Li L, Bissyandé TF, Wang HY *et al.* On identifying and explaining similarities in Android apps. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 34(2): 437–455 Mar. 2019. DOI 10.1007/s11390-019-1918-8

On Identifying and Explaining Similarities in Android Apps

Li Li[1], Tegawendé F. Bissyandé[2], Hao-Yu Wang[3], and Jacques Klein[2]

[1]*Faculty of Information Technology, Monash University, Melbourne 3168, Australia*
[2]*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg 2721, Luxembourg*
[3]*School of Computer Science, Beijing University of Posts and Telecommunications, Beijing 100876, China*

E-mail: li.li@monash.edu; tegawende.bissyande@uni.lu; haoyuwang@bupt.edu.cn; jacques.klein@uni.lu

Received March 5, 2018; revised December 7, 2018.

Abstract App updates and repackaging are recurrent in the Android ecosystem, filling markets with similar apps that must be identified. Despite the existence of several approaches to improving the scalability of detecting repackaged/cloned apps, researchers and practitioners are eventually faced with the need for a comprehensive pairwise comparison (or simultaneously multiple app comparisons) to understand and validate the similarities among apps. In this work, we present the design and implementation of our research-based prototype tool called SimiDroid for multi-level similarity comparison of Android apps. SimiDroid is built with the aim to support the comprehension 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 dissection 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.

Keywords Android, similarity analysis, app clone

1 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[1,2] or to efficiently construct and spread malware[3−5].

The research community has recently proposed a large body of studies dealing with the detection of cloned/repackaged apps in the Android ecosystem[6−9]. Such studies 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 for 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 or cause apps to be removed from markets[5]. Finally, researchers can build change recommendation approaches by mining app versions, and propose detection approaches for locating malicious payloads in repackaged malware samples[10].

Unfortunately, the state-of-the-art studies on repackaged/clone app detection built on internal heuristics are tedious to replicate, while the associated pro-

Regular Paper

A preliminary version of the paper was published in the Proceedings of TrustCom 2017.

This work was supported by the Luxembourg National Research Fund (FNR), Luxembourg, under Grant Nos. CHARACTERIZE C17/IS/11693861 and Recommend C15/IS/10449467, and also by the National Natural Science Foundation of China under Grant No. 61702045 and the Beijing University of Posts and Telecommunications (BUPT) Youth Research and Innovation Program of China under Grant No. 2017RC40.

totype tools are not available for furthering research in these directions[11]. Most of repackaged app detection studies[6–9,12–44] indeed do not come with reusable tools for the research community. To the best of our knowledge, Androguard[45] and FSquaDRA[46] 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., the 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 such as understanding the evolution of Android app vulnerabilities[47] or dissecting piggybacked malicious Android apps[3].

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

• We present the design and implementation of SimiDroid, contributing with a reusable tool[1] to the community for detecting similar Android apps and explaining the identified similarities at different levels which can be further enriched via plugin implementations.

• 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.

This paper is an extended version of a conference paper entitled "SimiDroid: Identifying and Explaining Similarities in Android Apps", which has been published at the 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications. In the previous version, we have introduced our research-based prototype tool called SimiDroid for supporting the identification and explanation of similarities between two given Android apps. In this work, we additionally introduced two parameters to customise the artefacts to-be analysed. These two parameters provide a means for users to perform customised analyses so as to perform similarity analysis in a way that common libraries or certain resource files are not considered. We also extended SimiDroid to support the similarities analysis among multiple Android apps simultaneously, which subsequently allows the users of SimiDroid to cluster Android apps into different categories based on their similarities.

Except for the tool extension, we have refined our experiments with more empirical findings. For example, we enhanced our exploration study in RQ3 (refer to Subsection 4.3) by discussing a new category of changes that are recurrently targeted by repackagers. Furthermore, we added a new research question specifically for evaluating the capability of conducting similarity analysis for multiple Android apps at the same time. Last but not the least, we also updated many sections such as the abstract and introduction sections to reflect the latest state of this work.

The remainder of this paper is structured as follows. Section 2 provides necessary background information on Android app packaging and clarifications on the terminology related to similar Android apps. Section 3 details the architecture and workflow of SimiDroid. Section 4 presents the evaluation results of this work, including the research questions and answers. Section 5 discusses the threats to validity. Section 6 provides concluding remarks.

2 Background

We provide an overview of the structure of an Android app package and clarify the terminology used in the literature related to the topic of similar Android apps.

[1]https://github.com/lilicoding/SimiDroid, Jan. 2019.

2.1 Android App Packaging

An Android app (packaged as an APK file) is actually an archive (i.e., ZIP file) assembling various files (cf. Listing 1), mainly including the followings:

1) a configuration file named *AndroidManifest.xml*: important information such as the declared permissions and the list of components are specified in this file;

2) a bytecode file named *classes.dex* representing the main app code in DEX format;

3) a *res* directory storing various resource files, including pictures and layouts that define the app's "look and feel";

4) others, such as the META-INF directory storing the certificate of authors, and the assets directory which can be used to store raw data.

```
Example.apk
|-- AndroidManifest.xml // main config
|-- classes.dex // main app code
|-- res // resource files
|    |-- drawable
|    |-- layout
|    |-- ... ... // menu, raw, etc.
|-- META-INF // signature
|-- ... ... // lib, assets, etc.
```

Listing 1. Main structure of an Android APK.

This structure simplifies similarity analysis either on the code included in DEX files or on resources available in the dedicated directories.

2.2 Terminology on Similar Android Apps

State-of-the-art studies have used various terms in the literature to refer to the concept of similar Android apps. In particular, reusing (and cloning) is (are) often used to describe the process of leveraging parts or the entirety of existing code to build new programs. This reuse process is also referred to as code plagiarism in markets when third-party developers have no right to exploit other developers' efforts. In the Android community, repackaging is consistently used for referring to the process of cloning Android apps. In this case, developers first unpack the app and then perform necessary changes on the disassembled files before repackaging them back into a new app version (which is now referred to as a repackaged app). Generally, repackaging is different from reusing, where repackaging will likely reuse the original resources (of the original app) while reusing may not necessarily be involved in a repackaging process. Indeed, developers can reuse some code snippets of other apps without repackaging them. Moreover, repackaging processes do not necessarily involve a change in the code of a given app. Only modifying metadata or resource files could be sufficient to divert app ownership and, hence, associated revenues. The literature reserves the term "piggybacking" for repackaging cases where additional code manipulation (e.g., insertion of a malicious payload) is performed on the original app[34].

2.3 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 Subsection 2.3.1 and Subsection 2.3.2, respectively.

2.3.1 Identifying Similar Android Apps

Similarity identification of Android apps, which is also referred to by literature studies as repackaged/cloned apps identification (or reuse/plagiarize detection), has been recurrently addressed by the state-of-the-art work. As an example, AndroidSOO[19] leverages the "string offset order" symptom to quickly flag if a given Android app is repackaged. Similarly, Li *et al.* showed 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[3].

Excepting symptom-based approaches, researchers also rely on dynamic analysis to identify similar Android apps[48]. For example, DroidMarking[29] and AppInk[38] leverage watermarking techniques to check at runtime if the installing app is repackaged from other apps. Comparably, DIVILAR[21] provides a self-defense technique, where the app itself is instrumented with diversified virtual instructions that will eventually be executed in a specialized engine, to mitigate the spread of similar (but fake) apps.

