Measuring code reuse in Android apps

Hugo Gonzalez and Natalia Stakhanova and Ali A. Ghorbani
Faculty of Computer Science, University of New Brunswick
hugo.gonzalez,natalia,ghorbani@unb.ca

***Abstract*—The appearance of the Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of mobile threats. Leveraging openness of Android app markets and the lack of security testing, malware authors commonly plagiarize Android applications through code reuse, boosting the amount of malware on the markets and consequently the infection rate. In the last few years the number of studies focused on detection of mobile app code reuse has drastically increased. Ranging from lightweight detection of suspicious signs to more sophisticated and computationally expensive methods assessing apps' similarity, the studies treated the presence of code reuse as a sign of plagiarized apps and maliciousness.**

In this work, we revisit this assumption and investigate code reuse in legitimate and malicious mobile apps. The main questions that this study aims to answer are what it is that is being reused, what we can learn from this reuse and consequently how we can use this knowledge. To answer these questions we measure code uniqueness and identify common components originating from third-party sources. We further analyze and correlate reused code extracted from over 60,000 apps from ten markets around the world and commonly used app repositories. As our analysis shows, understanding code reuse can shed some light on app origin and evolution.

Keywords: Android, code reuse

I. INTRODUCTION

The appearance of Android platform and its popularity has resulted in a sharp rise in the number of reported vulnerabilities and consequently in the number of threats. According to Kaspersky's report, the number of mobile malware targeting the Android platform tripled in 2013, reaching 98.1% of all mobile malware [5]. In 2014, Android malware continued its unprecedented growth reaching the 600% increase.

The further analysis of this behavior shed light into this phenomenon [26], [9]. Attributing this increase to the lack of comprehensive security protection on the Android platform, the studies emphasized the ease of applications' reverse engineering that allows malware authors to rapidly produce apps reusing legitimate code and repackaging it with added malicious functionality. Quickly becoming a popular vehicle for malware propagation through Android markets, repackaged apps are overwhelming markets. For example, the official GooglePlay market was reported to host 2% of repackaged malware, while on several unofficial markets this number is reaching 50% [9]. Such popularity of repacking has drawn significant research attention, and a number of studies have emerged focused especifically on the detection of app repackaging [7], [25], [24], [22], [15] and methods to detect tools to perform repackage [10].

In spite of variability in complexity and depth of analysis, all these studies exhibit similar characteristics. Focusing on malware detection, these works rely on the underlying assumption that repackaging is associated with malicious functionality. In other words, measuring code similarities between apps can indicate code reuse and hence potential repackaging. This creates several challenges: First, discovered code similarity (code reuse) might in fact be attributed to common libraries that are meant to be shared and thus should be excluded from analysis and subsequent detection. Second, while the large percentage of malware apps are repackaged, there is a significant amount of non-repackaged malware. This practice is also common among legitimate apps that may be repackaged with different resource files customized to local markets, or with advertisements to bring additional profit to app owners. Finally, code may exhibit similarity due to reasons independent of repackaging such as code evolution, or the same origin or developer. Thus building defenses based on repackaging signs, i.e., similarity of code, without a proper understanding of the nature of this similarity, i.e., code being reused, introduces bias into detection and limits the approach's efficiency.

In this work, we investigate code reuse in legitimate and malicious mobile apps in Android marketplaces. In particular, this study aims to shed light on several questions: What type of code is being reused? What is the difference between reused code in legitimate and malware apps? How much of this malicious reused code could be identifiable? and finally What else could we learn from the reused code?

Answering these questions will allow us to better understand the specifics of code shared among mobile applications and will give us leverage in developing better, more efficient and accurate malware detection.

To this end, we developed a system called CodeCheck to measure the amount of shared code. Specifically, given an app, CodeCheck extracts the .dex file and breaks it into smaller components corresponding to its structure (e.g., classes, methods, strings). To measure uniqueness of components CodeCheck converts their opcodes into hash representations. These representations are then further analyzed and correlated to reveal common code and components originating from third-party sources.

We use CodeCheck to study ten Android markets worldwide including official Google Play and nine third-party marketplaces. We also analyze a large set of over 45 000 Android malware apps identified as botnet malware, samples from VirusTotal stream and well known datasets employed in previous studies: Android Malware Genome Project [26],

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

and Drebin [4]. Through this analysis we created a database of common code components shared among the majority of Android apps. Our analysis shows that approximately 50% of all analyzed code is common, and despite the common beliefs, the legitimate mobile apps feature more code reuse that malicious apps.

II. Related Work

The past decade has seen extensive research in the area of mobile security. A broad study looking at a variety of mobile technologies (e.g., GSM, Bluetooth), their vulnerabilities, attacks, and the corresponding detection approaches was conducted by Polla et al. [16]. More focused studies surveying characteristics of mobile malware were offered by Alzahrani et al. [3], Felt et al. [8] and Zhou et al. [26].

