Duet: Library Integrity Verification for Android Applications

Wenhui Hu, Damien Octeau, and Patrick McDaniel
Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA, USA
{whu, octeau, mcdaniel}@cse.psu.edu

Peng Liu
College of Information Sciences and Technology
Pennsylvania State University
University Park, PA, USA
pliu@ist.psu.edu

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 paper, 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. To mitigate these threats, we propose *Duet*, 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. We implemented a full working prototype of *Duet*. In a dataset with 100,000 Android applications downloaded from Google Play between February 2012 and September 2013, we verify integrity of 15 libraries. On average, 80.50% of libraries can pass the integrity verification. In-depth analysis indicates that code insertion, obfuscation, and optimization on libraries by application developers are the primary reasons for not passing integrity verification. The evaluation results not only indicate that *Duet* is an effective tool to mitigate library-centric attacks, but also provide empirical insight into the library integrity situation in the wild.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive Software*

Keywords

Smartphone; Android; third-party library; library-centric security threat; library integrity verification

WiSec'14, July 23–25, 2014, Oxford, UK.

http://dx.doi.org/10.1145/2627393.2627404.

1. INTRODUCTION

The Android operating system holds the biggest market share in the world of smartphones [22]. For such a success, third-party applications play an important role in the whole ecosystem [14]. 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 [39].

Besides advertising libraries, Android application developers also include other libraries into their applications. For example, Zebra crossing (ZXing) [9] is a library that provides functionality of 1D/2D barcode image processing. Another example is the license verification library (LVL) [26] 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* [11] 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 [46], *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 [43], 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 followup news [44], 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 [30, 19], anti-virus software [5, 3] and anti-repackaging techniques [15, 45] 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 have 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* [37] as our reverse-engineering tool. The reverse-engineering is also called re-targeting in this paper.

Duet need 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 illustrate the library modification by application developers in the real world.

2. BACKGROUND AND PROBLEM STATEMENT

2.1 Background

2.1.1 Libraries in Android

The libraries on a smartphone can be broken down into two categories. Some libraries enable execution of the op-

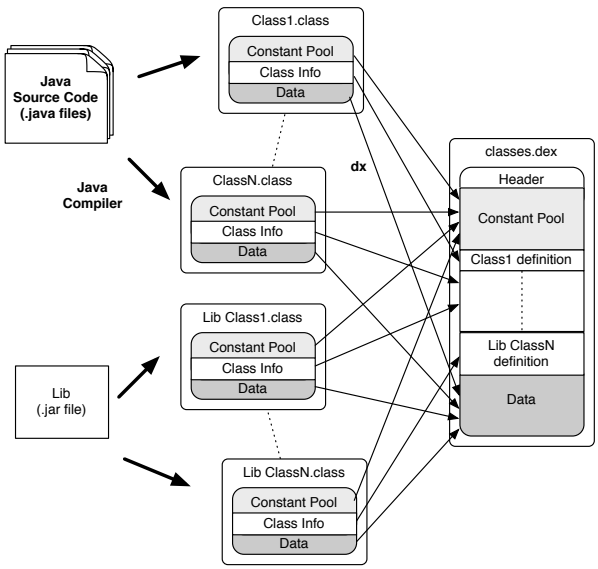


Figure 1: Compilation process for Android applications.

erating system. The second type of libraries are 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 [4]. The other libraries are generated within Android Open Source Project (AOSP) [2]. For instance, *libbinder.so* is the Binder library for Android interprocess 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 paper, we do not discuss attacks on these libraries or the verification of their integrity.

This paper 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 Section 1 target these libraries. In this paper, we propose a new integrity verification technique for these libraries, and this new technique can address these three threats.

2.1.2 Android Application Compilation

Android applications are developed in Java but compiled to Dalvik bytecode [10]. 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 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 [19, 37].

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.

2.1.3 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 paper, 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 paper, 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 6.

2.2 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 [24] 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 paper, 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

Section 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 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 [40] 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 [19]. 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.

2.3 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).

2.4 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 [11, 15] and Google is the victim of the masquerading threat [46]. 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 [43], benign application developers need library integrity verification to prove their innocence [44]. 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.