Another recent direction of detecting similar Android apps is to leverage machine learning based techniques. Indeed, both supervised learning[12,23,33] and unsupervised learning[27,36,43] have been investigated by state-of-the-art studies. As an example of supervised learning, DroidLegacy[23] 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[27] 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 attempt to detect similar apps in a way that they do not need the knowledge of original apps. The results of these 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 the literature in detecting similar Android apps at the moment is still based on pairwise similarity comparison[6–9,13–15,18,20,22,24,25,28,30–32,34,37,39–42,44–46,49,50].

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 the given two apps are similar (or dissimilar).

2.3.2 *Explaining Similar Android Apps*

To the best of our knowledge, there is no systematized work on explaining similarities in Android apps. However, there do exist several studies that perform manual or empirical understanding related to the similarity of Android apps. The most advanced work has been presented recently by Li *et al.*, who empirically dissected the piggybacking processes of Android apps[3]. 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, Mojica *et al.*[24,44] 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.*[49] and Linares-Vasquez *et al.*[25] investigated the Android reuse studies in the context of library usages[51,52]. As experimentally illustrated by Li *et al.*[49], 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.

3 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 answering to questions such as "to what extent are app X and app Y 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. Fig.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 related 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).

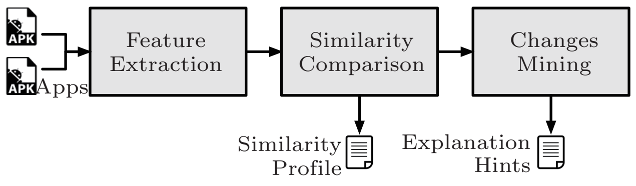
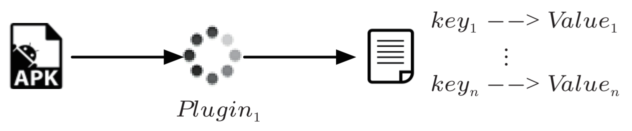


Fig.1. Overview of the working process of SimiDroid.

SimiDroid works in three steps by first extracting the necessary features, then generating a similarity profile for the two compared apps, and finally mining changes for providing hints for analysts to explain the similarities (or dissimilarities). We now detail these three steps in Subsection 3.1, Subsection 3.2, and Subsection 3.3 respectively.

3.1 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 with a set of key/value mappings of the selected features. Fig.2 illustrates the case of a plugin considering code statements as features.



Key/Value Concrete Example:
key: setSortOrderSummary()
value: {InvokeStmt, AssignStmt|0, InvokeStmt|2131099690}

Fig.2. 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.

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 comparison, and resource-based comparison. Fig.3 showcases pairwise comparison results on these aspects, for which we now detail them as follows.

3.1.1 *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 the statement's type (e.g., `if-statement`, `invoke-statement`) instead of the exact statement string. These features have been introduced in the previous work[3] 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., hiding the real method call through reflection[53]). MPlugin further extracts all constants (numbers and strings) as features for comparison.

Fig.3(a) presents a concrete example of 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).

For some cases, where a certain part of the code should not be considered for similarity analysis, e.g., alleviating the impact of common libraries, we should provide the flexibility to support that. To this end, we introduce into this plugin a parameter, namely *LibrarySetPath*, to support customised code-level similarity analysis. The value of parameter *LibrarySetPath* should point to a file containing a list of Java packages

```
$r3 = $r1.getEntry();                                                              $r3 = $r1.getEntry();
                                   InvokeStmt   ==   InvokeStmt
$r2[0] = $r3;                                                                              $r2[0] = $r3;
                                AssignStmt|0   ==   AssignStmt|0
$r4 = $r0.getString(2131099690,$2);                           $r4 = $r0.getString(2131034217,$2);
                     InvokeStmt|2131099690   !=   InvokeStmt|2131034217
```

(a)

```
activity:bander.fileman.TextViewer                                activity:bander.fileman.TextViewer
    android.intent.action.EDIT:android.intent.action.VIEW   ==   android.intent.action.EDIT:android.intent.action.VIEW
```

(b)

```
res/layout/main.xml                                                              res/layout/main.xml
                   E40448C6ECED431A2FCC20017CCDE920   !=   0240F5D10CA2140B9D3866247A2FE5D3
```

(c)

Fig. 3. Examples on (a) method-based, (b) component-based, and (c) 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*().

(one package per each line). When parameter *LibrarySetPath* is enabled, SimiDroid will ignore all the methods under the packages configured via the parameter.

3.1.2 *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 is 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 the malicious piece of code injected into piggybacked apps[3].

Fig.3(b) presents a concrete example of 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.

3.1.3 *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 Fig.3(c), the resource files share the same name but have different hashes).

Similar to *LibrarySetPath*, a parameter introduced in the MPlugin for customising the code to be excluded, we also introduce a parameter, namely *ResExtensionSetPath*, to customise the resource files that are not wanted for the similarity analysis. The value of parameter *ResExtensionSetPath* should point to a file containing a list of file extensions (one per each line). When the parameter is enabled, all the files having extensions configured via the parameter will be ignored by SimiDroid.

3.2 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 to be 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 to be 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, given 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 (1). The similarity score is computed based on the ratio of items that are identical (i.e., kept the same) between the two compared apps, where $total-new$ denotes all the items available in app_1 while $total-deleted$ stands for all the items available in app_2. Consequently, $\frac{identical}{total-new}$ is the retained ratio from app_1's point of view while $\frac{identical}{total-deleted}$ is the unchanged ratio from app_2's point of view. In this work, we simply consider the larger one as the similarity score for the pair. 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., $similarity \geqslant t$).

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

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 illustrate 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.

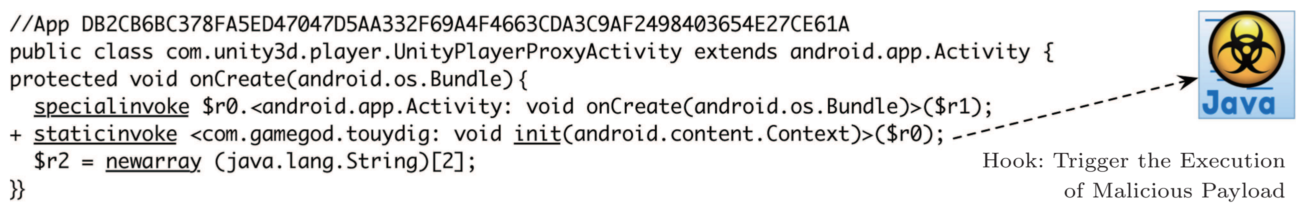
3.3 Changes Mining

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. The 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 Fig.4 representing a code snippet extracted from an Android app whose sha256 starts with DB2CB6[2]. The added line (starting with "+" symbol) is actually a hook, i.e., a piece of code injected to trigger the malicious payload, during the 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.



```
//App DB2CB6BC378FA5ED47047D5AA332F69A4F4663CDA3C9AF2498403654E27CE61A
public class com.unity3d.player.UnityPlayerProxyActivity extends android.app.Activity {
protected void onCreate(android.os.Bundle){
  specialinvoke $r0.<android.app.Activity: void onCreate(android.os.Bundle)>($r1);
+ staticinvoke <com.gamegod.touydig: void init(android.content.Context)>($r0);
  $r2 = newarray (java.lang.String)[2];
}}
```

Fig.4. Hook example.