With the recent burst of research interest in the area of Android device security, there have been a number of general studies offering methods for malicious app detection. These methods can be broadly divided into those focused on the detection prior to app installation (e.g., market analysis) and those that monitor app behavior directly on a mobile device. Among the studies in the first group are RiskRanker [12], DroidRanger [27], DroidScope [21], that dynamically monitor mobile apps behavior in an isolated environment collecting detailed information that might indicate maliciousness of a sample. Similarly, DroidMat [20] and DroidAPIMiner [2] look at the identification of malicious apps using machine learning techniques. Since these techniques are computationally expensive for a resource-constraint environment of a mobile platform, they are mostly intended for offline detection. Among studies that focus on malware detection on a mobile device directly is Drebin [4]. This approach employs static analysis in combinat1ion with machine learning to detect suspicious patterns in app behavior.

With a recent wave of repackaged applications, a number of studies looked at the problem of mobile apps similarity. The majority of the existing methods look at the content of .dex files for app comparison. Juxtapp [13] evaluates code similarity based k-grams opcodes extracted from selected packages of the disassembled .dex file. The generated k-grams are hashed before they undergo a clustering procedure that groups similar apps together. Similarly, DroidMOSS [25] evaluates app similarity based of fuzzy hashes constructed based on a series of opcodes. DroidKin [11] detects unique apps analyzing the similarity of opcodes and metadata of apps. ViewDroid [22] leverages user interface-based birthmarks for detecting app repackaging on the Android platform. Several methods were developed to fingerprint mobile apps to facilitate the detection. For example, AnDarwin [6], DNADroid [7], and PiggyApp [24] employ Program Dependance Graphs (PDG) that represent dependencies between code statements/packages. Potharaju et al. [17] computes fingerprints using Abstract Syntax Tree (AST). In traditional malware omain, BitShred [14] also looked at similarity of apps. Aiming to increase the speed of analysis, BitShred offered a scalable comparison of malware features to determine similarity among samples of different families.

header	Structural information
string_ids	Offset list for strings
type_ids	Index list into the string_ids for types
proto_ids	Identifiers list of prototypes
field_ids	Identifiers list of fields
method_ids	Identifiers list of methods
class_defs	Structure list of classes
data	Code and data
link_data	Data in statically linked files

Fig. 1. The structure of a .dex file

The main focus of these tools is the detection of similar apps. Such similarity is the first indication of potential app plagiarism or repackaging. Our work is complementary to all of these approaches, aiming to understand the nature of code that is being reused among multiple apps, we provide these similarity approaches with tools to only focus on unique code not present in any of the publicly available sources (code most relevant for similarity analysis) and thus potentially improve their efficiency.

Our work is most closely related to PlayDrone study by Viennot et al. [19]. Focusing on characteristics of GooglePlay market, the authors looked at the libraries' usage. Although within our study we also provide insight into the use of libraries by developers, we broaden our focus to shared code in general rather than the use of libraries. In a similar vein, AdRob [9] looks at library usage and popularity of markets among cloned applications. As opposed to AdRob and PlayDrone, we look at various markets and contrast code reuse between both malicious and legitimate applications to give further foundation for the development of efficient repackaging detection.

III. Background

An Android app is in essence a package (.apk file) that contains the executable file in Dalvik executable format(.dex file); a manifest file (AndroidManifest.xml) that describes the content of the package, some of the behavior information, the entry points for the code and the permissions information; optional native code (in form of executable or libraries) that usually is called from the .dex file using the java native interface or jni; a digital certificate authenticating an author; and the resources that the app uses (e.g., image, sound files etc.). Each .apk file is annotated with additional information, so called meta-data, such as the app creation date and time, version, size, etc.

Theoretically, any part of an .apk file can be reused. Practically, code reuse is commonly seen on several levels:

- *Meta-data level*: these are the surface code characteristics easily reachable from various components of .apk package, such as AndroidManifest.xml, a digital certificate authenticating an author, the meta-data. These components

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

are best described as application settings and thus are often reused among developers.

- *Resource level*: visual features of Android apps are commonly embedded in resources files (e.g., images, sounds, UI aspects). Some of these resources have their origins on the Android SDK and sample applications. Sometimes, the reuse of these visual aspects aims at human recognition of Android applications and thus commonly used by malware authors to hide malicious code under legitimate and recognizable interfaces.
- *Binary level*: code reuse at this level is the most commonly seen practice. It is typically due to the inclusion of different libraries, the wide availability of sample and open source apps.

Both meta-data and resource level reuse are complementary to reuse at a binary level. As such in this work we explore code reuse at the binary level and focus specifically on .dex file. The .dex file is a binary container for the Java code and the associated data. It includes a header containing meta-data about the executable followed by identifier lists that contain references to strings, types, prototypes, fields, methods and classes employed by the executable. The final part of the .dex file is the data section that contains code and data. The structure of .dex file is shown in Figure 1.

One of the primary motivations for this work lies in the substantial overhead incurred by SDKs and various libraries in many cases automatically included in an app. In general, depending on the complexity of an Android app and the libraries included, the number of methods could vary from 10 to 10,000.

For example, a simple "Hello World" Android program compiled with default settings results in 8,095 various methods, most of which are common, i.e., meant to be reused and found in the majority of existing apps. On top of this, a quick n-gram based similarity analysis of this program reveals a close resemblance of Zitmo malware [1]. Giving insight and meaning into this overhead can help to improve accuracy and speed of the analysis.

