7720</td>
<td>97.97%</td>
</tr>
<tr>
<td>18</td>
<td>AdMob (old version)</td>
<td>34</td>
<td>100.00%</td>
<td>1592</td>
<td>100.19%</td>
</tr>
<tr>
<td>19</td>
<td>acra</td>
<td>30</td>
<td>93.33%</td>
<td>7866</td>
<td>102.91%</td>
</tr>
<tr>
<td>20</td>
<td>codehars</td>
<td>25</td>
<td>100.00%</td>
<td>9755</td>
<td>91.78%</td>
</tr>
</tbody>
</table>
6.6 Discussions

6.6.1 Application Dependency

With the application to manage the policy, other applications are relying on the policy management application. This introduces the application dependency in Android applications. In fact, the framework could be extended to provide a service for end users to configure the privacy to different libraries in applications.

6.6.2 Policy for libraries

This chapter does not answer one question: what is a reasonable policy for a given third-party libraries. This is an important issue, but it is out of our scope. According to our knowledge, the way to get reasonable policies for different libraries could rely on the community. To use a community to generate widely accepted policies is not a new idea. Existing works, such as AdBlocks for web browsing, use the same methodology.

This technique could be misused to hurt the different stakeholders in the ecosystem. Advertising network could be hurt for several reasons: 1) less advertisements will be showed in the phone; 2) target advertising will not be as successful as what it is today. For publisher, they will get less impact with the same payment to advertising network. For application developers, their revenue could be reduced. In order to solve this problem, we really need community to create suitable policies.
6.6.3 Legal Discussion

The application retrofit process will break the digital signature of the application. Only when the application developers could sign the retrofitted application; all others have to sign it as a different developer. This is illegal in some countries. The application store needs an agreement with application developers if it wants to use *LibGuardian*.

6.6.4 Limitation

**Native Library is not a problem.** *LibGuardian* cannot handle native library. But this is not really shortcoming. For Android applications, sensitive behaviors have been invoked via Android framework APIs, which has been taken care by *LibGuardian*. Native code can only write to SD card. Plus, according to our measurement study, native libraries are not widely used in third-party libraries.

**Java reflection:** Java reflection can be used to replace calls of Android framework APIs. When using reflection, the APIs can be figured out by its name in String. Considering that reflection is also Android framework APIs, we can hook these Android framework calls by analysis on the string. But, this is not supported by *LibGuardian* at the moment.

**Dynamically loading:** Android applications can download the (.jar) or (.class) and load them during runtime. Our technique cannot handle these cases. We rely on exiting work that detects these dynamically downloading behaviors.

**Other Limitation:** *LibGuardian* uses the static analysis tool in Chapter 5 and the result from PScout. Hence, *LibGuardian* also has limitations of these tools.
In recent years, using third-party libraries in mobile applications cause library-centric security threats. In addition, these libraries are detected to contain potentially malicious behaviors. There are a few reasons why these problems spring up now. First, software developers have the tradition to use third-party libraries from almost the beginning of software industry. Second, the rise of the application store model reduces the bar of delivering mobile applications to end users. Third, the whole ecosystem of mobile applications is messed up: there are so many application stores, application developers, and library providers in the world. As a result of these three facts, tons of mobile applications containing third-party libraries are now used by millions of end users.

With the Android security model, the core logic and the third-party library share permissions. Solutions are provided to enforce library isolation. Unfortunately, libraries in the isolation still share some permissions with the core logic. In this dissertation, we aim to distinct policies to use third-party libraries. We learned a few lessons, and there are obviously future directions as the improvement.