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

```
receiver: "com.kuguo.ad.MainReceiver"
 intent-filter
  action: "android.intent.action.PACKAGE_ADDED"
  data: "package"
 intent-filter
  action: "android.net.conn.CONNECTIVITY_CHANGE"

receiver: "net.crazymedia.iad.AdPushReceiver"
 intent-filter
  action: "android.intent.action.PACKAGE_ADDED"
  data: "package"
 intent-filter
  action: "android.net.conn.CONNECTIVITY_CHANGE"
 intent-filter
  action: "android.intent.action.BOOT_COMPLETED"
}
```

Listing 2. Example of duplicated component capabilities.

3.4 SimiDroid for Multiple Apps

Towards supporting similarity analysis of multiple Android apps, we go one step further in this work by extending SimiDroid to take as input multiple Android apps that analysts would like to dissect at the same time. Given a set of Android apps, the working process of the extension is straightforward. For every two apps in the given set, SimiDroid performs its default pairwise comparison (for whatever plugins it has configured) as detailed in the previous section (cf. Subsection 3.2) and records its analysis results. The results of each pair are then merged in the end to highlight and thereby explain the similarities (or dissimilarities) of the considered apps. The output of this extension is a matrix which demonstrates the similarity scores of any two given apps and a list of explanation hints that summarize the changes among some of the considered apps. Table 1 presents an example of a possible matrix, which involves the similarity results of four Android apps (i.e., $a1$, $a2$, $a3$, and $a4$).

Table 1. Example of a Similarity Matrix

App	$a1$	$a2$	$a3$	$a4$
$a1$	-	0.81	0.74	0.2
$a2$	0.81	-	0.92	0.3
$a3$	0.74	0.92	-	0.4
$a4$	0.20	0.30	0.40	-

Clustering Similar Apps. Based on the aforementioned similarity matrix, we go one step deeper in this work to group highly similar apps into clusters. More specifically, we consider the matrix representing an undirected graph, where each app represents a node while each similarity between two apps represents an edge (the similarity is then the weight of this edge). To this end, as shown in Fig.5(a), we are able to construct a strongly connected graph. Given a threshold t, we consider two apps are similar as long as their similarity is bigger than t. Then, we break all the edges that connect two dissimilar apps (i.e., their similarity is less than t). After this step, the remaining sub-graphs represent different clusters of similar apps that can be further leveraged to perform more advanced analyses. Indeed, as shown in Fig.5(b), with a threshold at 0.8, four out of six edges are excluded from the graph, resulting in two sub-graphs (i.e., two clusters: one with $a1$, $a2$, $a3$ and the other with $a4$). Despite this clustering approach is straightforward, it does become useful when the number of considered apps increases (e.g., over 1 000) as well as the threshold changes (especially when we need to dynamically adjust it).

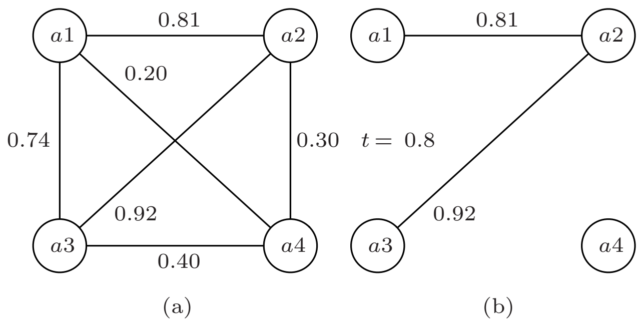


Fig.5. Constructed graphs for clustering similar Android apps.

3.5 Implementation

SimiDroid, along with the current MPlugin, CPlugin and RPlugin plugins, is 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[54]. 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[55], 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.

4 Evaluation

Our evaluation addresses the following research questions.

• *RQ*1. Can the prototype implementation of SimiDroid detect similar apps in a set of real-world apps?

• *RQ*2. Can SimiDroid be used to detect similar apps without taking common libraries into consideration? If so, what is the impact of excluding common libraries on the performance of SimiDroid's similarity analysis?

• *RQ*3. How is SimiDroid compared with existing tools?

• *RQ*4. What is the extent of details that SimiDroid can provide to support the comprehension of similarities within a pair of apps?

• *RQ*5. Can SimiDroid be leveraged to cluster multiple Android apps into categories based on their similarities?

All the experiments discussed in this section are conducted on a Core i7 CPU running a Java VM with 8 GB of heap size.

4.1 RQ1: Detection

For a start, we acknowledge that pairwise similarity analysis in general (including the one explored by SimiDroid) cannot scale to market datasets[3]. For example, for the 2 million apps available on Google Play, there are $C^2_{2\times10^6}$ 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 will show later how SimiDroid is useful for identifying similarities among a (relatively big) number of apps.

We evaluate the detection ability of SimiDroid using an established comprehensive benchmark[3] of piggybacked apps with about 1 000 pairs of apps[3]. 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 into SimiDroid.

Fig.6 shows the distribution of similarity scores that SimiDroid computes for method-based, component-based, and resource-based comparisons, where the median values are 0.999 6, 1[4], and 0.866 1 respectively[5].

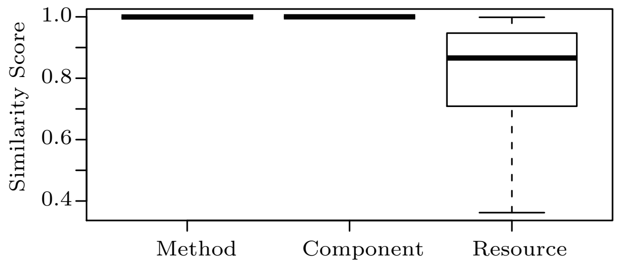


Fig.6. Distribution of similarity scores computed through method-based, component-based, and resource-based comparisons.

By far, the similarity scores based on resource-based comparison are less than those 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[6]. This finding is also in line with recent findings in [3], revealing that resource files can be extensively manipulated during piggybacking.

Both method- and component-based comparisons have achieved high similarity scores (cf. Fig.6), 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 are slightly higher than the scores computed through the 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.

[3]The real apps are downloaded from AndroZoo[56].

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

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

[6]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.

In order to present a fair comparative study, we also compute the similarity scores via SimiDroid for a set of 1 000 pairs of Android apps, which are randomly selected from Google Play. Since the selection is conducted randomly, we expect that for these pairs the similarity results reported by SimiDroid would be low. Indeed, the median similarities are 0, 0, and 0 for the method-based, component-based, and resource-based comparisons respectively, showing that SimiDroid is capable of flagging similar (or dissimilar) Android apps.

*Result of RQ*1. SimiDroid is reliable to detect similar Android apps. In general, code-based similarity analysis is more accurate than resource-based similarity analysis.

