# Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark

Li Li, Tegawendé F. Bissyandé, Jacques Klein

**Abstract**—Repackaging is a serious threat to the Android ecosystem as it deprives app developers of their benefits, contributes to spreading malware on users' devices, and increases the workload of market maintainers. In the space of six years, the research around this specific issue has produced 57 approaches which do not readily scale to millions of apps or are only evaluated on private datasets without, in general, tool support available to the community. Through a systematic literature review of the subject, we argue that the research is slowing down, where many state-of-the-art approaches have reported high-performance rates on closed datasets, which are unfortunately difficult to replicate and to compare against. In this work, we propose to reboot the research in repackaged app detection by providing a literature review that summarises the challenges and current solutions for detecting repackaged apps and by providing a large dataset that supports replications of existing solutions and implications of new research directions. We hope that these contributions will re-activate the direction of detecting repackaged apps and spark innovative approaches going beyond the current state-of-the-art.

**Index Terms**—Android, Repackaging, Clone, Literature Review, Benchmark.

## 1 Introduction

Mobile applications, especially Android apps, are straightforward to reverse engineer, copy and resubmit to markets [1]. The Android application packaging system indeed relies on the ZIP open compression format to archive apps' resource files and decompilable bytecode, making it easy for anyone with the adequate tool support to unpack any app, modify its contents, and repackage it. *Repackaging* is thus a common threat to the Android ecosystem, where it is used by *plagiarists* who *clone* apps from other developers, e.g., in order to redirect advertisement revenue [1], [2], and by malware writers who *piggyback* malicious payloads on popular apps to spread malware [3].

Android app repackaging has been raised as a serious problem by various authors in the literature as well as by different stakeholders in the app development industry. For example, large-scale application plagiarism [4] has led to the shutdown of several non-official app markets. Last year, Ustwo Games, developer of the popular "Monument Valley" game, has reported that only 5% of Monument Valley installations on Android are paid for [5], with various copies being available from different "authors" in the same market. Very recently, the famous Pokemon Go app has been repackaged in different ways and for different reasons as mentioned in the Lookout blog [6]. In a different category, Jung et al. [7] have presented a serious case of repackaging attacks on Korea's Banking Android apps, demonstrating how it was possible to redirect money transfers without having to illegally obtain any of the sender's personal information such as bank accounts.

- *L. Li is with the Faculty of Information Technology, Monash University, Australia.*
- *T. Bissyandé and J. Klein are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg.*

*Manuscript received XXX; revised XXX.*

To limit repackaging and its impacts, different steps must be taken by all concerned parties. For example, developers may explore different techniques for watermarking their apps and denying some functionalities when it becomes obvious that the running copy is a cloned version [8], [9]. Nevertheless, most of the workload is carried by market maintainers who must employ powerful tools to catch repackaged apps in a fast and accurate way so as to remove them from markets [10]. A number of research studies in the literature have investigated a variety of repackaging detection approaches without convincingly demonstrating that the problem is now well addressed.

In this paper, we revisit the state of research on repackaged app detection, insisting on the practical challenges that the community must focus on for implementing effective repackaged app detection solutions for Android markets. Overall we make the following contributions:

- We propose a systematic review of the state-of-the-art literature on repackaged app detection and highlight their shortcomings in terms of the impracticality of the approaches, lack of reproducibility, and suboptimal evaluation scenarios.
- We build the RePack dataset and release it to the community to encourage proper assessment of repackaged app detection approaches. Our work builds upon the popular AndroZoo repository [11], [12], which can serve as an exchange repository for describing one's dataset using the hash values of apps. We also enumerate research directions that the community should take up for advancing the state-of-the-art in the topic.

The remainder of this paper is organised as follows. Section 2 clarifies the terminology used, and explains the procedure for the Systematic Literature Review (SLR) that we have conducted on the topic of Android repackaged app detection. Section 3 discusses the prominent challenges in taming app repackaging. Then, Section 4 summarises the different contributions made in the literature and highlight some issues in their approaches and evaluations. Section 5 describes our efforts in addressing some important challenges. Notably, we discuss the construction of a large dataset of repackaging pairs from the AndroZoo app repository. Sections 6 and 7 discuss priority research directions and potential threats to validity of this study respectively. Section 8 enumerates closely related work and Section 9 finally provides concluding remarks.

All artefacts of our research are available in the RePack repository at:

`https://github.com/serval-snt-uni-lu/RePack`

## 2 Literature Search

In this section, we provide introductory information on the lightweight Systematic Literature Review (SLR) that we have performed to assess the advances that were made in the area of repackaged app detection. We first clarify the terminology used in the field before giving statistics on the collected literature corpus.

### 2.1 Terminology

Several terms are used in the literature, notably in paper titles and abstracts, to indicate actions that somehow involve a repackaging process:

*Repackaging* refers to the core process of unpacking a software package, then repackaging it after a probable modification of the decompiled code and/or of other resource files (e.g. logos, Permission list, etc.). Because all Android app packages (APKs) are signed with the developer certificate, a repackaging pair, formed by an original app and its repackaged version, can be differentiated by their checksums even when no modification of the code has been performed by the repackager, who should be different from the original app developer. Following this principle, **we consider in this work two apps as a repackaged app pair as long as (1) they share at least 80% of the code (i.e., code similarity exceeds 80%) and (2) they are signed by different developers**.

*Cloning* is the process of building a software by reverse engineering another software or by reimplementing it based on documentation or usage experience. Theoretically, cloning is different from repackaging because it does not need to package an app based on its cloning version while repackaging always form the app based on its original counterpart. Nevertheless, in the Android ecosystem, this difference is negligible as it is straightforward to clone an app via repackaging and, in most cases, the whole app code (rather than partial code) is manipulated.

*Reusing* is the action of producing apps from existing code rather than developing from scratch. The existing code can vary from small parts like several methods to big parts such as whole app functionalities.

*Plagiarism* consists in wrongfully appropriating the work of another developer, e.g., by cloning her/his APK to benefit from, for example, advertisement revenues. Comparing to the term reusing, plagiarism emphases on the part that the cloned code is wrongfully leveraged.

*Piggybacking* is defined in the literature as a malware development activity where a given benign app is repackaged to include a malicious payload. Piggybacked apps thus constitute a subset of repackaged apps.

*Camouflage* is a technique used by malware writers to trick users into installing malware sample, which is presented as a well-known popular app, e.g., by repackaging an app to replace its main functionality with the malicious implementation. In this work, we consider camouflage as a special case of piggybacking, where the app interfaces are not modified (to keep the same looks) but the app code has been manipulated.

### 2.2 Systematic Literature Review (SLR) Methodology

Our work evolves around an investigation of the state-of-the-art research on repackaged app detection. We search for the relevant literature in a systematic way, following the guidelines provided by Keele [13], and Brereton et al. [14]. Thus, in a first step, after outlining the relevant research questions, we search for potential related work (up to the end of 2017) in four well-known online repositories: ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect. We use two groups of keywords (in the form of regular expression) enumerated in Table 1. The search string[1] is formed as a conjunction $g_1$ `AND` $g_2$ where $g_1$ and $g_2$ are themselves formed each as a disjunction of the keywords respectively of groups G1 and G2. The goal of this step is to collect as many related papers as possible, taking into account most well-recorded proceedings. We consolidate[2] the collected list of relevant work by manually going through all the papers, examining the title and abstract, to ensure that they deal with repackaged app detection. Following the same guidelines suggested by Barbara Kitchenham [17], short papers[3] such as the one presented by Ayush Kohli [18] will not be considered in this study.

In a second step, we perform a backwards-snowballing on the remaining papers in an attempt to account for influential papers that may not have been recorded in the aforementioned repositories or that did not mention the used keywords. To that end, we carefully read the related work section of the papers collected at the end of the first step.

TABLE 1: Repository Search Keywords.

| **Group** (`AND`) | **Keywords** (`OR`) |
|---|---|
| G1 | android, mobile, *phone* |
| G2 | clon*, repackag*, piggyback*, plagiari*, reus*, camouflag* |

At the end of the SLR search, we had collected 57 papers that present work dealing, in one way or another, and to any extent, with research on Android app repackaging. Table 2 enumerates all the papers, highlighting their publication year, publication venues and the accompanying tool name.