A. Transforming the apps

Obfuscation is commonly used approach to protect an app/malware from reverse engineering or detection. The obfuscation transformations can be broadly divided into the following categories[2]:

- *Level 0. Trivial transformations*: that change the final hashing signature without any code modifications. These modifications include repackaging, re-alignment, re-signing, rebuilding, files replacement (as such as icon), and addition of junk files.
- *Level 1. Syntactic code transformations*. These transformations target analysis activities and affect the amount of effort needed to understand the app once it has been reverse engineered. This category includes changes in the package name, identifier renaming, method renaming, class renaming, string encryption, inserting defunct methods. ProGuard and DexGuard perform some of these transformations.
- *Level 2. Optimization transformations*. These transformations are not designed to obfuscate code, but rather optimize it. At this level of transformations, common technique are shrinking, code rearrangement and removal of code duplication. Proguard also offers Level 2 optimization.
- *Level 3. Semantical code transformations.* This category includes code reordering, junk code insertion, encrypting payloads and native exploits, function inlining and outlining, reflection, bytecode hiding, bytecode encryption, call indirections.

[1]We employed publicly available web interface of DroidKin [11]

[2]Rastogi et al. [18] described three different classes of transformations for Android apps where they included the majority of the transformations described by Zheng et al. [23]. Since these classifications have a narrow view of transformation, we present a broad list of obfuscation transformations ranging from simple to more advanced techniques.

IV. Design

The proposed approach aims to systematically identify common code, i.e., the code actively reused by Android app developers. As such CodeCheck design has dual purpose. On the one hand, it generates a collection of reused code. On the other hand, it provides a benchmark for a given app to assess its uniqueness, i.e., indicate amount and nature of its code reuse.

In both cases, the proposed system encompasses three steps: *feature extraction*, *hashing*, and *assessment*. For each app, the system derives relevant features and forms vectors that serve as a basis in the assessment where code components' reuse information is accumulated and measured. Once the stream of apps is processed, code usage statistics is analyzed and distributed among corresponding repositories. These repositories are the main benchmarks for further analysis of individual apps.

A. Processing stream of apps

In this work, we focus primarily on binary code reuse and specifically on analysis of .dex file. Due to the nature of the .dex structure, code is distributed among various parts of an executable file. We follow class and method identifiers lists to obtain information on classes and methods. This approach gives us access to all methods referred to by .dex file, whether specifically defined in the file or not. Although methods with identical names should have unique content, it has been observed that malicious functionality is often hidden under familiar method's names. Moreover, methods' names are often obfuscated. As such we proceed to extract both method's name and its corresponding opcodes.

Opcodes are generally favored in representing low-level semantics of the code. Although extracting opcodes alone might abstract specific details describing a program control transfer or an arithmetical operation, enhancing them with the corresponding operands creates variability in code. We thus resort to using opcodes alone.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

To provide unique and easily matched fingerprint for each method, extracted opcodes are hashed using MD5 algorithm. The resulting hashes are then correlated and analyzed by categories.

B. Common code repositories

Commonly reused code is stored in several repositories according to its origin and classification. The origin is closely related with where the code was found, while classification is based on analysis of verified non-suspicious apps.

Once created these repositories are maintained and updated following the same procedure[3].We distinguish the following categories of common code:

- **Android SDK related code** (a.k.a Baseline filter): The development of an Android app requires a set of libraries and essential tools that are provided in Android SDK environment. These include a debugger, a handset emulator based on QEMU, libraries, sample code with examples on the use of these libraries, documentation, and tutorials. We create a baseline from the android.jar libraries and samples distributed with different versions of the Android SDK. These versions include API 22 MNC preview, API 22/Android 5.1.1, API 21/Andorid 5.0.1, API 20, L preview, API 20/Android 4.4W.2, API 19/ Android 4.4.2, API 17/Android 4.2.2, API 16/Android 4.1.2, API 12/Android 3.1, API 10/Android 2.3.3, API 8/Android 2.2 and API 7/Android 2.1. All those include Android support libraries with more features included in each new release.
- **Open source apps**: The code in this category cames from free open source apps. These samples do not include ads libraries or commercial libraries. Some open source libraries might be present in these apps. These category only includes full functional apps, i.e., no code samples.
- **Internet apps code samples**: This category includes app samples illustrating Android app development and collected from online repositories such as github, online communities, Android tutorials and personal blogs. This category also includes clients for well known services (e.g., banking app), open source games and utilities. Note that open source libraries collected from these sources are not included in this category.
- **Market-based apps**: This category includes various filters, based on the markets we analyzed. Non-suspicious apps from each market are used to create a filter for that specific market. Note that all methods found the Baseline filter are removed prior to creating market filters. We also compile an extra filter with the commmon code appearing in different markets.
- **Generic methods**: This filter includes all methods that typically appear in the majority of libraries, and those that are only in obfuscated classes. Usually these methods are small and very generic. init() and cinit() are examples of these methods. These methods are extracted from the apps after applying Baseline filter.
- **Open source libraries**: This filter includes free open source libraries created or migrated to Android to facilitate app development such as kawa, apache, actionbarsherlock, spongycastle. These are libraries focused on communications, security, interface enhancement, data processing, graphics, etc. These libraries were extracted from non-suspicious apps from all the markets after applying Baseline filter.
- **Ads libraries**: A common approach to profit from Android apps is to incorporate advertisements in an application. Many advertisement libraries now offer analytic support and the corresponding libraries to be included in an app. Some examples of these libraries are Acra, Sponsorpay, Localytics. Since the use of certain libraries often leads to aggressive app behavior, many Android antivirus engines flag apps with specific ads libraries as unwanted. In addition, several studies pointed out that the use of advertisement SDKs is potentially indicative of app's maliciousness [9]. This filter includes all the ads libraries identified in all market apps.
- **App creators**: This category refers to the platforms that allow us to generate a complete Android app from prebuild components. Some examples of these generators are Appcelerator, Titanium, Andromo. We identified several classes commonly used by this platforms and created a filter to reflect their presence.
- **Commercial third party libraries**: There are libraries that do not incorporate any advertisements, are not open source, e.g., game engines unity3d or cocos2d. The use of such libraries must be accompanied by a paid license.
- **Social Network APIs**: These APIs commonly provide access to social networks and services from within an app.

