Revealing Similarities in Android Malware by Dissecting their Methods

Michele Pasetto
University of Verona
Verona, Italy
michele.pasetto_01@studenti.univr.it

Niccol Marastoni
University of Verona
Verona, Italy
niccolo.marastoni@univr.it

Mila Dalla Preda
University of Verona
Verona, Italy
mila.dallapreda@univr.it

***Abstract*—One of the most challenging problems in the fight against Android malware is finding a way to classify them according to their behavior, in order to be able to utilize previously gathered knowledge in analysis and prevention.**

In this paper we introduce a novel technique that discovers similarities between Android malware samples by comparing fragments of executed traces (strands) generated from their most suspect methods. This way we can accurately pinpoint which (possibly) malicious behaviors are shared between these different samples, allowing for easier analysis and classification.

We implement this approach in a tool, *StrAndroid*, that we evaluate on a few dataset of malware and ransomware samples, comparing its results to an existing similarity tool.

***Index Terms*—Program Analysis, Similarity, Android Malware**

1. Introduction

Android is the most widespread mobile OS in the world with an estimated 85% market share as of 2020 [1]. Combined with the fact that around 3.5 billion people nowadays own a smartphone [2] results in close to 3 billion active Android users and many more Android devices. These numbers are helpful in explaining why Android is a highly coveted attack vector for malware developers.

Even though the security model of Android is complex [3] and new versions of the system tend to keep up with new threats, it is rare for users to consistently have the new version installed on their devices [4]. Securing the app distribution system is also not always a sound solution for the security of the ecosystem, as users from all over the world regularly access third-party stores that are not known for their thorough app vetting process [5]. All of these factors combined result in an ecosystem that presents many potentially lucrative attack vectors.

Recent work on Android malware focuses on developing new intelligent and adaptive methods for malware detection and classification [6], [7], generally by adopting machine learning models. These academic endeavors seldomly reflect in the actual usage by anti-malware vendors, where the main techniques used for the classification of malware is still signature-based [8] and thus easily circumvented by simple code modifications [9]. Indeed it is necessary to find better ways to discover the similarities between different malware samples.

A recent report published by MalwareBytes [10] shows that Android malware is getting "stealthier and more aggressive", which should result in research focused on more precision and accuracy. The approach that we present in this paper is meant as a step towards applying more sophisticated methods to the analysis of Android malware, possibly leading to more precision in the analysis results.

We take direct inspiration from [11], where the authors applied similarity by composition to binaries in order to discover their similarity even when compiled with different toolchains and optimizations. The idea behind similarity by composition comes from a work in image recognition [12] and leverages the fact that two images are similar if they are made of similar components. Thus finding the similarities between the components is a key part of establishing if the two original images are also similar. The goal of [11] was to find pieces of code that were semantically similar to a query fragment mainly to allow the search of bugs or known exploits in a benign scenario where binary samples are generated with different compilers or different versions of the same compiler. This use case is of course different from malware analysis where we need to take into consideration the use of malicious code modifications explicitly designed to prevent analysis. For this reason the methodology differs in some key areas that we will expand upon in Section 4.

Motivation. We believe the similarity by composition approach might work for Android malware because malicious behaviors are often just different combinations of the same few base components.

For example, there are approaches that aim at finding Android clones that are similar wrt their behaviors [13] . For this purpose, Object Based Actions (OBAs) are defined as all the API calls that can be grouped into a common semantic group: i.e. HTTP-based actions, TelephonyManager-based actions, SMSManager-based actions, etc. A malicious behavior can then be summarized by the combinations of these OBAs into common patterns. For example, if the goal of the malware is to steal the user data and send it to a remote server then the corresponding behavior can be described with TelephonyManager-based + HTTP-based actions, while if the data was sent through SMS texts then we would have TelephonyManager-based + SMSManager-based actions. These are malicious behaviors that can help classify the malware into different classes and they are all combinations of smaller components.

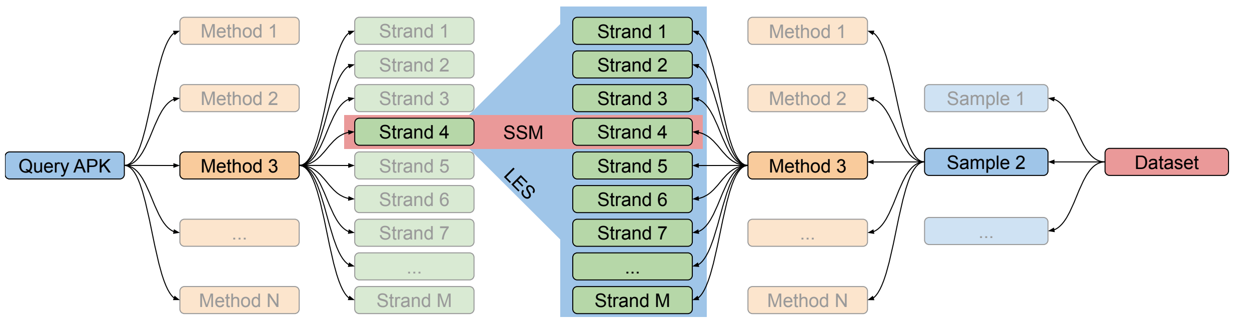


Figure 1. A diagram capturing the workflow of *StrAndroid*

With the similarity by composition approach of [11] we can give a more precise characterization of the behaviors hidden in Android methods.

Contributions. The contributions of this paper are:

- a novel similarity algorithm for Android Malware
- the implementation of a tool for similarity by composition, *StrAndroid*
- the evaluation of *StrAndroid* on datasets of Android malware and ransomware

2. Background

In this section we will give a brief rundown of the main concepts needed to understand our methodology and the problem it sets to solve. For readers familiar with the static analysis of Android malware we suggest skipping to Section 4.

2.1. Android Environment

Android is an open source operating system meant for mobile devices that adopts a very strict security model [3]. Every app installed in an Android device is packaged in an APK file (which is an archive for distribution) and it has to conform to the model in order to interact with the most vulnerable parts of the system, and this is enforced through the use of *permissions* [14] and specific system *APIs*. Throughout the paper we will use the terms *app* and *APK* interchangeably.

APIs and Permissions. Android offers a comprehensive list of APIs to its developers in order to better control the behavior of apps wrt the system itself. We analyzed a number of Android malware samples and created a list of APIs that can be used to execute malicious attacks on the device, we call these *risky APIs*. These are functions that range from network APIs, necessary to send packets through network protocols, to device-specific APIs, which are used to access private information about the user of the device. This is a small sample of our *risky APIs* list:

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

Other works share our view that some APIs have more weight than others, for example [15] calls them "sensitive" APIs and uses them as meaningful features to automatically characterize Android malware.

