GroupDroid: Automatically Grouping Mobile Malware by Extracting Code Similarities

Niccolò Marastoni
Università di Verona
niccolo.marastoni@univr.it

Andrea Continella
Politecnico di Milano
andrea.continella@polimi.it

Davide Quarta
Politecnico di Milano
davide.quarta@polimi.it

Stefano Zanero
Politecnico di Milano
stefano.zanero@polimi.it

Mila Dalla Preda
Università di Verona
mila.dallapreda@univr.it

ABSTRACT

As shown in previous work, malware authors often reuse portions of code in the development of their samples. Especially in the mobile scenario, there exists a phenomena, called *piggybacking*, that describes the act of embedding malicious code inside benign apps. In this paper, we leverage such observations to analyze mobile malware by looking at its similarities. In practice, we propose a novel approach that identifies and extracts code similarities in mobile apps. Our approach is based on static analysis and works by computing the Control Flow Graph of each method and encoding it in a feature vector used to measure similarities. We implemented our approach in a tool, GroupDroid, able to group mobile apps together according to their code similarities. Armed with GroupDroid, we then analyzed modern mobile malware samples. Our experiments show that GroupDroid is able to correctly and accurately distinguish different malware variants, and to provide useful and detailed information about the similar portions of malicious code.

CCS CONCEPTS

• **Security and privacy** → *Malware and its mitigation*;

KEYWORDS

Mobile, Malware, Similarity

ACM Reference Format:
Niccolò Marastoni, Andrea Continella, Davide Quarta, Stefano Zanero, and Mila Dalla Preda. 2017. GroupDroid: Automatically Grouping Mobile Malware by Extracting Code Similarities. In *SSPREW-7 : 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop , December 4–5, 2017, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3151137.3151138

1 INTRODUCTION

The great diffusion and usage of smartphones raised, in the last years, new security and privacy issues. In fact, these devices are particularly attractive for cybercriminals, since compromising them can be extremely lucrative, e.g., allowing attackers to defeat two factor authentication, or leaking sensitive information. As a consequence, we observed an increase in the development and diffusion of mobile malware, mainly targeting the Android platform. Android is nowadays the most spread mobile operating system. According to Statista [31], in the first quarter of 2017 87% of smartphone sales were devices running Android.

In 2016, Kaspersky Lab detected 8,526,221 malicious installation packages [21]. In particular, they observed a great increase in the diffusion of mobile ransomware, identifying 261,214 new mobile ransomware samples. Zhou et al. [39] studied the overall health of existing Android Markets, including both official and unofficial (third-party) ones, showing that most of malware is present in alternative markets. However, despite Google's effort (which uses a tool called Bouncer [28]), many malicious apps managed to avoid detection and were published on the Google Play Store [1].

Researchers studied this emerging phenomena and proposed solutions to mitigate security problems, such as malware detection [2, 5, 26], privacy leak detection [6, 11, 17, 29, 34], or mitigation of specific mobile attacks [18, 24].

From a different point of view, Lindorfer et al. [23] studied the evolution of Windows malware demonstrating that malware authors share their malicious code across different malware variants, and constantly update their samples to release new versions. Similarly, Android malware authors are known for leveraging techniques like repackaging and piggybacking, in which malicious code is embedded into benign apps [38].

Leveraging such observations, previous work proposed approaches to identify app similarities in order to perform malware classification [15, 35] or, for instance, detect vulnerable apps [20]. However, such approaches provide just a "black-box" classification, without providing any details as to why two or more apps belong to the same class. This has also been proved in [14], where authors showed that mobile malware detectors apply black-box signatures that do not reliably give insight about the malicious activity.

In this paper, we propose a novel technique to identify code similarities among Android apps, recognizing and extracting similar code that produces similar behaviors. Our approach is based on static analysis and works at the method level. Specifically, we extract the Control Flow Graph (CFG) of every method and encode each CFG in a vector of features that we use to measure the similarity. Chen et al. introduced the concept of 3D-CFG and its relative centroid to build a scalable method for app clone detection in [9],

SSPREW-7 , December 4–5, 2017, San Juan, PR, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5387-8/17/12…$15.00
https://doi.org/10.1145/3151137.3151138

focusing on applications that share most of the code, or at least the core functionality. One of their main motivations for working on clone detection is the fact that malware prefers to use app clones as “carriers” for propagation, while our focus is almost the opposite. We look for similarities in potentially small portions of malicious code, those that are deemed interesting for our particular analysis.

We implemented our approach in a tool called GroupDroid able to group Android malware samples on the basis of their code similarities and to extract the portions of similar code, providing useful and detailed feedback of the classification and helping in the reverse engineering process. We evaluated GroupDroid against 4,211 malicious Android apps, showing that it is able to successfully identify different families. Our experiments showed that GroupDroid is not only able to group together malware samples of the same family, but it can also distinguish slightly different variants by identifying differences in the similarities. An example of this is clearly visible in Figure 1, where the methods only differ in their names, package names and some specific strings (like *url* that is transformed in *erwgerwg*).

In summary, we make the following contributions:

- We propose a static analysis-based approach to identify syntactic similarities between Android apps and extract portions of code that produce similar behaviors.
- We implement our approach in a tool, GroupDroid, that is able to find similarities in malicious Android apps and group them together according to such similarities.
- We evaluated GroupDroid on a dataset of 4,211 Android malware samples. Then, we present some case studies that show how mobile malware authors share and reuse their malicious code across different variants.

2 BACKGROUND AND MOTIVATION

Malware lives in a complex ecosystem that, similarly to an industry environment, includes malware developers, managers, maintenance, and business strategies. This ecosystem is, of course, stimulated by the financial incentives that revolve around it. Common trends in the mobile scenario include: stealing and selling user information, stealing user credentials, premium-rate calls, SMS spam, ransoms, advertising click frauds, and in-app billing frauds. Armin [4] studied the mobile underground market finding an alive and thriving ecosystem that benefits from the existence of an established modus operandi for desktop malware, which is well-structured and successful. Such a market is based on a crime-as-a-service model, in which resources, such as customizable malware, are sold and rented online. For instance, a Trojan called “Exo Android Bot” was heavily advertised in forums in 2016. For $400 per week or $3,000 per year the author promised Android malware that could intercept SMS, use screen overlays, and had 24/7 support [33]. Unfortunately, sometimes malicious apps manage to evade detection and appear on official stores. For example, in February 2017, an `Android.Fakebank.B` variant masked as a weather app called “Good Weather” was published on the official Google Play Store and was downloaded by approximately 5,000 users [33].

Motivation: One of the distinctive aspects of malware, especially on mobile, is its evident need to fit in among legitimate apps, to entice the user and spread faster. Since its appearance needs to resemble goodware, a big part of the app is dedicated to behaviors that aren’t malicious at all, and often the benevolent part of the app varies in different samples of the same malware family. A malware family is thus defined by the only component that is maintained constant among every sample: the malicious payload (Figure 2). This phenomena, known as *piggybacking*, in which cybercriminals embed malicious code into benign apps, has been observed and studied by researchers in previous works [22, 36, 37]. Moreover, as we previously described, malware authors re-use (part of) their malicious code across different malware versions and variants, which are constantly released on the underground market.

On the base of such observations, if we find a good and efficient way to analyze code similarities between many apps, we can potentially isolate the malicious behaviors shared among the different malware samples, and group apps together according to these similarities.