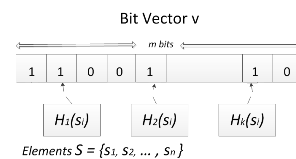

Fig. 2. An illustration of a Bloom filter

*a) **Bloom filter**:* In order to maintain efficiency in storage and speed, common code repositories are implemented as Bloom filters. A Bloom filter is a simple data structure that allows space-efficient representation of set elements to support fast memberships queries. Given a set $S = s_1, s_2, \ldots, s_n$ of n elements, the Bloom filter relies on a bit vector and a set of hash functions. The idea of a Bloom filter is illustrated in Figure 2.

Given a vector v of m bits, initially set to 0, and a set of k hash functions $H = h_1, h_2, \ldots, h_k$, an element $s_i \in S$ will set all bits of vector v corresponding to the following positions $h(s_i)_1, h(s_i)_2, \ldots, h(s_i)_k$ to 1. Note that the same

[3]We make these repositories publicly available to a wider academic community for further research: link is blinded for review

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

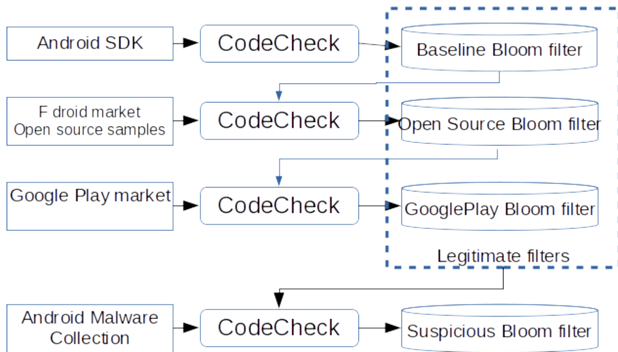

Fig. 3. The process of building Bloom filters

bit might be set to 1 multiple times either through different hash functions for the same element s_i or different elements of S. The process of verifying if an element b is present in the Bloom filter is analogous to the insertion process with one difference: instead of setting the bits corresponding to the hashes to 1 we verify if the bits corresponding to positions $h(b)_1, h(b)_2, \ldots, h(b)_k$ are already set to 1. If at least one bit is set to 0 then an element b is not in the Bloom filter.

*b) **Analysis of individual apps**:* Once repositories are in place, analysis of an individual app is expedited. An Android app is processed in fashion similar to processing a stream of apps. Opcodes are extracted from each of the discovered .dex file methods and hashed to produce feature vectors. These vectors are then matched against Bloom filters to discover the presence and nature of code reuse. To assess the amount of code reuse, feedback from Bloom filters is summarized as a percentage of code reuse in each filter category.

V. Data

To evaluate the performance of the proposed approach and study the characteristics of code reuse, we gathered a large collection of Android applications from ten markets around the globe, VirusTotal, and three datasets: Android Malware Genome project [26], Drebin [4] and Android botnet dataset, provided for our study by an antivirus vendor.

Market apps were collected between September of 2014 and August of 2015 and represent the most popular apps in the corresponding markets. All apps were inspected by ESET anti-virus scanner to ensure the absence/presence of malware. As such malware apps retrieved from GooglePlay market were filtered out to retain benign samples only. The final set of market apps contains 17,743 apps: 4574 apps from GooglePlay, 1395 apps from Fdroid, an open-source Android market, 5631 apps from six Chinese markets, 1248 from Swedish market, called Appland and 4895 from independent Aptoide market.

We combined the malicious apps from the Android Malware Genome project, Drebin and the Android botnet dataset; and removed all apps with duplicated .dex files to eliminate redundancy. The remaining apps were combined in one set that we refer to as Android Malware Collection.

