SimiDroid: Identifying and Explaining Similarities in Android Apps

Li Li, Tegawendé F. Bissyandé, Jacques Klein
Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg
{li.li, tegawende.bissyande, jacques.klein}@uni.lu

***Abstract*—App updates and repackaging are recurrent in the Android ecosystem, filling markets with similar apps that must be identified and analyzed to accelerate user adoption, improve development efforts, and prevent malware spreading. Despite the existence of several approaches to improve the scalability of detecting repackaged/cloned apps, researchers and practitioners are eventually faced with the need for a comprehensive pairwise comparison to understand and validate the similarities among apps. This paper describes the design of SimiDroid, a framework for multi-level comparison of Android apps. SimiDroid is built with the aim to support the understanding of similarities/changes among app versions and among repackaged apps. In particular, we demonstrate the need and usefulness of such a framework based on different case studies implementing different analyzing scenarios for revealing various insights on how repackaged apps are built. We further show that the similarity comparison plugins implemented in SimiDroid yield more accurate results than the state-of-the-art.**

I. Introduction

Android OS has attracted a considerable number of developers and users in recent years. App markets are thus now filled with millions of diversified Android apps offering similar functionalities. While many of such apps are revised versions of one another that are distributed by the same developers to meet user requirements on updated functionalities or to adapt to third-party market opportunities, a large proportion of apps however represent cloned or repackaged versions built by third party developers to redirect advertisement revenues or to efficiently construct and spread malware [1].

The literature has recently proposed a large body of works dealing with the detection of cloned/repackaged apps in the Android ecosystem [2]–[4]. Such works generally output a verdict (Yes/No) on whether an app is a repackaged version of another, without actionable details on how the decision was made and where the similarity lies. Yet, there is a need in the research, development and even user communities for understanding the differences among app versions. For example, market maintainers and users often need to identify what has been modified in the latest app release, in order to ensure that the updated code is in line with the "what's new" descriptions. Developers can benefit from casual impact analyses assessing whether some specific modifications may impact app ratings. Finally, researchers can build change recommendation approaches by mining app versions, and propose detection approaches for locating malicious payloads in repackaged malware samples.

Unfortunately, the state-of-the-art on repackaged/clone app detection builds on internal heuristics are tedious to replicate, while the associated prototype tools are not available for furthering research in these directions [5]. Most of repackaged app detection works [3], [6] indeed do not come with reusable tools for the research community. To the best of our knowledge, Androguard [7] and FSquaDRA [8] are the main publicly available tools for app similarity analysis. The former performs pairwise comparison at the Dalvik bytecode level while the latter conducts its similarity analysis based on resource files. Both approaches however do not offer any explanation on the differences among similar apps, thus failing to provide opportunities for further analysis.

Detecting repackaged apps is a challenging endeavour. In recent years, the community has focused on meeting market scalability requirements with approaches that leverage fast resource-based similarity comparisons or machine learning techniques. Nevertheless, the results of such approaches must eventually be vetted and further broken down via a pairwise comparison of suspicious repackaging pairs.

In this work, we propose to fill the gap in repackaged app research by designing and prototyping a framework for automated, comprehensive, multi-level identification of similarities among apps with facilities for explaining the differences and similarities. SimiDroid is designed as a plugin-based framework integrating various comparison methods (e.g., code-based comparison at the statement level or at the component level, and resource-based comparison). By considering various aspects in a pairwise similarity check, SimiDroid offers opportunities for a fine-grained comprehension of app updating and repackaging scenarios.

Overall, in this paper, we make the following contributions:

- We present the design of SimiDroid, contributing with a reusable tool to the community for detecting similar Android apps and explaining the identified similarities at different levels which can be further enriched via plugin implementations. SimiDroid is publicly available at [9].
- We have implemented several similarity comparison methods as plugins for the current release of SimiDroid. These methods are borrowed from descriptions in the state-of-the-art literature, covering code-based and resource-based similarity comparisons.
- Finally, we investigate a number of case studies on real-world apps to demonstrate the suitability of SimiDroid in providing explanation hints for different usage scenarios.

II. Approach

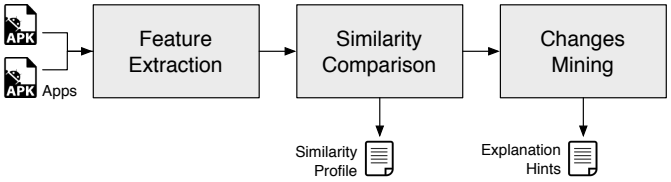
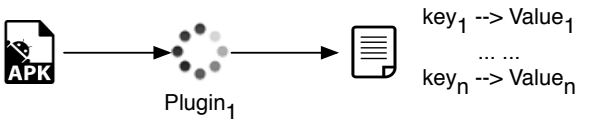


Fig. 1. An Overview of the Working Process of SimiDroid.