7.1 Library Integrity Verification

When we analyze the integrity of libraries in the real world, we prove 13.70% of libraries have been modified by the application developers. These modifications are special because they are done by obfuscation tool (12.73%) or optimization tool (0.97%). Our analysis cannot tell whether these libraries have any other modifications besides the obfuscation or optimization. To simplify, we use obfuscation to represent both obfuscation and optimization in the remaining of this section.
During our further research, we find that detecting whether these obfuscated libraries having other modification is quite difficult. One approach is to compare program dependency graphs of original libraries and those of libraries in applications. The comparison could help us to map a method or a class in obfuscated to the corresponding one in original library. However, this approach does not guarantee the integrity because some details have to be left out during the mapping process. For instance, a class file with less than 5 methods is hard to be mapped correctly. Therefore, such an approach is normally used to do clone detection. For the purpose of library integrity verification, this type of methods does not work.

The other potential direction is based on such a statement. One obfuscated application has not been modified before the obfuscation if it satisfies the following condition: 1) with the same input, the obfuscated application has the same output as the original application. Such an assumption is used to detect the detect software plagiarism [95]. Another potential direction is to do integrity verification after deobfuscation. The state of the art of deobfuscation is a recent work [96].

Both approach requires a complete control flow graph, which is quite reasonable for the whole application. However, the library integrity verification assumes that we don’t have the source code of core logic. Therefore, we cannot build a complete control flow graph and cannot borrow their technique to achieve the library integrity verification.

In fact, the library integrity verification tool does detect modifications on libraries after it release. The 13.70% of libraries are modified by the developers, but in a special way. We suggest the application developer should not perform obfuscation to libraries that have already been obfuscated. ProGuard [77] does have options to achieve this function.

The library integrity verification could be totally solved if we have both a new building process and a new installer in mobile phones. During the building process, the .jar file could be just packaged into the .apk file. In this situation, a library verification is straightforward by the comparison of the .jar file in applications and the .jar file from the library provider. To reduce the performance penalty, during the installing the installer should optimize the combination of the core logic and the libraries, similar as Android runtime (ART) does optimization during installation.
7.2 Bidirectional Callsite-Aware Static Taint Analysis

The bidirectional callsite-aware static taint analysis is a cross-stakeholder analysis. Each stakeholder may have several Android components. On the other hand, one component may belongs to different stakeholders. For instance, a service in the core logic is inherited from a service in the library. Hence, the cross-stakeholder analysis and the inter-component communication analysis are different. But, the cross-stakeholder analysis has to support the inter-component communication for precision.

Android supports both the explicit inter-component communication and the implicit intent-based communication. It is straight-forward to support explicit inter-component communication by creating the method of sending intents as sinks and the corresponding callbacks as sources. However, supporting implicit communications with reasonable precision is not that easy. Epicc [61] and COAL [62] are tools to analyze the inter-component communication systemically. As a future work, we should integrate these tools with our analysis tool.

During the evaluation, we also find that our analysis is not precise in terms of modelling applications’ running environment. The limitation comes from a few things: 1) modelling of the life cycle; 2) modelling of the callback; and 3) using taint wrapper to avoid analysis in Android framework. DroidSafe [64] is more precise in terms of modelling the applications’ running environment. This is one potential future work to get a better precision.

Our analysis relies on a list of sensitive sources and sensitive sinks in the Android framework API. If the attack uses any APIs not in the list, the tool cannot detect it. However, how to get a complete list of sources and sinks is an interesting work itself. Susi [97] uses a machine learning approach to identity the sources and sinks automatically. In contrast, DroidSafe [64] chooses to generate the sources and sinks by reviewing source code manually. How to get a suitable list of sources and sinks are out of the scope of this dissertation. But, this is an interesting work for the future.

To build a high quality static taint analysis tool should consider a lot of things. In our example, in future we should consider at least “inter-component communication”, “applications’ running environment modelling”, and “sources and sinks”. But, there is another important fact: the tradeoff between the performance and the precision. This trade-off has to be considered reasonably under different situations.
7.3 Polices for Third-party Libraries

One major contribution of this dissertation is that our tool generates distinct polices. In particular, our tool uses the data dependencies to know which Android framework APIs in libraries are used by core logic. It is a step to generate better polices. But, our tool cannot tell which policy is the best for the Android framework APIs in libraries that are not used by core logic.