1. (android OR mobile OR *phone*) AND (clon* OR repackag* OR piggyback* OR plagiari* OR reus*)

2. Online repository search engines often list irrelevant results presence potential irrelevant articles [15], [16].

3. Less than five pages in double column or nine pages in single column.

TABLE 2: Full List of Collected and Examined Papers

| Tool/Reference | Year | Venue$^{\alpha}$ |
|---|---|---|
| CodeMatch [19] | 2017 | ESEC/FSE |
| DR-Droid2 [20] | 2017 | TDSC (J) |
| DAPASA [21] | 2017 | TIFS (J) |
| FUIDroid [22] | 2017 | MISY (J) |
| APPraiser [23] | 2017 | IEICE TIS (J) |
| RepDroid [24] | 2017 | ICPC |
| SimiDroid [25] | 2017 | TrustCom |
| GroupDroid [26] | 2017 | SSPREW@ACSAC (W) |
| CLANdroid [27] | 2016 | ICPC |
| DR-Droid [28] | 2016 | MoST@S&P (W) |
| DroidClone [29] | 2016 | DICTAP |
| FSquaDRA2 [30] | 2016 | NordSec |
| Li et al. [31] | 2016 | SANER |
| Niu et al. [32] | 2016 | ICSAI |
| RepDetector [33] | 2016 | ESSoS |
| SUIDroid [34] | 2016 | TrustCom |
| Kim et al. [35] | 2015 | ASE (J) |
| AndroidSOO [36] | 2015 | EuroSec@EuroSys (W) |
| AndroSimilar2 [37] | 2015 | JISA |
| Chen et al. [38] | 2015 | JCST (J) |
| DroidEagle [39] | 2015 | WiSec |
| ImageStruct [40] | 2015 | ISPEC |
| MassVet [41] | 2015 | USENIX Security |
| PICARD [42] | 2015 | CCPE (J) |
| Soh et al. [43] | 2015 | ICPC |
| Wu et al. [44] | 2015 | SCN (J) |
| WuKong [45] | 2015 | ISSTA |
| AnDarwin2 [46] | 2014 | TMC (J) |
| AndRadar [47] | 2014 | DIMVA |
| Chen et al. [48] | 2014 | ICSE |
| DIVILAR [49] | 2014 | CODASPY |
| DroidKin [50] | 2014 | SecureComm |
| DroidLegacy [51] | 2014 | PPREW@POPL (W) |
| DroidMarking [9] | 2014 | ASE |
| DroidSim [52] | 2014 | IFIP SEC |
| FSquaDRA [53] | 2014 | DBSec |
| Kywe et al. [54] | 2014 | ICISC |
| Linares-Vsquez et al. [55] | 2014 | MSR |
| PLayDrone [56] | 2014 | SIGMETRICS |
| ResDroid [57] | 2014 | ACSAC |
| Ruiz et al. [58] | 2014 | IEEE Software (J) |
| ViewDroid [59] | 2014 | WiSec |
| AdRob [60] | 2013 | MobiSys |
| AnDarwin [61] | 2013 | ESORICS |
| AndroSimilar [62] | 2013 | SIN |
| AppInk [63] | 2013 | AsiaCCS |
| AppIntegrity [64] | 2013 | CODASPY |
| DroidAnalytics [65] | 2013 | TrustCom |
| PiggyApp [66] | 2013 | CODASPY |
| SCSdroid [67] | 2013 | CompSec (J) |
| Androguard [68] | 2012 | HICSS |
| DNADroid [2] | 2012 | ESORICS |
| DroidMat [69] | 2012 | AsiaJCIS |
| DroidMOSS [70] | 2012 | CODASPY |
| JuxtApp [71] | 2012 | DIMVA |
| Potharaju et al. [72] | 2012 | ESSoS |
| Ruiz et al. [73] | 2012 | ICPC |

$^{\alpha}$: (J) and (W) stand for Journal and Workshop venues respectively

## 2.3 Statistics on State-of-the-art Work

The research topic around repackaged apps has been initiated in the Android community after a presentation of Desnos and Gueguen [74] at Black Hat, Abu Dhabi 2011, where they discussed Android app reverse engineering and decompilation, and the associated security implications. Fig. 1 illustrates some statistical trends of the research publications on Android repackaged app detection.

It appears from the collected data that repackaged app detection has been tackled first and mostly by security researchers. Then, Software Engineering researchers have picked up on the problem, leading to a peak of publications in 2014. After 2014, the volume of published research started to steadily decrease, although no new data has shown that the problem has been solved in practice.

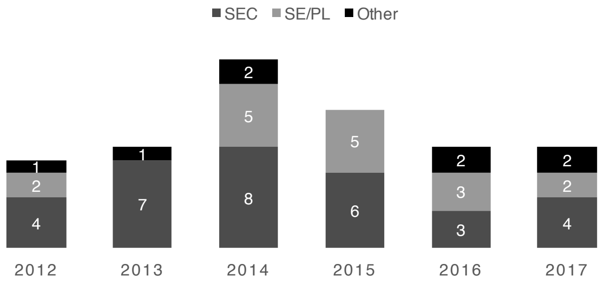


Fig. 1: Distributions of the State-of-the-art literature per Year.

It is noteworthy that only 19% (11 out of 57) of the state-of-the-art work have been archived in a Journal volume, suggesting that very few extensive investigations into the problem are available. Conference proceedings, which provide a faster visibility of researcher's work on a competitive topic such as Android, account for over 80% of the publications. The four workshop papers [51], [36], [28], [26] that we have identified in the SLR are providing radically new approaches to the problem of repackaged apps detection, but only focus on repackaged malware.

Except for the considered approaches that explicitly target the detection of repackaged Android apps, our literature search also identified several papers focusing on detecting third-party libraries [75], [76], [77]. Although these papers are not considered in this work, we believe their approaches can generally be adapted to detect repackaged apps as well. For example, Li et al. [75] have introduced LibD to identify third-party libraries, including multi-package ones, which are categorised based on the internal code dependencies of candidate libraries. Interested readers are encouraged to follow their research paper for more details.

## 3 Overview of Challenges in Repackaged App Detection

Before detailing the different solutions presented in the literature, we propose to review the challenges that researchers should seek to address in repackaged app detection. These challenges are brought up by the realities in the app industry, practical requirements for assessing a detection approach, as well as specificities of Android development.

**(1)Meeting market-scale constraints.**
Android developers have produced millions of apps distributed in several markets, raising scalability issues in the detection of repackaged apps. In this work, we consider that the scalability of detecting repackaged apps is referred to two challenges: (1) combinatorial explosion and (2) impractical aspect, e.g., to have the original app counterparts in the dataset.

Regarding combinatorial explosion, let us take Google Play as an example, the official Android app market has hosted already over 3 million Android apps[4], while several

4. https://www.appbrain.com/stats/number-of-android-apps

alternative markets such as AppChina have also passed the one-million mark. A simple and intuitive, pairwise similarity comparison between apps (combinatorial explosion) is thus not scalable in practice. For example, using such an approach to detect repackaged apps in Google Play alone, one would need to perform about $C^2_{3*10^6}$ comparisons. If we consider a computing platform with 10 cores each starting 10 threads to compare pairs of apps in parallel, it would still take several months to complete the analysis when optimistically assuming that each comparison would take about 1ms.

From a practical point of view, unfortunately, the problem is further exacerbated by the fact that repackaged apps and their original counterparts are often hosted on different markets. A pairwise comparison approach would then require collecting as many apps as possible across all markets and repositories. Failing to collect such app set would result in an unfair evaluation for repackaged app detection approaches. Indeed, on the one hand, because of missing original apps, some repackaging detection approaches (e.g., pairwise-based approaches) would be unsuccessful for flagging some repackaged apps, although those approaches by themselves are capable of achieving that. On the other hand, given a flagged repackaged app (e.g., by ML-based classifiers), not being able to identify its original counterpart in the testing app set does not necessarily mean the flagged app is a false alarm, because it could be the case that the testing app set is just not big enough, where the original counterpart happens to not be in it.

**(2)Having a reference dataset.**
Despite the awareness in academia and industry on the problem of app repackaging, the community lacks relevant datasets to support research work. Building a large and consistent ground truth of repackaging pairs indeed requires substantial efforts. Unfortunately, unless such efforts are made, we can hardly expect significant advances towards producing reliable approaches and tools for addressing repackaged app detection. Indeed, on the one hand, in the absence of a reference dataset, which can serve as a pseudo ground truth, state-of-the-art work cannot be benchmarked and compared one against the other. On the other hand, existing approaches that claim to be successful cannot convince the reader on the precision of their techniques, since confirmation is manually performed by the authors and the detected repackaged apps are not disclosed to the community for additional checking.

**(3)Recognising the original app in a repackaging pair.**
Given a repackaging pair, constituted by two similar apps, one being a repackaged version of the other, it is commonly accepted that it remains challenging to distinguish which app is the original [60], [66]. Instead, the literature often relies on heuristics such as app packaging/compilation time (e.g., timestamp of *classes.dex* ). Although such heuristics are intuitive, they are not fully reliable in a sophisticated malware development scenario. Indeed, it is possible for malware writers to manipulate compilation time of their repackaged apps [78]. Yet, a number of state-of-the-art approaches depend on such heuristics to flag repackaged apps in the wild.

**(4)Accounting for potential obfuscation.**
Obfuscation is known to be effective to help developers to hide their code logic for preventing potential plagiarism. Many obfuscation algorithms have been implemented in frameworks such as DexGuard [79] and SandMark [80] which are already used in the Android community. At the same time, however, obfuscation can be leveraged by pirates and malware writers to evade the detection of their repackaging operations. As shown by Huang et al. [81], most static approaches for repackaged app detection are ineffective in the presence of obfuscated apps.