Our objective is to provide to the community an extensible framework for supporting the comprehension of similarities among Android apps. The framework aims at contributing to answer to questions such as "to what extent app X and app Y are similar?" and "what are the changes that have been applied to app X in order to build app Y?". We expect the answers to these questions to consider different aspects of Android app packages and to propose different granularity of details.

We design SimiDroid as a plugin-based system which can independently load various comparison techniques at different levels. As introduced earlier, SimiDroid implements pairwise comparison schemes to dissect the similarities and differences among suspected updates of app pairs. Figure 1 illustrates the overall working process in SimiDroid. Two apps are provided as inputs and SimiDroid yields a *similarity* profile and some explanation hints as output. The similarity profile summarizes similarity facts relating to the similarity scores at different levels. The explanation hints highlight the detailed changes revealing the differences among the apps (e.g., string encryption has been applied).

SimiDroid works in three steps by first extracting the necessary features, then based on them to generate a similarity profile for the compared two apps, and finally to mine changes for providing hints for analysts to explain the similarities (or dissimilarities). We now detail these three steps in Section II-A, Section II-B, and Section II-C respectively.



key/value concrete example:

`key: setSortOrderSummary()`

`value: {`**InvokeStmt, AssignStmt|0, InvokeStmt|2131099690**`}`

Fig. 2. The Working Process of a Plugin of SimiDroid. The key/value Concrete Example is Extracted from App *FFE44A*, for which We will Provide more Details on how the Value is Formed in Fig. 3.

A. Feature Extraction

A plugin implements a similarity computation approach by providing heuristics for extracting the features that it considers for comparing apps. In general, a SimiDroid plugin provides a representation of an app as a set of key/value mappings of the selected features. Figure 2 illustrates the case of a plugin considering code statements as features.

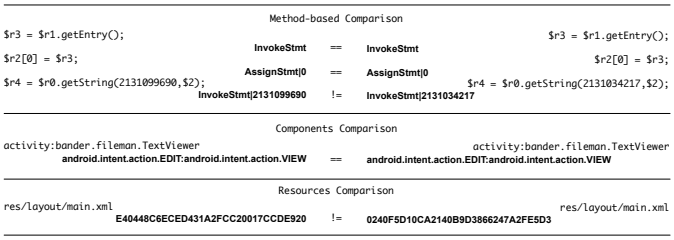


Fig. 3. Examples on Method-based, Component-based, and Resource-based Comparison. The Compared Two Apps are *FFE44A* (Left) and *1CA20C* (Right). The Code Snippet Shown in the Method-based Comparison block is Extracted from Method *setSortOrderSummary()*.

With this schema, SimiDroid offers a straightforward way for practitioners to integrate new plugins implementing comparisons that take into account a variety of app aspects. In practice, there are a few *classes* that could be extended (overriding some methods) to integrate the plugin logic (i.e., how features are extracted) into the framework. Currently, we have developed three different plugins in SimiDroid implementing similarity computation following the aspects suggested by the literature: method-based comparison, component-based, and resource-based comparison. Fig. 3 showcases pairwise comparison results on these aspects, for which we now detail them as follows:

MPlugin - *Method-based comparison:* The first plugin implements a common similarity computation method based on app code, at the level of methods. We design the feature extraction of this plugin to yield method signatures and abstract representations of statements. The latter representations are derived from statement's type (e.g., `if-statement`, `invoke-statement`) instead of the exact statement string. These features have been introduced in previous work [1] not only to implement fast pairwise comparison but also to be resilient, to some extent, to obfuscation, i.e, the comparison will not be impacted in cases where variable names differ but will be impacted in cases where code structure changes (e.g., hide the real method call through reflection [10]). MPlugin further extracts all constants (numbers and strings) as features for comparison.

The first block of Fig. 3 presents a concrete example on how method values (statement types in particular) are formed and compared. By considering constant strings/numbers, SimiDroid is capable of identifying fine-grained changes. For example, as shown in Fig. 3, SimiDroid spots that the constant number in the *getString()* method call is different between the pair of apps, giving hints for analysts on where to focus to understand the motivation behind the change (e.g., the value of $\$r4$ could eventually be changed).

CPlugin - *Component-based comparison:* The second plugin extracts app features at the component level, where key/value mappings are inferred from component names, and other Android package information that are component capabilities including *action*, which describes the type of

behaviour matched by the component (e.g., MAIN component) and *category*, which specifies what the component represents (e.g., LAUNCHER). CPlugin, although it appears to offer a higher-level overview than MPlugin, can be leveraged to better understand the types and capabilities of malicious piece of code injected into piggybacked apps [1].

The second block of Fig. 3 presents a concrete example on how components are compared. This comparison will identify changes in the capabilities reported of an existing or a new component, providing hints to further the analysis when there is a suspicion on the mismatch between one app behaviour and the capability exposed by the other. For example, if the LAUNCHER component is switched from one component to another, there is a hint of piggybacked app writer that intends to divert user attention for triggering malicious code execution.