3. SYSTEM OVERVIEW

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 nor-

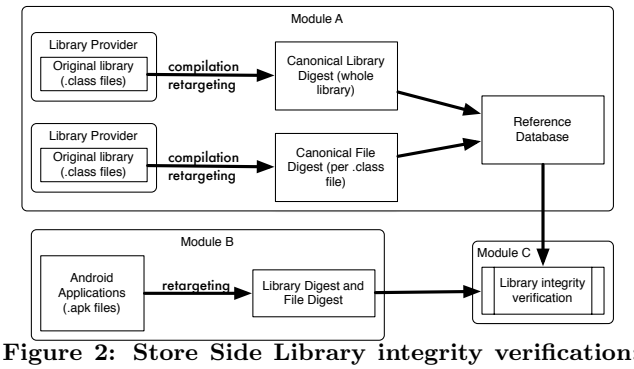


Figure 2: Store Side Library integrity verification: system overview.

mal 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.

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).

3.3 Architecture of Duet

As shown in Figure 2, *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 [25] 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. DESIGN AND IMPLEMENTATION

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 [37] 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 [36] 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.

4.1.1 Potential Attach 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 cases 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 [37], 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.

4.2 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.

4.2.1 Canonical Library Digest

As shown in Figure 3(a), *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.

Retargeting: We use *Dare* [37] 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 [37]. 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* [27] 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.

4.2.2 Canonical File Digest

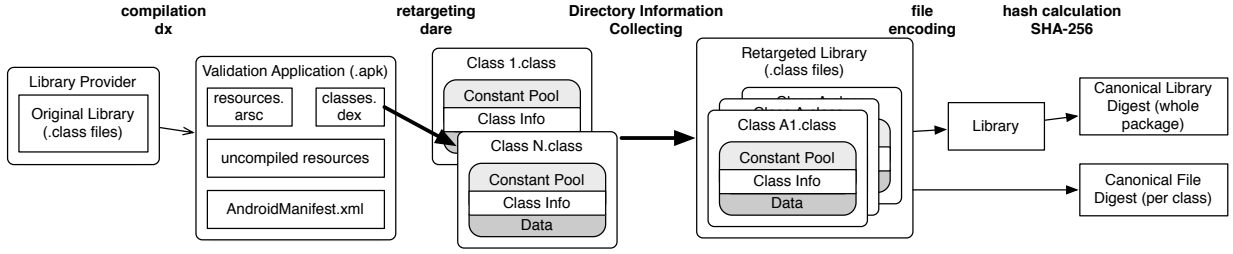
For each original library, *Duet* calculates a cryptographic hash for each retargeted *.class* file, which is a canonical file digest. These canonical file digests and the names of *.class* files are also stored into the reference database.

4.2.3 Issues

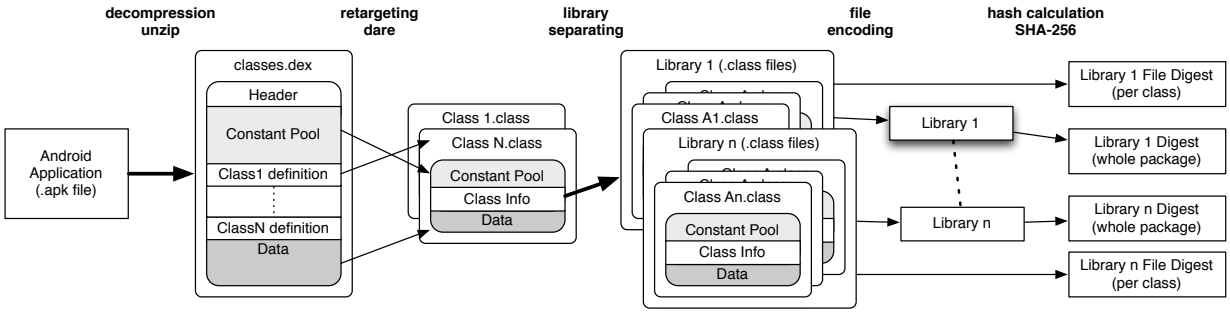
For the reference database, it is critical to store all legitimate canonical library digests and legitimate canonical file digests. Otherwise, *Duet* will make wrong conclusions for the integrity verification. It requires *Duet* not only supports legitimate behaviors of library providers but also supports 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* solve these three issues in Section 5.3.

4.3 Library Integrity Verification in Applications

As shown in Figure 3(b), the integrity verification in applications also needs five steps: decompression, retargeting,



(a) Calculating the canonical library digest and canonical file digest for original libraries.



(b) Calculating library digest and file digest for libraries in Android applications.

Figure 3: Calculation of library digest and file digest.

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 *classes.dex* from the *.apk* files of Android applications. The .apk files are in zip format. We use "unzip" to get the *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 *.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.

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 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.