**(5)Dealing with noise of common libraries.**
Common libraries, such as the popular *com.google.ads* and *com.revmob* advertisement libraries, which are extensively used across many apps, can significantly impact the effectiveness in repackaged app detection [31]. Indeed, when common libraries are substantially larger than core functionality code, a pairwise comparison approach can lead to false positives, presenting two different apps, but with similar libraries, as a repackaging pair. Similarly, when a large library is replaced during repackaging by another library, a pairwise comparison will fail to detect the repackaging scenario, leading to a false negative. To overcome this challenge, researchers build whitelists of common libraries that are filtered out during repackaged detection processes [48]. Unfortunately, it is also challenging to build an exhaustive list of common libraries.

**(6)Constructing and exploiting a call/dependency graph.**
Call and dependency graphs are appealing for repackaged app detection as they can abstract the behaviour implemented in a software to allow effectively identifying similar behaviour [82], [83]. Unfortunately, there are several challenges in constructing an Android call/dependency graph that will be reliable for comparison experiments. First, Android is event-based and most behavioural actions are performed via user-triggered events (e.g., clicking a button) or system events (e.g., incoming phone call), through callback methods. A call graph may not properly account for the sequences in which callback methods are called. Second, the Inter-component communication mechanism further involves the use of callback methods to allow interaction among different parts of an app. Since those parts are not directly linked at the code level, the constructed call/dependency graph of the app will eventually be incomplete, depending on the choice of starting point for exploring the app. Finally, heavy use of reflection further complicates the building of sound call/dependency graphs [84].

The size of the graphs can also challenge the detection of repackaged apps. Indeed, when the graphs are small (e.g., with less than four nodes), comparisons often lead to numerous false positives [48]. When the graphs are very large, the challenge of finding isomorphisms may become prohibitive.

**(7)Dealing with corner-cases.**
Besides the aforementioned challenges, some corner cases are often eluded in the literature of app analysis. First, some apps cannot be decompiled by popular Baksmali and Soot de-compilers. Indeed, malware writers may intentionally include code which is specifically engineered to prevent such compilers to work [85]. Second, all apps are not strictly packaged with common assets: for example, some apps do not have layouts, which may cause failures for approaches

that assume that apps always do. Finally, app hardening techniques, where the main app code in *classes.dex* is encrypted and loaded at runtime through Java Native Interface (JNI), also raises the bar for the research in repackaged app detection [86].

**(8)Dealing with legal issues.**
Aside from technical barriers, legal concerns, including copyright/licensing issue in exposing third-party apps and liability in redistributing malware can impact the advances in research on repackaged app detection. Indeed, as previously warned by Bodden et al. [87], data security and user privacy on one side and intellectual property rights on the other side, are slowly emerging in the field of informational self-protection. Thus, for example, researchers are often forced to hold back their dataset, hindering adequate comparisons that could lead to tangible improvement of the state-of-the-art. As suggested by Rasthofer et al. [88], there is a need in our community to analyze legal issues.

## 4 Review of State-of-the-art Approaches

The papers collected for the SLR include an approach and experiments related to repackaged apps detection. We characterise the different approaches and discuss their evaluation scenarios.

### 4.1 Taxonomy of Approaches

Table 3 provides details on categorisation of the different state-of-the-art approaches, highlighting the various features each leverages in its proposed approach. Repackaged app detection can be performed statically or dynamically. We also note that there are static approaches which do not analyse the bytecode in the app package for their detection process. Instead, they solely rely on the resource files accompanying the code. Various information from apps are leveraged as features for identifying repackaged apps. Such features can be extracted from metadata (e.g., permissions recorded in the Manifest file), from the code (e.g., call graphs), or from runtime data (e.g., execution traces).

Based on our review of the 57 state-of-the-art studies, we propose a taxonomy of 5 categories for the design of state-of-art approaches:

***Similarity computation***-based approaches, developed in 42 out of 57 papers, are the most common methodology adopted in the literature. Since Androguard [90], [74], which has proposed algorithms for pairwise comparison of apps, several variants using code information (e.g., DNADroid [2] with dependence graphs, DroidMOSS [70] with fuzzy hashing-based fingerprints), layout/resource information (e.g., DroidEagle [39]) or a combination of both (e.g., ResDroid [57], ViewDroid [59]) have been developed. Several approaches have further been proposed to improve the scalability of the state-of-the-art in pairwise comparison. Generally, these involve a two-step process during which the apps are first pre-processed to extract features that best summarise them. PiggyApp [66] builds vectors using normalised values of extracted features. Thus, instead of computing the similarity between apps based on their code, the authors compute the distance of vectors associated with the apps. Although this state-of-the-art work attempts to escape the scalability problem with pairwise comparisons by relying on the Vantage Point Tree data structure to partition the metric space, it still requires the dataset to contain exhaustively the original apps as well as their repackaged versions. Later, Chen et al. [48], [41] have proposed to abstract app method code into a single geometric characteristic value, a graph score, to allow even faster comparisons. Their approaches are however also unusable in practice in the context of the myriads of Android markets which are difficult to crawl [11] at once so as to have all potential original and repackaged apps in the search space.

***Runtime monitoring***-based approaches, used in 5 approaches, record or/and extract specific information during dynamic execution (or installation) of apps to check whether an app is repackaged or not. Most of those approaches (e.g., AppInk [63]) aim at repackaging deterrence by providing means for market maintainers to arbitrate/validate whether a watermarked app has been repackaged.

***Supervised learning***-based approaches, implemented in 5 approaches, extract feature vectors from app data and train classifiers that will be used to predict whether an app is repackaged or not. DR-Droid [28] reuses known features from the malware detection community and applies it to each of the statically identified loosely-coupled parts of a repackaged app. SCSDroid [67] compares dynamically recorded system call sequences against some pre-learned runtime information of known families for detecting repackaged malware.

***Unsupervised learning***-based approaches, developed in 4 approaches, regroup apps in different sets using advanced learning algorithms with features that can split apps based on their similarity. We differentiate these approaches from simple Similarity computation-based ones, as they radically try to improve the scalability of the pairwise comparison between apps, by focusing on apps that are likely to be repackaged from one another.

***Symptom discovery***-based approaches, implemented in only one recent workshop paper, build on the intuitive assumption that repackaging processes leave marks on the repackaged apps. If such marks can be fully characterised, it is possible to spot the symptoms in apps. AndroidSOO [36] has recently introduced and explored a novel and easily extractable attribute called String Offset Order, which is extracted from string identifiers list in the *classes.dex* bytecode file. Such approaches can normally provide promising results as they can manage to solve most challenges at once, especially the requirement to have the original apps available in the test set, and are unlikely suffering from false positive results if the corresponding symptoms are well-defined.

During SLR paper examination, we have attempted to identify which challenges, among those enumerated in Section 3, authors have strived to address. In particular, we check that the proposed approach/methodology 1) meets market constraints (in terms of scalability and usability in practice), 2) is evaluated based on a constructed reference dataset (whatever its size and representativeness), 3) explicitly accounts for app obfuscation (to any extent), and 4) attempts to reduce the noise of common libraries. Details in Table 3 show that no approach addresses all challenges, with Market-scale constraints being the least tackled in the

TABLE 3: Summary of Examined Approaches.