The usage of system APIs in Android is regulated by requiring the use of each API to be accompanied by a request for permission to the user. These permissions are stored in an XML document (the *manifest*) at the root of the application and are often used in malware analysis to spot which APKs could contain malicious behaviors [16]–[18]. Naturally, applications that do not request permission to use any *risky API* will not be able to affect the behavior of the device nor infringe on the privacy of the user in any way. It is hard to draw conclusions from the sole inclusion of permissions in the *manifest*, since app developers tend to be overzealous and request more permissions than necessary [14].

For this reason we focus on the API calls encountered in the code without checking the permissions, as we believe this gives us an edge when it comes to the precision of the analysis.

Smali. Before being run by Dalvik, the JVM implementation of Android, every app is compiled in *dex* bytecode. This format is not very user-friendly, and for this reason most static analysis on Android code is done on the *smali* format. *StrAndroid* uses *APKTool* [19] in order to translate the native bytecode into smali files, which will then be parsed and analyzed by our tool.

A detailed description of the parsing process and our analysis methodology is in Section 4.

2.2. Android Malware

There are many types of Android malware found in the various markets, and they can be grouped by behavior [20] as: 1) spyware, which is malware that allows an external entity to acquire private information about the unsuspecting user, 2) ransomware, encrypting the private data of the user and requiring a payment (ransom) to decrypt it, and 3) adware, showing unwanted ads to the user. Of course many other types of malware exist, but in this work we target a small dataset of ransomware samples in order to gauge the effectiveness of *StrAndroid*.

2.3. Static Analysis

Program analysis is mainly divided into static and dynamic analysis, where the former studies the properties of programs from the code itself, while the latter executes the program and analyzes the execution traces. The approach described in this paper is static, as we analyze the smali code obtained from the APKs. We now give a brief description of the basic components of static analysis that are required to understand the inner workings of *StrAndroid*.

Basic Blocks. Basic blocks are fragments of code uninterrupted by control flow instructions. They usually are represented as nodes in the control flow graph.

Program Slicing. Program slicing is a static analysis procedure that allows the isolation of code fragments (slices) containing only instructions that are directly related to the slicing criterion through data flow and control flow [21]. The slicing criterion can be a variable in a specific location, a list of variables or an instruction that contains at least a variable.

In this paper we will focus on backward slicing as our preferred technique to extract strands from the APKs, meaning that we identify a program location of interest (slicing criterion) and perform a backward scan of the code, saving a list of instructions that are correlated to at least one of the variables in the slicing criterion.

Strands. The concept of strand was first introduced in [11] (section 3.2) and simply defines a basic block-level program slice. Isolating slices in a basic block means that all the variables therein contained are connected with only data flow (as basic blocks do not contain any control flow by definition). This allows for a less precise representation of the code itself but a more fine-grained kind of analysis, where it is easier to discard non important elements. Section 4 goes in more depth about our specific strands implementation, while two examples of strands can be seen in Figure 3.

3. State of The Art

In this section we will cover some existing works that closely relate to our approach.

Android malware analysis. In [22], the authors highlight how important it is to develop more precise descriptions of the behaviors for existing malware datasets. They conduct a large-scale study where they analyze specific samples in various families that have been classified by existing anti-virus scans. Our approach shares the same goal of achieving a more precise analysis, but it differs in the methodology.

Many Android malware analysis tools have been developed recently using Machine Learning. There are works such as [15] that we already mentioned in Section 2 for sharing our view on *sensitive* APIs, and more recent works that employ modern ML techniques such as LSTM and autoencoders in order to better classify Android malware [6], [23]. Even if our approach is static, with no ML influence, we share the goal of finding the best features that can predict the maliciousness of an Android malware.

Program similarity. The work that most relates to ours is [11], where the authors envisioned an approach based on similarity by composition to find code clones in binaries that have either been compiled with different toolchains or that belong to different systems altogether. Our work is a direct translation of their effort to Android malware analysis. We make some changes to the strand similarity methodology (see Section 4 for details) and we apply the approach to a very different problem domain.

4. Methodology

In this section we will describe our approach in detail, including some of the methodology in [11] that initially inspired this work and delving deeper in the differences between our work and [11].

A detailed workflow can be seen in Fig 1. A query APK is compared to the rest of the dataset by first extracting all its methods and dividing them into basic blocks. Then, from every method we generate the strands (see Section 2 and [11]) which are placed in buckets. This is done for every sample in the dataset (only once, then it is saved for future runs). Every strand from the query APK is compared with all the strands coming from each method of the dataset samples using the *Strand Similarity Measure*. The *Local Evidence Score* is then computed to gauge how significant the strand matching really is. Finally, the *Global Evidence Score* will simply sum all the LES from the various strands in the methods and generate a score that indicates how similar two methods are. In the rest of the section we go into detail for each of the steps just described.

Use Case. *StrAndroid* is a tool focused on the analysis and reverse engineering of malware. In order to use the tool at its fullest we first need a dataset of Android malware samples, then we can input a query sample q_s into the system and *StrAndroid* will show which sample best matches q_s wrt its potentially malicious behaviors.

This comparison is done at the method level: the sample that shares the highest number of methods with q_s is selected as the result of the analysis. Along with this, *StrAndroid* will show a comprehensive list of all the common methods between the two samples and their relative similarity scores, allowing the analysts to focus the reverse engineering efforts on these methods.

Method Filtering and Collection. As a first step in the analysis *StrAndroid* builds an internal representation of all the APKs in the dataset by collecting their methods. We filter out the methods that do not meet two basic requirements: 1) a minimum length, and 2) presence of Risky APIs (see Section 2 and [24] for a brief introduction).

The minimum length of the methods is a variable parameter that can be set at the beginning of the analysis and is meant to filter out common methods that are used in most APKs (such as *init* methods) and would muddy the detection of actual malicious behaviours. In order to further speed up the analysis, *StrAndroid* will not consider methods that do not contain at least one invocation to a *risky API*, since these methods only contain the internal logic of the app and cannot exhibit malicious behaviors targeted towards either the user or the device itself. Both of these heuristics have been adopted due to our previous experience with *GroupDroid* [24] and they have been successfully tested empirically. During this initial scan of the methods in the dataset, the code is parsed by *StrAndroid* in order to speed up the next part of the analysis.

```
strands = []
for inst in reverse(BB):
    s <- new Strand
    s <- s + inst
    d <- defined_vars(inst)
    u <- used_vars(inst)
    BB <- BB - inst
    for inst_1 in reverse(BB):
        d_1 <- defined_vars(inst_1)
        u_1 <- used_vars(inst_1)
        for v in u:
            if v in d_1:
                u <- u + u_1
                s <- s + inst_1
                BB <- BB - inst_1
    strands <- strands + reverse(s)
return strands
```

Figure 2. Strand Extraction Algorithm

Parsing. Program slicing requires isolating instructions that are correlated to the slicing criterion through data flow [21] (control flow can be ignored since strands are extracted from basic blocks), thus the first component of the tool is a parser for Smali. In order to generate def-use chains it is essential to know which variables are used and defined in each line. For this purpose, our smali code parser extracts these variables using the pattern of the particular opcode: we define 6 procedures covering all the possible Def-Use behaviors of the instructions and apply to each opcode the correct routine.