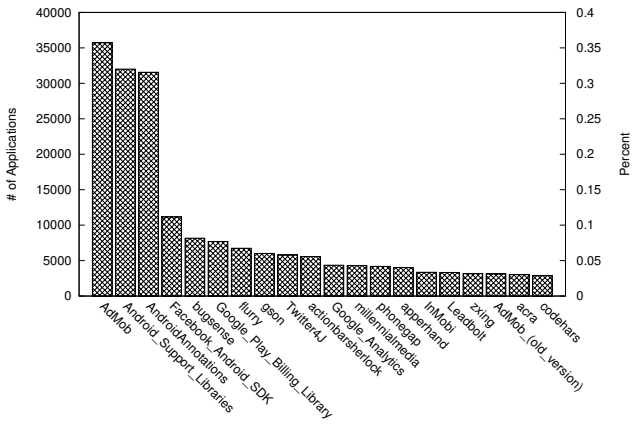
Table 1: Categories of the top 100 detected libraries, and their source code available situation.

	# of Libraries	# of Source Available	Source Available Percent (%)
App-Dev	39	34	87.18%
Advertising	34	0	0.00%
Service	10	2	20.00%
Analytics	9	0	0.00%
Game	8	0	0.00%
Total	100	36	36.00%

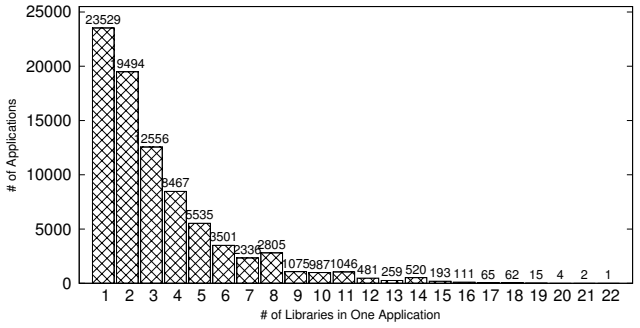
5.1 Libraries in Android Applications

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(a) 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 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.



(a) Popularity of the Top 20 Libraries in Our Dataset.



(b) Number of Libraries Contained by Each Application.

Figure 4: Library usage information.

The next popular 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(b), 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 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 security threats is very important for building a healthy smartphone application ecosystem.

Table 2: Usage information of the top 100 detected libraries.

	# of Category Usage	Percent in Total Usage (%)
App-Dev	149,291	54.03%
Advertising	84,617	30.62%
Service	16,762	6.07%
Analytics	17,458	6.32%
Game	8,189	2.96%
Total	276,317	100.00%

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 [1] 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.

Attack Methods	Class Modification	Repackaging
API Removal	Not Pass	Not Pass
API Addition	Not Pass	Not Pass
API Modification	Not Pass	Not Pass
Field Removal	Not Pass	Not Pass
Field Addition	Not Pass	Not Pass
Field Modification	Not Pass	Not Pass
.class File Removal	Not Pass	Not Pass
.class File Addition	Not Pass	Not Pass

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.

Table 3: Information of the Reference Database.

	Rank	# of Version	# of Canonical Library Digest	# of Canonical File Digest
AdMob	1	16	256	25,104
Android Support	2	41	656	130,080
AppBrain	40	89	1,424	56,316
AdFonic	57	9	144	5,280
Others		63	252	35,624
Total		218	2,732	252,404

Table 4: In-the-wild Library Integrity Verification Results. (All Known Versions)

	Rank	# of Detection	Library Digest	File Digest	Compilation with Customized Options	# of Passings	# of not Passings
AdMob	1	35,726	28,264(79.11%)	40(0.11%)	10(0.03%)	28,314(79.25%)	7,412(20.75%)
Android Support	2	32,002	23,243(72.63%)	37(0.12%)	24(0.07%)	23,304(72.82%)	8,698(27.18%)
AppBrain	40	1,522	1,050(68.99%)	235(15.44%)	0(0.00%)	1,285(84.43%)	237(15.57%)
AdFonic	57	1,025	876(85.46%)	0(0.00%)	0(0.00%)	876(85.46%)	149(14.54%)
Average			(76.55%)	(3.92%)	0(0.03%)	(80.50%)	(19.50%)

Building the Reference Database: As we explained in Section 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. 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 [27]. 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 are 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 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 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 passing, however, do not always mean malicious library modification or masquerading library. In Section 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 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.

Table 5: In-the-wild Library Integrity Verification Results. (Some Versions)

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

In both Table 4 and Table 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 3.3 by introducing the file digest as the supplement of the library digest.

In Table 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 customised *dx* options. This indicates that (1) *Duet* does tolerate different *dx* compiler options; (2) Very few application developers customise the *dx* compiler options.

5.4 Implications of Passing Rates

As we mentioned in Section 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

[1] *OldAdMob* was the library released by AdMob when it was an independent company.

unmodified, even though it was in reality modified. In Section 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 6.