| Tool | Category | Features | Dynamic | Bytecode | MS$^{\alpha}$ | GT$^{\beta}$ | OR$^{\gamma}$ | LN$^{\delta}$ |
|---|---|---|---|---|---|---|---|---|
| AndroidSOO [36] | Symptom discovery | string offset order | | | ✓ | | | |
| CodeMatch [19] | Similarity Comparison | code fuzzy hash | | ✓ | | ✓ | ✓ | ✓ |
| FUIDroid [22] | Similarity Comparison | layout tree | | | | ✓ | ✓ | ✓ |
| APPraiser [23] | Similarity Comparison | resource files | | | | | ✓ | |
| RepDroid [24] | Similarity Comparison | layout group graph | ✓ | | | ✓ | ✓ | |
| SimiDroid [25] | Similarity Comparison | method statements, resource files, components | | ✓ | | ✓ | | |
| GroupDroid [26] | Similarity Comparison | control flow graph | | ✓ | | ✓ | | |
| CLANdroid [27] | Similarity Comparison | Identifiers, APIs, Intents, Permissions, and Sensors | | ✓ | | ✓ | | ✓ |
| Li et al. [31] | Similarity Comparison | method-level signature | | ✓ | | | | ✓ |
| RepDetector [33] | Similarity Comparison | inputs/outputs of methods | | ✓ | | | ✓ | ✓ |
| Wu et al. [44] | Similarity Comparison | HTTP distance | ✓ | ✓ | | | ✓ | ✓ |
| FSquaDRA2 [30] | Similarity Comparison | signature of resource files | | | | ✓ | ✓ | |
| SUIDroid [34] | Similarity Comparison | layout tree | | | | | ✓ | ✓ |
| DroidClone [29] | Similarity Comparison | control flow pattern | | ✓ | | | ✓ | |
| Niu et al. [32] | Similarity Comparison | method-level signature | | ✓ | | | ✓ | |
| AndroSimilar2 [37] | Similarity Comparison | entropy of byte block | | ✓ | | | ✓ | |
| AndroSimilar [62] | Similarity Comparison | entropy of byte block | | ✓ | | | ✓ | |
| DroidEagle [39] | Similarity Comparison | visual resources | | | | ✓ | ✓ | |
| ImageStruct [40] | Similarity Comparison | images | | | | | | |
| Soh et al. [43] | Similarity Comparison | user interfaces | ✓ | ✓ | | ✓ | ✓ | ✓ |
| Chen et al. [38] | Similarity Comparison | method-level signature powered by NiCad [89] | | ✓ | | | | |
| MassVet [41] | Similarity Comparison | centroid of UI structures, method-call graphs | | ✓ | | ✓ | ✓ | ✓ |
| DroidKin [50] | Similarity Comparison | meta-info and n-gram bytecode/opcode | | ✓ | | | ✓ | |
| Ruiz et al. [58] | Similarity Comparison | count-, set-, sequence-, and relationship-based objects | | ✓ | | | ✓ | ✓ |
| Linares-Vásquez et al. [55] | Similarity Comparison | count-, set-, sequence-, and relationship-based objects | | ✓ | | | | |
| Chen et al. [48] | Similarity Comparison | centroid of control flow graph (CFG) | | ✓ | | ✓ | | ✓ |
| PLayDrone [56] | Similarity Comparison | signature of resource files | | | | | | |
| FSquaDRA [53] | Similarity Comparison | signature of resource files | | | | ✓ | | |
| ViewDroid [59] | Similarity Comparison | ICC-based view graph | | ✓ | | | ✓ | ✓ |
| DroidSim [52] | Similarity Comparison | component-based control-flow graph | | ✓ | | | ✓ | ✓ |
| AndRadar [47] | Similarity Comparison | method-level signature | | ✓ | | | | |
| Kywe et al. [54] | Similarity Comparison | app name, description, icon, screenshot | | | | | | |
| PiggyApp [66] | Similarity Comparison | APIs, permissions, Intents | | ✓ | | | ✓ | ✓ |
| DroidAnalytics [65] | Similarity Comparison | API sequences | | ✓ | | | ✓ | ✓ |
| AdRob [60] | Similarity Comparison | data-dependency graph | | ✓ | | | | ✓ |
| DroidMOSS [70] | Similarity Comparison | opcode sequences, developer certificate | | ✓ | | | ✓ | ✓ |
| JuxtApp [71] | Similarity Comparison | k-grams of opcode sequences | | ✓ | | | | ✓ |
| DNADroid [2] | Similarity Comparison | program/data dependency graph | | ✓ | | | ✓ | ✓ |
| Androguard [68] | Similarity Comparison | method-level signature | | ✓ | | | ✓ | |
| Potharaju et al. [72] | Similarity Comparison | abstract syntactic tree | | ✓ | | | ✓ | |
| Ruiz et al. [73] | Similarity Comparison | count-, set-, sequence-, and relationship-based objects | | ✓ | | | | |
| WuKong [45] | Similarity Comparison | API call sequences , variable occur times | | ✓ | | ✓ | ✓ | ✓ |
| Kim et al. [35] | Similarity Comparison | runtime API invocations | ✓ | | | | ✓ | |
| ResDroid [57] | Unsupervised Learning | activities, permissions, intent filters, event handlers, etc. | | ✓ | | ✓ | ✓ | ✓ |
| AnDarwin2 [46] | Unsupervised Learning | program dependency graph | | ✓ | | | ✓ | ✓ |
| AnDarwin [61] | Unsupervised Learning | program dependency graph | | ✓ | | | ✓ | ✓ |
| DroidMat [69] | Unsupervised Learning | permissions, intents, components, API calls, ICC | | ✓ | | | | |
| DAPASA [21] | Supervised Learning | coefficient/distance of sensitive subgraph/motifs | | ✓ | ✓ | | ✓ | ✓ |
| DR-Droid2 [20] | Supervised Learning | user interactions, sensitive APIs, permissions | | ✓ | ✓ | | ✓ | ✓ |
| DR-Droid [28] | Supervised Learning | user interactions, sensitive APIs, permissions | | ✓ | ✓ | | ✓ | ✓ |
| DroidLegacy [51] | Supervised Learning | frequency of API calls in primary module | | ✓ | ✓ | ✓ | ✓ | |
| SCSdroid [67] | Supervised Learning | system call sequences | ✓ | ✓ | | | | |
| PICARD [42] | Runtime Monitoring | execution trace | ✓ | ✓ | | | | |
| DIVILAR [49] | Runtime Monitoring | virtualization-based protection | ✓ | ✓ | | | | |
| DroidMarking [9] | Runtime Monitoring | watermarking | ✓ | ✓ | | | | |
| AppIntegrity [64] | Runtime Monitoring | package name | ✓ | | | | | |
| AppInk [63] | Runtime Monitoring | watermarking | ✓ | ✓ | | | | |

MS$^{\alpha}$: Market-Scale, GT$^{\beta}$: Ground Truth, OR$^{\gamma}$: Obfuscation Resilience, LN$^{\delta}$: Library Noise.

literature.

We further study the most represented similarity computation-based approaches. Details enumerated in Table 4 show that the simple Jaccard index is the most shared similarity metric. A plethora of approaches are then trying different algorithms for computing the similarity scores.

Overall, the literature mostly describes approaches that statically detect repackaged apps by analysing the app bytecode. Most of the approaches are based on pairwise similarity comparison, which is unfortunately not suitable for market-scale analyses. We remind the readers that in this work the scalability issue is referred to the problems of (1) combinatorial explosion and (2) absence of original apps. Indeed, even when some approaches find relevant features for efficiently speeding the comparison (e.g., Andarwin [46] can analyse an app in 109 seconds), pairwise comparison-based and unsupervised learning-based approaches are still facing the issue of requiring the presence of the original app to find its repackaged versions. Nevertheless, pairwise similarity comparison is still useful and is often necessary. Actually, in general, the analysis results of any advanced approaches, which may meet market scalability requirements, must still be vetted and confirmed via a pairwise comparison that validates the high similarity score between the suspicious repackaged app and another app.

Most state-of-the-art approaches cannot scale to millions of Android apps. They are thus not practical for market maintainers.

## 4.2 Review of Evaluation Setups and Artefacts

Authors of state-of-the-art approaches have argued that their proposed features enumerated in Table 3 are effective for identifying repackaged apps. However, in the absence of a comprehensive comparative assessment of existing approaches, one question remains open for the community:

TABLE 4: Distance metrics used in Similarity computation approaches.

| Algorithm | Formula[1] | Approaches | Count |
|---|---|---|---|
| Jaccard | $\frac{\lvert X \cap Y \rvert}{\lvert X \cup Y \rvert}$ | PLayDrone [56], FSquaDRA [53], ViewDroid [59], PiggyApp [66], JuxtApp [71], Wu et al. [44], Ruiz et al. [73], Ruiz et al. [58], Linares-Vásquez et al. [55], Li et al. [31], SimiDroid [25], APPraiser [23], Kim et al. [35] | 13 |
| Euclidean | $\sqrt{\sum_1^n (x_i - y_i)^2}$ | Potharaju et al. [72], Soh et al. [43] | 2 |
| Normalized Compression | $\frac{L_{X\|Y} - min\{L_X, L_Y\}}{max\{L_X, L_Y\}}$ | Androguard [68], AndRadar [47] | 2 |
| Mahalanobis[2] | $\sqrt{\sum_1^n \frac{(x_i - y_i)^2}{s_i^2}}$ | RepDetector [33] | 1 |
| Manhattan | $\frac{\sum_1^n \lvert x_i - y_i \rvert}{\sum_1^n (x_i + y_i)}$ | WuKong [45] | 1 |
| Cosine | $\frac{\sum_1^n x_i y_i}{\sqrt{\sum_1^n x_i^2}\sqrt{\sum_1^n y_i^2}}$ | Kywe et al. [54], CLANdroid [27] | 2 |
| Customized/Other | FUIDroid [22], DroidClone [29], Niu et al. [32], DroidAnalytics [65], DNADroid [2], DroidMOSS [70], ImageStruct [40] RepDroid [24], SUIDroid [34], MassVet [41], AdRob [60], DroidSim [52], DroidEagle [39], DroidKin [50], AndroSimilar2 [37] GroupDroid [26], Chen et al. [38], Chen et al. [48], FSquaDRA2 [30], AndroSimilar [62], CodeMatch [19] | | 21 |

[1] $x_i$, $y_i$ are elements of feature set $X$, $Y$.
[2] $s_i$ in Mahalanobis algorithm is the standard deviation of $x_i$, $y_i$ over the sample set.

*what is the minimal feature set that is most effective in discriminating repackaged apps from non-repackaged ones?*

We survey the origin of datasets used for experiments, their sizes, and the availability of tools and data from state-of-the-art approaches for use by other researchers. Table 5 provides the details of this assessment information for the reviewed publications.

**Tool availability** Among the 57 publications proposing approaches for detecting repackaged apps, only 7 have made their tool support available.

**Datasets availability** Only 4 approaches have publicly released a ground truth of similarities among Android apps. Five (5) approaches have released the original set of apps where they searched for repackaged apps.