4.2 RQ2: Exclusion of Libraries

Recall that we have introduced two parameters for SimiDroid to perform customised similarity analysis, e.g., discarding unwanted code and resource files. We would like to also evaluate the validity of these two parameters. To this end, we re-launch SimiDroid on 10 randomly selected benchmark app pairs with these two parameters enabled, respectively. The experimental results confirm that both parameters have significantly impacted the experimental results. Our manual investigation further confirms that the changes brought along by these parameters are also valid.

Now, let us demonstrate the usefulness of the newly introduced parameters via a concrete example, i.e., leveraging parameter *LibrarySetPath* to investigate the impact of common libraries to pairwise similarity analysis of Android apps. We enable the parameter by giving as input the list of common libraries released by Li *et al.*[49][7]. As long as a library is identified in a given app, the library code will not be taken into account for computing the overall similarity of Android apps.

We again launch SimiDroid on 500 randomly selected app pairs from the benchmark dataset. Among the 500 app pairs, without considering library code, 75.8% of them have their similarity scores increased, 18.6% of them have no change, while around 6% of them (i.e., 28 pairs) have their similarity scores reduced. Fig.7 further presents the distribution of similarity scores computed via method-based comparisons, where common libraries are considered and are discarded, respectively. It is worth to note that, three of 28 pairs with reduced scores have their similarity scores even down to under 80%, the threshold used by Li *et al.*[49] to flag potential repackaged Android apps. That is to say, these three app pairs would have been falsely included in the benchmark as repackaged app pairs. This evidence, on the one hand, shows that our newly introduced library exclusion parameter is effective for SimiDroid to discard common libraries when performing similarity analysis, and on the other hand, demonstrates that the consideration of common libraries can indeed impact the performance of pairwise similarity analysis of Android apps (sometimes even introduce false positive results).

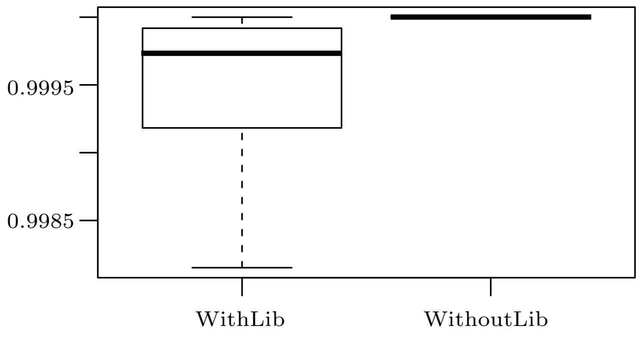


Fig.7. Distribution of similarity scores computed via method-based comparisons, where common libraries are considered (WithLib) and excluded (WithoutLib), respectively.

*Result of RQ*2. The performance of pairwise similarity analysis of Android apps can indeed be impacted by common libraries.

4.3 RQ3: Comparison

We compare SimiDroid against the available implementation of two state-of-the-art studies, namely AndroGuard[45] and FSquaDRA[46], covering code-based and resource-based similarity analysis respectively.

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. Subsection 3.2). However, the comparison between the content of two methods is 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).

[7]We also added *android.support* package into the whitelist because it has been explicitly neglected.

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 RQ1). Fig.8 comparatively plots the distribution of similarity scores calculated by SimiDroid, AndroGuard, and FSquaDRA, respectively. The similarity results computed by the state-of-the-art work 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.

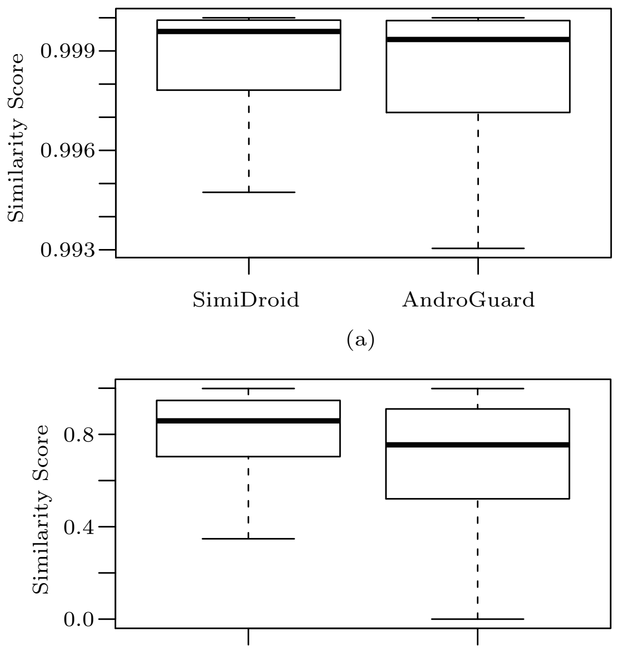


Fig.8. Comparison results among the similarity scores of (a) SimiDroid (code-based) and AndroGuard, and (b) SimiDroid (resource-based) and FSquaDRA.

As shown in Fig.8, 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 10 pairs for manual investigation. Table 2 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 have decided to focus only on the reported similar methods. These 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 2, eight 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 find 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 on 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 Fig.4 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 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. Fig.4).

Case Study 2: *2326A8* $\rightarrow$ *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 inserting a call to *dywtsbn*(), implemented as part of the newly injected payload within the same class as *onCreate*().

*Result of RQ*3. SimiDroid outperforms both code-based (AndroGuard) and resource-based (FSquaDRA) similarity analysis tools for detecting similar Android apps.

Table 2. Randomly Selected 10 Piggybacking Pairs and Their Code-Based Similarity Results Yielded by AndroGuard and SimiDroid

Original	Piggybacked	AndroGuard					SimiDroid				
		Identical	Similar	New	Deleted	Score (%)	Identical	Similar	New	Deleted	Score (%)
FFDE8B	DB2CB6	618	1	875	0	99.84	1 043	2	1 300	0	99.81
2326A8	7D6D97	1 422	0	1 727	0	100.00	2 445	1	4 384	0	99.96
E2CEED	E9B8EE	264	5	1 178	0	98.14	390	5	2 299	0	98.73
8C23C6	5ADAE7	92	1	950	0	98.92	124	1	1 730	0	99.20
A0087E	296792	3 143	1	276	0	99.97	7 090	1	460	0	99.99
1B8441	172F27	3 965	1	834	0	99.97	8 488	1	1 300	0	99.99
00C381	2DC271	905	1	294	0	99.89	1 418	1	460	0	99.93
93E50D	664F22	1 225	1	1 210	0	99.92	2 042	1	2 786	0	99.95
9E49AE	29A23A	1 386	1	1 000	0	99.93	2 172	1	1 892	0	99.95
321DA9	86E88F	829	1	184	0	99.88	1 390	1	474	0	99.93

4.4 RQ4: 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 1) to characterize suspicious intent in repackaging, 2) to recognize symptoms of piggybacking, 3) to hint on malicious payload code, or 4) to measure the impact of library code in app similarity computation.

4.4.1 Constant String Replacement

Online documentation of advertisement integration into Android app exposes how ad revenues are 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 presents a suspected case of redirecting 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 34 in Listing 3) on the call to the API method *setAdUnitId*(), so as to redirect the revenue generated by app EF2BDA.

The constant string replacement analysis has also allowed confirming 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 3, 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 1 000).