TABLE I
Analyzed Data

Origin	Total apps	Legitimate	Malware	Unwanted	Unsafe
Android SDK related samples	50	50	0	0	0
Internet Samples	100	100	0	0	0
Fdroid	1395	1395	0	0	2
GooglePlay	4574	4572	2	82	31
3gyu (anruan)	1564	1551	13	228	96
anzhi	2192	2132	60	623	345
appland	1248	1235	13	252	23
aptoide	4895	4891	4	729	51
gfan	454	454	0	69	83
jimi	330	329	1	25	3
mumayi	45	44	1	10	2
nduoa	1046	1036	10	117	293
Android Malware Collection	3928	0	3928	-	-
Virus Total stream	41448	0	41448	-	-
Total:	63269	17789	45480	2135	929

TABLE II
Statistics of the baseline filter code reuse

Source	Total # of methods	Common methods	Other methods	Common methods	Other methods
Internet Samples	642462	436001	206461	67.86%	32.14%
Fdroid	5375424	3629595	1745829	67.52%	32.48%
GooglePlay	94946363	62022265	32924098	65.32%	34.68%
3gyu	10109255	5753632	4355623	56.91%	43.09%
anzhi	16783573	9588928	7194645	57.13%	42.87%
appland	15758682	9972300	5786382	63.28%	36.72%
aptoide	105087308	68132509	36954799	64.83%	35.17%
gfan	4281811	2375660	1906151	55.48%	44.52%
jimi	850013	421208	428805	49.55%	50.45%
mumayi	614394	350402	263992	57.03%	42.97%
nduoa	7404078	4189538	3214540	56.58%	43.42%
Android Malware Collection	14104241	7221264	6882977	51.20%	48.80%

We also collected and compiled 100 apps from github, various online tutorials and blogs. These samples are bundled under the 'Internet code samples' category. 50 app samples were retrieved from official Google framework examples.

The resulting set of 63,269 samples employed for our experiments contains only apps with unique MD5 hash values. The details of our datasets are shown in Table I. The Table also shows the number of apps labeled by ESET as unwanted, unsafe or malware.

VI. Experimental results

To understand code reuse, we conducted several experiments. In the first round of experiments, we create a baseline filter and populated the filters with the apps from the specified markets. In this round, we aim to evaluate existence of common code in the markets. In the second round of experiments, we analyze the content of filters in more details and create new Bloom filters based on the functionality of the methods (e.g., advertisement libraries). In the third round, we measure common, rare and unique code, and the objectives of the methods based on the name of the libraries that it belongs. In the final round, we study 15 well-known apps with respect to the created filters.

A. Baseline filter

In the first experiment, we aim to measure the presence of baseline common code between the markets. The baseline filter is created from the Android SDK related code. Since most of the apps are build using Google Android SDK, we envision this filter will represent a large portion of the code reused in the apps. The filter contains 18,532 unique methods.

The application of this filter shows that the majority of apps across different markets contain over 50% of Android SDK

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

related code (Table II). Removing even this amount of code during similarity analysis will allow to significantly improve the performance of the method.

After removing code present in baseline filters, we take a further look at the remaining code (referred to in Table II as 'other methods'). In this remaining portion we measure frequent code (if the method appears in at least 5% of the apps), rare code (if it appears in more than 1%) and unique code (if the method appears in less than 1% of the apps). Table III shows the code breakdown in three datasets. GooglePlay market apps contain the smallest amount of rare and unique code (13%), while apps in Android Malware Collection feature above 21%, of such code. In essence, this is the code that malware detection analysis should focus on.

TABLE III
STATISTICS OF THE CODE OCCURRENCE

Source	Freq. code	Rare code	Unique code
GooglePlay	87.31%	7.63%	5.06%
Fdroid	87.94%	2.5%	9.56%
Android Malware Collection	73.17%	21%	5.83%

B. Detailed content of frequent code

A more detailed insight into the common code reused from GooglePlay data is shown in Table IV. As expected the majority of common methods come from various libraries, while 5% of code is effectively borrowed from other legitimate apps.

Further analysis of this borrowed code showed that most of the employed methods are longer and about 25% of them are named obfuscated. Class/method name obfuscation approach was mentioned as Level 1 or our transformations propose. This transformation is commonly used by Proguard, the well known Android obfuscator, integrated into the Android framework. Such Proguard transformation is typically recommended by various tutorials and manuals, and as our result shows consistently followed by developers, i.e., 40.93% of all methods in common code were obfuscated.

In spite of slightly larger methods being reused from other apps, distribution of common methods' size (in terms of their opcode representation) across categories seems to be fairly uniform (Figure 4). It is interesting to note that on average smaller methods are reused more frequently and usually under various names, i.e., although class/method name changes the content remains exactly the same. This was noted before as a sign of malicious functionality being hidden under familiar method's names. As this experiment showed such practice is common across legitimate apps as well, and thus cannot be relied upon in malware detection.

C. Open source filter

After application of baseline filters, the remaining code is used to create a new filter that contains the common code found in open source apps available on the Internet. Fdroid and collected open source apps serve as a primary source of apps for this filter. This filter contains 20814 methods. Applying this filter to the apps retrieved from various markets shows that a fairly small amount of code (a little over 5%) seems to be borrowed from open source apps (Table V).