This is also an important issue, but it is out of our scope. According to our knowledge, the way to get reasonable policies for different libraries could rely on the community. To use a community to generate widely accepted policies is not a new idea. Existing works, such as AdBlocks for web browsing, use the same methodology.

Without the knowledge for unnecessary part, the technique could be misused to hurt the different stakeholders in the ecosystem. Advertising network could be hurt for several reasons: 1) less advertisements will be showed in the phone; 2) target advertising will not be as successful as what it is today. For publisher, they will get less impact with the same payment to advertising network. For application developers, their revenue could be reduced. In order to solve this problem, we really need the community to create suitable policies.

7.4 Concluding Remarks

The application store model has bought millions of applications into the world. Third-party libraries not only enable fast development, but also allow developers to integrate services into applications. Understanding whether the library has been modified, and creating distinct polices for the library isolation are therefore indispensable as a basis to guarantee the security of mobile applications.
The Top 100 Libraries in the DataSet

Table A.1, Table A.2, Table A.3, and Table A.4 show the top 100 libraries detected from the dataset of 100,000 applications. Every library has been verified by finding the library provider manually.
### Table A.1. The Top 100 Libraries (Part 1).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Namespace</th>
<th>Frequency</th>
<th>Library Name</th>
<th>Open Source</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>com.google.ads.</td>
<td>35726</td>
<td>AdMob</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>2</td>
<td>android.support.</td>
<td>32002</td>
<td>Android Support Libraries</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>3</td>
<td>android.annotation.</td>
<td>31563</td>
<td>AndroidAnnotations</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>4</td>
<td>com.facebook.</td>
<td>11170</td>
<td>Facebook Android SDK</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>5</td>
<td>com.bugsense.trace.</td>
<td>8156</td>
<td>bugsense</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>6</td>
<td>com.android.vending.</td>
<td>7680</td>
<td>Google Play Billing Library</td>
<td>Y</td>
<td>Service</td>
</tr>
<tr>
<td>7</td>
<td>com.flurry.android.</td>
<td>6741</td>
<td>flurry</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>8</td>
<td>com.google.gson.</td>
<td>6007</td>
<td>gson</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>9</td>
<td>twitter4j.</td>
<td>5795</td>
<td>Twitter4J</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>10</td>
<td>com.actionbarsherlock</td>
<td>5573</td>
<td>actionbarsherlock</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>11</td>
<td>com.google.analytics.</td>
<td>4344</td>
<td>Google Analytics</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>12</td>
<td>com.millennialmedia.android.</td>
<td>4296</td>
<td>millennialmedia</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>13</td>
<td>com.phonegap.</td>
<td>4179</td>
<td>phonegap</td>
<td>N</td>
<td>App-Dev</td>
</tr>
<tr>
<td>14</td>
<td>com.apperhand.</td>
<td>4017</td>
<td>apperhand</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>15</td>
<td>com.inmobi.</td>
<td>3344</td>
<td>InMobi</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>16</td>
<td>com.pad.android.</td>
<td>3330</td>
<td>Leadbolt</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>17</td>
<td>com.google.zxing.</td>
<td>3156</td>
<td>zxing</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>18</td>
<td>com.admob.</td>
<td>3126</td>
<td>AdMob (old version)</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>19</td>
<td>org.acra.</td>
<td>3023</td>
<td>acra</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>20</td>
<td>org.codehaus.</td>
<td>2878</td>
<td>codehaus</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>21</td>
<td>org.jsoup.</td>
<td>2807</td>
<td>jsoup</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>22</td>
<td>com.google.gdata.</td>
<td>2745</td>
<td>Google Data Protocol</td>
<td>N</td>
<td>App-Dev</td>
</tr>
<tr>
<td>23</td>
<td>oauth.</td>
<td>2670</td>
<td>OAuth2</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>24</td>
<td>com.adwhirl.</td>
<td>2458</td>
<td>adwhirl</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>25</td>
<td>com.mobclix.android.sdk.</td>
<td>2403</td>
<td>MobClix</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>Rank</td>
<td>Namespace</td>
<td>Frequency</td>
<td>Library Name</td>
<td>Open Source</td>
<td>Category</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>-----------</td>
<td>--------------</td>
<td>-------------</td>
<td>----------------</td>
</tr>
<tr>
<td>26</td>
<td>com.viewpagerindicator.</td>
<td>2370</td>
<td>viewpagerindicator</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>27</td>
<td>com.google.common.</td>
<td>2352</td>
<td>guava</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>28</td>
<td>com.flurry.org.</td>
<td>2148</td>
<td>flurry</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>29</td>
<td>com.google.protobuf.</td>
<td>2027</td>
<td>Protocol Buffers</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>30</td>
<td>biz.source_code.base64Coder.</td>
<td>2022</td>
<td>Base64Coder</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>31</td>