4.4.2 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[3], 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 17) or when it must be stopped/destroyed (line 6). Actually, SimiDroid has found 2 259 cases (within 523 pairs)

```
//Case Study 1: The second similar method identified between apps \emph{FFDE8B} and \emph{DB2CB6}.
public class com.unity3d.player.UnityPlayerActivity extends android.app.Activity {
 protected void onDestroy() {
  $r0 := @this: com.unity3d.player.UnityPlayerActivity;
  specialinvoke $r0.<android.app.Activity: void onDestroy()>();
+ staticinvoke <com.gamegod.touydig: void destroy(android.content.Context)>($r0);
  $r2 = $r0.<com.unity3d.player.UnityPlayerActivity: com.unity3d.player.UnityPlayer a>;
  virtualinvoke $r2.<com.unity3d.player.UnityPlayer: void configurationChanged(android.content.res.Configuration)>($r1);
  return;
}}
//Case Study 2: The unique similar method identified between apps \emph{2326A8} and \emph{7D6D97}.
public class com.platoevolved.socialpluginunity.SocialPluginUnityActivity extends com.unity3d.player.UnityPlayerActivity {
  public void onCreate(android.os.Bundle) {
   $r0 := @this: com.platoevolved.socialpluginunity.SocialPluginUnityActivity;
   $r1 := @parameter0: android.os.Bundle;
   specialinvoke $r0.<com.unity3d.player.UnityPlayerActivity: void onCreate(android.os.Bundle)>($r1);
+  virtualinvoke $r0.<com.platoevolved.socialpluginunity.SocialPluginUnityActivity: void dywtsbn()>();
   return;}
+ public void dywtsbn(){
+ com.platoevolved.socialpluginunity.SocialPluginUnityActivity $r0;
+ android.sowsyr.RerhnAndroid $r1;
+ $r0 := @this: com.platoevolved.socialpluginunity.SocialPluginUnityActivity;
+ $r1 = new android.sowsyr.RerhnAndroid;
+ specialinvoke $r1.<android.sowsyr.RerhnAndroid: void <init>(android.content.Context)>($r0);
+ virtualinvoke $r1.<android.sowsyr.RerhnAndroid: void GVern()>();
+ return;
+}
}
//Case Study 3: Redirecting ad revenue from between \emph{EF2BDA} and \emph{87880D}.
public class com.gameneeti.game.deckbowling.Start extends android.app.Activity {
 void callAdds() {
  $r1 = $r0.<com.gameneeti.game.deckbowling.Start: com.google.android.gms.ads.AdView adView>;
- virtualinvoke $r1.<com.google.android.gms.ads.AdView: void setAdUnitId(java.lang.String)>("a1522d5c390a573");
+ virtualinvoke $r1.<com.google.android.gms.ads.AdView: void setAdUnitId(java.lang.String)>("ca-app-pub-8182614411920503/1232098473");
}}
```

Listing 3. Case study illustrative code snippets extracted from real Android apps.

where new method call is introduced during repackaging (cf. Table 3).

Table 3. Explanation Statistics

Explanation Type	Number of Pairs	Number 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

4.4.3 *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[49]. 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 [49]. Among the analyzed pairs, SimiDroid reports different similarity scores for 422 pairs when common libraries are excluded (cf. Table 3). 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 84%, leading to a decision that these apps constitute a repackaging pair (if we consider also 80% as the threshold). Compared with 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.

4.4.4 *Duplicated Component Capabilities*

Building on findings in [3], 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 the same event, or that can realize the same action (e.g., play videos). In our experiments, we have shown (cf. Listing 2 for 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 surprisingly 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 ($A^2_{213} = 45\,156$).

4.4.5 *Resource File Rename*

Finally, we have implemented an analysis rule to hint on extensive resource file renaming. This practice is indeed more common in piggybacking processes than in mere app updates[3]. The possible intention behind such a behaviour could be that attackers attempt to fool resource-based comparisons and hence to bypass the detection of resource-based repackaged detection approaches which are easily applicable at market scale. Fig.9 outlines the distribution of resource directories involving resource file renames. The most favoured resources are inside the res directory where the majority of files are pictures. In total, as shown in Table 3, SimiDroid harvests 994 cases where a resource file is renamed among 160 piggybacking pairs.

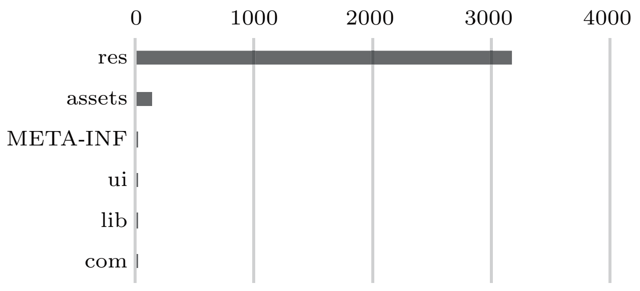


Fig.9. Distribution of resource directories ranked based on the number of resource files (inside the directory) renamed.

Case Study 7: *740E84* $\rightarrow$ *938A1D* (*Resource File Rename*). SimiDroid reports a case where two resource files are exactly the same in content (same hash code) but have different names between the compared two apps. Particularly, the resource file named *assets/skeleton/skeleton_kaboom_blue.json* in the first app has been renamed to *assets/skeleton/skeleton_kaboom_red.json* in the second app.

*Result of RQ*4. SimiDroid is capable of supporting the comprehension of similarities within a pair of Android apps. The explanation hints reported by SimiDroid can indeed provide an opportunity for analysts to readily identify noteworthy changes, offering explanations on the likely intent of such changes.

4.5 RQ5: Similarity App Cluster

As mentioned in Subsection 3.4, one benefit that our extension to SimiDroid for simultaneously analyzing multiple Android apps can be leveraged is to group similar apps into clusters among a large set of Android apps. In this work, we evaluate this hypothesis through a real challenge where security analytics would like to understand the diversity of malicious apps belonging to a given family. Although those apps, which are from the same malware family, should semantically share the same malicious behaviour, their implementation could vary dramatically (e.g., they may be variants of different root malware that are developed from scratch). Therefore, it is useful to have an automated approach that systematically groups apps (e.g., from the same malware family) based on their characteristics into different clusters. Indeed, the clustering result provides a similarity overview of all the involved apps, and hence can be leveraged to quickly answer the following questions: how many malware variants exist in the family, or how many kinds of malware share the same code structure.

In this work, we randomly select 2 000 malware from VirusShare[8], a well-known source sharing viruses including Android malicious apps. Since our goal here is to conduct similarity analyses for such apps that are from the same malware family, we need to cluster the 2 000 apps into different families based on their malicious behaviour. To this end, we resort to a malware labelling tool called AVClass[57] to assign Android malware to different families. AVClass takes as input the anti-virus labels and outputs the most likely family name for each malware sample, where the anti-virus labels can be obtained from VirusTotal.

Among the 2 000 randomly selected malware, AVClass flags 89 of them as "SINGLETON", indicating that there is no family name found for these apps. In this work, we exclude these 89 apps from consideration. The remaining 1 811 malware are grouped by AVClass into 150 clusters, where the number of variants in each family varies from 1 to 429. Fig.10 illustrates the detailed distribution of the number of variants available in those clustered families. The median value indicates that around half of the families have at least two variants in their families.