Strand Extraction and Normalization. Once all the interesting methods have been collected and parsed for every APK in the dataset, they are split into basic blocks by a simple heuristic. At this point, strands are extracted from the basic blocks. Strands are static slices of the code contained in a basic block, obtained via a simple backward slicing algorithm (see pseudocode at Figure 2). These strands then go through a *normalization* process, where every variable encountered is renamed wrt its order of appearance in the strand.

As an example we show this on a snippet from a simplified method (its statements are shorter):

```
move-object v4, v2
const/high16 v5, 0x10000000
invoke-virtual {v4, v5}, setFlags(I)
move-result-object v4
move-object v6, v1
move-object v7, v4
invoke-virtual {v6}, LstartService()
move-result-object v6
```

Assuming that the last statement is the slicing criterion, starting from it, the algorithm walks backwards and collects all the statements that may affect the value of $v6$, which is the only variable in the slicing criterion. This is the strand extracted:

```
move-object v6, v1
invoke-virtual {v6}, LstartService()
move-result-object v6
```

Clearly all the statements ignored do not contain a data-flow connection with $v6$. At this point we normalize the strand by renaming the variables:

```
move-object v1, v2
invoke-virtual {v1}, LstartService()
move-result-object v1
```

This last step is done in order to thwart the all too common occurrence of statement reordering, which happens both due to obfuscation attempts and due to the decompilation process. In our tests strand normalization has proven to be a necessary step, since the similarity measure is very syntactic.

Strand Similarity Measure (SSM). In [11] the similarity between two strands was computed via a program verifier that checks for input-output equivalence between the strands while pairing each variable from a strand with the corresponding one in the other. This is done by lifting the binary procedure into BoogieIVL [25] by first going through IDA pro, then LLVM IR [26] via BAP [27] and lastly SMACK [28] is used to translate LLVM IR into BoogieIVL.

Lacking such a peculiar toolchain for our Android use case we opted to simplify the strand similarity measure and adopt the very common Jaccard index. Two strands s_1 and s_2 are compared wrt the Jaccard index of the instructions they contain. Let us recall the mathematical definition of the Jaccard index between two sets:

$$J(A,B) = \frac{|A \bigcap B|}{|A \bigcup B|} = \frac{|A \bigcap B|}{|A| + |B| - |A \bigcap B|} \tag{1}$$

Then our strand similarity measure (*SSM*) can be stated as:

$$SSM(s_1, s_2) = J(lines(s_1), lines(s_2)) \tag{2}$$

Where $lines(s_1)$ and $lines(s_s)$ denote the set of statements contained in the normalized strand s_1 and s_2 respectively. For example, given the normalized strand in the previous section ($strand_A$) we can have a similar strand extracted from a different method and then normalized ($strand_B$):

```
move-object v1, v2
move-object v2, v1
invoke-virtual {v1}, LstartService()
move-result-object v1
```

This strand is almost equivalent to the previous but has an added line *move-object v2, v1* that does not change the semantics of the strand (we found these types of meaningless insertions in our manual investigation). Calculating the Jaccard Index between these strands we have 0.75, since:

$$\frac{|strand_A \bigcap strand_B|}{|strand_A \bigcup strand_B|} = \frac{3}{4} = 0.75 \tag{3}$$

This can be thought of as the likelihood of the two strands being similar. While 75% likelihood might seem too low of a value, it is mainly dictated by the reduced length of this example, for longer strands the Jaccard Index of two similar strands is usually higher.

The design choice of using the Jaccard index as similarity between two strands comes from the need to have an efficient method to compute the function, as it necessarily needs to be computed for every strand in every method. It is also convenient that the Jaccard index gives a value between 0 and 1, giving a sort of likelihood to the similarity of two strands. We will discuss possible problems with this approach in Section 6.

```
iget-object v5, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-interface {v5}, Lorg/apache/http/HttpResponse;->getEntity()Lorg/apache/http/HttpEntity;
move-result-object v1
.local v1, "entity":Lorg/apache/http/HttpEntity;
invoke-interface {v1}, Lorg/apache/http/HttpEntity;->getContent()Ljava/io/InputStream;
move-result-object v5
invoke-virtual {p0, v5}, Lcom/xxx/yyy/BBBB;->generateString(Ljava/io/InputStream;)Ljava/lang/String;
```

```
iget-object v9, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-interface {v9}, Lorg/apache/http/HttpResponse;->getEntity()Lorg/apache/http/HttpEntity;
move-result-object v2
.local v2, "entity":Lorg/apache/http/HttpEntity;
iget-object v9, p0, Lcom/xxx/yyy/BBBB;->response:Lorg/apache/http/HttpResponse;
invoke-virtual {p0, v9}, Lcom/xxx/yyy/BBBB;->getContentCharset(Lorg/apache/http/HttpResponse;)
move-result-object v0
.local v0, "charset":Ljava/lang/String;
invoke-interface {v2}, Lorg/apache/http/HttpEntity;->getContent()Ljava/io/InputStream;
move-result-object v9
invoke-virtual {p0, v9, v0}, Lcom/xxx/yyy/BBBB;->generateString(Ljava/io/InputStream;)
```

Figure 3. Two semantically equivalent strands from method HppGet

Method Similarity Measure. When every strand is extracted both in the dataset and in the query sample q_s, *StrAndroid* checks every method in q_s against every method of the dataset with the following algorithm. First we define a function that, given a query strand s_q and a target method t, gives us the likelihood of finding a strand $s_t \in strands(t)$ such that s_q and s_r are semantically similar. This is equal to the maximum of the *SSM* between the query strand s_q and every strand contained in t:

$$\mathcal{P}(s_q|t) = max_{s_t \in t}(SSM(s_q, s_t)) \tag{4}$$

Continuing with the methodology first described in [11] we implement a function that measures the statistical significance of a strand wrt the entire dataset. This is done in order to give more weight to strands that are not common in the dataset, and should result in a similarity measure that focuses on the more unique parts of the code. Given a query strand s_q and all the strands in the dataset $s_t \in T$, we define:

$$\mathcal{P}(s_q, T) = \frac{\sum_{s_t \in T} SSM(s_q|s_t)}{|T|} \tag{5}$$

A lower value of $\mathcal{P}(s_q, T)$ represents a higher statistical relevance, as it means that s_q has few semantically similar strands in the dataset.