**Dataset size** There is a huge variation among the sizes of datasets used in the evaluation setups of state-of-the-art approaches. Five studies have gone over the one-million apps mark, 10 studies have analysed more than 100 thousand apps (although less than 1 million), 30 studies have analysed between 1000 and 100,000 apps, 7 papers have analysed between 100 and 1000 apps. Three papers have even been assessed on less than 100 apps. Given the size of the official market alone, there is room to improve the scale of the experiments performed in the literature.

**Dataset diversity** We also checked the origin of the datasets and found that many approaches collect their experimental datasets from 1 or few sources. We note that those approaches that used several sources are not necessarily the ones that assessed on the largest datasets.

We have further investigated to what extent state-of-the-art approaches have been compared in the literature. Given the lack of data sharing and the unavailability of tool support from competitor approaches, little comparative evaluation has been presented in the literature. Among the 57 publications, only nine (9) have performed a comparative study against the similarity scores of other approaches (e.g., the Androguard [90] tool). Fig. 2 summarises the graph of comparison among the different state-of-the-art approaches. We note that other comparative assessments are performed on authors' previous studies (which, by the way, are not available to others) or by replicating, to the best of their effort, some basic similarity computation-based approach.

TABLE 5: Summary of Examined Approaches based on Their Evaluation Metrics.

| Publication | Tool Available | Dataset Available | #. of Apps | Genome |
|---|---|---|---|---|
| CodeMatch [19] | ✓ | ✓ | (10000,100000) | |
| DR-Droid2 [20] | | | (1000,10000) | ✓ |
| DAPASA [21] | | | (10000,100000) | ✓ |
| FUIDroid [22] | | | (10000,100000) | |
| APPraiser [23] | | | (1000000, $\infty$) | |
| RepDroid [24] | ✓ | ✓ | (100,1000) | |
| SimiDroid [25] | ✓ | ✓ | (1000,10000) | |
| GroupDroid [26] | | | (1000,10000) | ✓ |
| CLANdroid [27] | ✓ | ✓ | (10000,100000) | |
| DR-Droid [28] | | | (1000,10000) | ✓ |
| DroidClone [29] | | | (100,1000) | |
| FSquaDRA2 [30] | ✓ | | (1000,10000) | |
| Li et al. [31] | | ✓$^\alpha$ | (1000000, $\infty$) | |
| Niu et al. [32] | | | - | |
| RepDetector [33] | | | (1000,10000) | |
| SUIDroid [34] | | | (100000,1000000) | |
| Kim et al. [35] | | | (100,1000) | |
| AndroidSOO [36] | | | (10000,100000) | ✓ |
| AndroSimilar2 [37] | | | (10000,100000) | ✓ |
| Chen et al. [38] | | | (1000,10000) | ✓ |
| DroidEagle [39] | | | (1000000, $\infty$) | |
| ImageStruct [40] | | | (10000,100000) | ✓ |
| MassVet [41] | | | (1000000, $\infty$) | |
| PICARD [42] | | | (0,100) | |
| Soh et al. [43] | | | (100,1000) | |
| Wu et al. [44] | | | (1000,10000) | |
| WuKong [45] | | | (100000,1000000) | |
| AnDarwin2 [46] | | | (100000,1000000) | |
| AndRadar [47] | | | (100000,1000000) | ✓ |
| Chen et al. [48] | | | (10000,100000) | |
| DIVILAR [49] | | | (0,100) | |
| DroidKin [50] | | | (1000,10000) | ✓ |
| DroidLegacy [51] | | | (1000,10000) | ✓ |
| DroidMarking [9] | | | (100,1000) | |
| DroidSim [52] | | | (100,1000) | ✓ |
| FSquaDRA [53] | ✓ | | (10000,100000) | |
| Kywe et al. [54] | | | (10000,100000) | |
| Linares-Vásquez et al. [55] | | ✓$^\alpha$ | (10000,100000) | |
| PlayDrone [56] | | ✓$^\alpha$ | (1000000, $\infty$) | |
| ResDroid [57] | | | (100000,1000000) | |
| Ruiz et al. [58] | | | (100000,1000000) | |
| ViewDroid [59] | | | (10000,100000) | |
| AdRob [60] | | | (100000,1000000) | |
| AnDarwin [61] | | | (100000,1000000) | |
| AndroSimilar [62] | | | (10000,100000) | ✓ |
| AppInk [63] | | | (0,100) | |
| AppIntegrity [64] | | | (10000,100000) | |
| DroidAnalytics [65] | | ✓$^\alpha$ | (100000,1000000) | |
| PiggyApp [66] | | | (10000,100000) | |
| SCSdroid [67] | | | (100,1000) | |
| Androguard [68] | ✓ | | - | |
| DNADroid [2] | | | (10000,100000) | |
| DroidMat [69] | | | (1000,10000) | |
| DroidMOSS [70] | | | (10000,100000) | |
| JuxtApp [71] | | | (100000,1000000) | |
| Potharaju et al. [72] | | | (1000,10000) | |
| Ruiz et al. [73] | | ✓$^\alpha$ | (1000,10000) | |

✓$^\alpha$: dataset without repackaging labels

Finally, we investigate how the accuracy of repackaged app detection is evaluated in state-of-the-art literature. In

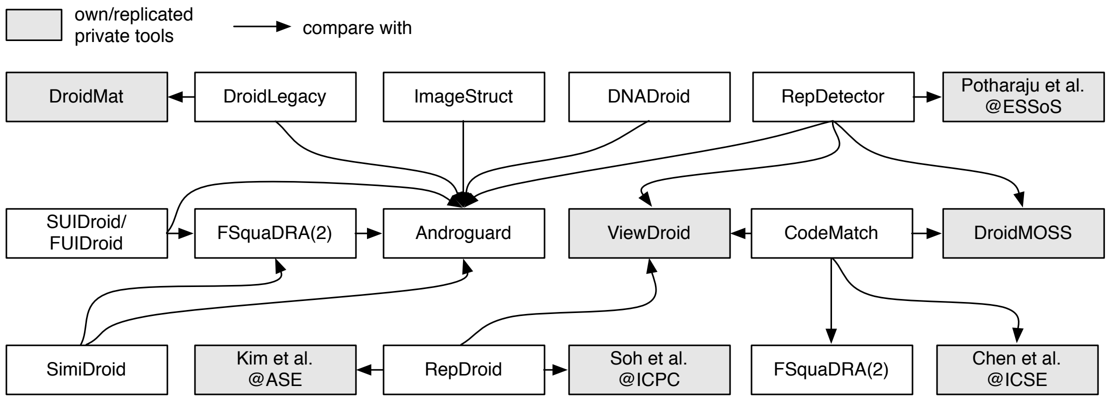


Fig. 2: Relationship of comparison among state-of-the-art approaches.

the absence of an external ground truth, authors apply their approach on random datasets and then manually verify the findings, on a sampled subset, to compute efficiency, leading to the introduction of potential researcher bias. For evaluating their recent centroid-based approach, Chen et al. [48] have randomly selected and checked 359 apps among thousands of apps that are flagged as cloned and found that their approach has zero false positive. A few approaches (7, cf. Fig. 2) use Androguard as a proxy to check the similarity of the detected repackaging pairs, adding some confidence to their evaluation setup. Supervised Learning approaches rely on the Genome dataset or other malware from VirusTotal to build training and test sets.

Non-disclosure of tools and datasets is leading to redundant research and does not encourage innovation since there is limited opportunities to reproduce, validate and compare.

## 5 Dataset Construction

State-of-the-art work in the literature claims high-performance rates for their proposed approaches. Unfortunately, their shortcomings in opening their datasets and implementations to comparative assessment by other researchers are actually blocking further research into the problem. Since Android app repackaging remains relevant today[5], we propose to reboot this research topic, in the hope of encouraging novel contributions that will tackle efficiently the different challenges enumerated in Section 3.

To that end, we propose to **build an extensive dataset**, namely RePack, for assessing repackaged detection algorithms. Such a dataset includes a set of repackaged apps accompanied with a "proof" of their repackaged state by providing the original apps with which they form repackaging pairs, following the same definition: A repackaged app pair (1) has at least 80% of code similarity between its two apps and (2) has its two apps are signed by different developers. Our work builds upon the popular AndroZoo repository, which can serve as an exchange repository for describing one's dataset using the hash values of apps.

The collection of RePack is done in a systematic way. Fig. 3 illustrates an overview of the construction process. We leverage the AndroZoo dataset [11], which (by the time of this study) includes over 5 million apps continuously crawled from 13 markets including the official Google Play and several alternatives markets such as AppChina, as well as online repositories such as F-Droid and the Genome project. To find repackaging pairs, we take the traditional, time-consuming, but most accurate, approach of performing similarity computations. To optimize the process, we devise a two-phase approach for identifying repackaging pairs. We now detail these two phases separately.