Previous works proposed approaches to classifying apps by looking at their similarities [15, 20, 35]. However all these approaches mainly provide black-box tools, which do not generate detailed information about the behaviors the apps share. Instead, we aim at proposing a novel approach that allows to obtain practical and detailed feedback of the classification. In practice, our approach identifies and extracts the similar portions of code that cause two or more apps to be grouped together.

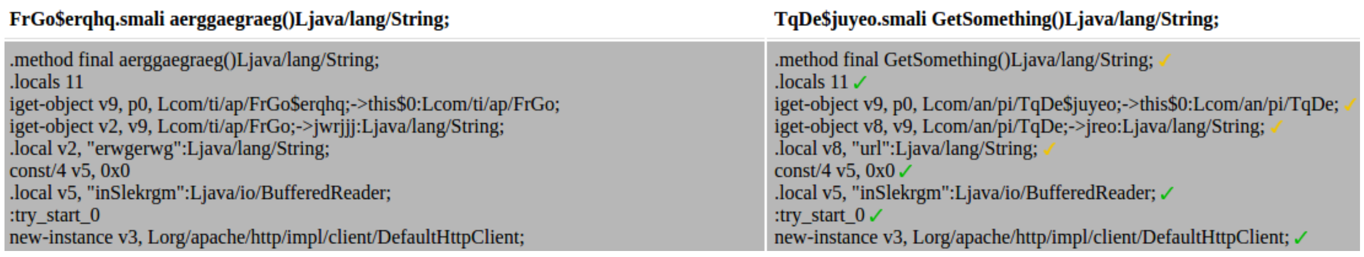
3 APPROACH AND METHODOLOGY

GroupDroid at its core takes in input smali files from Android apps and filters the individual methods according to some fixed thresholds that allow us to be selective in regards to the behaviours we want to observe. Then the code is parsed to extract some static features, 3D-CFG centroids and API vectors, which will then be used to compare apps at the method level, to check for similarities. When GroupDroid completes all the comparisons, it then groups the apps together according to how much of the code they share. In the next sections, we describe the phases of the approach of GroupDroid, following the workflow depicted in Figure 3.

3.1 Filter

As a first step of our approach, apps are unpacked using apktool and the smali files are parsed to extract methods. The filtering phase considers some thresholds that are variable and can be tuned in the *settings* section.
The first one is the *min_method_size*, which counts the minimum number of instructions in a method for it to be extracted. This tells the parser to ignore those methods that do not contain enough statements.
The filter can be adjusted at each analysis, but it’s usually set at 6 to weed out some methods that are prevalent in every Android app, mainly *init*() type of methods. These do not have any control flow information but they usually just initiate a couple of variables and invoke one additional method, so they become the main source for false positives and usually make the results of our analysis less interesting.

The second filter checks if a method invokes any of the *Risky APIs*. If it does not, we do not consider it since it cannot possibly have any interesting behavior. *Risky APIs* is just a collection of all the



FrGo$erqhq.smali aerggaegraeg()Ljava/lang/String;	TqDe$juyeo.smali GetSomething()Ljava/lang/String;
.method final aerggaegraeg()Ljava/lang/String;	.method final GetSomething()Ljava/lang/String; ✓
.locals 11	.locals 11 ✓
iget-object v9, p0, Lcom/ti/ap/FrGo$erqhq;->this$0:Lcom/ti/ap/FrGo;	iget-object v9, p0, Lcom/an/pi/TqDe$juyeo;->this$0:Lcom/an/pi/TqDe; ✓
iget-object v2, v9, Lcom/ti/ap/FrGo;->jwrjjj:Ljava/lang/String;	iget-object v8, v9, Lcom/an/pi/TqDe;->jreo:Ljava/lang/String; ✓
.local v2, "erwgerwg":Ljava/lang/String;	.local v8, "url":Ljava/lang/String; ✓
const/4 v5, 0x0	const/4 v5, 0x0 ✓
.local v5, "inSlekrgm":Ljava/io/BufferedReader;	.local v5, "inSlekrgm":Ljava/io/BufferedReader; ✓
:try_start_0	:try_start_0 ✓
new-instance v3, Lorg/apache/http/impl/client/DefaultHttpClient;	new-instance v3, Lorg/apache/http/impl/client/DefaultHttpClient; ✓

Figure 1: An example of GroupDroid's main output

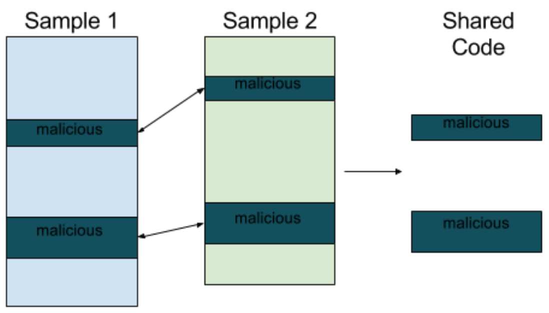


Figure 2: Isolating Malicious Behaviors

APIs offered to Android developers to interact with the phone, and they range wildly between APIs that allow apps to write SMS and others that grant access to the phone's filesystem. With the way the Android OS is structured, apps just do not have any way to do anything harmful without using these APIs.

The API filter also allows us to choose which APIs to focus on during analysis, making it easier to spot partial similarities in different samples. We collected the *Risky APIs* from various sources on the net and divided them in classes for different types of malware. For example, spyware samples will usually use some of these APIs:

```
android.telephony.TelephonyManager
android.provider.Contacts
java.net.ServerSocket
org.apache.http.impl.client.DefaultHttpClient
```

3.2 Feature Extraction

We consider similarity at the method level, so we can encode every method and then compare it to others.
Since our similarity computation needs to be both fast and resilient to code transformations (to a certain degree), we decided to first consider the structure of the code. We extract the CFG of every method and encode each CFG in a vector of features that we use for the final similarity measure.

The first 4 features are the 3D-CFG's weighted centroid of the method, as first defined in [9].
The idea behind 3D-CFG centroids is borrowed directly from physics, to be more specific it's a reinterpretation of the *center of mass* of an ensemble of rigid bodies.

3D-CFG. Each method in a sample app gets transformed in its CFG, which represents the rigid structure, and every node (basic block) in the CFG is treated like an object with mass, connected to the other objects via weightless sticks (the edges of the CFG).
At this point an *end* node will be added, and every basic block with a return statement will naturally flow into it, providing a single exit node that will have its weight set to 0.

Before assigning any mass to the nodes of the CFG, this needs to get transformed even further to make it fit into a 3D space.
First, each node p needs to have $< x_p, y_p, z_p, >$ coordinates, where:

- x_p = sequence number
  The choice of the sequence number depends on the order in which every basic block in the CFG will be executed. This is not particularly easy to judge with static analysis, but the goal of the 3D-CFG conversion is to provide a 1-to-1 conversion between code and 3D-CFG, thus it can rely on simple heuristics that will assure the same conversion every time copies of the same method will be fed to the algorithm. If a branch node has sequence number n, the first node in the branch with more nodes will have n + 1.
- y_p = number of outgoing edges
- z_p = loop depth
  The nesting level of the basic block.

3D-CFG centroids. Once the basic blocks are successfully converted into nodes in the CFG, every node p is then connected to its successor q via an edge $e(p, q)$ according to the original structure of the CFG.
The weight $\overline{w_p}$ of every node p is given by the number of statements (instructions) in it. We count every instruction apart from `nop`, since we don't want to be fooled by simple padding.