Following [11] we can now define the Local Evidence Score (LES) between a strand and a method as:

$$\begin{aligned} LES(s_q, t) &= Log(\frac{\mathcal{P}(s_q, t)}{\mathcal{P}(s_q, T)}) \\ &= Log(\frac{max_{s_t \in t}(SSM(s_q, s_t))}{\mathcal{P}(s_q, T)}) \end{aligned} \tag{6}$$

The last function that we need to define in order to obtain a similarity between methods is the Global Evidence Score (GES). Given a query method q contained in the query sample and a target method t extracted from one of the samples in the database we have:

$$GES(q|t) = \sum_{s_q \in q} LES(s_q|t) \tag{7}$$

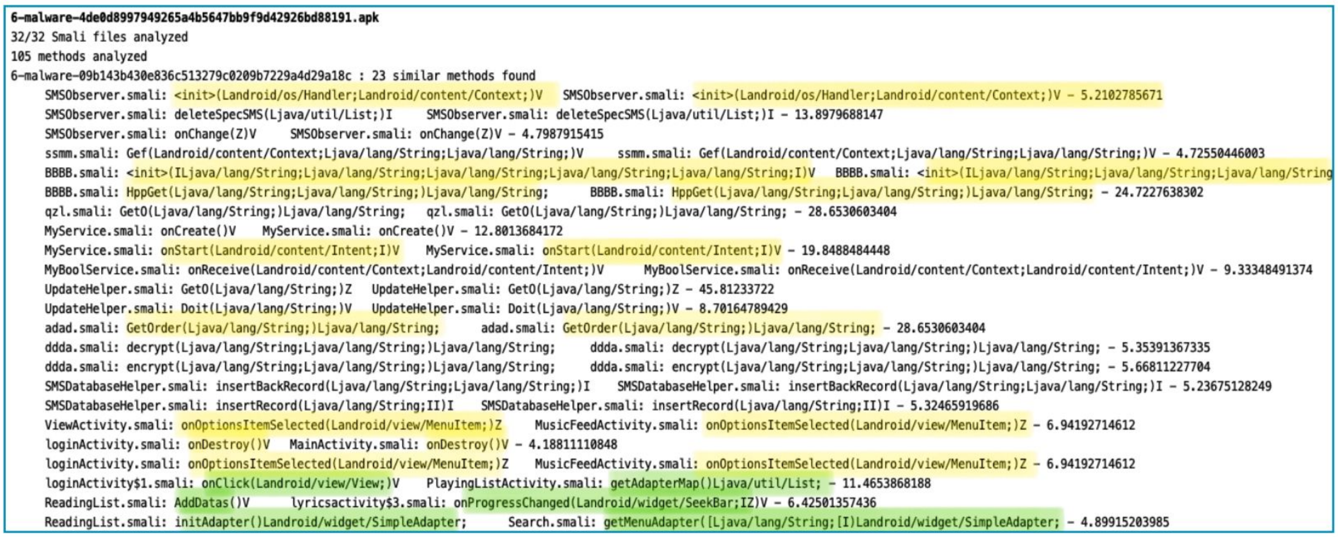
The measure of similarity between two methods is given by the sum of the individual values of LES for every strand in the query method. This sum is of course only lower-bounded by 0 but does not have an upper bound, which can induce significant errors that we discuss in Sections 5 and 6. For each method in the query sample, *StrAndroid* computes the GES for every method in the dataset and returns only the method that generates the highest score, provided it exceeds the set threshold of 4.

The sample r_s that matches the most methods with the query sample q_s is then returned as the result of the analysis, with a list of all similar methods between the two samples.

5. Evaluation

As a first step, *StrAndroid* has been evaluated against the well known GENOME dataset [29] simulating a classification task. Since our tool requires an existing dataset of known malware, this proved to be the easiest test bed to ascertain its efficacy, even though the dataset itself is showing its age. The GENOME dataset mostly contains malware that has not been thoroughly modified, so these results, while encouraging, were not sufficient for a proper evaluation. For this reason a second evaluation has been conducted on the PraGuard dataset [30], which contains various program transformation such as string encryption, class encryption and reflection.

We then tested *StrAndroid* on samples selected from a dataset of Android ransomware and malware previously used in the evaluation of *GroupDroid* [24], this allowed us to easily spot any inconsistencies between the two approaches. Unlike our previous approach, *StrAndroid* adopts a more precise similarity algorithm that has virtually zero false positives, thus it easily refines the results we had with *GroupDroid*. Figure 5 shows a comparison between the two tools by comparing the number of similar methods that were found among 8 pairs in the dataset. *GroupDroid* almost always over-approximates and returns more similar methods than *StrAndroid*, except a few cases where *StrAndroid* actually discovers some methods that



```
6-malware-4de0d8997949265a4b5647bb9f9d42926bd88191.apk
32/32 Smali files analyzed
105 methods analyzed
6-malware-09b143b430e836c513279c0209b7229a4d29a18c : 23 similar methods found
     SMSObserver.smali: <init>(Landroid/os/Handler;Landroid/content/Context;)V   SMSObserver.smali: <init>(Landroid/os/Handler;Landroid/content/Context;)V - 5.2102785671
     SMSObserver.smali: deleteSpecSMS(Ljava/util/List;)I     SMSObserver.smali: deleteSpecSMS(Ljava/util/List;)I - 13.8979688147
     SMSObserver.smali: onChange(Z)V     SMSObserver.smali: onChange(Z)V - 4.7987915415
     ssmm.smali: Gef(Landroid/content/Context;Ljava/lang/String;Ljava/lang/String;)V     ssmm.smali: Gef(Landroid/content/Context;Ljava/lang/String;Ljava/lang/String;)V - 4.72550446003
     BBBB.smali: <init>(ILjava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;I)V   BBBB.smali: <init>(ILjava/lang/String;Ljava/lang/String;Ljava/lang/String
     BBBB.smali: HppGet(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;      BBBB.smali: HppGet(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String; - 24.7227638302
     qzl.smali: GetO(Ljava/lang/String;)Ljava/lang/String;   qzl.smali: GetO(Ljava/lang/String;)Ljava/lang/String; - 28.6530603404
     MyService.smali: onCreate()V    MyService.smali: onCreate()V - 12.8013684172
     MyService.smali: onStart(Landroid/content/Intent;I)V    MyService.smali: onStart(Landroid/content/Intent;I)V - 19.8488484448
     MyBoolService.smali: onReceive(Landroid/content/Context;Landroid/content/Intent;)V      MyBoolService.smali: onReceive(Landroid/content/Context;Landroid/content/Intent;)V - 9.33348491374
     UpdateHelper.smali: GetO(Ljava/lang/String;)Z   UpdateHelper.smali: GetO(Ljava/lang/String;)Z - 45.81233722
     UpdateHelper.smali: Doit(Ljava/lang/String;)V   UpdateHelper.smali: Doit(Ljava/lang/String;)V - 8.70164789429
     adad.smali: GetOrder(Ljava/lang/String;)Ljava/lang/String;      adad.smali: GetOrder(Ljava/lang/String;)Ljava/lang/String; - 28.6530603404
     ddda.smali: decrypt(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;     ddda.smali: decrypt(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String; - 5.35391367335
     ddda.smali: encrypt(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;     ddda.smali: encrypt(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String; - 5.66811227704
     SMSDatabaseHelper.smali: insertBackRecord(Ljava/lang/String;Ljava/lang/String;)I    SMSDatabaseHelper.smali: insertBackRecord(Ljava/lang/String;Ljava/lang/String;)I - 5.23675128249
     SMSDatabaseHelper.smali: insertRecord(Ljava/lang/String;II)I    SMSDatabaseHelper.smali: insertRecord(Ljava/lang/String;II)I - 5.32465919686
     ViewActivity.smali: onOptionsItemSelected(Landroid/view/MenuItem;)Z     MusicFeedActivity.smali: onOptionsItemSelected(Landroid/view/MenuItem;)Z - 6.94192714612
     loginActivity.smali: onDestroy()V   MainActivity.smali: onDestroy()V - 4.18811110848
     loginActivity.smali: onOptionsItemSelected(Landroid/view/MenuItem;)Z    MusicFeedActivity.smali: onOptionsItemSelected(Landroid/view/MenuItem;)Z - 6.94192714612
     loginActivity$1.smali: onClick(Landroid/view/View;)V    PlayingListActivity.smali: getAdapterMap()Ljava/util/List; - 11.4653868188
     ReadingList.smali: AddDatas()V      lyricsactivity$3.smali: onProgressChanged(Landroid/widget/SeekBar;IZ)V - 6.42501357436
     ReadingList.smali: initAdapter()Landroid/widget/SimpleAdapter;      Search.smali: getMenuAdapter([Ljava/lang/String;[I)Landroid/widget/SimpleAdapter; - 4.89915203985
```