<td>com.revmob.</td>
<td>1864</td>
<td>RevMob</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>32</td>
<td>com.urbanairship.</td>
<td>1851</td>
<td>urbanairship</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>33</td>
<td>com.airpush.android.</td>
<td>1850</td>
<td>airpush</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>34</td>
<td>org.fmod.</td>
<td>1783</td>
<td>fmod</td>
<td>N</td>
<td>game</td>
</tr>
<tr>
<td>35</td>
<td>com.unity3d.</td>
<td>1742</td>
<td>unity3d</td>
<td>N</td>
<td>game</td>
</tr>
<tr>
<td>36</td>
<td>com.adobe.</td>
<td>1728</td>
<td>Adobe PDF Library SDK</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>37</td>
<td>com.mobfox.</td>
<td>1679</td>
<td>mobfox</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>38</td>
<td>com.chartboost.</td>
<td>1566</td>
<td>chartboost</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>39</td>
<td>org.slf4j.</td>
<td>1530</td>
<td>SLF4J</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>40</td>
<td>com.appbrain.</td>
<td>1522</td>
<td>appbrain</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>41</td>
<td>com.senddroid.</td>
<td>1510</td>
<td>Opt-Out</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>42</td>
<td>com.google.api.</td>
<td>1488</td>
<td>Google APIs Client Library</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>43</td>
<td>com.tapjoy.</td>
<td>1475</td>
<td>tapjoy</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>44</td>
<td>com.badlogic.gdx.</td>
<td>1468</td>
<td>libgdx</td>
<td>N</td>
<td>Game</td>
</tr>
<tr>
<td>45</td>
<td>com.paypal.</td>
<td>1460</td>
<td>Mobile Payment Libraries</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>46</td>
<td>kankan.wheel.</td>
<td>1457</td>
<td>android-wheel</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>47</td>
<td>com.adsdk.sdk.</td>
<td>1398</td>
<td>MobFox</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>48</td>
<td>com.j256.ormlite.</td>
<td>1333</td>
<td>OrmLite</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>49</td>
<td>com.nostra13.universalimageloader.</td>
<td>1319</td>
<td>Android-Universal-Image-Loader</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>50</td>
<td>com.loopj.android.</td>
<td>1276</td>
<td>Android Asynchronous Http Client</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>Rank</td>
<td>Namespace</td>
<td>Frequency</td>
<td>Library Name</td>
<td>Open Source</td>
<td>Category</td>
</tr>
<tr>
<td>------</td>
<td>---------------------------</td>
<td>-----------</td>
<td>------------------------</td>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>51</td>
<td>com.smaato.SOMA.</td>
<td>1222</td>
<td>SMAATO</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>52</td>
<td>com.umeng.</td>
<td>1202</td>
<td>umeng</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>53</td>
<td>org.jaxen.</td>
<td>1178</td>
<td>jaxen</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>54</td>
<td>com.jumptap.adtag.</td>
<td>1175</td>
<td>Jumptap</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>55</td>
<td>org.kxml2.</td>
<td>1173</td>
<td>kxml</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>56</td>
<td>com.Leadbolt.</td>
<td>1063</td>
<td>LeadBolt</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>57</td>
<td>com.adfonic.android.</td>
<td>1025</td>
<td>Adfonics Android SDK</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>58</td>
<td>org.ksoap2.</td>
<td>1017</td>
<td>KSOAP</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>59</td>
<td>org.kobjects.</td>
<td>1015</td>
<td>kobjects</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>60</td>
<td>com.applovin.</td>
<td>977</td>
<td>AppLovin</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>61</td>
<td>org.appcelerator.</td>
<td>975</td>
<td>Titanium</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>62</td>
<td>com.amazon.</td>
<td>965</td>
<td>AWS SDK</td>
<td>Y</td>
<td>Service</td>
</tr>
<tr>
<td>63</td>
<td>com.mopub.mobileads.</td>
<td>963</td>
<td>MoPub</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>64</td>
<td>jp.Adlantis.</td>
<td>956</td>
<td>AdLantis</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>65</td>
<td>org.anddev.</td>
<td>868</td>
<td>AndEngine</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>66</td>
<td>net.rbgrn.</td>
<td>841</td>
<td>GLWallpaperService</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>67</td>
<td>com.markupartist.</td>
<td>808</td>
<td>Actin Bar</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>68</td>
<td>com.inneractive.api.ads.</td>
<td>752</td>
<td>inneractive</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>69</td>
<td>com.openfeint.</td>
<td>743</td>
<td>openfeint</td>
<td>N</td>
<td>game</td>
</tr>
<tr>
<td>70</td>
<td>net.nend.android.</td>
<td>737</td>
<td>nendSDK</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>71</td>
<td>com.crittercism.</td>
<td>723</td>
<td>crittercism</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>72</td>
<td>jp.co.imobile.android.</td>
<td>711</td>
<td>imobile</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>73</td>
<td>com.commonsware.</td>
<td>676</td>
<td>commonsware</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>74</td>
<td>com.huntmads.admobadaptor.</td>
<td>670</td>
<td>HuntMAds</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>75</td>
<td>org.cocos2dx.</td>
<td>669</td>
<td>cocos2d-x</td>
<td>N</td>
<td>game</td>
</tr>
</tbody>
</table>
Table A.4. The Top 100 Libraries (Part 4).