TABLE IV
ANALYSIS OF COMMON METHODS (EACH REPOSITORY SEPARATELY)

Category	Number of methods		
	GooglePlay	Fdroid	Android Malware Collection
Android libraries	218294 (78.63%)	11319 (83.50%)	155546 (71.53%)
Ads libraries	7563 (2.72%)	24 (0.18%)	1931 (2.49%)
Open source libraries	10811 (3.89%)	2469 (18.21%)	2434 (3.13%)
App creators	3978 (1.43%)	27 (0.20%)	72 (0.09%)
Third party libraries	571 (0.21%)	0	6 (0.01%)
Social network APIs	29166 (10.51%)	164 (1.21%)	1230 (1.58%)
Android apps	14057 (5.06%)	113 (0.83%)	16611 (21.39%)
Overall obfuscated	40.93%	6.68%	12%

TABLE V
STATISTICS OF THE OPEN SOURCE FILTER CODE REUSE

Source	Total # of methods	Common OS methods	Common OS methods %
GooglePlay	94946363	6597425	6.95%
3gyu	10109255	763882	7.56%
anzhi	16783573	1127782	6.72%
appland	15758682	1091356	6.93%
aptoide	105087308	5906044	5.62%
gfan	4281811	271498	6.34%
jimi	850013	68187	8.02%
mumayi	614394	44286	7.21%
nduoa	7404078	544668	7.36%
Android Malware Collection	14104241	1331438	9.44%

D. GooglePlay market analysis

After filtering out baseline and open source code, a filter with most frequently used methods (appearing in at least 2.5% of apps) is created using 4574 apps retrieved from GooglePlay market. We consider this filter as benign as all malicious/unsafe samples were removed prior to analysis. It is important to note that none of the previous filters include methods from advertisement libraries, third-party libraries or malicious code. This filter contains 154443 methods. Table VI presents the statistics of GooglePlay code.

TABLE VI
STATISTICS OF THE GOOGLEPLAY FILTER CODE REUSE

Source	Total # of methods	Common GP methods	Common GP methods %
3gyu	10109255	1037433	10.26%
anzhi	16783573	1576796	9.39%
appland	15758682	3261629	20.70%
aptoide	105087308	18684876	17.78%
gfan	4281811	449081	10.49%
jimi	850013	75732	8.91%
mumayi	614394	80808	13.15%
nduoa	7404078	755617	10.21%
Android Malware Collection	8018472	836425	10.43%

E. Suspicious Filter

The suspicious Bloom filter was build on code derived from Android Malware Collection and not present in baseline, open source, and GooglePlay filters. This filter contains 15317 unique methods. Note that code present in this filter is not necessarily malicious, but clearly not commonly present in legitimate apps.

Application of this filter to other markets reveled interesting tendencies. The summary is given in Table VII. The amount of code shared by these apps on alternative markets is less than 1%. In other words, roughly 1% of apps contains code that resembles known malware.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

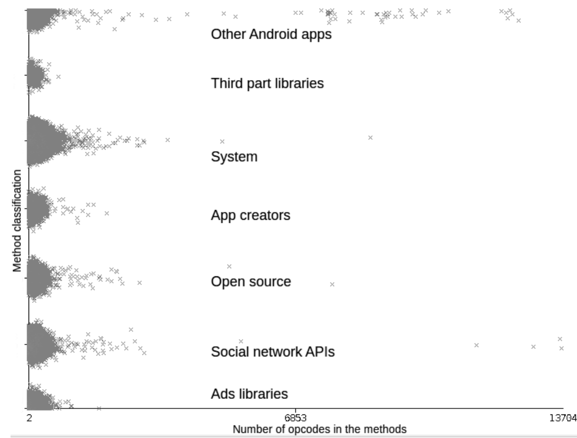

Fig. 4. The size of the methods grouped by category.

Manual analysis of malicious Bloom filters revealed several large methods distributed across only a few apps. The largest methods with 12130 and 4521 opcodes were found in six different malicious samples. These samples were repackaged versions of an open source game enriched with DroidKunfu malware functionality.

Following previously reported observation that the use of advertisement SDKs is potentially indicative of app's maliciousness [9], we notice no visible distinction between the use of ad code/libraries in benign and malicious apps.

TABLE VII
STATISTICS OF THE SUSPICIOUS FILTER APPLICATION

Source	Total # of methods	Common suspicious methods	Common suspicious methods %
3gyu	10109255	70466	0.70%
anzhi	16783573	100218	0.60%
appland	15758682	27809	0.18%
aptoide	105087308	55039	0.05%
gfan	4281811	10633	0.25%
jimi	850013	38784	4.56%
mumayi	614394	2504	0.41%
nduoa	7404078	30673	0.41%

F. Duplicated methods in Android Malware Collection

We analyzed how much of the code had been reused in Android malicious datasets used in previous studies. Figures 5 show the true amount of code duplication in these datasets. The left hand side demonstrates the amount of unique methods as the dataset being matched to legitimate filter. This is contrasted by the right hand side that shows possible reduction in code if duplicate hashes were to be stored in Bloom filter. Both Drebin and Botnet data show a reduction of unique methods by half, indicating a significant code reuse among apps within those datasets. It should be noted though that this unique code does no imply malicious functionality, but rather indicates the code that is not commonly seen in legitimate apps.

VII. CODECHECK IN ACTION

To analyze the practicality of CodeCheck, we experimented with 10 legitimate and malware apps. The aim of this analysis is to demonstrate the potential benefits of using CodeCheck, for example in initial stages of analysis.