RPlugin - *Resource-based comparison:* The third plugin builds on resource file comparisons to detect similar apps. The assumption in the literature is that, during repackaging and cloning, these files are unlikely to be modified. Although, some recent experiments have shown that resource files can be manipulated during app repackaging, such modifications are generally not extensive. The feature extraction process generates key/value mappings using hash values of the files' content. RPlugin can thus identify when a resource file has been "compromised" (e.g., as shown in the third block of Fig. 3, the resource files share the same name but have different hashes).

B. Similarity Comparison

At the end of the feature extraction step, for a given pair of Android apps (app_1, app_2), SimiDroid conducts the similarity comparison on top of the two sets of extracted key/value mappings (map_1 and map_2). The computation is implemented in SimiDroid to quantify and qualify the extent of similarity between the pair of apps. We adopt the following four metrics to measure similarity:

- **identical,** when a given key/value entry is matched exactly the same in both maps. For example, given $key_x \in keys(map_1)$, we consider it as identical as long as it exists also in map_2 and its value is exactly the same between the two compared maps, (i.e., $map_1[key_x] = map_2[key_x]$).
- **similar,** when a given key/value entry slightly varies from one app to the other in a pair, more specifically when the key is the same but values differ. For instance, given an entry from app_1 with key $key_x \in keys(map_1)$, we consider it to be similar to an entry from app_2 when key_x exists also in map_2 but its value is different from the one in map_1 (i.e., $map_1[key_x] \neq map_2[key_x]$).
- **new,** when a given key/value entry exists only in map_2 but not in map_1. Thus, given a key $key_x \in keys(map_2)$, we consider it as new as long as it does not exist in map_1 (i.e., $key_x \notin keys(map_1)$).
- **deleted,** when a given entry existed in map_1, but is no longer found in map_2. For instance, give a key $key_x \in keys(map_1)$, we consider it as deleted as long as it does not exist in map_2 (i.e., $key_x \notin keys(map_2)$).

Based on these metrics, we can now compute the similarity score of the given two apps (app_1, app_2) using Formula 1. Given a pre-defined threshold t, which can be computed based on a set of known repackaging pairs, it is then possible to conclude with confidence that the given two apps are similar (i.e., $similarity \geq t$).

$$similarity = max\{\frac{identical}{total - new}, \frac{identical}{total - deleted}\} \quad (1)$$

where

$$total = identical + similar + deleted + new \quad (2)$$

We remind the readers that this similarity comparison step is generic and common to all plugins. Thus, plugin developers do not need to modify the implementation of this step for supporting the similarity analysis of their plugins. However, in order to explain beyond the current metrics, which illustrates what entries are kept, modified, newly added or deleted, developers are enabled to extend this step as well for performing more fine-grained similarity analyses and therefore providing more detailed explanations.

C. Changes Mining

Finally, SimiDroid attempts to mine the changes, based on the generated similarity profile, to provide hints for analysts to quickly identify and thus explain the similarities between compared Android apps. This changes mining module cannot be fulfilled without the support of plugins integrated to SimiDroid. Plugin developers are expected to provide necessary auxiliary code in order to support this module to hunt for changes. These auxiliary code can be added before or after the similarity comparison. In order to achieve that, SimiDroid provides callback methods for plugin developers to implement (i.e., pre-comparison callback for such auxiliary code that needs to be executed before the similarity comparison and post-comparison callback for such auxiliary code that needs to be executed after the comparison). As an example, in order to perform a similarity analysis without considering the appearance of common libraries for our method-based comparison plugin, we implement a pre-comparison callback to exclude common libraries, where the pre-comparison callback will be excluded before the similarity comparison is conducted.

In the current implementation of MPlugin (i.e., the Method-based comparison plugin), we have implemented a post-comparison callback for inferring the changes between two similar methods. Information on those changes can provide fine-grained explanations on what has been modified between the considered pair of apps and, to some extent, why those changes are made. As a use case, given a pair of similar apps ($a_1 \rightarrow a_2$), where a_2 is a piggybacked version of a_1 with some malicious payloads injected, by inferring the changes between similar methods, we would be able to understand how the injected malicious payloads are triggered.

Consider the example depicted in Listing. 1 representing a code snippet extracted from an Android app whose sha256 starts with *DB2CB6*[1]. The added line (starting with '+' symbol) is actually a hook, i.e., a piece of code injected to trigger the malicious payload, during execution of original benign code (here from an app whose sha256 starts with *FFDE8B*). This example illustrates that the malicious payloads could be triggered by a single method call. By following the execution path of this hook, analysts can locate the malicious payload and understand the grafted malware behaviour.