Figure 4. A screenshot of *StrAndroid*, highlighted in yellow are the new similar methods that were missed by *GroupDroid*. In green the only false positives of *StrAndroid*

eluded detection in *GroupDroid*. An example of the analysis results can be seen in Figure 4.

GENOME. We extracted around 600 samples from the GENOME dataset, excluding mainly the families that contained less than 20 samples each, then we randomly chose one sample from each family and removed it from the dataset. We then used *StrAndroid* with each one of these removed samples as the query APKs, in order to find similar APKs in the dataset. Our tool paired each query APK with samples of their original family on 100% of the cases, thus validating the approach.

PraGuard. The PraGuard dataset [30] provided a few different code obfuscations applied to the GENOME dataset. We tested *StrAndroid* with these by extracting one sample from each class in the obfuscated database and used it as a query APK against the entire original GENOME dataset. The results with the samples modified with string encryption have been positive, with every sample extracted randomly from each malware class being classified correctly with other samples in the same class. This test proved that the string comparison, used to gauge the equivalence of statements for the *SSM* between strands, does not impact negatively the effectiveness of our tool when used for classification.

The next two obfuscation classes obtained from PraGuard, class encryption and reflection, uncovered a lot of flaws in the approach. Both classes resulted in unsatisfactory classification, with the samples obfuscated with class encryption resulting in zero similar APKs for many of them. This negative result is unavoidable as the APKs are obfuscated with DexGuard [31], which encrypts and compresses (with GZIP) every class in the APK. The content of the classes is thus completely hidden to a static analyzer and is only revealed at run-time. Our approach relies on extracting data-flow information from methods statically, which means that the only methods available for analysis were the ones used for run-time decryption.

The samples obfuscated with reflection instead generated many false positives, which is easily explainable by the confined nature of our analysis (every strand comes from a single basic block). In order to correctly calculate the similarity between strands, the method invocation has to be part of the strand itself.

Use Case Dataset. We used a dataset of 20 Android malware and ransomware samples, a reduced version of the one collected in 2017 for [24], where each sample is similar to at least one other sample in the dataset, giving a total of 10 semantically-similar program pairs (or families). By *similar* samples, in this context, we mean that they contain some of the same malicious behaviors, while the rest of the application (usually a piggy-backed legitimate app) is not considered for the similarity. In Figure 5 we show a direct comparison between *StrAndroid* and *GroupDroid*. The latter almost always returns more similar methods but this is due to the presence of false positives, while *StrAndroid* is generally more precise for all the classes considered and sometimes obtains even less false negatives than *GroupDroid* (for example in *1-ransom*).

The dataset contains ransomware and malware samples and is fairly small to allow manual verification of the results, since the goal of this evaluation phase is to challenge the ground-truth extracted from an analysis by *GroupDroid*. In Section 7 we speculate on some possible improvements of this step.

Precision. The nature of the similarity measure implemented in *StrAndroid* should make it so that the tool is not affected by certain types of code obfuscation such as structural transformations (modifying the CFG of the methods) and dummy code insertion. We verified this in our reduced dataset by running both *GroupDroid* and *StrAndroid* and manually evaluating the results of the analyses.

The results of our tests are overwhelmingly positive, using *StrAndroid* we uncovered the source of some false negatives in the analysis with *GroupDroid*, mostly coming from samples employing the two aforementioned modifications.

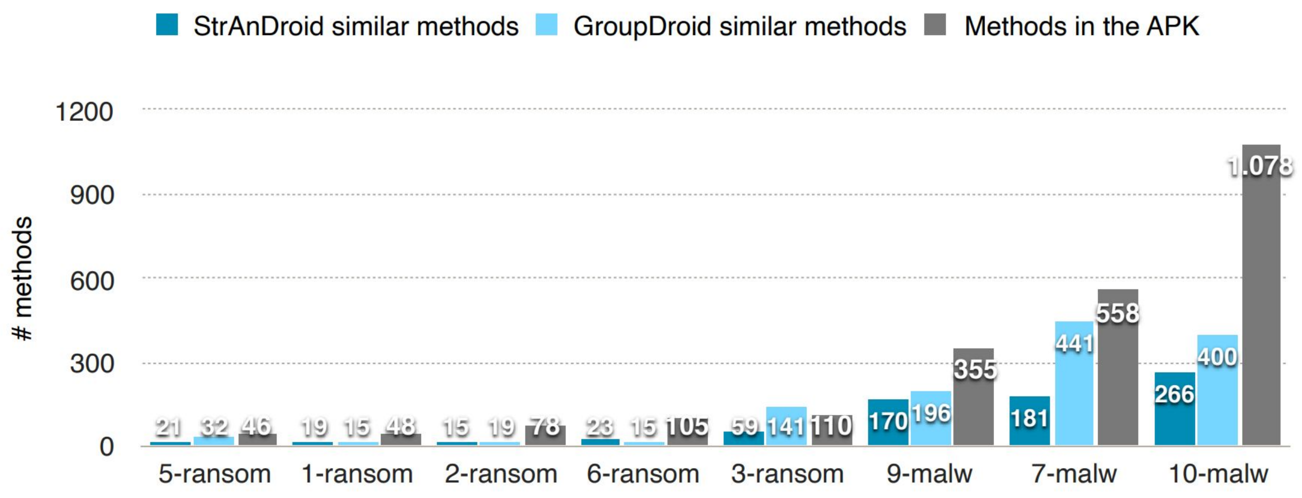


Figure 5. Comparing the number of similar methods found in 8 pairs of the dataset. *GroupDroid* often over-approximates and finds many false positives while *StrAndroid* is more precise.