<table>
<thead>
<tr>
<th>Rank</th>
<th>Namespace</th>
<th>Frequency</th>
<th>Library Name</th>
<th>Open Source</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>com.scoreloop.</td>
<td>667</td>
<td>scoreloop</td>
<td>N</td>
<td>game</td>
</tr>
<tr>
<td>77</td>
<td>crittercism.android.</td>
<td>657</td>
<td>crittercism (old version)</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>78</td>
<td>com.prime31.</td>
<td>648</td>
<td>prime31</td>
<td>N</td>
<td>game</td>
</tr>
<tr>
<td>79</td>
<td>net.daum.adam.publisher.</td>
<td>611</td>
<td>Ad@mPublisherSDK</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>80</td>
<td>com.mobclick.android.</td>
<td>594</td>
<td>mobclix (old version)</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>81</td>
<td>com.tapit.</td>
<td>587</td>
<td>tapit</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>82</td>
<td>com.sun.mail.</td>
<td>582</td>
<td>JavaMail</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>83</td>
<td>org.achartengine.</td>
<td>575</td>
<td>acharttengine</td>
<td>N</td>
<td>App-Dev</td>
</tr>
<tr>
<td>84</td>
<td>myjava.awt.datatransfer.</td>
<td>548</td>
<td>javamail</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>85</td>
<td>com.google.i18n.</td>
<td>545</td>
<td>libphonenumber</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>86</td>
<td>com.zestadz.android.</td>
<td>521</td>
<td>ZestADZ</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>87</td>
<td>mediba.ad.sdk.android.</td>
<td>514</td>
<td>mediba ad</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>88</td>
<td>com.cauly.android.ad.</td>
<td>504</td>
<td>cauly</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>89</td>
<td>com.MdotM.</td>
<td>496</td>
<td>MdotM</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>90</td>
<td>com.heyzap.</td>
<td>469</td>
<td>HeyZap</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>91</td>
<td>com.Localytics.android.</td>
<td>411</td>
<td>Localytics</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>92</td>
<td>com.admarvel.android.</td>
<td>410</td>
<td>AdMarvel</td>
<td>N</td>
<td>App-Dev</td>
</tr>
<tr>
<td>93</td>
<td>org.joda.</td>
<td>399</td>
<td>Joda</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>94</td>
<td>com.google.inject.</td>
<td>391</td>
<td>google-guice</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>95</td>
<td>net.robotmedia.</td>
<td>364</td>
<td>robotmedia/AndroidBillingLibrary</td>
<td>N</td>
<td>Service</td>
</tr>
<tr>
<td>96</td>
<td>com.pontiflex.mobile.</td>
<td>299</td>
<td>pontiflex</td>
<td>N</td>
<td>Advertising</td>
</tr>
<tr>
<td>97</td>
<td>com.omniture.</td>
<td>292</td>
<td>Marketing Cloud for Android</td>
<td>N</td>
<td>Analytics</td>
</tr>
<tr>
<td>98</td>
<td>com.github.droidfu.</td>
<td>282</td>
<td>DroidFu</td>
<td>Y</td>
<td>App-Dev</td>
</tr>
<tr>
<td>99</td>
<td>com.sponsorpay.sdk.android.</td>
<td>237</td>
<td>SponsorPay</td>
<td>N</td>
<td>App-Dev</td>
</tr>
<tr>
<td>100</td>
<td>com.getjar.sdk.</td>
<td>203</td>
<td>getjar</td>
<td>N</td>
<td>Advertising</td>
</tr>
</tbody>
</table>