For CPlugin (i.e., the Component-based plugin implementation), we have also implemented a post-comparison callback to check if the newly added components have shared the same capabilities as such of the original components. Doing so is indeed suspicious since there is no need for a benign app to implement several components with the same capabilities (e.g., two PDF reader components in the same app). Consider again the piggybacked app (*DB2CB6*) whose code excerpt was provided in the previous example. The analysis has revealed that this app has declared two broadcast receivers (cf. lines 1 and 8 of Listing 2) to be notified of both *PACKAGE_ADDED* and *CONNECTIVITY_CHANGE* events. In other words, when one of these two events comes, both components (receivers) will be triggered to handle the events. Such a behaviour is suspicious as in a typical development scenario, there is no need for a duplication of event listening.

```
1 class UnityPlayerProxyActivity extends Activity {
2 protected void onCreate(Bundle) {
3     specialinvoke $r0.onCreate($r1);
4 + staticinvoke <com.gamegod.touydig: void
      init(android.content.Context)>($r0);
5   $r2 = newarray (java.lang.String)[2];
6 }}
```

Listing 1. A Hook Example (from app DB2CB6).

```
1 receiver: "com.kuguo.ad.MainReceiver"
2  intent-filter
3   action: "android.intent.action.PACKAGE_ADDED"
4   data: "package"
5  intent-filter
6   action: "android.net.conn.CONNECTIVITY_CHANGE"
7
8 receiver: "net.crazymedia.iad.AdPushReceiver"
9  intent-filter
10  action: "android.intent.action.PACKAGE_ADDED"
11  data: "package"
12 intent-filter
13  action: "android.net.conn.CONNECTIVITY_CHANGE"
14 intent-filter
15  action: "android.intent.action.BOOT_COMPLETED"
16 }
```

Listing 2. An Example of Duplicated Component Capabilities.

D. Implementation

SimiDroid, along with the current MPlugin, CPlugin and RPlugin plugins, are implemented in Java. MPlugin, the method-based comparison plugin, is implemented on top of Soot, a framework for analyzing and transforming Java and Android apps [11]. Code statements in MPlugin are processed at the Jimple code level, an intermediate representation (IR) provided by Soot in default. The transformation from Android bytecode to Jimple code is done by Dexpler [12], which has now been integrated into Soot as a plugin. CPlugin, the component-based comparison plugin, leverages the *axml* library to directly extract component information from the compressed *Android Manifest* file in order to facilitate the extraction process.

[1]Through this paper, we uniquely name an app with the first six letters of its sha256.

III. Evaluation

Our evaluation addresses the following research questions:

- RQ1: Can the prototype implementation of SimiDroid detect similar apps in a set of real-world apps?
- RQ2: How does SimiDroid compare with existing tools?
- RQ3: What is the extent of details that SimiDroid can provide to support comprehension of similarities within a pair of apps ?

A. RQ1: Detection

For a start, we acknowledge that the similarity analysis explored by SimiDroid is focused on pairwise comparison, and thus cannot scale to market datasets [1]. For example, for the 2 million apps available on Google Play, there are $C^2_{2*10^6}$ candidate pairs to compare. Therefore, we emphasize at this point that the objective of SimiDroid is not to identify all the similar apps among a large set of apps, but rather to confirm suspicions on a pair of apps and provide details, at different levels, for supporting explanations on their similarity.

We evaluate the detection ability of SimiDroid using an established comprehensive benchmark [1] of piggybacked apps with about 1,000 pairs of apps: each pair is formed by an original benign app and its counterpart piggybacked malware (i.e., a malware built by grafting a malicious payload to the original benign app). The assessment thus consists in computing the capability of SimiDroid to identify each pair in the set. This evaluation is performed based on each of the plugins integrated in SimiDroid.

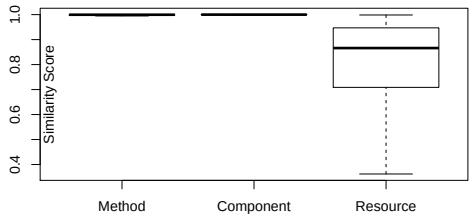


Fig. 4. Distribution of Similarity Scores Computed through Method-based, Component-based, and Resource-based Comparisons.

Fig. 4 shows the distribution of similarity scores that SimiDroid computes for Method-based, Component-based and Resource-based comparisons, where the median values are 0.9996, 1[2], and 0.8661 respectively[3].

[2]Roughly speaking, over 50% of the pairs have no modification at the component level.

[3] Note that on some corner case apps, a plugin may fail to compute the similarity of a given pair (e.g., fail to extract features). We have dropped such pairs from the results.

By far, the similarity scores based on resource-based comparison are lesser than that provided by code-based approaches (including both method and component based comparisons). Using Mann-Whitney-Wilcoxon (MWW) tests, we further confirm that the difference of similarity scores between resource-based and code-based comparison is statistically significant [4]. This finding is also in line with recent findings in the literature [1], revealing that resource files can be extensively manipulated during piggybacking.

Both method and component based comparisons have achieved high similarity scores (cf. Fig. 4), suggesting that app cloning will unlikely modify the app code in an invasive manner. This finding is also in line with the practice of repackaging and code reuse where repackagers have shown to pay the least efforts in code changes, to allow easier automation of the repackaging process.