A centroid is a vector $< c_x, c_y, c_z, \overline{w} >$ where:

$$c_x = \frac{\sum_{e(p,q)\in 3D-CFG} (\overline{w_p}x_p + \overline{w_q}x_q)}{\overline{w}} \tag{1}$$

$$c_y = \frac{\sum_{e(p,q)\in 3D-CFG} (\overline{w_p}y_p + \overline{w_q}y_q)}{\overline{w}} \tag{2}$$

$$c_z = \frac{\sum_{e(p,q)\in 3D-CFG} (\overline{w_p}z_p + \overline{w_q}z_q)}{\overline{w}} \tag{3}$$

$$\overline{w} = \sum_{e(p,q)\in 3D-CFG} (\overline{w_p} + \overline{w_q}) \tag{4}$$

Modified Centroid. Because of the nature of methods in Android apps and their reliance on invocations to the framework APIs, a

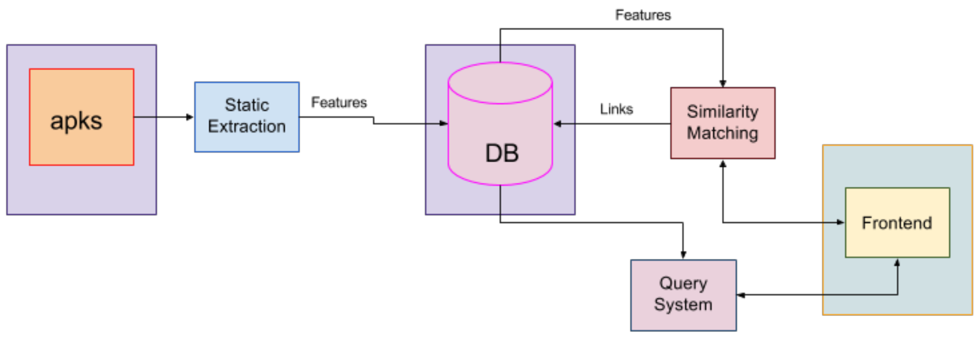


Figure 3: Workflow of GroupDroid.

second type of centroid is introduced in [27], where the weights of each node are the sum of the number of statements and the number of invocations found in the basic block.
This allows for better distinction between possibly cloned methods, enhancing the underlying differences, and does not impact the complexity of the calculation (both centroids can be calculated at the same time with no added overhead).

Hence, $\overline{w}' = \overline{w} +$ # of invocations, and c' will be our new weighted centroid, with the same process applied above.

Centroid Difference Degree. At this point the methods of our Android apps have been reduced to simple 3D vectors (technically 4D, since it's 3 dimensions plus the weight) so now we can define a simple distance measure between the centroids.

$$\mathbf{CDD}(\overrightarrow{c_1}, \overrightarrow{c_2}) = max(\frac{|c_{1_x} - c_{2_x}|}{c_{1_x} + c_{2_x}}, \frac{|c_{1_y} - c_{2_y}|}{c_{1_y} + c_{2_y}}, \frac{|c_{1_z} - c_{2_z}|}{c_{1_z} + c_{2_z}}, \frac{|\overline{w_1} - \overline{w_2}|}{\overline{w_1} + \overline{w_2}}) \tag{5}$$

The first 4 features are the modified centroid, and as we experimented with it, we found out it has way better performances than the normal one or a combination of the two.

The 3D-CFG centroid is a structural feature, and not a very precise one (since the centroid only keeps part of the structural information of the CFG [9]), so it forgets a lot of what happens in the methods. This, coupled with android method's tendency to be rather simple, gives rise to a lot of false positives.
In the original paper, the authors added a vector of statement types (couting how many different types of statements were found, declarations, assignments, invocations etc.) to face this problem, but after extensive testing with our datasets it became clear that it did not solve most of the false positives, if any.

+Our proposed solution is to look at the APIs that are called during the method run: if two pieces of code are similar, they need to exhibit some of the same behaviors. To check this, we extract the API invocations during the first parse of the smali code, annotating every method with a 5th feature.

API Vectors. This 5th feature in our vector is a number between 0 and $(2^{23} - 1)$, it's a binary encoding of the APIs (*APIBool*). We isolated 23 *Risky APIs* mostly from literature and our own experience, and stored them in a list. Every time a method invokes an API in the list, the 5th feature (which is initialized to 0) will be added to 2^a (where a is the index of the API). So an invocation to the first API will add 1 (2^0), to the 4th will add 8 (2^3) and so on. This is done mostly to provide a new feature that does not slow down the extraction process any further and that still allows us to check in O(1) the new property.

During the extraction of the 5th feature we also save a vector (*APIVector*) where each index represents how many times a given API is invoked in the method's body. Example: if we only consider 4 APIs, NFC, TELEPHONY, NET and CRYPTO and the methods makes 3 calls to methods related to NET and one to a method related to TELEPHONY, then the *APIVector* will be [0, 1, 2, 0].

This is our last group of features, at this time it's a vector of up to 23 elements with each element being an integer in the range 0+, and it will be used eventually in the very last step of our similarity function. The vector can of course be smaller if fewer APIs are selected during the filtering phase.

3.3 Code Similarity

This is the algorithm for our code similarity measure, given 2 methods m_1 and m_2:
$s(m_1, m_2) = \mathbf{CDD}(m_1.centroid, m_2.centroid) < cThreshold$
$\wedge\ \mathbf{BVE}(m_1.APIBool, m_2.APIBool)$
$\wedge\ \mathbf{VDD}(m_1.APIVector, m_2.APIVector) < vThreshold$

Where **CDD** (the Centroid Difference Degree) is a very fast operation that lets us filter methods that share at least some common structure. It returns a float value between 0 and 1.0, where 0 means complete similarity and 1.0 no similarity at all (it's a weighted distance). The threshold for the final similarity can be changed at will, but it's usually set to 0.4. This value is the result of many experiments and lets the search space reduction algorithm work (it would become ineffective at 1.0) while allowing structural dissimilarities introduced by code transformations. The next steps of the algorithm will eventually catch any discrepancies produced by this lax approach.

BVE is the Boolean Vector Evaluation:

$$\mathbf{BVE}(bv_1, bv_2) = bv_1 \& bv_2 \tag{6}$$

This is an $O(1)$ function, incredibly simple and designed to act as a rough filter to avoid analyzing methods that do not share any API invocation. As previously explained, our 5th feature is a number

between 0 and $2^{23}-1$ that is mined without any additional overhead during the parsing of the method and it encodes succinctly which APIs are called in the body of the method.
Consider this practical example with methods q.run() and i.run(), extracted from the debug output of GroupDroid:

```
FAILED API CHECK:
(4.384615384615385, 1.3205128205128205,
0.08974358974358974, 78)
32, [0, 0, 0, 0, 0, 3], 'q.smali', 'run()V'
AND
(3.375, 1.5416666666666667, 0.16666666666666666, 72)
18, [0, 1, 0, 0, 2, 0], 'i.smali', 'run()V'
MDD = 0.168574812475
```

The first 4 numbers between parenthesis are the 3D-CFG centroid of the method, while the first number in the next line is our 5th feature. Methods $q.run()$ and $i.run()$ passed the **CDD** test, meaning their structure is fairly similar, or the threshold was set too high (in fact it was set to 0.2). Method $q.run()$ calls an API that is located at position 5 in the risky APIs vector, so its binary feature is the number 100000 ($1 * 2^5 = 32$), while method $i.run()$ calls two APIs that can be found at position 1 and 4, producing the binary feature 010010 ($1 * 2^4 + 1 * 2^1 = 18$).
At this point the **BVE** function simply applies a bitwise AND operator to the binary features and discovers that they do not share any API call ($32\&18 = 0$), meaning that the structural similarity resulted in a false positive. Hence, we can declare the methods not similar immediately without further computations.