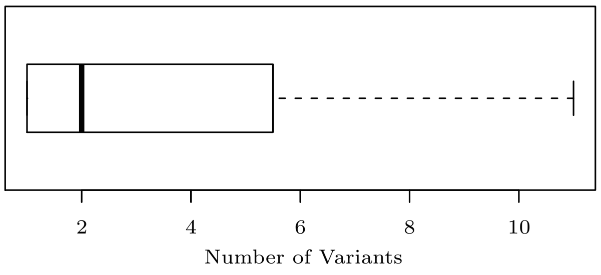


Fig.10. Distribution of the number of variants available in clustered families.

Table 4 presents the top 10 families ranked based on their number of variants. Family *fakeinst* is the most favoured one that has 429 variants appeared in our randomly selected 2 000 malware samples, accounting for roughly 20% of Android malware. As revealed by Kaspersky[9], *fakeinst* apps are the first type of active SMS trojan that targets users in 66 countries. Thanks to SimiDroid, as shown in the third column of Table 4, we find that those 429 variants are actually from 15 root malicious apps, which are recurrently copied and repackaged to form new variants (with small changes). From Table 4 we can also observe that for each family there is a cluster that contains significantly more variants compared with others (e.g., 205 for *fakeinst* and 90 for *dowgin*), suggesting that attackers attempt to implement their malicious apps based on popular ones, making this cluster of malware variants even more popular in the family.

We further go one step deeper to investigate whether the apps in the largest cluster of each family are actually cloned from the same app (i.e., sharing the same package name) or are developed by the same attacker (i.e., sharing the same developer signature). Table 5 summarises the statistics we have obtained, where x/y in the second column shows that we could not obtain the metadata for $y - x$ apps due to tooling errors, e.g., our tool-chain cannot successfully extract metadata for 4 (i.e., $15 - 11$) apps in family *airpush*). Interestingly, for some clusters such as *smsreg* and *smsagent*, all their apps (or the majority of them) are actually modified

[8]https://virusshare.com, Jan. 2019.

[9]http://securityaffairs.co/wordpress/24427/malware/fakeinst-first-sms-trojan.html, Jan. 2019.

Table 4. Top 10 Malware Families Ranked Based on Their Numbers of Variants

Family	Number of Variants	Number of Clusters	Numbers of Variants in Clusters
fakeinst	429	15	205, 56, 31, 16, 19, 25, 24, 6, 6, 4, 4, 4, 3, 2, 2
dowgin	156	11	90, 8, 4, 4, 4, 3, 3, 3, 2, 2, 2
adwo	148	4	80, 50, 3, 2
gappusin	117	13	19, 7, 6, 6, 4, 3, 2, 2, 2, 2, 2, 2, 2
kuguo	108	4	65, 3, 2, 2
smsreg	75	4	19, 15, 3, 2
airpush	52	4	15, 2, 2, 2
smsagent	49	2	37, 2
admogo	48	2	43, 2
opfake	47	8	15, 8, 6, 4, 4, 3, 2, 2

Note: Clusters containing only a single malware are ignored.

from the same root app and are manipulated by the same developer. This evidence suggests that attackers are continuously updating their malware so as to bypass the emerging malware-detecting approaches. Instead of updating from the same original app, some attackers are preferred to introduce into a malware family of different apps. For example, all the 80 apps in cluster *adwo* are developed by the same attacker, while are developed based on different root apps, i.e., they all share different unique package names.

Table 5. Statistics of the Apps in the Largest Cluster of Each Family

Family	Variants	Number of Package Names	Number of Signatures
fakeinst	191/205	143	143
dowgin	88/90	2	13
adwo	80/80	1	80
gappusin	19/19	8	9
kuguo	65/65	1	18
smsreg	19/19	19	19
airpush	11/15	1	2
smsagent	37/37	37	37
admogo	43/43	1	43
opfake	15/15	4	15

Since the clustering approach of SimiDroid adopts the same similarity analysis algorithms from the pairwise comparison approach, which has been experimentally shown reliable, the performance of the clustering approach should be also reliable. Indeed, by taking as input the 80 *adwo* apps mentioned before, SimiDroid would still group all of them into the same cluster. Similarly, as another experimental example, SimiDroid groups 10 randomly selected Google Play apps into 10 clusters, i.e., one app in a cluster. Our manual observation confirms that those 10 apps are indeed different from each other.

*Result of RQ*5. The capability of analyzing simultaneously multiple Android apps makes SimiDroid capable of clustering Android apps into different categories based on their similarities, and hence eases the job of code analysts, e.g., it can help security analysts to quickly understand the implementation differences of a set of malware, though they may belong to the same family.

5 Threats to Validity

Our approach and the experiments presented in this paper introduce a few threats to validity. First of all, the current implementation of SimiDroid does not provide strong obfuscation resilience. For example, if the method names (or component names) are changed due to obfuscation, SimiDroid can no longer compute a reliable similarity score for these candidate apps. Nevertheless, the main usage of SimiDroid is not to detect cloned apps in the wild within a large scale of apps, but to vet and confirm the similarity of suspicious candidate pairs, giving by other coarse approaches.

Second, the comparison between AndroGuard and SimiDroid may not be perfect since these two tools leverage totally different fundamental tools to compute the similarities between two apps. For example, as shown in Table 2, for app DB2CB6, the total number of methods considered by AndroGuard is 1 494 (618 + 1 + 875) while by SimiDroid is 2 345 (1 043 + 2 + 1 300). The reason why AndroGuard yields fewer methods than SimiDroid is that AndroGuard attempts to perform $1 \rightarrow n$ comparisons while SimiDroid performs $1 \rightarrow 1$ comparisons. As a result, n similar methods will be counted only once. Furthermore, thanks to our manual verification, we have also confirmed that AndroGuard will likely yield both false positive and false negative results in terms of identifying similar methods.

Finally, the similarity analysis among multiple Android apps provided by SimiDroid is quite straightforward, where only the similarity (i.e., the weight between two nodes) is considered at the moment. In our future work, we plan to consider more artefacts such as the degrees of the nodes in the graph to make the clustering results more persuasive.

6 Conclusions

We introduced a new framework, SimiDroid, for supporting researchers and practitioners in the analysis of similar apps (by performing either pairwise comparison for two apps or multiple apps comparison). SimiDroid integrates plugins implementing the extraction of features, at different levels, for the computation of 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 showed 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 contributed to supporting the community in the realisation of extensive studies on app similarities to further experiment in their fast, accurate and scalable approaches.

References

[1] Dong F, Wang H Y, Li L, Guo Y, Xu G A, Zhang S D. How do mobile apps violate the behavioral policy of advertisement libraries? In *Proc. the 19th Workshop on Mobile Computing Systems and Applications,* February 2018, pp.75-80.

[2] Dong F, Wang H Y, Li L, Guo Y, Bissyandé T F, Liu T M, Xu G A, Klein J. FraudDroid: Automated ad fraud detection for Android apps. In *Proc. the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2018, pp.257-268.

[3] Li L, Li D, Bissyandé T F, Klein J, Le Traon Y, Lo D, Cavallaro L. Understanding Android app piggybacking: A systematic study of malicious code grafting. *IEEE Transactions on Information Forensics and Security*, 2017, 12(6): 1269-1284.