The scores of component-based comparison is slightly higher than the scores computed through method-based comparison. This indicates that in contrast to methods, component capabilities are even rarely changed during app cloning. Indeed, in our experiments, 85% of investigated pairs do not modify the component capabilities of the original apps.

In order to present a fair comparative study, we also compute the similarity scores via SimiDroid for a set of 1000 pairs of Android apps, which are randomly selected from Google Play. Since the selection is conducted randomly, we therefore expect that for these pairs the similarity results reported by SimiDroid would be low. Indeed, the median similarities are 0, 0, and 0 for Method-based, Component-based, and Resource-based comparisons respectively, showing that SimiDroid is capable of flagging similar (or dissimilar) Android apps.

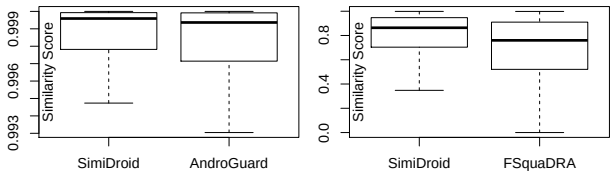
B. RQ2: Comparison

We compare SimiDroid against available implementation of two state-of-the-art works, namely AndroGuard [7] and FSquaDRA [8], covering respectively code-based and resource-based similarity analysis.

AndroGuard. AndroGuard is probably the first available tool presented to the community for detecting the similarity of two Android apps. Like with MPlugin in SimiDroid, the similarity of AndroGuard is computed at the method level and is calculated based on the same four metrics leveraged by SimiDroid (cf. Section II-B). However, the comparison between the content of two methods are different. Instead of comparing all the statements inside a given method, AndroGuard leverages state-of-the-art compressors to compute the similarity distance between two methods. AndroGuard currently uses the Normalized Compression Distance (NCD).

FSquaDRA. FSquaDRA is an approach that detects repackaged Android apps based on the resource files available in app packages. It performs a quick pairwise comparison with an attempt to measure how many identical resource files are shared by a candidate pair of apps.

We run both AndroGuard and FSquaDRA on the same benchmark ($\approx 1,000$ pairs that we have used in previous RQ. Fig. 5 comparatively plots the distribution of similarity scores calculated by SimiDroid, AndroGuard, and FSquaDRA, respectively. The similarity results computed by the state-of-the-art works are also in line with the conclusions reached previously in answering RQ1: code-based similarity results (i.e., AndroGuard) are generally better than resource-based similarity results (i.e., FSquaDRA). We have also confirmed that the differences are significant using MWW tests at the significance level of 0.001.



(a) Code-based. (b) Resource-based.

Fig. 5. Comparison Results among the Similarity Scores of SimiDroid (Code-based), AndroGuard, and of SimiDroid (Resource-based), FSquaDRA.

As shown in Fig. 5, the median value of SimiDroid is slightly higher than the median value of AndroGuard, although the difference between the two is not statistically significant when checked with MWW tests (i.e., p-value > 0.001). In order to compare the precision of these two code-based similarity analysis tools, we plan to manually compare the results yielded by these two apps. To this end, we randomly select ten pairs for manual investigation. Table I enumerates the randomly selected pairs.

In this work, instead of manually investigating all the methods, which needs a lot of efforts and is hard to perform in practice, we decide to focus only on the reported *similar* methods. Those *similar* methods are actually quite suitable for our purpose, as they have embraced the exact changes between the compared two apps. As shown in Table I, 8 out of 10 of the selected app pairs share the same number of similar methods (per pair) by both AndroGuard and SimiDroid. We then manually investigate the cases of app pairs where the reported numbers of *similar* methods differ by AndroGuard and SimiDroid. We found that this is mainly due to false negative results of AndroGuard, which has failed to report a *similar* method for both cases. We now provide more details for these two candidate pairs.

Case Study 1: *FFDE8B* $\rightarrow$ *DB2CB6*. For this app pair, SimiDroid reports two similar methods while AndroGuard reports only one similar method. The two similar methods reported by SimiDroid are *onCreate()* in class *UnityPlayerProxyActivity* and *onDestroy()* in class *UnityPlayerActivity*. We have shown in Listing 1 as a motivating example that the first similar method, namely *onCreate()*, has indeed been manipulated to trigger the execution of package *com.gamegod.touydig*. Now we present the code snippet of

[4]The reported p-value indicates that the difference is significant at a significance level $\alpha = 0.001$. Because p-value $< \alpha$, there is one chance in a thousand that the difference between the compared two datasets is due to a coincidence.

TABLE I
THE RANDOMLY SELECTED TEN PIGGYBACKING PAIRS AND THEIR CODE-BASED SIMILARITY RESULTS YIELDED BY ANDROGUARD AND SIMIDROID.