### 5.1 Splitting the Search Space

Because of time and computing resources constraints, it is virtually infeasible to perform pairwise comparisons for all possible app combinations in a dataset such as AndroZoo. Instead, we propose to rely on a clustering-based approach to split the search space, so as to focus on comparing only likely similar apps. As illustrated in Fig. 3, this phase is actually made up of three steps:

- Step (1): Feature Extraction. We abstract each app into a representative feature vector. To ensure processing speed, we focus on features that are easily extractable from an APK file. Those include class names, declared permissions, declared actions and intent-filters. These features will prevent for example from regrouping a game app with a messenger app and will ensure that similar apps are included together in the same cluster.
- Step (2): EM Clustering. We leverage the Expectation Maximization (EM) algorithm [91] to regroup the AndroZoo apps into different clusters. EM is preferred to other popular clustering algorithms, such as K-mean, because it does not require to be parameterised with the number of clusters that should be produced.
- Step (3): Candidate Pairing. At the end of this phase, we consider the set of apps in each cluster and form candidate combinations of repackaging pairs. Given two apps in a cluster, the candidate pair is formed by considering the one created before (based on the creation time of the DEX file) as the original app while the remaining one as the repackaged version. Because we

5. As demonstrated in the Lookout blog [6], which details their analysis of the various repackaging versions of the popular Pokemon Go app (repackaging with a trojan included, repackaging for cheating, repackaging with adware included, etc.)

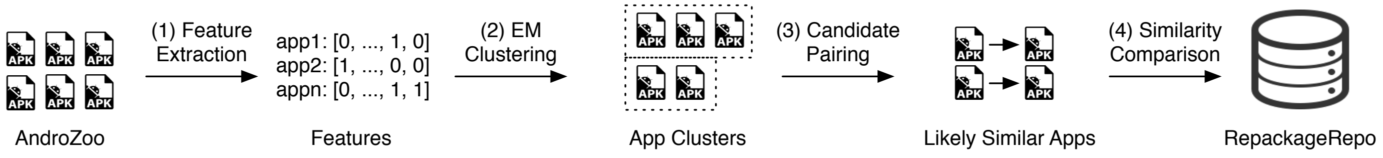


Fig. 3: Overview of RePack Construction. Step (1), (2), and (3) for Splitting the Search Space. Step (4) for Fast, Approximate, Brute-force Similarity Comparison.

consider repackaging to be mostly a parasite activity, we drop candidate pairs where the apps have been signed with the same developer certificate. Indeed, apps that are signed by the same certificate are usually considered to be app variants of a same company or app versions of the same app, which are unlikely to be repackaged versions.

## 5.2 Fast, Approximate, Brute-force Similarity Comparison

Given a candidate pair of apps ($app_1$, $app_2$) found in a cluster, we compute four similarity metrics, and calculate a score based on Formula (1).

- *identical* ($I$), which represents the number of methods (signature + body) which are shared by both apps;
- *similar* ($S$), representing the number of similar (same signature but different body contents) methods between two apps;
- *new* ($N$), representing the number of new methods that were added in $app_2$ in comparison with $app_1$; and
- *deleted* ($D$), representing the number of such methods that exist in $app_1$ but not in $app_2$.

$$similarityScore := max\{\frac{|I|}{|I|+|S|+|D|}, \frac{|I|}{|I|+|S|+|N|}\} \quad (1)$$

To ensure a fast computation of the above metrics, we use an approximative representation of app method contents by mapping the different statement types to alphabet characters. For example, the simplified code snippet presented in Listing 1 could be represented by string $acc$, where an interface and virtual invocation statement are respectively mapped to character $a$ and $c$. All variables are thus dropped from the comparison. The contents of the different methods thus go through a code-to-text transformation leading to short strings for which efficient similarity analysis algorithms exist. By ignoring easily-manipulable variable names, our code-to-text transformation enables the brute-force comparison to be resilient to simple obfuscations which are commonly performed during repackaging. Although we have attempted to fasten the pairwise comparisons, our similarity analysis still takes roughly one month to finish on all the candidate pairs.

Finally, after confirming the effectiveness of our pairwise comparison methodology, we further set the similarity threshold to 80% to decide that a pair of apps is a repackaging pair. We aim to be conservative by selecting a threshold that is more strict than those used in the literature [53], [2]. We have performed experiments to validate that the approximation in Phase 2 of the approach is preserving

```
$r2 = interfaceinvoke $r1.<WindowManager: Display
     getDefaultDisplay()>();
virtualinvoke $r2.<Display: int getWidth()>();
virtualinvoke $r2.<Display: int getHeight()>();
```

Listing 1: Simplified Code Snippet of Android Apps. This Code Snippet is Presented at the Jimple Level, where Our Similarity Analysis is Implemented on top of Soot, in which Jimple is the Default Intermediate Representation (IR) Code.

similarity scores. These experiments consisted in computing exact statements similarity and comparing the similarity scores against those obtained after approximations of method contents. Overall, we ran the experiments on all the repackaged app pairs obtained in this work. We found no difference between the two experiments, i.e., with exact statements compared and the same threshold at 80%, all the RePack app pairs are still flagged as repackaged pairs. Furthermore, we also validate that the scores that we obtain are similar to those obtained with state-of-the-art tools such as AndroGuard, which is basically in agreement with our collected pairs (more details are given in Section 5.5).

## 5.3 Overall Results

Based on the AndroZoo dataset, we are able to collect 15,297 repackaging pairs to be shared as repackaging reference dataset. We find that many apps are repackaged several times by different attackers, with a minimal, mean, and maximum times of 1, 2.168, and 176, respectively. Overall, our RePack dataset includes 15,297 repackaged apps for 2,776 original apps. Table 6 shows the top three original apps that are repackaged by over 100 different attackers. Interestingly, all those three apps are from the official Google Play store, suggesting that Google Play apps are somehow more favoured by attackers to repackage and distribute.

Fig. 4 plots the distribution of DEX size of all the collected RePack apps, where the size ranges from 3.67 KB (minimal) to 16,180 KB (maximum), with a median and mean size of 965.2 KB and 88.67 KB respectively. This distribution suggests that RePack is quite diverse, containing small-size, middle-size, and large-size Android apps. We then go one step deeper to investigate the changes of DEX size between the two apps of a given pair. Among the 15,297 app pairs, over 70% of them have shown that repackaging will eventually enlarge the DEX file, suggesting additional code is usually injected during repackaging. However, for nearly 30% of repackaging cases, the DEX size of repackaged apps are smaller than that of the original apps.

Fig. 5 and Fig. 6 further respectively plots the distribution of the number of Resource Files and Java Files for the benchmark apps. The number of resource files ranges

TABLE 6: Top Three Original Apps that have been Repackaged by Over 100 Different Attackers.

| SHA256 (Original App) | Package Name | Market | Repackaged Certs |
|---|---|---|---|
| 9CC2EAEF8636AE77794ACDF085A2C241A98E620581391D41FBC5D39D69528E53 | com.algorythmicstudios.droid | play.google.com | 176 |
| 34084F29D69F2056E776B1F6BA3B1174D07C192F4EF2AF7CE793B0DE97C517C9 | com.theindievelopers.stacktothirty | play.google.com | 109 |
| D178AA7FC82311AF6536ECD7872FAEC9C1111E233EF25798F1E157F375862FCC | com.HatchWorks.BabyDiscoverAquatic | play.google.com | 107 |

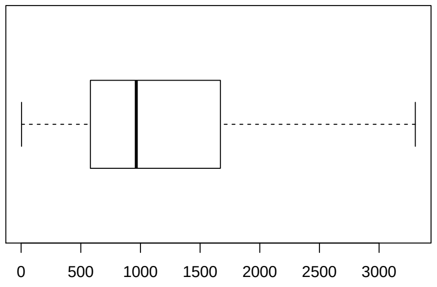


Fig. 4: Distribution of DEX Size of RePack Apps (in KB).

from 12 to 10,529[6], where the median and mean values are 150 and 247, respectively. For Java files, the number ranges from 6 to 7225, where the median and mean values are 662 and 1050, respectively. These two distributions once again suggest that the RePack dataset is quite diverse, where both apps with a small number of resource/Java files and with a large number of resource/Java files are included. Interestedly, Spearman's rank correlation coefficient (i.e., $\rho < 0.35$) suggests that there is no strong correlation between the number of resource files and the number of Java files, further confirming the diversity of our benchmark dataset.

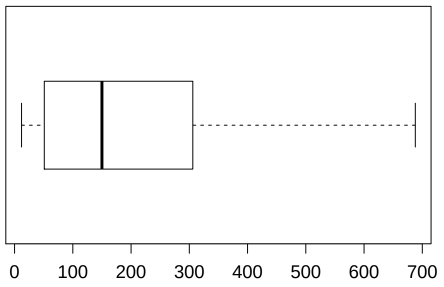


Fig. 5: Distribution of the Number of Resource Files.

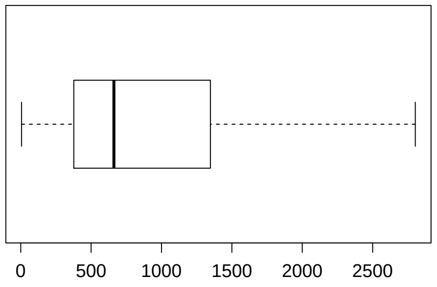


Fig. 6: Distribution of the Number of Java Files.