[4] Wang H Y, Liu Z, Guo Y, Chen X Q, Zhang M, Xu G A, Hong J. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proc. the 26th International Conference on World Wide Web*, April 2017, pp.163-172.

[5] Wang H Y, Li H, Li L, Guo Y, Xu G A. Why are Android apps removed from Google play? A large-scale empirical study. In *Proc. the 15th International Conference on Mining Software Repositories*, May 2018, pp.231-242.

[6] Chen J, Alalfi M H, Dean T R, Zou Y. Detecting Android malware using clone detection. *Journal of Computer Science and Technology*, 2015, 30(5): 942-956.

[7] Wang H Y, Guo Y, Ma Z A, Chen X Q. WuKong: A scalable and accurate two-phase approach to Android app clone detection. In *Proc. the 2015 International Symposium on Software Testing and Analysis*, July 2015, pp.71-82.

[8] Chen K, Liu P, Zhang Y J. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proc. the 36th International Conference on Software Engineering*, May 2014, pp.175-186.

[9] Zhou W, Zhou Y J, Jiang X X, Ning P. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proc. the 2nd ACM Conference on Data and Application Security and Privacy*, February 2012, pp.317-326.

[10] Li L, Li D Y, Bissyandé T F, Klein J, Cai H P, Lo D, Traon L Y. On locating malicious code in piggybacked Android apps. *Journal of Computer Science and Technology*, 2017, 32(6): 1108-1124.

[11] Li L, Bissyandé T F, Papadakis M, Rasthofer S, Bartel A, Octeau D, Klein J, Traon L. Static analysis of Android apps: A systematic literature review. *Information and Software Technology*, 2017, 88: 67-95.

[12] Tian K, Yao D F, Ryder B G, Tan G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In *Proc. the 2016 IEEE Security and Privacy Workshops*, May 2016, pp.262-271.

[13] Guan Q L, Huang H Q, Luo W Q, Zhu S C. Semantics-based repackaging detection for mobile apps. In *Proc. the 8th International Symposium on Engineering Secure Software and Systems*, April 2016, pp.89-105.

[14] Wu X P, Zhang D F, Su X, Li W W. Detect repackaged android application based on HTTP traffic similarity. *Security and Communication Networks*, 2015, 8(13): 2257-2266.

[15] Sun M. S, Li M M, Lui J. DroidEagle: Seamless detection of visually similar Android apps. In *Proc. the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, June 2015, Article No. 9.

[16] Jiao S B, Cheng Y, Ying L Y, Su P R, Feng D G. A rapid and scalable method for Android application repackaging detection. In *Proc. the 11th International Conference on Information Security Practice and Experience*, May 2015, pp.349-364.

[17] Aldini A, Martinelli F, Saracino A, Sgandurra D. Detection of repackaged mobile applications through a collaborative approach. *Concurrency and Computation: Practice and Experience*, 2015, 27(11): 2818-2838.

[18] Soh C, Tan H B. K, Arnatovich Y L, Wang L. Detecting clones in Android applications through analyzing user interfaces. In *Proc. the 23rd International Conference on Program Comprehension*, May 2015, pp.163-173.

[19] Gonzalez H, Kadir A A, Stakhanova N, Alzahrani A J, Ghorbani A A. Exploring reverse engineering symptoms in Android apps. In *Proc. the 8th European Workshop on System Security*, April 2015, Article No. 7.

[20] Chen K, Wang P, Lee Y J, Wang X F, Zhang N, Huang H Q, Zou W, Liu P. Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale. In *Proc. the 24th USENIX Security Symposium*, August 2015, pp.659-674.

[21] Zhou W, Wang Z, Zhou Y J, Jiang X X. DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform. In *Proc. the 4th ACM Conference on Data and Application Security and Privacy*, March 2014, pp.199-210.

[22] Gonzalez H, Stakhanova N, Ghorbani A A. DroidKin: Lightweight detection of Android apps similarity. In *Proc. the 10th International Conference on Security and Privacy in Communication Systems*, September 2014, pp.436-453.

[23] Deshotels L, Notani V, Lakhotia A. DroidLegacy: Automated familial classification of Android malware. In *Proc. ACM SIGPLAN on Program Protection and Reverse Engineering Workshop*, January 2014, Article No. 3.

[24] Mojica I J, Adams B, Nagappan M, Dienst S, Berger T, Hassan A E. A large-scale empirical study on software reuse in mobile apps. *IEEE Software*, 2014, 31(2): 78-86.

[25] Vásquez L M, Holtzhauer A, Bernal-Cárdenas C, Poshyvanyk D. Revisiting Android reuse studies in the context of code obfuscation and library usages. In *Proc. the 11th Working Conference on Mining Software Repositories*, May 2014, pp.242-251.

[26] Crussell J, Gibler C, Chen H. AnDarwin: Scalable detection of Android application clones based on semantics. *IEEE Transactions on Mobile Computing*, 2015, 14(10): 2007-2019.

[27] Shao Y R, Luo X P, Qian C X, Zhu P F, Zhang L. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Proc. the 30th Annual Computer Security Applications Conference*, December 2014, pp.56-65.

[28] Zhang F F, Huang H Q, Zhu S C, Wu D H, Liu P. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proc. the 7th ACM Conference on Security and Privacy in Wireless & Mobile Networks*, July 2014, pp.25-36.

[29] Ren C G, Chen K, Liu P. Droidmarking: Resilient software watermarking for impeding Android application repackaging. In *Proc. the 29th ACM/IEEE International Conference on Automated Software Engineering*, September 2014, pp.635-646.

[30] Sun X, Zhongyang Y B, Xin Z, Mao B, Xie L. Detecting code reuse in Android applications using component-based control flow graph. In *Proc. the 29th IFIP TC 11 International Conference on ICT Systems Security and Privacy Protection*, December 2014, pp.142-155.

[31] Lindorfer M, Volanis S, Sisto A, Neugschwandtner M, Athanasopoulos E, Maggi F, Platzer C, Zanero S, Ioannidis S. AndRadar: Fast discovery of Android applications in alternative markets. In *Proc. International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2014, pp.51-71.

[32] Kywe S M, Li Y J, Deng R H, Hong J. Detecting camouflaged applications on mobile application markets. In *Proc. the 17th International Conference on Information Security and Cryptology*, December 2014, pp.241-254.

[33] Lin Y D, Lai Y C, Chen C H, Tsai H C. Identifying Android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 2013, 39(B): 340-350.

[34] Zhou W, Zhou Y J, Grace M, Jiang X X, Zou S H. Fast, scalable detection of piggybacked mobile applications. In *Proc. the 3rd ACM Conference on Data and Application Security and Privacy*, February 2013, pp.185-196.

[35] Vidas T, Christin N. Sweetening Android lemon markets: Measuring and combating malware in application marketplaces. In *Proc. the 3rd ACM Conference on Data and Application Security and Privacy*, February 2013, pp.197-208.

[36] Crussell J, Gibler C, Chen H. AnDarwin: Scalable detection of semantically similar Android applications. In *Proc. the 18th European Symposium on Research in Computer Security*, September 2013, pp.182-199.