		AndroGuard					SimiDroid				
Original	Piggybacked	Identical	Similar	New	Deleted	Score	Identical	Similar	New	Deleted	Score
FFDE8B	DB2CB6	618	1	875	0	99.84%	1043	2	1300	0	99.81%
2326A8	7D6D97	1422	0	1727	0	100.00%	2445	1	4384	0	99.96%
E2CEED	E9B8EE	264	5	1178	0	98.14%	390	5	2299	0	98.73%
8C23C6	5ADAE7	92	1	950	0	98.92%	124	1	1730	0	99.20%
A0087E	296792	3143	1	276	0	99.97%	7090	1	460	0	99.99%
1B8441	172F27	3965	1	834	0	99.97%	8488	1	1300	0	99.99%
00C381	2DC271	905	1	294	0	99.89%	1418	1	460	0	99.93%
93E50D	664F22	1225	1	1210	0	99.92%	2042	1	2786	0	99.95%
9E49AE	29A23A	1386	1	1000	0	99.93%	2172	1	1892	0	99.95%
321DA9	86E88F	829	1	184	0	99.88%	1390	1	474	0	99.93%

```
1  //Case Study 1: Pair (FFDE8B, DB2CB6)
2  class UnityPlayerActivity extends Activity {
3   protected void onDestroy() {
4    $r0 := @this: UnityPlayerActivity;
5    specialinvoke $r0.<Activity: void
       onDestroy()>();
6  + staticinvoke <touydig: void
       destroy(Context)>($r0);
7    $r2 = $r0.<UnityPlayerActivity: UnityPlayer a>;
8    virtualinvoke $r2.configurationChanged($r1);
9    return;
10 }}
11 //Case Study 2: Pair (2326A8, 7D6D97)
12 class SocialPluginUnityActivity extends
       UnityPlayerActivity {
13   public void onCreate(android.os.Bundle) {
14    $r0 := @this: SocialPluginUnityActivity;
15    $r1 := @parameter0: android.os.Bundle;
16    specialinvoke $r0.onCreate(bundle)($r1);
17 +  virtualinvoke $r0.dywtsbn();
18    return;}
19 + public void dywtsbn() {... ...}
20 }
21 //Case Study 3: Pair (EF2BDA, 87880D)
22 class Start extends Activity {
23  void callAdds() {
24   $r1 = $r0.<Start: AdView adView>;
25 - virtualinvoke
       $r1.setAdUnitId("a1522d5c390a573");
26 + virtualinvoke $r1.setAdUnitId(String)(
       "ca-app-pub-8182614411920503/1232098473");
27 }}
```

Listing 3. Illustrative code snippets extracted from real Android apps.

the second similar method, namely *onDestroy()*, in Listing 3, where one statement (line 6) has been added to the original app. The purpose of this injection is to clean the changes due to the execution of injected malicious payloads, which are triggered by the first similar method *onCreate()* (cf. Listing 1).

Case Study 2: *2326A8 → 7D6D97*. For this candidate pair, AndroGuard reports no similar method while SimiDroid yields one similar method, which is *onCreate()* of class *SocialPluginUnityActivity*. Through manual investigation, as shown in Listing 3, we confirm that *onCreate()* of class *SocialPluginUnityActivity* is indeed a similar method which has been tampered with to insert a call to *dywtsbn()*, implemented as part of the newly injected payload within the same class as *onCreate()*.

C. RQ3: Support for Comprehending Repackaging/Cloning changes

We now investigate the enabling potential of SimiDroid for comprehending the details in Android app similarities. To the best of our knowledge, little work has focused on systematizing the explanation of similarities among apps.

On top of the *detection* module (i.e., feature extraction plugin + similarity comparison plugin), a *change mining* module implements specified analyses (before or after the comparison) for providing insights into the nature and potential purpose behind the changes. Those analyses are specified by leveraging archived knowledge from the literature and can be extended by practitioners based on their manual investigation findings. We now enumerate and discuss several analysis directions that are currently implemented in SimiDroid and that have been used i) to characterize suspicious intent in repackaging, ii) to recognize symptoms of piggybacking, iii) to hint on malicious payload code, or iv) to measure the impact of library code in app similarity computation.

TABLE II
EXPLANATION STATISTICS.

Explanation Type	#. of Pairs	#. of Times
Constant String Mismatch	110	476
Constant Number Mismatch	122	2,447
New Method Call	523	2,259
Library Impact	422	422
Duplicated Component Capability	611	60,312
Resource File Rename	160	994

1) Constant String Replacement:
Online documentation of advertisement integration into Android app exposes how ad revenues is forwarded on the basis of an ad ID tied to the app owner. We have implemented an analysis in SimiDroid that focuses on changes related to constant string replacement: we focus on cases where only the string varies while the associated code statement (i.e., statement type and statement context method) does not vary. This analysis hinted on a suspected case of a redirection in ad revenues, illustrated by the following case study.

Case Study 3: *EF2BDA → 87880D* (Redirect ad revenue). When repackaging app *EF2BDA* into *87880D*, attackers have also changed the ad ID ('a1522d5c390a573' in *EF2BDA*) to

match their own (line 36 in Listing 3) on the call to API method *setAdUnitId()*, so as to redirect the revenue generated by app *EF2BDA*.