[80] **AppBrain**, “Number of available Android applitions,” .


Vita
Wenhui Hu

EDUCATION

Doctor of Philosophy, Computer Science and Engineering  
Pennsylvania State University, University Park, PA  
Advisor: Dr. Patrick McDaniel and Dr. Peng Liu  
October 2016

Master of Science, Computer Science and Engineering  
Helsinki University of Technology, Espoo, Finland  
Graduation with Distinction  
January 2007

Bachelor of Engineering, Computer Science and Engineering  
Southeast University, Nanjing, China  
June 2002

ACADEMIC HONORS

- 1st Prize, Cisco Innovation Challenge (as group leader), Pennsylvania State University, Feb 2012.
- Travel Grant for Trusted Infrastructure Workshop, 2010.
- M.Sc. (Tech) with Distinction, Helsinki University of Technology, Jan 2007.
- Outstanding Graduate in Southeast University, 2002.
- Outstanding Graduation Project and Thesis in Southeast University, 2002.
- The first-class scholarship for outstanding students in Southeast University, 2001.
- The third-class scholarship for outstanding students in Southeast University, 1999, and 2000.

PROFESSIONAL EXPERIENCE

Research Assistant to Patrick McDaniel and Peng Liu, Pennsylvania State University, University Park, PA.  
20013 - 2016

Teaching Assistant Pennsylvania State University, University Park, PA.  
2007 - 2016

Software Designer Caner Oy, Helsinki, Finland,  

Network Administrator Changshu Information Center, Changshu, China,  

Software Engineer MacroTech Co. Ltd, Nanjing, China,  