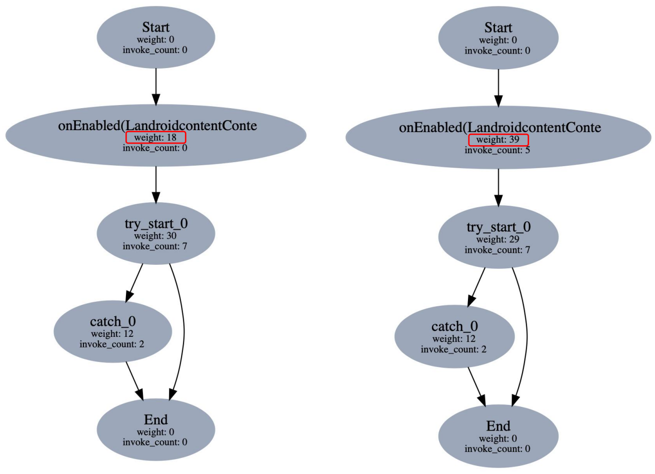


Figure 6. CFGs extracted from two semantically equivalent methods with bogus code insertions

In Section 5.1 we explore two specific examples of these tests.

Normal Code Evolution. Since most malware nowadays consists of modified versions of existing malware, it is possible that some of the transformations that we noticed in the samples are not always the result of an attempt to obfuscate the code, but they could simply be the result of updated and refined code for the new versions. Malware developers could add code to their samples not just as dummy filler to fool signature based approaches (although that seems to work well [32]), but also to add new behaviors. It is our opinion that this has to be investigated more, as it is possible to use *StrAndroid* to analyze the evolution of malware in the same family over time (we will expand on this in Section 7).

Threshold. As anticipated in Section 4, the similarity between two methods is given by their GES, which is a summation of all the local LES between the strands and the method itself. This causes the similarity measure to assume theoretically unbounded values (since it depends on the number of strands in a methods), which means that setting a predefined threshold Tr that works on all methods is not a trivial task. If Tr is too small it can cause false positives among methods that are not similar but contain a lot of strands (as we will see in Section 6) and false negatives if the methods are indeed similar but too small to reach Tr. We thoroughly experimented with our dataset and reached the conclusion that $Tr = 4$ is a good threshold to decide the similarity between two methods, as we encountered 71 methods in the class $7 - malware$ that are similar between the two samples and returned a GES between 4 and 4.5.

5.1. Case Studies

The nature of strands, mainly their existence confined in basic blocks, allows *StrAndroid* to be very precise when finding similarities between methods even when one of the control flow is modified substantially. Another advantage of using buckets of strands is evident when evaluating the similarity of methods where dummy code has been inserted, as the original code (the code that is semantically relevant) is still present in the form of a composition of strands. We now show two specific examples of these cases and highlight how focusing on strands helped the analysis.

Modified CFG. HppGet() is a method that is present in two similar malware samples ($sample_1$ and $sample_2$) in our reduced dataset and is used to communicate with a remote server with the use of the Apache HTTP API. The peculiarity of the two versions of this method is that the one in $sample_2$ is heavily modified wrt its CFG compared to the version in $sample_1$ (both CFGs can be seen in Figure 7).

This proved to be a challenge for the structural similarity approach taken by *GroupDroid*, while *StrAndroid* still recognizes the meaningful strands in the code, ignoring the modifications to the CFG. One of the meaningful strands from each ransomware sample can be seen in

	N. of APKs	N. of Methods	Avg. Time (per method)
TS_1	100	20489	0.9s
TS_2	150	27273	1.32s
TS_3	200	40277	2.15s

TABLE 1. TEST SET FOR PERFORMANCE EVALUATION

Figure 3, it is easy to see how they describe the same behavior.

Dummy code insertion. onEnable() is a method extracted from two ransomware samples ($sample_3$ and $sample_4$). Both versions of the method have been manually investigated and evidently perform the same function, but the one in $sample_4$ is almost double the size of its original version in $sample_3$. The CFG of the methods has not been modified, but around 20 lines of dummy code have been inserted. This can be seen in Figure 6 where the *weight* parameter in the second basic block shows that the amount of statements therein contained has doubled.

This is usually done in order to fool automatic malware recognition tools that rely on exact file signature and it also proved to be a challenge for *GroupDroid*, since the similarity measure relies heavily on the *weight* of the basic blocks (the number of statements in it). As expected the strand approach taken with *StrAndroid* works flawlessly with transformations that modify the structure of the CFG.

5.2. Performance

The method parsing and the strand generation are relatively simple operations, as they only require one pass for each smali file. Doing this to every sample in the dataset still yields a linear complexity, meaning that even with hundreds of samples the extraction times are fairly small. The true complexity of *StrAndroid* comes from strand comparisons, as each strand from every method in the query APK has to be compared (via *SSM*) with every strand from every method in every sample of the dataset.

To gauge the actual performance of the tool we ran tests on a MacBook Pro with a 2.3 GHz i5 dual-core processor and 8GB of RAM, against a test set composed of 3 different subsets of the GENOME dataset with randomly extracted samples. The specifics of the test set can be seen in Table 1, along with the average time it took to analyze a single method in the query sample against every method in the test set. The query samples were also extracted randomly from the original dataset, one for each of the 5 classes *BeanBot*, *DroidDream*, *DroidKungFu*, *Geinimi* and *GoldDream*. The specific execution times for the tests on each sample against the 3 subsets can be observed in Table 2.

	TS_1	TS_2	TS_3	Risky methods in query sample
BB43	4m 18s	6m 15s	10m 39s	361
DDL7	3m 35s	5m 18s	8m 50s	234
DKF14	3m 58s	5m 44s	9m 45s	289
GEIN37	2m 50s	4m 23s	6m 58s	163
GD17	2m 24s	3m 58s	5m 55s	166
Avg.	3m 25s	5m 7s	8m 25s	

TABLE 2. RUNNING TIMES OF THE ANALYSIS ON THE TEST SET

Improvements. Strand comparisons are independent of each other, which means that the performance of our tool could be increased by a great factor if we employed code parallelization. The tool also suffers from the bare-bones Python implementation, where a great number of string comparisons means a great decrease in performance. We are currently looking into *Cython* [33] in order to leverage the faster C implementation for string comparison.