The constant string replacement analysis has also allowed to confirm obfuscation of code to prevent repackaging detection. In addition to constant strings, SimiDroid also harvests the replacement of constant numbers between similar methods. The method-based comparison in Fig. 3 has actually demonstrated the case where a constant number in a method of app *FFE44A* is updated in app *1CA20C*, leading eventually to a change in the selected entry. As shown in Table II, SimiDroid has identified 476 cases (within 110 pairs) where constant strings are replaced and 2,447 cases (within 122 pairs) where constant numbers are replaced among the evaluated benchmark pairs (nearly 1000).

2) New Method Call:
A new method call in a cloned app code is a relevant starting point for tracking a potential injected payload. Indeed, repackagers, as established in a previous study [1], often modify existing code to insert a single method call for triggering the redirection of control flow from the execution of original benign code into the newly added (likely malicious) code. Listing 3 shows examples of such method call insertions identified by SimiDroid at key points of an Android program, i.e., when an activity is created/launched (line 18) or when it must be stopped/destroyed (line 6). Actually, SimiDroid has found 2,259 cases (within 523 pairs) where *new method call* is introduced during repackaging (cf. Table II).

3) Library Impact:
As shown by Li et al., the presence of common libraries can cause both false positives and false negatives when attempting to detecting repackaged/cloned apps [13]. We have specified a change analysis after the identification of similarities to further differentiate changes within libraries from those within app core code. We thus use a library exclusion filter based on a whitelist of libraries borrowed from [13]. Among the analyzed pairs, SimiDroid reports different similarity scores for 422 pairs when common libraries are excluded (cf. Table II). This analysis further allows to avoid false positives and to reduce the rate of false negatives in making a detection decision on whether two apps constitute a repackaging pair.

Case Study 4: *29C2D4* $\rightarrow$ *287198* (False Positive). By considering common libraries, the similarity of these two apps is 86%. Giving a threshold of 80%, we have reasons to believe that these two apps are cloned from one another. However, after excluding common libraries, the similarity of these two apps falls down to 0, demonstrating that a naive similarity analysis could be misled by common libraries and yield false positive results.

Case Study 5: *F3B117* $\rightarrow$ *25BC25* (False Negative). After excluding common libraries, the similarity of these two apps reaches to 84%, leading to a decision that these apps constitute a repackaging pair (if we consider also 80% as the threshold). Comparing to the case where libraries are considered (47% similarity score), one would have missed the chance to suspect the pair of apps, resulting in a false negative result.

4) Duplicated Component Capabilities:
Building on findings in the literature [14], we identify hints on repackaging in similar apps by focusing on duplication in Manifest entries. In particular, duplicated component capabilities can be taken as a symptom to quickly confirm piggybacking as it is indeed suspicious for a normal benign app, developed from scratch, to implement several components that listen to a same event, or that can realize a same action (e.g., play videos). In our experiments, we have shown (cf. Listing 2 example) fine-grained changes in 611 piggybacking apps presenting such a symptom, in contrast to their original counterparts.

Case Study 6: *3FC49C* $\rightarrow$ *A02FE8* (Duplicated Capabilities). When analyzing this pair, SimiDroid yields surprisedly 45,682 duplicated capability cases, which are mainly contributed by action *android.intent.action.VIEW*, which has been declared in total 243 times for 213 components ($A^2_{213} = 45,156$).

IV. Related Work

The related work of this paper lies mainly in two folds: 1) identifying similar Android apps and 2) explaining similar Android apps. We now detail them in Section IV-A and Section IV-B, respectively.

A. Identifying Similar Android Apps

Similarity identification of Android apps, which is also referred by literature works as repackaged/cloned apps identification (or reuse/plagiarize detection), has been recurrently addressed by state-of-the-art works. As an example, AndroidSOO [15] leverages the *string offset order* symptom to quickly flag if a given Android app is repackaged. Similarly, Li et al. show that *duplicated permissions* and *duplicated capabilities*, which can be extracted from the Android manifest file, could be also taken as reliable symptoms to achieve the same purpose [1]. Excepting symptom-based approaches, researchers also rely on dynamic analysis to identify similar Android apps (e.g., DroidMarking [16]).

Another recent direction of detecting similar Android apps is to leverage machine learning based techniques. Indeed, both supervised learning [17]–[19] and unsupervised learning [20]–[22] have been investigated by state-of-the-art works. As an example of supervised learning, DroidLegacy [18] takes the frequency of API calls as features to conduct 10-fold cross validation for the purpose of automatically classifying malware samples, including repackaged ones. As an example of unsupervised learning, ResDroid [20] adopts a clustering-based approach to coarsely group similar Apps into same clusters, so as to reduce the computing space of other fine-grained comparison approaches.