[37] Zheng M, Sun M S, Lui J. DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate android malware. arXiv:1302.7212, 2013. https://arxiv.org/pdf/1302.7212.pdf, September 2018.

[38] Zhou W, Zhang X W, Jiang X X. Appink: Watermarking Android apps for repackaging deterrence. In *Proc. the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, May 2013, pp.1-12.

[39] Gibler C, Stevens R, Crussell J, Chen H, Zang H, Choi H. AdRob: Examining the landscape and impact of Android application plagiarism. In *Proc. the 11th Annual International Conference on Mobile Systems, Applications, and Services*, June 2013, pp.431-444.

[40] Hanna S, Huang L, Wu E, Li S, Chen C, Song D. Juxtapp: A scalable system for detecting code reuse among Android applications. In *Proc. the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, July 2012, pp.62-81.

[41] Crussell J, Gibler C, Chen H. Attack of the clones: Detecting cloned applications on Android markets. In *Proc. the 17th European Symposium on Research in Computer Security*, September 2012, pp.37-54.

[42] Potharaju R, Newell A, Nita R C, Zhang X Y. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proc. the 4th International Symposium on Engineering Secure Software and Systems*, February 2012, pp.106-120.

[43] Wu D J, Mao C H, Wei T E, Lee H M, Wu K P. DroidMat: Android malware detection through manifest and API calls tracing. In *Proc. the 7th Asia Joint Conference on Information Security*, August 2012, pp.62-69.

[44] Ruiz I J M, Nagappan M, Adams B, Hassan A E. Understanding reuse in the Android market. In *Proc. the 20th IEEE International Conference on Program Comprehension*, June 2012, pp.113-122.

[45] Desnos A. Android: Static analysis using similarity distance. In *Proc. the 45th Hawaii International Conference on System Sciences*, February 2012, pp.5394-5403.

[46] Zhauniarovich Y, Gadyatskaya O, Crispo B, La S F, Moser E. FSquaDRA: Fast detection of repackaged applications. In *Proc. the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy*, July 2014, pp.130-145.

[47] Gao J, Li L, Kong P F, Bissyandé T F, Klein J. On vulnerability evolution in Android apps. In *Proc. the 40th International Conference on Software Engineering: Companion Proceedings*, May 2018, pp.276-277.

[48] Kong P F, Li L, Gao J, Liu K, Bissyandé T F, Klein J. Automated testing of Android apps: A systematic literature review. *IEEE Transactions on Reliability*. doi:10.1109/TR.2018.2865733.

[49] Li L, Bissyandé T F, Klein J, Le T Y. An investigation into the use of common libraries in Android apps. In *Proc. the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, March 2016, pp.403-414.

[50] Viennot N, Garcia E, Nieh J. A measurement study of Google play. In *Proc. ACM International Conference on Measurement and Modeling of Computer Systems*, June 2014, pp.221-233.

[51] Ma Z, Wang H Y, Guo Y, Chen X Q. LibRadar: Fast and accurate detection of third-party libraries in Android apps. In *Proc. the 38th ACM/IEEE International Conference on Software Engineering Companion*, May 2016, pp.653-656.

[52] Wang H Y, Guo Y. Understanding third-party libraries in mobile app analysis. In *Proc. the 39th IEEE/ACM International Conference on Software Engineering Companion*, May 2017, pp.515-516.

[53] Li L, Bissyandé T F, Octeau D, Klein J. DroidRA: Taming reflection to support whole-program analysis of Android apps. In *Proc. the 25th International Symposium on Software Testing and Analysis*, July 2016, pp.318-329.

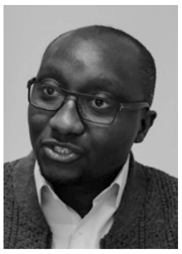
[54] Lam P, Bodden E, Lhoták O, Hendren L. The Soot framework for Java program analysis: A retrospective. In *Proc. Cetus Users and Compiler Infrastructure Workshop*, October 2011, Article No. 35.

[55] Bartel A, Klein J, Le Traon Y, Monperrus M. Dexpler: Converting Android Dalvik bytecode to jimple for static analysis with soot. In *Proc. the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, June 2012, pp.27-38.

[56] Li L, Gao J, Hurier M, Kong P F, Bissyandé T F, Bartel A, Klein J, Traon Y L. Androzoo++: Collecting millions of Android apps and their metadata for the research community. arXiv:1709.05281, 2017. https://arxiv.org/pdf/1709.05281.pdf, September 2018.

[57] Sebastián M, Rivera R, Kotzias P, Caballero J. AVCLASS: A tool for massive malware labeling. In *Proc. the 19th International Symposium on Research in Attacks, Intrusions, and Defenses*, September 2016, pp.230-253.

Li Li is a lecturer (a.k.a., assistant professor) and a Ph.D. supervisor at Monash University, Melbourne, Australia. He received his Ph.D. degree in computer science from the University of Luxembourg, Luxembourg, in 2016. His research interests are in the fields of Android security and reliability, static code analysis, machine learning and deep learning. Dr. Li received an ACM Distinguished Paper Award at ASE 2018, a FOSS Impact Paper Award at MSR 2018, and a Best Paper Award at the ERA track of IEEE SANER 2016. He is an active member of the software engineering and security community serving as reviewers or co-reviewers for many top-tier conferences and journals such as ICSME, SANER, TOSEM, TSE, TIFS, TDSC, and TOPS.



Tegawendé F. Bissyandé is a research scientist with SnT (Interdisciplinary Centre for Security, Reliability and Trust), University of Luxembourg, Luxembourg. He received his Ph.D. degree in computer science from the University of Bordeaux, Bordeaux, France, in 2013. His work is mainly related to software engineering, specifically empirical software engineering, reliability and debugging as well as mobile app analysis. His studies were presented in major conferences such as ICSE, ISSTA and ASE, and published in top journals such as Empirical Software Engineering and IEEE TIFS. He has received a Best Paper Award at ASE 2012, and has served in several program committees including ASE-Demo, ACM SAC, and ICPC.

Hao-Yu Wang is currently an assistant professor at Beijing University of Posts and Telecommunications, Beijing. He received his Ph.D. degree from Peking University, Beijing, in 2016. His research interest lies at the intersection of mobile system, privacy and security, and program analysis. His studies were presented in major conferences such as ESEC/FSE, ISSTA, WWW, UbiComp, IMC, and published in top journals such as ACM Transactions on Information Systems (TOIS).

Jacques Klein is a senior research scientist at the University of Luxembourg, Luxembourg, and at the Interdisciplinary Centre for Security, Reliability and Trust (SnT). He received a Ph.D. degree in computer science from the University of Rennes, Rennes, France, in 2006. His main areas of expertise are threefold: 1) mobile security (malware detection, prevention and dissection, static analysis for security, vulnerability detection, etc.); 2) software reliability (software testing, semi-automated and fully-automated program repair, etc.); 3) data analytics (multi-objective reasoning and optimization, model-driven data analytics, time series pattern recognition, etc.). In addition to academic achievements, Dr. Klein has also standing experience and expertise on successfully running industrial projects with several industrial partners in various domains by applying data analytics, software engineering, information retrieval, etc., to their research problems.