Fig. 7 plots the distribution based on the diff of Creation time between the two apps of a given repackaging pair. Comparing to the creation time of the original apps, the repackaging delay ranges from several days to several years with an average, 88.67 days. This distribution also suggests that our collected RePack apps are diverse, containing different repackaging cases that could be interesting to malware analysts to investigate.

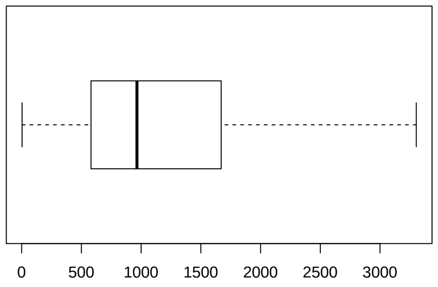


Fig. 7: Distribution of Creation Time Diff between RePack Pairs (in Day).

Finally, we use the AVClass [92] labelling tool to assess the diversity of repackaged malicious apps in our dataset. Given a repackaged app, we consider it as malicious as long as one of VirusTotal hosted anti-virus products flags it as such. For a given app and its labels from VirusTotal anti-virus engines, AVClass outputs a unique name of malware or adware family. We feed AVClass with all the repackaged malicious apps in our dataset. AVClass has successfully classified 5,960 repackaged apps of our dataset into 45 known families, while the Genome dataset includes 49 families. Moreover, our dataset includes about 9,337 repackaged apps that AVClass is not able to categorize into a known malware or adware family.

Overall, all the aforementioned studies, covering different aspects, suggest that our collected repackaged pairs, namely RePack, is quite diverse, and therefore is reliable to be leveraged to support various analyses. Last but not the least, as discussed in our previous work [31], for detecting repackaged Android apps, common libraries may cause both false positive and false negative results. Hence, we extend our benchmark to also provide library information for each repackaged pairs. We hope the extended information can encourage the community to innovate in-depth analyses for better understanding the facts between libraries and app clones, including malicious ones. The library usage (i.e., the 1,113 libraries summarised by Li et al. [31]) of each app in the benchmark has been also made publicly available in our replication dataset. Moreover, to facilitate the use of library information for Android-based analyses, we provide a research-based prototype tool called LibExcluder for generating library-free versions of given Android apps. The goal of LibExcluder is to remove library code from a given Android app. Given a whitelist of common libraries, LibExcluder takes as input an Android app and outputs a new app version, which is generally as same as the inputted

6. The maximum number is considered as an outlier so that it is not presented at the boxplot. This explanation also applies to other boxplots.

TABLE 7: Manual validation results of randomly selected repackaged app pairs (five examples).

| Repackaged App Pair | Code Similarity | Resource Similarity | Manual Observation (Same) Package Name | App Name | Icon | Main UI Skeleton |
|---|---|---|---|---|---|---|
| CCAC0E/3CDCEF | 0.990 | 0.982 | ✗ | ✗ | ✗ | ✓ |
| AE064D/6F2081 | 0.905 | 0.357 | ✗ | ✗ | ✗ | ✓ |
| CA9ABD/78940D | 0.987 | 0.982 | ✓ | ✓ | ✓ | ✓ |
| 207372/707ED8 | 0.866 | 0.929 | ✗ | ✗ | ✗ | ✓ |
| FF57A0/74E764 | 0.995 | 0.776 | ✗ | ✗ | ✗ | ✓ |

one except that the code belonging to libraries configured in the whitelist are excluded. Therefore, LibExcluder presents to existing state-of-the-art approaches a new app version where library code no longer exists. Without any modification (i.e., being non-invasive), existing approaches such as IccTA [93] can benefit from this work to perform library-free analyses.

Our dataset is, to the best of our knowledge, the largest one containing repackaged app pairs. It is built from a representative set of apps, and includes a diverse set of repackaged apps.

## 5.4 Manual Validation of Random Samples

One of the major goals of constructing a benchmark of Repackaged Android apps is to support replication and comparison studies by the community. Towards demonstrating this ability, we need to ensure in the first place that our constructed benchmark is reliable. To this end, we use a statistical formula to compute our sample size for manual checking. This formula, extracted from [94] (page 75), is presented in Equation 2, where the following parameters are used:

- $population$ is set to $15,297$, the size of our dataset;
- The confidence interval $c$ is set to 10% (i.e., $c = 0.1$);
- $p$ is related to the variability in the population. Since we don't know the variability, $p$ is set to $0.5$, i.e., maximum variability.
- $z$ is related to the confidence level. As explained in [95] (page 3), "$z^2$ *is the abscissa of the normal curve that cuts off an area* $\alpha$ *at the tails* ($1-\alpha$ *equals the desired confidence level, e.g., 95%*)". We set the confidence level at 95%, and thus $z$ is set to $1.96$ (as explained in [95]).

$$ss := \frac{\frac{z^2 * p * (1-p)}{c^2}}{1 + \frac{\frac{z^2 * p * (1-p)}{c^2} - 1}{population}} \quad (2)$$

As a result, the sample size $ss$ is equal to $94.48$. Eventually, we select a round number of 100 app pairs and manually validate their similarities.

It is actually non-trivial to decide whether a given two apps (which share over 90% of the code and are signed by different developers) are repackaged from one to another manually. We hence resort to dynamic analysis to validate the selected pairs. Given a pair of apps, we manually install them respectively on two emulators set up with exactly the same configurations. After the apps are installed, we manually launch and play with them and observe the similarity and difference between the two apps. Among the 100 selected pairs, our manual validation confirms that 89 of them are clearly repackaged pairs (sharing exactly the same UI page or at least similar UI skeleton), giving a precision of 89% at least for our harvested benchmark[7]. The remaining 11 pairs have big changes in their UI pages that cannot be soundly confirmed. Nonetheless, we remind the readers that those app pairs, despite having different UI pages, have shared over 80% of code and thus could be repackaged app pairs as well.

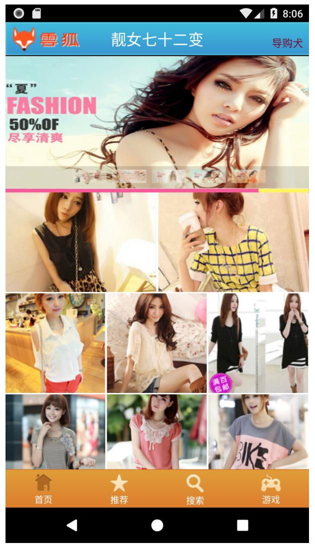
Table 7 illustrates five samples of the details we have observed from our manual validation process. The first column presents the hash values[8] of the selected pairs. The second and third columns illustrate the similarity scores yielded by SimiDroid, based on its method-based and resource-based similarity analyses, respectively. The last four columns show whether the package name, app name, icon and the main UI skeleton are respectively the same between the two apps in a pair. Regarding the main UI skeleton, we consider it is the same as long as the layout is more or less the same. For example, in this work, we consider the two pages shown in Fig. 8 (collected from pair *FF57A0/74E764*) have the same UI skeleton. If two apps (1) have over 90% of code similar from one another, (2) are signed by different developers (or teams), and (3) have similar look and feel (i.e., similar UI skeleton), we consider these two apps as true repackaged app pair. Hence, our manual validation confirms that 89% of the randomly selected app pairs from our benchmark are true repackaged app pairs, suggesting that our benchmark is quite reliable. Subsequently, our benchmark should be capable of supporting replication and comparison studies between state-of-the-art approaches.

Table 7 further reveals that a repackaged app may (or may not) change the package name, app name and icon of the original app. These findings once again suggest that our benchmark is diverse and is representative to different types of repackaging cases.

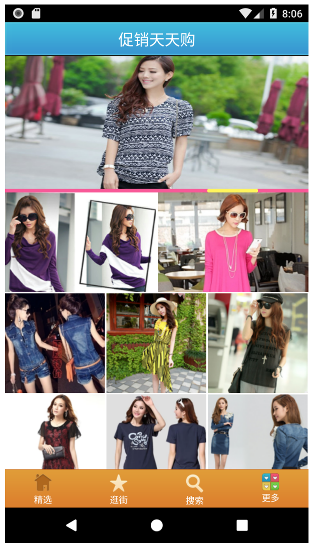
## 5.5 Supporting Replication and Comparison of State-of-the-art Work

Towards demonstrating the ability to support replication and comparison studies, we revisit a couple of literature approaches for repackaging detection which have made their associated tools available. As shown in Table 5, only seven research approaches have made some tools available.

7. Recall is not computed due to the complexity of confirming non-repackaged app pairs (i.e., false negative results).
8. SHA256s, only the first six letters are shown.



(a) Original App.



(b) Repackaged App.

Fig. 8: The first page of apps FF57A0 (left) and 74E764 (right), which more or less share the same UI Skeleton.

Nevertheless, not all of them are applicable for our study: CLANdroid, CodeMatch and RepDroid cannot be directly executed as the former two approaches require a corpus preprocessing step and while the last one expects a complicated runtime environment with hard-coded platform dependencies. FSquaDRA2 and FSquaDRA share the same basics in their approaches, experimenting one of them should be enough. As a result, we focus on replicating the experiments of FSquaDRA (resource-based comparison), Androguard (approximate code-based comparison), and SimiDroid (exact code-based comparison) based on our RePack dataset.