If the **BVE** function returns a TRUE value (which is any non-zero number), then the last function is applied to the remaining features. The **VDD** is the API Vector Distance Degree:
$av_1 = [v_{1,1}...v_{1,23}]$
$av_2 = [v_{2,1}...v_{2,23}]$

$$VVD(av_1, av_2) = max\left\{\frac{|v_{1,i} - v_{2,i}|}{|v_{1,i} + v_{2,i}|} | i \in [0, 22]\right\} \tag{7}$$

This is very similar to the **CDD** function and again outputs a value between 0 and 1.0, where 0 is an exact match between the API vectors, while 1.0 this time means that at least one of the elements in one vector did not have a match in the other. We use this function to allow for future relaxing of the API vector threshold, but for now it's set at a firm 0 (meaning we want an exact match all the time).

If all 3 the functions return True then the two methods are deemed similar.

3.4 App Similarity

The app similarity score is calculated as the ratio between the number of shared methods and the size of the first app, calculated as number of methods.
The reason the similarity is asymmetrical is to account for the difference in size between different apps, if an app shares 20% of its code with another, it's not always a given that the second one shares 20% of its code with the first.

$$score(app_1, app_2) = \frac{|\{(m_1, m_2)|m_1 == m_2 \wedge m_1 \in app_1 \wedge m_2 \in app_2\}|}{|\{m_1|m_1 \in app_1\}|} \tag{8}$$

Two apps are considered not similar at all when their score is exactly 0, which means that they share no common method. Two apps are considered equal when their score is 1, and that only happens when every method from the first app has a match in the other app.

```
for sample ∈ analyzed_apps do
        for group ∈ groupset do
                score ⟵ similarity_score(group, sample)
                if score > threshold then
                        group ⟵ group∪sample
                else
                        new group
                        group ⟵ sample
                        groupset ⟵ groupset∪group
                endif
        endfor
endfor
```

Figure 4: Grouping algorithm.

3.5 Grouping

The grouping phase of GroupDroid consists of a general clustering algorithm that takes into account the app similarity score as a distance measure.
It starts by iterating on each analyzed app, creating a cluster for the first one and then adding to it all other apps that have a good similarity score, taking into account the asymmetry of the score we always check for the best one of the pair. The similarity score threshold is one of the parameters that we can play with and it's usually set at 1.0 if we only want to consider groups of apps that share 100% of the code we care about. This particular value has proved to be very valuable in our analysis, as it gives more concise results when few APIs are filtered, but it can also severely impair the grouping accuracy when we are analyzing for more APIs. For example we found the samples in case study 9 after lowering the grouping threshold and allowing the samples to share only part of the code.

Figure 4 shows the pseudocode of our grouping algorithm.

4 SYSTEM DETAILS

4.1 Search Space Reduction

To analyze an entire dataset, GroupDroid needs to encode every method of every sample and compare it to the others. Numerically, this means that if the dataset contains n methods, there will be $O(n^2)$ comparisons, which becomes a problem very fast when analyzing big datasets.

For example, the GENOME dataset (which isn't particularly big) has more than 1,000 apps, and each one of them has up to 3,000 methods in it (mostly some members of the DroidKungFu family), which totals at about $3*10^6$ methods with $9*10^{12}$ comparisons. Our algorithm can compute 10^5 comparisons per second on average, so that would take close to $9 * 10^7$ seconds (slightly less than 3 years). Of course this is not an acceptable running time for any kind of analysis, thus it became necessary to restrict the search space.

Since the first step of the algorithm considers structural similarity between methods by converting the CFG into a 3-dimensional vector (4-dimensional if we consider the weight), restricting the search space literally means that we may consider only the regions of the 4-dimensional space that contain centroids closer to the one in input.

The first step of the dimensionality reduction occurs during the preliminary phase, when every method is coded in its centroid, GroupDroid updates a nested dictionary-like data structure that will act as a hash-table to allow for fast searching.
The dictionary is thus updated:

```
c = (x, y, z, w)

update_centroid_dict(Centroid c):
dict[floor(x)][floor(y)][floor(z)][w].append(c)
```

This way a centroid c with the coordinates `[1.342,3.45,8.01,12]` will be added to a list of other centroids in `dict[1][3][8][12]`.

The algorithm to search for a matching centroid now is pretty straightforward, we just need to calculate a valid range of coordinates to check, and then look into their respective lists. The range of coordinates is calculated using the CDD function. For example, given the previous centroid c in input and the standard thresholds for GroupDroid, its matching centroids will be searched in the following ranges: $x = (1, 1), y = (2, 4), z = (6, 10), w = (9, 15)$. This gives us 48 possible lists of centroids, with most of them realistically being empty.

For a more practical example we'll run our algorithm on the first method of the first sample in the GENOME dataset: 86 actual MDD checks and only 9 API checks, in a dataset with about 70k methods. Normally checking every single method in the dataset against the others would take more than 13 hours, this way it could take 18 seconds, assuming that the search space was equally distributed (a pretty bold and unrealistic assumption).

Since there isn't any theoretical reason why methods should stack up in a particular spot, we ran some tests and the worst performance by far was in a method that had to be checked against 290 other methods. With this experimental worst case in mind (assuming that every single one of the methods had to find 290 suitable doubles), the run time of our algorithm would still take a little more than 3 minutes. Compared to 13 hours it's still a pretty big improvement.

4.2 Implementation

We implemented GroupDroid in about 3K lines of Python as a web application, to make it easier to deploy on remote servers. What follows is a simple description of the implementation concerning its core elements.

For the server side aspects of GroupDroid we used the web framework Flask, which allows for faster initial development and launch on a local machine. Different technologies will be considered for an eventual future production deployment.

The system works by leveraging apktool to extract the apps and translate the dex files into smali, a human readable format that is produced by the disassembler baksmali. Then a simple parser scans through the smali files and generates the features needed for the similarity measure. Each method is transformed into its respective CFG (as better explained in Section 3.2) and saved in a dictionary that will remain in RAM for the duration of the analysis. We used to store everything on disk and then call it back whenever needed but it created unnecessary bottlenecks, now the only disk I/O operations happen on server start up and when the analysis is over. This means that the dictionary containing the features and all the reports are saved on disk in plain text using the Python library Pickle, so that GroupDroid can reload them in memory at each startup.

5 EXPERIMENTAL EVALUATION

In the next sections, we describe our datasets and the experiments we performed to evaluate GroupDroid. More precisely, we first provide the results of the analysis we performed using GroupDroid. To evaluate the grouping accuracy, we leveraged AVClass [30], a malware labeling tool that determines the most likely family of a given sample by clustering the AV labels obtained through VirusTotal. Then, we present some interesting case studies that show how malware samples reuse malicious code. Finally, we assess the runtime performance of our tool.

5.1 Datasets

We used four different datasets for our evaluation. First, we got access to 675 ransomware samples from the Heldroid [3] dataset (**Dataset_1**). Then, between July and August 2017, we used the VirusTotal Intelligence API to obtain two datasets: (1) the 500 most recent Android ransomware labeled as ransomware by at least 5 AVs (**Dataset_2**); (2) the 1,000 most recent generic malware labeled as malicious by at least 5 AVs (**Dataset_3**). Finally, we got access to data from AndroTotal [25]. Specifically, we obtained 2,036 apps labeled as malicious by at least 25% of the AVs in AndroTotal (**Dataset_4**).