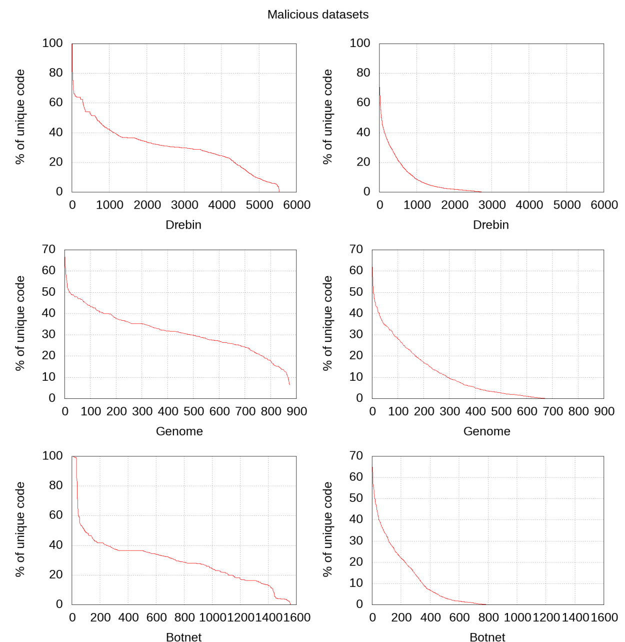

Fig. 5. The datasets from Android Malware Collection.

TABLE VIII
CODECHECK ANALYSIS OF BENIGN ANDROID APPS

App	Total # of methods	Baseline filter	Open Source	GooglePlay	Suspicious	Unique
Facebook app	2553	1235 (48%)	129 (5%)	213 (8%)	2 (0%)	974 (38%)
Viber app	38302	20882 (54%)	2005 (5%)	2698 (7%)	11 (0%)	12706 (33%)
Trivia crack app	34547	19472 (56%)	1762 (5%)	5102 (14%)	2 (0%)	8209 (23%)
Angry birds 2 app	41145	27545 (66%)	1894 (4%)	9435 (22%)	1 (0%)	2270 (5%)
Flixter app	49494	29061 (58%)	2276 (4%)	9055 (18%)	4 (0%)	9098 (18%)

Legitimate apps. The legitimate apps were retrieved from GooglePlay. The selected apps and their corresponding results are showed in Table VIII.

The following legitimate apps were analyzed:

Facebook app (com.facebook.katana) contains the majority of the unique code under "facebook" package, which is expected behavior.

Viber app (com.viber.voip) contains the majority of the code under "android" class. It contains a large portion (33%) of never seen before code that upon manual inspection turned out to be viber package and code from webrtc project.

Trivia crack app (com.etermax.preguntados.lite) includes unseen code from eight different advertisement libraries, four social media sites and a third part library for games "unity3d". The unclassified code is obfuscated with level 1 transformation.

Angry birds 2 app (com.rovio.baba) include six advertisement libraries, two paid libraries for games, "prime31" and "unity3d"; social media from three sites and a large amount of code under "android" package.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

Flixter app (net.flixster.android) with its 18% of unique code contains three ads libraries, code from five open source libraries, four social media sites and two analytical engines focused on digital media. The remaining 15% is obfuscated code.

TABLE IX
CODECHECK ANALYSIS OF MALWARE APPS

App	Total # of methods	Baseline filter	Open Source	GooglePlay	Suspicious	Unique code
Ransomware/Locker	102	3 (2 %)	0 (0 %)	0 (0 %)	0 (0 %)	99 (97 %)
HackingTeam	4310	1438 (33 %)	167 (3 %)	453 (10 %)	2 (0 %)	2250 (52 %)
ProxyTrojan	60	29 (48 %)	4 (6 %)	0 (0 %)	0 (0 %)	27 (45 %)
Ransomware	6919	5898 (85 %)	595 (8 %)	159 (2 %)	0 (0 %)	267 (3 %)
HijackRAT	3436	1520 (44 %)	238 (6 %)	1443 (41%)	0 (0 %)	235 (6 %)
Android Malware Collection	14104241	51.20%	9.44%	10.43%	N/A	5.83%

Malware apps. We obtained the five most recent Android malicious samples from ContagioMinidump blog [1] and used CodeCheck to investigate the uniqueness of their code. The apps were analyzed through each of the created filters. Unique code category refers to the amount of code that remained after application of baseline filter and occurred in less than 1% of apps. The results are showed in Table IX. For comparison, this Table also includes the numbers received for Android Malware Collection (these numbers are discussed in previous sections). The following malware samples were analyzed:

Android Ransomware/Locker (MD5:735B4E78B334F6B9EB19E700A4C30966) is a very small app. As we can see almost all the code found in the app is unique. Manual analysis of unique code revealed that it is obfuscated, and mostly contained within two classes.

HackingTeam (MD5:904ED531D0B3B1979F1FDA7A9504C882) is an Android sample that reused common code. The large portion of 52% of unique unseen code is located under "android/app" package, that hides app's real objective. Manual analysis of unique code revealed a presence of the lua interpreter (never seen in any other apps) and obfuscated code.

Android Proxy Trojan (MD5:D05D3F579295CD5018318072ADF3B83D) is another small malicious app. 48% of unique code is hidden under "android" package. No obfuscation was found in this sample.

Locker Ransomware (MD5:F836F5C6267F13BF9F6109A6B8D79175) is a large app, with only 3% of unique code, all of which is under "android" package.