All the aforementioned approaches, which detect similar apps in a way that they do not need the knowledge of original apps. The results of those approaches, however, also need to be vetted through a comprehensive pairwise comparison (e.g., to confirm the final accuracy). Actually, like SimiDroid, the

majority work in detecting similar Android apps at the moment are still based on pairwise similarity comparison [3], [6], [13].

However, these approaches do not provide a means for analysts to quickly explain how and why the compared two apps are similar (or dissimilar). SimiDroid is thus presented to fill this gap, aiming for not only detecting similar Android apps but also explaining why a given two apps are similar (or dissimilar).

B. Explaining Similar Android Apps

To the best of our knowledge, there is no systematized work on explaining similarities in Android apps. However, there do have several works that perform manual or empirical understanding related to similarity of Android apps. The most advanced work is presented recently by Li et al., who have empirically dissected the piggybacking processes of Android apps [1]. Unfortunately, their empirical investigations are mainly done in manual and there is no supporting tool associated. Our work, namely SimiDroid, can actually be leveraged to support their findings.

Despite the piggybacking processes, researchers are also interested in understanding code reuse in Android markets. Indeed, Ruiz et al. [23], [24] have empirically investigated thousands of apps across five different categories, in an attempt to understand the code reuse (in class level) of Android apps. Li et al. [13] and Linares-Vasquez et al. [25] have investigated the Android reuse studies in the context of library usages. As experimentally illustrated by Li et al., the appearance of common libraries could cause both false positives and false negatives for detecting piggybacked apps.

The objective of this paper is to provide a generic framework for automated, comprehensive, and multi-level identification of similarities (or reuses) among apps. Our work, along with other plugins, can be taken as a keystone for supporting the replication of existing similarity-based studies and for facilitating the development of new similarity-based studies.

V. Conclusion

We introduce a new framework, SimiDroid, for supporting researchers and practitioners in the analysis of similar apps. SimiDroid integrates plugins implementing the extraction of features, at different level, for the computation of pairwise similarity scores. This framework is targeted at confirming that two apps are indeed similar and at detailing not only the similarity points but also the modifications in changed code.

Using a benchmark of piggybacking pairs, we have shown how SimiDroid is accurate in detecting similar apps, and the extent to which it can support the analysis of changes performed by malicious app writers when repackaging a benign app. With this framework, we contribute to supporting the community in the realisation of extensive studies on app similarities to further experiment in their fast, accurate and scalable approaches.

Acknowledgments

This work was supported by the Fonds National de la Recherche (FNR), Luxembourg, under projects AndroMap C13/IS/5921289 and Recommend C15/IS/10449467.

References

[1] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. Understanding android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics & Security*, 2017.

[2] Jian Chen, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. Detecting android malware using clone detection. *JCST*, 2015.

[3] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *ISSTA*, 2015.

[4] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE*, 2014.

[5] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 2017.

[6] Quanlong Guan, Heqing Huang, Weiqi Luo, and Sencun Zhu. Semantics-based repackaging detection for mobile apps. In *ESSoS*, 2016.

[7] Anthony Desnos. Android: Static analysis using similarity distance. In *HICSS*, 2012.

[8] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. Fsquadra: fast detection of repackaged applications. In *DBSec*, 2014.

[9] Simidroid. https://github.com/lilicoding/SimiDroid.

[10] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *ISSTA*, 2016.

[11] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *CETUS*, 2011.

[12] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *SOAP*, 2012.

[13] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. An investigation into the use of common libraries in android apps. In *SANER*, 2016.

[14] Li Li, Daoyuan Li, Tegawendé François D Assise Bissyande, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting malicious code from piggybacked android apps. Technical report, SnT, 2016.

[15] Hugo Gonzalez, Andi A Kadir, Natalia Stakhanova, Abdullah J Alzahrani, and Ali A Ghorbani. Exploring reverse engineering symptoms in android apps. In *EuroSec*, 2015.

[16] Chuangang Ren, Kai Chen, and Peng Liu. Droidmarking: Resilient software watermarking for impeding android application repackaging. In *ASE*, 2014.

[17] Ke Tian, Danfeng (Daphne) Yao, Barbara G. Ryder, and Gang Tan. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *MoST@S&P (W)*, 2016.

[18] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: automated familial classification of android malware. In *PPREW@POPL (W)*, 2014.

[19] Ying-Dar Lin, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *CompSec*, 2013.

[20] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. In *ACSAC*, 2014.

[21] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar android applications. In *ESORICS*, 2013.

[22] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *AsiaJCIS*, 2012.

[23] Israel J. Ruiz, Meiyappan Nagappan, Bram Adams, and Hassan Ahmed E. Understanding reuse in the android market. In *ICPC*, 2012.

[24] Israel J. Ruiz, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. A large scale empirical study on software reuse in mobile apps. *IEEE Software*, 2014.

[25] Mario Linares-Vasquez, Andrew Holtzhauer, Carlos Bernal-Cardenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. In *MSR*, 2014.