6. Limitations

Our tests with the PraGuard dataset [30] and the manual assessment of the results with our reduced malware/ransomware dataset have unveiled some limitations of our approach.

String Comparison. When calculating the SSM as the Jaccard Index between two strands, the union operator considers the strands as sets and the statements therein contained as strings. This means that our algorithm will judge the uniqueness of a statement in the set by using exact string equivalence. This has proven to not be a problem when analyzing most samples in the GENOME dataset [29] but has resulted in a slightly reduced number of equivalent methods found when using samples from PraGuard obfuscated with string encryption.

Static Thresholds. The value 4 as a threshold for the GES between two methods has proven to be a good estimate for method similarity throughout the tests. This value is highly dependent on the size of the methods and on the number of strands, thus it could be useful to have a threshold that adapts to these parameters, or conversely implement an algorithm that learns the correct threshold given the parameters.

Unbounded Similarity Measure. Related to the previous point, the GES is calculated as a summation of the LES between all strands. This makes its value theoretically unbounded, which further exacerbates the problem of having a static threshold for the similarity measure. Future evolutions of this work could consider a measure of central tendency such as the arithmetic mean.

Unique Result. As introduced in Section 4, *StrAndroid* returns only one method as a result of the similarity analysis for each method in the query APK. This effectively means that, given a dataset containing families of malware and a query apk that belongs to one of the families, the result APK is going to be unique. In other words, no other APKs containing less similar methods (but still similar) is going to be returned. This might be too strict of a design choice, as our similarity measure is in no way perfect.

7. Conclusions and Future Work

We implemented a tool *StrAndroid* to assess the similarity of Android malware wrt their malicious behaviors, taking as inspiration the approach of similarity by composition first introduced in [12] for image recognition and in [11] for binary code. We tested the approach on a selected dataset of 20 Android malware and ransomware samples and we compared the results to *GroupDroid*.

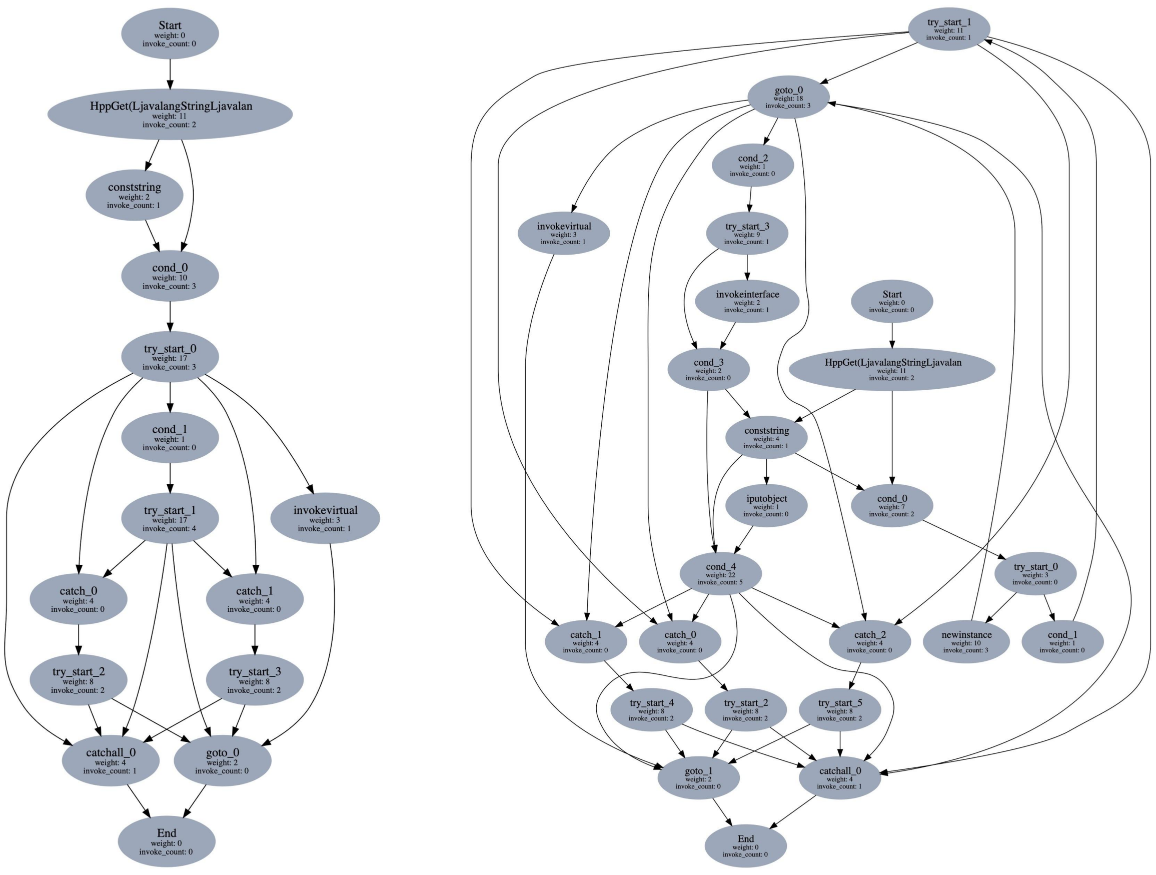

Start weight: 0 invoke_count: 0
HppGet(LjavalangStringLjavalan weight: 11 invoke_count: 2
conststring weight: 2 invoke_count: 1
cond_0 weight: 10 invoke_count: 3
try_start_0 weight: 17 invoke_count: 3
cond_1 weight: 1 invoke_count: 0
try_start_1 weight: 17 invoke_count: 4
invokevirtual weight: 3 invoke_count: 1
catch_0 weight: 4 invoke_count: 0
catch_1 weight: 4 invoke_count: 0
try_start_2 weight: 8 invoke_count: 2
try_start_3 weight: 8 invoke_count: 2
catchall_0 weight: 4 invoke_count: 1
goto_0 weight: 2 invoke_count: 0
End weight: 0 invoke_count: 0
try_start_1 weight: 11 invoke_count: 1
goto_0 weight: 18 invoke_count: 3
cond_2 weight: 1 invoke_count: 0
invokevirtual weight: 3 invoke_count: 1
try_start_3 weight: 9 invoke_count: 1
invokeinterface weight: 2 invoke_count: 1
Start weight: 0 invoke_count: 0
cond_3 weight: 2 invoke_count: 0
HppGet(LjavalangStringLjavalan weight: 11 invoke_count: 2
conststring weight: 4 invoke_count: 1
iputobject weight: 1 invoke_count: 0
cond_0 weight: 7 invoke_count: 2
cond_4 weight: 22 invoke_count: 5
try_start_0 weight: 3 invoke_count: 0
catch_1 weight: 4 invoke_count: 0
catch_0 weight: 4 invoke_count: 0
catch_2 weight: 4 invoke_count: 0
newinstance weight: 10 invoke_count: 3
cond_1 weight: 1 invoke_count: 0
try_start_4 weight: 8 invoke_count: 2
try_start_2 weight: 8 invoke_count: 2
try_start_5 weight: 8 invoke_count: 2
goto_1 weight: 2 invoke_count: 0
catchall_0 weight: 4 invoke_count: 1
End weight: 0 invoke_count: 0