AndroidOS Wroba / HijackRAT (MD5:A21FAB634DC788CDD462D506458AF1E4) is an app that contains only 6% of unique code, the majority of which is in "android/app" package. There is some obfuscation present in unique methods, but the container class and package name remains in plain text.

As opposed to legitimate apps, this set of malware seems to have slightly less SDK-related code. This number, however, is close to what we saw in apps from Android Malware Collection. What interesting here is the amount of unique code. Three out of five analyzed apps have significantly higher percentage of unique code, i.e., code that was not commonly seen in other apps before. Since CodeCheck extracts and labels known code focusing on this unique portion can expedite the analysis.

VIII. CONCLUSION

In this work we presented CodeCheck, a system that allows to analyze the amount and nature of code reuse in Android apps. We use CodeCheck to characterize code commonly shared among legitimate and malicious applications on a collection of over 60,000 Android apps. Our analysis revealed several aspects of code reuse that can be leveraged in developing better, more efficient and accurate malware detection approaches:

- Both benign and malicious apps contain over 80% of common code, thus relying on the presence of similarities between apps will not serve as a reliable predictor of apps repackaging.
- Unique code only represents 5% of all code in both legitimate and malware apps. Given this insight, focusing detection efforts on this portion of code can significantly reduce processing time allowing for more complex analysis.
- There is no distinction between the amount of advertisement SDKs used in benign and malicious apps. Both benign and malware data employed almost 3% of code attributed to various libraries.
- Although code obfuscation is often seen as a sign of malicious functionality, our results reveal that this practice is much more common across legitimate apps. Almost 41% of all analyzed methods from apps collected from GooglePlay market were obfuscated, compared to only 12% of methods derived from malicious apps.
- Our research is complementary to other approaches, where we could be helping to increase the performance and scalability of other systems, or explore new approaches using the composition of the apps.

To facilitate the following research in this area, we make the repositories generated in the course of this study together with the sets of unique code publicly available to a wider academic community.

REFERENCES

[1] Contagio mobile.

[2] Y. Aafer, W. Du, and H. Yin. DroidAPIMiner: Mining API Level Features for Robust Malware Detection in Android. In *Proceedings of the 9th SecureComm*, Sydney, Australia, September 25-27 2013.

[3] A. J. Alzahrani, N. Stakhanova, H. Gonzalez, and A. Ghorbani. Characterizing Evaluation Practices of Intrusion Detection Methods for Smartphones. *Journal of Cyber Security and Mobility*, 2014.

[4] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the 21th Annual NDSS*, 2014.

[5] V. Chebyshev and R. Unuchek. Mobile malware evolution: 2013, July 2014.

[6] J. Crussell, C. Gibler, and H. Chen. Scalable semantics-based detection of similar android applications. In *18th ESORICS*, Egham, U.K.

[7] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *17th ESORICS)*, volume 7459, pages 37–54. Springer, 2012.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.

[8] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on SPSM '11*, New York, NY, USA, 2011. ACM.

[9] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceedings of the 11th MobiSys*, Taipei, Taiwan, 2013.

[10] H. Gonzalez, A. A. Kadir, N. Stakhanova, A. J. Alzahrani, and A. A. Ghorbani. Exploring reverse engineering symptoms in android apps. In *Proceedings of the Eighth European Workshop on System Security*, page 7. ACM, 2015.

[11] H. Gonzalez, N. Stakhanova, and A. Ghorbani. Droidkin: Lightweight detection of android apps similarity. In *Proceedings of the 10th SECURECOMM*, 2014.

[12] M. C. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *The 10th MobiSys*, pages 281–294, 2012.

[13] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th DIMVA'12*, pages 62–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[14] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 309–320, New York, NY, USA, 2011. ACM.

[15] J. Jeong, D. Seo, C. Lee, J. Kwon, H. Lee, and J. Milburn. MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*, pages 50–57, Oct 2014.

[16] M. La Polla, F. Martinelli, and D. Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15, 2013.

[17] R. Potharaju, A. Newell, C. Nita-Rotaru, and X. Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 4th ESSoS'12*, pages 106–120, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] V. Rastogi, Y. Chen, and X. Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC*, pages 329–334. ACM, 2013.

[19] N. Viennot, E. Garcia, and J. Nieh. A measurement study of Google Play. In *Proceedings of SIGMETRICS 2014*, page 13, 2014.

[20] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 7th Asia JCIS*, pages 62–69, Aug 2012.

[21] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[22] F. Zhang, H. Huang, S. Zhu, D. Wu, and P. Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *7th ACM Conference on WiSec 2014*, Oxford, United Kingdom, 2014.

[23] M. Zheng, P. P. Lee, and J. C. Lui. Adam: an automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.

[24] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of "piggybacked" mobile applications. In *Proceedings of the CODASPY '13*, pages 185–196, New York, NY, USA, 2013. ACM.

[25] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the Second ACM CODASPY '12*, pages 317–326, New York, NY, USA, 2012. ACM.

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

[27] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *19th Annual NDSS*, 2012.

Authorized licensed use limited to: University of Saskatchewan. Downloaded on December 25,2021 at 20:03:46 UTC from IEEE Xplore. Restrictions apply.