In summary, our datasets total 4,211 malicious apps.

5.2 Grouping Results

Dataset_1 + Dataset_2. For this study we analyzed a dataset of 1,175 ransomware. The goal of the analysis was to group these samples by highlighting their shared code.

The biggest group in the dataset turned out to be 242 samples, all of them sharing the same methods to encrypt and decrypt files. This group goes well beyond code reuse, as all of them have the same structure and contain only one package with always exactly 22 smali files. The only difference is in the package name and in the file names, as all of the samples seem to contain different permutations of random strings.

The methods isolated as most relevant, using the API filter function, have been the `encrypt()`, `decrypt()` and `init()`, which contain all the code necessary to perform the main action of ransomware.

The second group has 94 members and is another case of extreme code reuse, where most of the samples contain exactly the same files, this time even with the same names and with only a couple different packages for most of it. GroupDroid was able to identify code reuse even among those few samples that were not exactly cloned, and again the only transformation done to the samples were different names for the packages and the smali files, but this time changing the whole structure of the application and adding a lot to it.
Table 1 shows how many different labels were applied to this particular group by AVclass.

All in all GroupDroid helped isolate at least 18 groups of apps in the dataset that share their core code. In Table 2, we show 12 interesting ones and their relationship with AVClass classes. From top to bottom it's easy to see that what AVClass labels as **koler** or **locker** are actually 9 different groups of applications, which probably exhibit the same behaviour (theyre all ransomware afterall) but do not really share enough code to be considered similar by GroupDroid. For example, all the samples in G1 contain the methods `init()` `encrypt()` and `decrypt()` and they try to hide them in randomly named smali files.

Another interesting observation can be done by looking at the table from left to right on G11 and G11, the families **simplocker** and **slocker** could be easily merged, and it could explain the similarity of their name. The same observation can be made by the samples in the families **svpeng** and **crosate**, that were both grouped together by GroupDroid on G3. Manual analysis has confirmed that indeed all these samples belong together.

This means that GroupDroid gave a more accurate representation of the dataset as a whole, giving more detailed granularity to the generic labels **locker** and **koler** while also grouping together unnecessary labels. This is simpler to see when looking at the graphs in Figure 5 and 6.

One of the best results was given by the second biggest group isolated by GroupDroid, all of which was comprised of samples of ransomware downloaded between july and August 2017. All 94 of them were correctly grouped together by GroupDroid (they're almost identical clones) but AVClass gives 5 different labels (**jisut, slocker, congur, lockscreen, locker**). It's very likely that labeling recently found malware is less consistent among AVs, as we can see the opposite trend during our evaluation of the tool with the GENOME dataset. Since the dataset has been around for so long the AVs have reached a consensus and they are able to consistently choose the right labels for every sample contained.

C	congur	jisut	lockscreen	slocker	locker
G1	25	42	7	19	1

Table 1: AVclass classification of group 2 in Dataset_1 + Dataset_2

C	koler	locker	simplocker	slocker	crosate	svpeng
G1	242	0	0	0	0	0
G2	69	0	0	0	0	0
G3	0	0	0	0	22	39
G4	40	0	0	0	0	0
G5	0	34	0	2	0	0
G6	27	0	0	0	0	0
G7	0	25	0	0	0	0
G8	0	23	0	0	0	0
G9	0	22	0	0	0	0
G10	0	0	18	2	0	0
G11	0	0	7	13	0	0
G12	0	18	0	2	0	0

Table 2: Classification of Dataset_1 + Dataset_2. On the x axis we have the classification by AVClass, while GroupDroid's grouping is on the y axis.

Dataset_3. Another dataset we worked on was downloaded from VirusTotal and comprises 1000 samples. GroupDroid identified about 20 unique groups when analyzing for TELEPHONY related behaviours.
In Table 3, we highlight only some of the most interesting groups. Looking at this table it's clear that the family **SmsReg** is actually divided in 4 groups and the families **SmsThief** and **smsforw** contain 2 distinct groups each, again making a case for a very weak ground truth. There is also a very faint hint at the fact that **SmsThief** and **SmsSpy** could be merged, but it's much clearer when changing the filter a little, looking at cryptographic APIs, in Table 4. The most interesting result from this analysis comes from group 2, where 8 samples were classified by AVClass as **lockscreen** and 20 as **singleton**. This means that AVClass did not have a label for them, so they are 20 unseen samples that have been correctly grouped together by GroupDroid as members of the family **lockscreen**.
Also a good indication that the previous classification for G2 (all members of the family **SmsReg**, in table 3) was correct, we now find the same exact samples in G3.

Dataset_4. Our last analysis is a dataset consisting of 1,790 samples, all downloaded from AndroTotal.
Here are some interesting groups of malware with telephony-related behaviours: The classification in this case seems to be much more consistent between GroupDroid and AVClass and the reason could be that these families have been around for a while, as they appear in the original GENOME dataset (apart from **Sandr**) so they have been thoroughly studied and analized.
It's also interesting to note that GroupDroid divided in 2 different groups what AVClass thought was only the family **DroidDream**, so we looked into the samples themselves, being aided by GroupDroid's handy method filtering tool. We noticed that, while all of the samples in the 2 groups mined the IMEI, IMSI and device ID from the unsuspecting user, they went about it in a completely different way: the samples in G4 divided each action in 3 different methods (Figure 7), while those in G5 did it in a single method

C	SmsThief	SmsReg	SmsSpy	smsforw	smsPay
G1	25	0	1	0	0
G2	0	22	0	0	1
G3	0	18	0	0	1
G4	0	4	0	0	2
G5	0	8	0	0	0
G6	0	0	0	9	0
G7	7	0	0	0	0
G8	0	0	0	0	0
G9	0	0	0	6	0

Table 3: Classification of Dataset_3 (AVClass on the x axis, GroupDroid on the y axis).

C	SmsThief	SmsReg	SmsSpy	lockscreen	singleton
G1	32	0	11	0	0
G2	0	0	0	8	20
G3	0	22	0	0	0

Table 4: Dataset_3 with Crypto APIs (AVClass on the x axis, GroupDroid on the y axis).

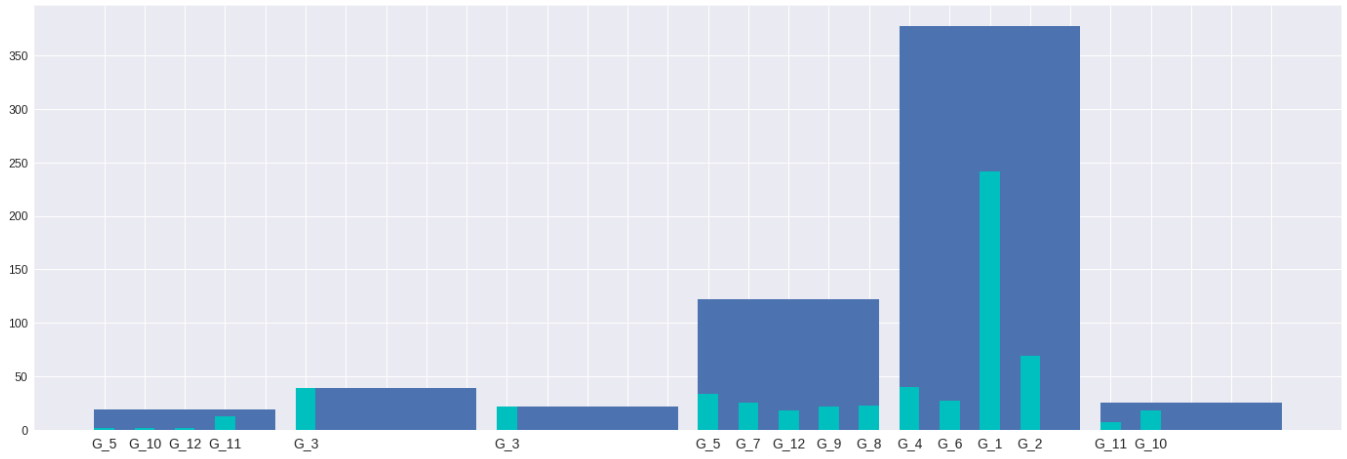


Figure 5: AVClass labels (darker shade) are more generic, GroupDroid helps refine the results by subdividing them.