Figure 7. CFGs extracted from two semantically equivalent methods, *StrAndroid*deems the methods similar enough

We assessed that *StrAndroid* produces results that are much more precise, with a low rate of false negatives and false positives, and is able to overcome certain types of structural code transformations such as dummy code insertion and CFG transformations. The strength of this approach comes from the fact that it combines semantic and syntactic types of analysis and benefits from each. Semantic analysis is applied in the parsing of the methods and extraction of the strands, while syntactic analysis is used in the similarity score calculations where each line in the strand is exactly compared to those in another strand. We believe our methodology shows promise as a concrete step towards helping analysts with the reverse engineering of Android malware. In the following we will list some of the possible improvements to this work that could be considered as future research directions.

Program Verifier for Smali. One of the original limitations of this work has been the inability to find a free program verifier written specifically for smali code. A useful direction would be to develop either a new program verifier for smali or a transpiler from smali to a similar language that already has a program verifier. This should counter some of the limitations of using the Jaccard index as similarity measure between strands, as stated in Sec. 6.

Machine Learning. Strands proved to be a great feature to ascertain similarity between malware samples in Android, but they are used in a rigid static algorithm with set thresholds. Strands could be used as features for a classification algorithm based on Machine Learning techniques, where the importance of a strand can be calculated through attention instead of its uniqueness in the dataset. This is similar to the approach in code2vec [34] where the authors used paths in the AST in order to train an algorithm that could infer method names from their code.

New Evaluation Testbed. The compact size of the dataset comes from a need to manually verify the precision of *StrAndroid* over *GroupDroid*, a future direction for this research could be to design a new evaluation methodology that automates such a task. This is not a trivial proposition, as it is hard to trust any automatically-generated ground-truth and it is even harder to find datasets that have already been analyzed with our same use-case in mind.

A Combination of *StrAndroid* and *GroupDroid*. As we discuss in Section 5, *GroupDroid* is a very fast tool that works very well for clustering a dataset of unknown malware even if it returns many false positives wrt the individual methods. *StrAndroid*, on the other hand, is very precise when it comes to recognizing similar methods in different malware samples, but its performance is quite lacking. We are actively investigating how to combine these two approaches in a new tool.

Monitoring Malware Evolution. Android malware is in constant evolution, new families are added at a slower rate [35] but new versions of existing malware are constantly being discovered on the distribution platforms. It is then of the utmost importance to be able to analyze and counteract the evolution of these new variants. We already discussed

(Section 5) how *StrAndroid* seems to be resilient to certain code transformation techniques, such as dummy code insertion, but its strength lies in focusing on specific strands that execute the malicious behavior. This means that it can recognize new versions of the same methods even when the code has been updated to modify or add to the existing behaviors, and highlighting the common strands between two cloned methods can bring into light the newly added behaviors. A new direction in the development of this approach is geared towards analyzing the new versions of existing malware, in order to assess their evolution.

Acknowledgment. The research has been partially supported by the project Dipartimenti di Eccellenza 2018 2022 funded by the Italian Ministry of Education, Universities and Research (MIUR).

References

[1] IDC, "Smartphone users worldwide," 2020, [accessed 13-March-2020]. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[2] S. O'Dea, "Smartphone users worldwide," 2020, [accessed 13-March-2020]. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide

[3] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[4] M. Mahmoudi and S. Nadi, "The android update problem: An empirical study," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 220–230.

[5] M. Xu, Y. Ma, X. Liu, F. X. Lin, and Y. Liu, "Appholmes: Detecting and characterizing app collusion among third-party android markets," in *Proceedings of the 26th International Conference on World Wide Web*, 2017, pp. 143–152.

[6] W. Wang, M. Zhao, and J. Wang, "Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *Journal of Ambient Intelligence and Humanized Computing*, vol. 10, no. 8, pp. 3035–3043, 2019.

[7] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, 2019.

[8] A. Kapoor, H. Kushwaha, and E. Gandotra, "Permission based android malicious application detection using machine learning," in *2019 International Conference on Signal Processing and Communication (ICSC)*. IEEE, 2019, pp. 103–108.

[9] M. Dalla Preda and F. Maggi, "Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology," *Journal of Computer Virology and Hacking Techniques*, vol. 13, no. 3, pp. 209–232, 2017.

[10] Malwarebytes, "State of malware report," 2020, [accessed 13-March-2020]. [Online]. Available: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf

[11] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, 2016.

[12] O. Boiman and M. Irani, "Similarity by composition," in *Advances in neural information processing systems*, 2007, pp. 177–184.

[13] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 306–317.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.

[15] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, 2016.

[16] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an, and H. Ye, "Significant permission identification for machine-learning-based android malware detection," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3216–3225, 2018.

[17] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[18] K. A. Talha, D. I. Alper, and C. Aydin, "Apk auditor: Permission-based android malware detection system," *Digital Investigation*, vol. 13, pp. 1–14, 2015.

[19] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," 2012.

[20] N. Kiss, J.-F. Lalande, M. Leslous, and V. V. T. Tong, "Kharon dataset: Android malware under a microscope," in *The {LASER} Workshop: Learning from Authoritative Security Experiment Results ({LASER} 2016)*, 2016, pp. 1–12.

[21] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[22] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 252–276.

[23] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and lstm," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.

[24] N. Marastoni, A. Continella, D. Quarta, S. Zanero, and M. D. Preda, "Groupdroid: Automatically grouping mobile malware by extracting code similarities," in *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, 2017, pp. 1–12.

[25] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A modular reusable verifier for object-oriented programs," in *International Symposium on Formal Methods for Components and Objects*. Springer, 2005, pp. 364–387.

[26] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[27] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[28] Z. Rakamarić and M. Emmi, "Smack: Decoupling source language details from verifier implementations," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 106–113.

[29] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 95–109.

[30] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[31] Guardsquare, "Dexguard," 2020, [accessed 14-June-2020]. [Online]. Available: https://www.guardsquare.com/en/products/dexguard

[32] P. Faruki, A. Bharmal, V. Laxmi, M. S. Gaur, M. Conti, and M. Rajarajan, "Evaluation of android anti-malware techniques against dalvik bytecode obfuscation," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 414–421.

[33] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[34] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[35] K. Tam, A. Feizollah, N. B. Anuar, R. Salleh, and L. Cavallaro, "The evolution of android malware and android analysis techniques," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–41, 2017.