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.

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 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, *Waston* [8] 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 admissive 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 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 contains special characters from Chinese or Japanese. We manually build a list of *.class* names for these obfuscated cases. As show in Table 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 [6] 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 [7] 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 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 [27]. 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 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

Table 6: In-depth Analysis of In-the-wild Verification Not Passing Samples.

	Rank	# of not Passing	Code Insertion	Obfuscation	Optimization	Missing Original Library	Remain
AdMob	1	7,412(20.75%)	63(0.18%)	5,772(16.16%)	509(1.42%)	451(1.26%)	617(1.73%)
Android Support	2	8,698(27.18%)	5,607(17.52%)	2,150(6.72%)	356(1.12%)	0(0.00%)	585(1.83%)
AppBrain	40	237(15.57%)	4(0.26%)	213(13.99%)	16(1.05%)	0(0.00%)	4(0.26%)
AdFonic	57	149(14.54%)	2(0.20%)	144(14.05%)	3(0.30%)	0(0.00%)	0(0.00%)
Average		(19.50%)	(4.54%)	(12.73%)	0(0.97%)	(0.32%)	(0.96%)

are willing to provide these 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.

7. RELATED WORK

Security and privacy in Android have become popular topics in the research community.

Smartphone platform security: Kirin [20] 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 [38] does runtime inspection on the permission state.

For application security, dynamic taint analysis is used in both Android OS [18] and iOS [17]. 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. [19] and Chan et al. [12] obtain the application source codes by decompiler and use existing tools to do analysis. Other works [13, 19, 21, 23, 28, 29, 30, 45, 47] do the analysis on either the class bytecodes or intermediate codes. These works have detected malware and give the direction of future research to improve the security on the Android platform.

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.

Android Advertising Library: Advertising libraries have been the focus of recent works. Grace et al. [28] and Steven et al. [42] 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.

Researchers proposed solutions to use different applications for application logic and for the advertising library [41, 34]. In this approach, the advertising library and the remaining application logic have their own protection domain. AdDroid [39] 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 paper, 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.

Library Detection: Library detection has been well studied. IDA Pro's Fast Library Identification and Recognition Technology (FLIRT) [32] is a popular library identification technique using byte pattern matching algorithms. In another work [31], 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. However, none of them can verify the library integrity.

Repackaging: Researchers have noticed repackaged Android applications. Zhou et al. [46] 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) [45, 15, 16]. The CFG and PDG are then used to detect the repackaged applications. These solutions are targeting to application plagiarism. They require to remove library code for making decisions on potential repackaging. Therefore, these techniques cannot be used to address the three library-centric security threats.

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 [33] and [35]. These techniques detect similarity between applications by comparing strings, tokens, trees, or semantics. None of them can be used to verify the software integrity.

8. CONCLUSION

There are three security library-centric threats in the world of Android. Since there is no existing technique that can effectively fully address these threats, we propose a novel technique for library integrity verification in application stores. In the evaluation, we use a dataset with 100,000 Android applications downloaded in the wild, and perform the library integrity verification on 15 libraries. Measurement results indicate that *Duet* is a useful technique to mitigate three security library-centric threats. Measurement results also provide the first empirical insight into the library integrity situation in the wild.

9. ACKNOWLEDGEMENTS

We thank Matthew Dering for providing our application samples. We also thank Stephen McLaughlin, Hayawardh Vijayakumar and our shepherd David Barrera for editorial comments during the writing of this paper. This work was supported by ARO W911NF-09-1-0525 (MURI), NSF CCF-1320605, and W911NF-13-1-0421 (MURI). This material is also based upon work supported by the National Science Foundation Grants No. CNS-1228700, CNS-0905447, CNS-1064944 and CNS-0643907. Any opinions, findings, and con-

clusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] android-apktool. http://code.google.com/p/android-apktool/.

[2] Android open source project. http://source.android.com.

[3] AntiVirus Security. https://play.google.com/store/apps/details?id=com.antivirus.

[4] Bionic. https://github.com/android/platform_bionic.

[5] Kaspersky Internet Security for Android. http://www.kaspersky.com/android-security.

[6] Proguard. Available at: http://proguard.sourceforge.net.

[7] Soot: a java optimization framework. Available at: http://www.sable.mcgill.ca/soot/.

[8] Watson library. http://grepcode.com/file/repo1.maven.org/maven2/com.octo.android.robospice/robospice-motivations/1.2.0/android/support/v4/app/Watson.java.

[9] Zebra crossing (zxing). https://github.com/zxing/zxing.