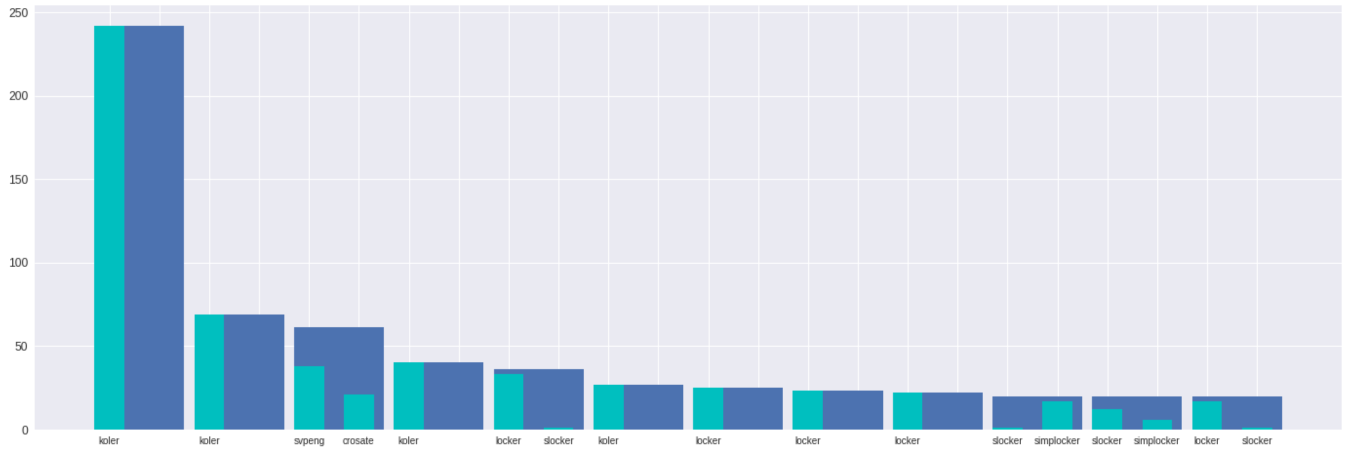


Figure 6: GroupDroid helps grouping together samples that were divided by AVClass.

C	BaseBridge	BeanBot	SMSpy	DroidDream	Sandr
G1	65	0	0	0	0
G2	0	49	0	0	0
G3	0	0	37	0	0
G4	0	0	0	36	0
G5	0	0	0	33	0
G6	0	0	0	0	31

Table 5: Classification of Dataset_4 (AVClass on the x axis, GroupDroid on the y axis).

(Figure 8).

But this is not the only difference, the samples in G4 can send SMS messages to steal the user's IDs, while those in G5 do not have this capability and just send everything through the net.
So in a way all 69 samples have some consistent behaviours, but they implement them in a much different way.

5.3 Grouping Accuracy

GroupDroid was first tested using the very well known GENOME dataset [38]. It contains 1200 malware samples of Android malware, categorized in 49 families, all of which were collected between 2010 and 2011.

This dataset has already been the subject of many studies, all of which have expanded the information gathered on the families and addressed the various problems that can be solved by classifying malware correctly ([2], [35], [7]). Since the dataset is already divided into malware families and it has a huge corpora of studies on it, this means that GroupDroid's accuracy can be assessed easily.

The fact that the samples in this dataset have been gathered in 2010 and 2011 implies that they should not perfectly reflect modern malware, but the dataset overall seems like a suitable candidate for initial testing.

The dataset contains 49 families, but 16 of these only contain one sample each. Since GroupDroid can group samples together with regard to their similarity, these 16 families cannot possibly be

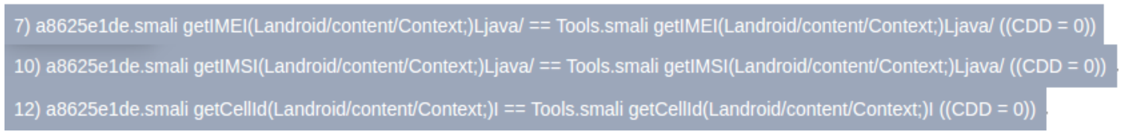

```
7) a8625e1de.smali getIMEI(Landroid/content/Context;)Ljava/ == Tools.smali getIMEI(Landroid/content/Context;)Ljava/ ((CDD = 0))
10) a8625e1de.smali getIMSI(Landroid/content/Context;)Ljava/ == Tools.smali getIMSI(Landroid/content/Context;)Ljava/ ((CDD = 0))
12) a8625e1de.smali getCellId(Landroid/content/Context;)I == Tools.smali getCellId(Landroid/content/Context;)I ((CDD = 0))
```

Figure 7: Method filtering tool, showing methods that extract IMEI, IMSI and ID in G4

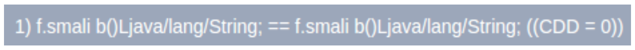

```
1) f.smali b()Ljava/lang/String; == f.smali b()Ljava/lang/String; ((CDD = 0))
```

Figure 8: Method filtering tool, showing a method that extracts IMEI, IMSI and ID in G5

	Precision	Recall	F1_S
CCD	0.6606	0.8909	0.7587
CCD + BVE	0.7008	0.9022	0.7889
CCD + BVE + VDD	0.7511	0.9021	0.8197

Table 6: Grouping accuracy for the GENOME dataset.

analyzed and so were taken out of the dataset.
The rest of the dataset was processed and we checked manually to see if the labels were consistent. The results for the remaining 33 families can be seen in Table 6, where:

$$Precision = \frac{TP}{TP + FP} \tag{9}$$

$$Recall = \frac{TP}{TP + FN} \tag{10}$$

$$F1\_Score = \frac{2 * TP}{(2 * TP) + FP + FN} \tag{11}$$

TP represents the True Positives, counting how many pairs of apks belonging in the same family were in fact grouped together, FP are the False Positives, pairs of apks that don't belong to the same family but were put in the same group and finally FN are the False Negatives, representing the number of pairs of apks that should have been grouped according to the ground truth but weren't.
The F1 score was chosen to ensure an equal weighting of the False Positives and False Negatives. In the end, our tool achieved an F1 score of ~82%.
All results are averaged throughout every family and precision is skewed by the fact that the dataset contains 5 variants of DroidKungfu that often enough are grouped together (thus inflating False Positive rates), while recall does not change that much because the false negatives are pretty consistent.
It's easy to see that accuracy of the grouping is improved by the addition of the BVE and VDD formulas to check for API vector similarity.