[10] Android Open Source Project. .dex - dalvik executable format, Nov 2007. http://source.android.com/devices/tech/dalvik/dex-format.html.

[11] AntiLVL. android cracking: Antilvl, 2013. Available at: http://androidcracking.blogspot.com/p/antilvl_01.html.

[12] Chan, P., Hui, L., and Yiu, S. Droidchecker: analyzing android applications for capability leak. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), ACM, pp. 125–136.

[13] Chin, E., Felt, A., Greenwood, K., and Wagner, D. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services* (2011), ACM.

[14] Cravens, A. A demographic and business model analysis of today's app developer. Available at: http://appdevelopersalliance.org/.

[15] Crussell, J., Gibler, C., and Chen, H. Attack of the clones: Detecting cloned applications on android markets. *Computer Security–ESORICS 2012* (2012), 37–54.

[16] Crussell, J., Gibler, C., and Chen, H. Andarwin: Scalable detection of semantically similar android applications. In *Computer Security–ESORICS 2013*. Springer, 2013.

[17] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium* (2011).

[18] Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., and Sheth, A. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), pp. 1–6.

[19] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A study of android application security. In *Proceedings of the 20th USENIX Security Symposium* (2011), vol. 2011.

[20] Enck, W., Ongtang, M., and McDaniel, P. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), ACM.

[21] Fuchs, A., Chaudhuri, A., and Foster, J. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland* (2009).

[22] Gartner. Android and samsung dominate the phone market in q1. Available at: http://www.engadget.com/2013/05/14/gartner-android-samsung-q1-2013/.

[23] Gibler, C., Crussell, J., Erickson, J., and Chen, H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. *Trust and Trustworthy Computing* (2012), 291–307.

[24] Google. Google Play In-app Billing. https://developer.android.com/google/play/billing/index.html?hl=en-Us.

[25] Google Inc. Andriod Support Library. http://developer.android.com/tools/support-library/index.html.

[26] Google Inc. Lvl: License verification library. http://developer.android.com/google/play/licensing/index.html.

[27] Google Inc. Release Notes - Google Mobile Ads SDK. https://developers.google.com/mobile-ads-sdk/docs/rel-notes.

[28] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks* (2012), ACM.

[29] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012).

[30] Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM.

[31] Griffin, K., Schneider, S., Hu, X., and Chiueh, T.-C. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection (RAID)* (2009), Springer.

[32] Guilfanov, I. Fast Library Identification and Recognition Technology (1997). https://www.hex-rays.com/products/ida/tech/flirt.shtml.

[33] Jiang, L., Misherghi, G., Su, Z., and Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering* (2007), IEEE Computer Society.

[34] Leontiadis, I., Efstratiou, C., Picone, M., and Mascolo, C. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications* (2012), ACM, p. 2.

[35] McMillan, C., Grechanik, M., and Poshyvanyk, D. Detecting similar software applications. In *Software Engineering (ICSE), 2012 34th International Conference on* (2012), IEEE.

[36] Meyer, J., Reynaud, D., and Kharon, I. Jasmin home page. http://jasmin.sourceforge.net/, 2004.

[37] Octeau, D., Jha, S., and McDaniel, P. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering* (November 2012).

[38] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In *2009 Annual Computer Security Applications Conference* (2009), IEEE.

[39] Pearce, P., Felt, A., Nunez, G., and Wagner, D. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of AsiaCCS* (2012).

[40] Poeplau, S., Fratantonio, Y., Bianchi, A., Kruegel, C., and Vigna, G. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS)* (2014).

[41] Shekhar, S., Dietz, M., and Wallach, D. Adsplit: separating smartphone advertising from applications. In *Proceedings of the 21st USENIX conference on Security symposium* (2012), USENIX Association.

[42] Stevens, R., Gibler, C., Crussell, J., Erickson, J., and Chen, H. Investigating user privacy in android ad libraries. In *Proceedings of IEEE Mobile Security Technologies (MoST)* (2012).

[43] Zhang, Y., Xue, H., Wei, T., and Song, D. Ad vulna: A vulnaggressive (vulnerable & aggressive) adware threatening millions. FireEye Blog, http://www.fireeye.com/blog/technical/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html, 2013.

[44] Zhang, Y., Xue, H., Wei, T., and Song, D. Monitoring vulnaggressive apps on google play'. FireEye Blog, http://www.fireeye.com/blog/technical/2013/11/monitoring-vulnaggressive-apps-on-google-play.html, November 2013.

[45] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM.

[46] Zhou, Y., and Jiang, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

[47] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of the 19th Annual Network and Distributed System Security Symposium (NDSS)* (2012).