5.4 Case Studies

Most of the code reuse we found only involve simple app cloning, sometimes with package and filename renaming, probably to avoid detection by simpler tools.
Here instead, we describe some of our most interesting findings, which are inter-group code reuse.

Code Removal. The group 2 of **Dataset_1 + Dataset_2** contains 94 samples and is one of the biggest ones. All its members share the methods `encrypt()`, `decrypt()`, `getKey()`, `getMD5string()`, and `init()` that are deemed as interesting by GroupDroid when filtering for cryptographic methods. However, when the analysis gets extended to other functions (such as display functions), there's another group that shares a lot of code with samples from group 2. In fact they share not only the code, but most of the application's structure (as shown in Figure 9). The reason they do not belong in the same group according to GroupDroid is that they're missing all the classes that deal with encryption, making every sample in the group more like a "scareware" as they lack the ability to do any harm to the filesystem. This highlights one of the challenges when trying to classify entire applications with regards to their code similarity, as we need to manually set how much shared code constitutes similarity to begin with.

Code Transformations. A lot of the samples encountered in our analysis are clones, for example all members of group 2 in **Dataset_1 + Dataset_2** share the same exact code and file structure, apart from 7 samples that are just a chinese variant of the malware (they still share the core code, but with a changed file structure of the app and additional classes). Other families change some basic properties to avoid detection, for example group 1 in **Dataset_1 + Dataset_2** keeps all the code and method names intact, while every file name changes between different samples.

The most interesting ones try to avoid detection by obfuscating everything, from the file structure to their code, as is the case of group 3 in **Dataset_4**, where every smali file is littered with tens of methods (with encrypted names) that do not do anything but call each other, thus obfuscating the CFG of the whole app. Control Flow obfuscation is also applied to the individual methods, making it a challenge for most similarity techniques. Despite this, GroupDroid was able to successfully group together these apps thanks to its relaxed CDD threshold (which allows greater control flow manipulation) and to the use of the API vector checks.

5.5 Performance

The performance of GroupDroid varies wildly in relation to the size of the dataset and the size of the samples in the dataset. We made an empiric study, adding few apks at a time from the GENOME dataset and timing how much it took for the system to analyze them. Figure 10 details how much GroupDroid scales in comparison to a thorough pairwise comparison (which would be quadratic) and both linear and n log(n) functions (ideals) in the worst case.
The total run of the similarity check for 38.149 methods was around 80 seconds, while it would have taken more than 800 (circa 14 minutes) without search space reduction.

```
smali
    com
        sssp
            BAH.smali
            R$string.smali
            R$xml.smali
            R$id.smali
            R$drawable.smali
            bbb.smali
            R$attr.smali
            R$raw.smali
            R.smali
            R$color.smali
            s$100000000.smali
            s.smali
            s$100000001.smali
            MyAdmin.smali
            R$style.smali
            R$layout.smali
            MD5Util.smali
            M.smali
            BuildConfig.smali
            R$anim.smali
            DU.smali
    adrt
        ADRTSender.smali
        ADRTLogCatReader.smali
```

```
smali
    com
        bangbangtang
            lock
                R$string.smali
                R$id.smali
                R$drawable.smali
                R$attr.smali
                b.smali
                R.smali
                c.smali
                a.smali
                R$style.smali
                R$layout.smali
                BuildConfig.smali
                b$100000000.smali
    adrt
        ADRTSender.smali
        ADRTLogCatReader.smali
```

Figure 9: Case study, Code Removal (Section 5.4). The main instructions that have been removed are highlighted in red.

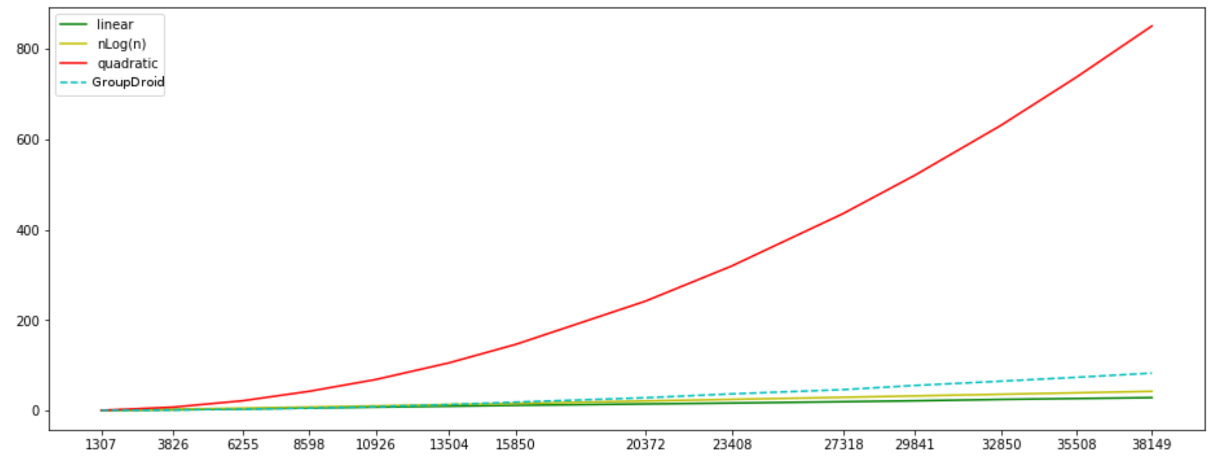


Figure 10: Performance of GroupDroid's optimized comparison algorithm.

6 LIMITATIONS & FUTURE WORK

Code Obfuscation. While GroupDroid managed to group together apps that where employing CFG obfuscation techniques, it still suffers from code obfuscation. This means that it might not be able to handle advanced obfuscated code. Some features could be better extracted in a dynamic way. For example our risky_api vector stores how many times a certain API is written in the method's body, while it would be far more interesting to know how many times it's actually called at runtime.
We could also explore other techniques to circumvent code obfuscation, such as studying the Data Dependency Graph.

Whitelisting Known Libraries. A lot of GroupDroid's false positives come from popular ad libraries (especially admob), since they often use a subset of the risky APIs and are common enough to be found in many samples. The worst cases happen when some parts of an application without ads have been cloned, and the clone has ads. If the cloned parts are less than the ads, the application will appear to be more similar to other applications that employ ads than to its own original. That's why it's usually good practice to whitelist popular ad packages.

Semantic API Grouping Some APIs are often used together to achieve certain goals, they can thus be grouped together and labeled with higher level behaviours (for example TELEPHONY used in conjunction with NET could mean privacy concerns). For now we manually choose which APIs are interesting for our analysis but it would be interesting to group them automatically.

Granularity. Right now, our analysis only considers similarity at the method's level, but instructions can easily be spread among different methods, or some methods could be grouped, making our approach unsound. A good approach could be to consider the entire application's CFG and search for common subgraphs. This is of course a problem that is incredibly hard to tackle (NP-hard in fact) and would require new heuristics and solutions to make it viable.

Performance. Almost every algorithm employed in GroupDroid is parallelizable, in the future we will adopt a cluster of machines with GPUs to divide the workload and speed up the application by quite a bit.

Implementation. As previously explained in Section 4.2, every feature and every report gets saved on disk in plaintext using some serialization libraries. It would be better to utilize a Database driver such as PostgreSQL or MySQL to store the results.

7 RELATED WORK

Previous work on Android systems focused on detecting *clones* and *code reuse* in Android apps, providing ways to cluster apks, and detect and classify Android malware. Many of the proposed methods employ static analysis to build features that can be used to cluster the malware, and strive to make the analyses scalable while keeping good accuracy results.

Clone Detection and Code Reuse. [16] employs *Normalized Compression Distance (NCD)* on each method to find similar methods, this work differs from our tool not only in the approach, but in the focus, here authors focused on determining the quality of an obfuscation process, determining if an application has been infected with malware, and extract the payload, and identify valid code updates. Juxtapp [20] builds an application feature matrix and proposes a scalable method to cluster and evaluate similarity between applications, it employs a scalable method and the evaluation has been run on a dataset of 95, 000 unique Andrdoid applications. Juxtapp can be used to detect code reuse, which allows to find instances of buggy code, and malware, or pirated applications. [12, 13] are methods for detecting cloned applications based on *Program Dependency Graph (PDG)*, the former measures similarity by first fitering methods, and then exploiting subgraph isomorphism to measure similarity. The latter instead exploits *Locality Sensitive Hashing* LSH to find approximate near-neighbors feature vectors. Wang et al. [32] use an approach based on filtering third party libraries, and then building a feature vector using API calls, the similarity is measured by pairwise comparison using the Manhattan distance, this two-steps process allows *WuKong* to be scalable and precise.

Mobile Malware Classification. Our proposed approach differes from other methods for mobile malware classification, this problem has been mainly tackled using machine learning: [15] extracts a signature to represent repackaged malware and tries to cluster malwares into families. [35] is based instead on a peculiar data structure called *Weighted Contextual API Dependency Graphs(WC-ADG)* to capture the semantics of the methods and use these to build a feature vector.

Mobile Malware Detection. Similar ideas have been proposed to tackle malware detection: [5] employs static analysis to extract a set of features that is then classified via linear Support Vector Machines to detect if a sample is malicious. An interesting instance is proposed by [19] and is based on feature space embedding based on call graphs. [2] evaluates how several machine learning algorithms score with an API-based features set. Previous work explored also the use of markov chains [26] for behavioral models, and *Hidden Markov Models (HMM)* and structural entropy [8] to achieve mobile malware detection. A slightly different approach is taken by [10], which employs a features set obtained via both static and dynamic analysis.

8 CONCLUSIONS

In this paper, we proposed a novel technique to identify Android apps that share similar behaviors. Our approach is based on code similarity and uses static features, such as 3D-CFG centroids and API vectors, to compare apps at the method's level. Leveraging such approach, we implemented a tool, GroupDroid, that groups Android malware according to code similarities. Exploiting the observation that malware authors reuse their malicious code across different malware samples—either because they use piggybacking techniques, or update their samples to release new versions—, GroupDroid is not only able to identify groups of malware families, but it can also precisely recognize different variants of the same family. Our experiments, against 4,211 known malicious apps, showed that GroupDroid is accurate and efficient. Moreover, during our evaluation we found several interesting cases of malware families sharing (almost) the same malicious code.

ACKNOWLEDGEMENTS

We would like to thank our reviewers for their valuable comments and input to improve our paper. This work has been supported by the MIUR FIRB project FACE (Formal Avenue for Chasing malwarE) RBFR13AJFT.

REFERENCES

[1] 2016. Google Play has hundreds of Android apps that contain malware. (2016). http://www.trustedreviews.com/news/malware-apps-downloaded-google-play
[2] Yousra Aafer, Wenliang Du, and Heng Yin. 2013. Droidapiminer: Mining api-level features for robust malware detection in android. In *International Conference on Security and Privacy in Communication Systems*. Springer, 86–103.
[3] Niccolò Andronio, Stefano Zanero, and Federico Maggi. [n. d.]. HelDroid: Dissecting and Detecting Mobile Ransomware. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2015-10) *(Lecture Notes in Computer Science)*, Vol. 9404. 382–404. https://doi.org/10.1007/978-3-319-26362-5_18
[4] Jart Armin. 2013. Mobile Threats and the Underground Marketplace. *APWG White Paper: Mobile* (2013).
[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.. In *NDSS*.
[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[7] Michael Bierma, Eric Gustafson, Jeremy Erickson, David Fritz, and Yung Ryn Choe. 2014. Andlantis: Large-scale Android dynamic analysis. *arXiv preprint arXiv:1410.7751* (2014).
[8] Gerardo Canfora, Francesco Mercaldo, and Corrado Aaron Visaggio. 2016. An hmm and structural entropy based detector for android malware: An empirical study. *Computers & Security* 61 (2016), 1–18.
[9] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/2568225.2568286
[10] Sen Chen, Minhui Xue, Zhushou Tang, Lihua Xu, and Haojin Zhu. 2016. Stormdroid: A streaminglized machine learning-based system for detecting android malware. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 377–388.
[11] Andrea Continella, Yanick Fratantonio, Martina Lindorfer, Alessandro Puccetti, Ali Zand, Christopher Kruegel, and Giovanni Vigna. 2017. Obfuscation-Resilient

Privacy Leak Detection for Mobile Apps Through Differential Analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[12] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets.. In *ESORICS*, Vol. 12. Springer, 37–54.

[13] Jonathan Crussell, Clint Gibler, and Hao Chen. 2013. Andarwin: Scalable detection of semantically similar android applications. In *European Symposium on Research in Computer Security*. Springer, 182–199.

[14] Mila Dalla Preda and Federico Maggi. 2017. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques* 13, 3 (2017), 209–232.

[15] Luke Deshotels, Vivek Notani, and Arun Lakhotia. 2014. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*. ACM, 3.

[16] Anthony Desnos. 2012. Android: Static analysis using similarity distance. In *System Science (HICSS), 2012 45th Hawaii International Conference on*. IEEE, 5394–5403.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[18] Yanick Fratantonio, Chenxiong Qian, Pak Chung, and Wenke Lee. 2017. Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. San Jose, CA.

[19] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 45–54.

[20] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. 2012. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 62–81.

[21] Kaspersky Lab. 2016. Mobile Malware Evolution 2016. (2016). https://securelist.com/files/2017/02/Mobile_report_2016.pdf

[22] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security* 12, 6 (2017), 1269–1284.

[23] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 349–358.

[24] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. (2017).

[25] Federico Maggi, Andrea Valdi, and Stefano Zanero. 2013. AndroTotal: A Flexible, Scalable Toolbox and Service for Testing Mobile Malware Detectors. In *Proceedings of the 3rd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*. ACM.

[26] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. 2016. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433* (2016).

[27] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. 2016. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 306–317.

[28] Jon Oberheide and Charlie Miller. 2012. Dissecting the android bouncer. *SummerCon2012, New York* (2012).

[29] Jingjing Ren, Ashwin Rao, Martina Lindorfer, Arnaud Legout, and David Choffnes. 2016. ReCon: Revealing and Controlling PII Leaks in Mobile Network Traffic. In *Proc. of the International Conference on Mobile Systems, Applications and Services (MobiSys)*.

[30] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. 2016. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 230–253.

[31] Statista. 2017. Global mobile OS market share in sales to end users from 1st quarter 2009 to 1st quarter 2017. (2017). http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[32] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 71–82.

[33] Candid Wueest. 2017. Financial Threats Review 2017. (2017). https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-financial-threats-review-2017-en.pdf

[34] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective Real-time Android Application Auditing. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[35] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1105–1116. https://doi.org/10.1145/2660267.2660359

[36] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 185–196.

[37] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 317–326.

[38] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 95–109.

[39] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. 2012. Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.. In *NDSS*, Vol. 25. 50–52.