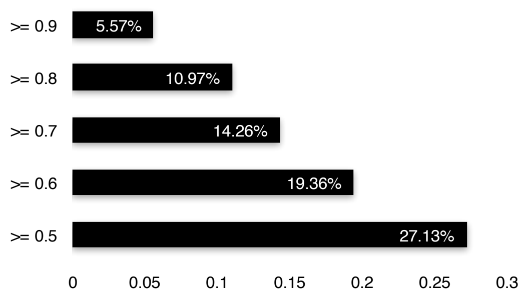


Fig. 9: FSquaDRA Results by Different Thresholds.

In this work, we use *recall* to characterise the ability of state-of-the-art tools to detect repackaged apps[9]. If a pair in the RePack benchmark is flagged as a repackaged pair, we consider it as a True Positive (TP) result. Otherwise, if the pair is flagged as a non-repackaged pair, we consider it as a False Negative (FN) result. Then, the recall of a given tool can be computed based on the following formula:

$$recall := \frac{TP}{TP + FN} \tag{3}$$

Due to exceptions thrown by AndroGuard, FSquaDRA and SimiDroid, we eventually consider results for 8,078 pairs, among the pairs in RePack, where all the three tools have successfully finished their analyses. Given the same threshold at *80%*, AndroGuard is in agreement with our collected dataset for 86% pairs while FSquaDRA only agrees for 11% pairs. In other words, while similarity scores by AndroGuard would allow identifying 86% of pairs in our dataset (i.e., recall is 86%), similarity by FSquaDRA is only aligned for 11% of the RePack pairs (i.e., recall is 11%). Regarding the similarity results of SimiDroid, we only observe seven pairs (out of in total 11,255 successfully finished analyses) that have their similarity scores less than *80%*, resulting in almost 100% recall. Fig. 9 further plots the results of FSquaDRA by different thresholds. The lower threshold considered, the higher the results achieved. Nonetheless, even with lower thresholds, the results of FSquaDRA are still not comparable to that of AndroGuard and SimiDroid. These results show that the similarity analysis we have performed for building possible repackaging pairs is highly in line with the analysis of AndroGuard and SimiDroid but not in line with the analysis of FSquaDRA. The disagreement of FSquaDRA could be explained by one of the findings summarized by Li et al. [96], where the authors experimentally demonstrate that repackaging may also largely manipulate the resource files which would thus lead to poor results for resource-based similarity analysis tools. Fig. 10 further shows the distribution of the similarity results of these three tools, where the median and mean values are 99%, 85.8% for AndroGuard, 30%, 35% for FSquaDRA, and 99.47%, 96.78% for SimiDroid, respectively. Mann-Whitney-Wilcoxon (MWW) test demonstrates that the difference between the obtained scores of AndroGuard and FSquaDRA (and between that of FSquaDRA and SimiDroid) are significant. Cliff's effective size estimation nevertheless suggests that the results of AndroGuard and FSquaDRA (and also that of SimiDroid and FSquaDRA) are largely and positively correlated, which has been also demonstrated by the authors of FSquaDRA.

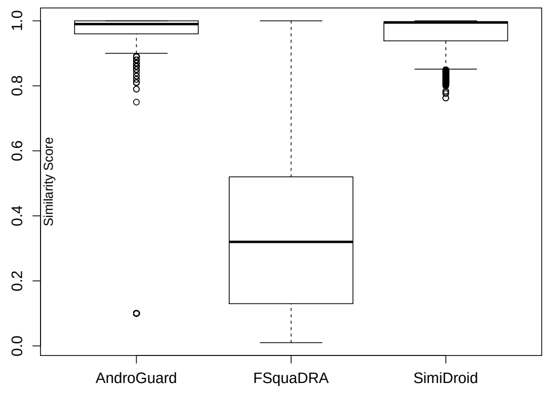


Fig. 10: Distribution of Similarity Scores Given by AndroGuard, FSquaDRA and SimiDroid.

9. The reason why *precision* is not considered is that in this work we assume all the repackaged pairs in our benchmark are true positives. Therefore, there will be no false positives reported and hence the precision of state-of-the-art tools will be always 100%.

Finally, we proceed ourselves to propose a straightforward scalable repackaged app detection approach based on our insights on analysing repackaging pairs in the RePack dataset. This approach uses classification technique (specifically machine learning based 10-Fold cross validation) based on symptoms (i.e., component capability declaration, app permission requests, mismatch between package and Launcher component names, package name diversity, favoured sensitive apis) of the repackaging process appearing in apps. In total, our approach extracts 521 features, including four boolean and int features presented and 107, 266, 144 features for repackaging favourite *new permissions*, *new capabilities* and *sensitive APIs*, respectively. Overall, this straightforward approach achieves 79% Recall for distinguishing repackaged apps from non-repackaged ones. Although these performance scores are lower than many performances (up to 100%) recorded in the literature, they are obtained, to the best of our knowledge, from the first readily reproducible experiments with an approach that is not based on pairwise similarity computation. The performance of this technique, measured on RePack, can be considered as a reasonable baseline to improve within the community. For the sake of space, we provide the details of this approach as a supplementary document for interested readers to refer to [97].

## 6 Priority Research Directions

Our investigation into the problem of repackaged app detection has yielded an enumeration of challenges that the research community must strive to address. We also propose in this section some priority research directions towards creating most efficient detection systems.

**1) Comprehending Repackaging Processes:** As discussed previously, machine-learning approaches are appealing to ensure scalability and practicality in repackaged app detection. Nevertheless, their design must be based on a solid feature engineering process which should identify features that are truly representative of the repackaging phenomenon. In this work, we have proposed preliminary investigations into such features and focused on features which were easy and fast to extract. More extensive analyses into a large ground truth of repackaging pairs can provide more insights on novel and discriminative features. For example, we have not studied the impact of repackaging on the density of call graphs (since edges are inserted/deleted by modifications of original apps).

A more extensive investigation into repackaging processes can further provide a taxonomy of repackaging similar to the types of code clones surveyed by Roy et al. [98]. Such a taxonomy can help researchers more precisely inform readers on the target of their research.

**2) Graph Analysis:** In static analysis, call/dependency graphs are known to be a reliable representation of app structure. Since repackaging is mostly not invasive, a repackaged app may be formed by loosely-coupled modules [28]. An efficient analysis of call/ dependency graphs can thus be leveraged in a precise and practical means to detect repackaged apps, i.e., without relying on the availability of the original apps for comparison.

**3) Dynamic Analysis:** Dynamic analysis, although accurate, is expensive to implement by all market maintainers. Furthermore, because malware can now easily detect when they are run in a sandboxed environment, they may hide the behaviour implemented in the payload added via repackaging. Nevertheless, new avenues of research involved a form of crowdsourcing can be explored, where market maintainers collect runtime information from users' devices for posterior analyses of similar/divergent app behaviour.

**4) Repackaging Deterrence:** Finally, besides providing market maintainers with means to screen markets for repackaged apps, researchers should provide repackaged app detection in markets, developers should be provided with means for protecting their apps from repackagers. AppInk [63] and DIVILAR [49] are rare examples of research work attempting to propose approaches for repackaged deterrence in the security community. We feel that the Software engineering community can heavily contribute in this area as well with watermarking techniques.

## 7 Threats to Validity

Although we have tried our best to collect relevant papers as much as possible by following a well-defined methodology for conducting SLR, our results may have still missed some publications. More specifically, the state-of-the-art repository search engines (e.g., the one provided by Springer) are not so accurate, usually resulting in many irrelevant papers and may also miss some relevant ones. To further mitigate this threat, we have also conducted a backwards-snowballing based on already considered publications.

Another threat to validity of our study lies in the exhaustiveness of our dataset. However, we have leveraged the AndroZoo largest available research dataset of Android apps to mitigate potential threats. Nevertheless, we have empirically shown that our collected dataset, namely RePack, is quite diverse, e.g., containing both small-size and large-size apps and containing over 40 distinct types of repackaged malware families.

As a known challenge, so far, this is no straightforward means to pinpoint whether a given app is the original version of a given repackaging app pair. Hence, the original apps in our collected dataset may not be the final original ones as the creation time of DEX files can be manipulated. Furthermore, as shown in a recent study, because of multi-generation repackaging, the identified original app of a given repackaging pair may also be a repackaged version of a previous app, making the identified original app even more wrongful. Furthermore, the creation time of Android apps can be also manipulated, making the identified original app even more wrongful. Nevertheless, it remains an interesting future work for our community to tackle.

Because of AV disagreements, the AV labels collected from VirusTotal may not be perfect, nor does the AVClass classifications. However, in this work, we only use VirusTotal to get quick insights on our constructed dataset. We thus encourage our fellow researchers to explore this direction to propose more promising approaches for pinpointing Android malware.

The fact that libraries are not excluded in this work could lead to inaccurate results as well. The rationale behind this decision is that (1) Despite much efforts have been put on investigating Android libraries, we feel that the momentum of Android research has not yet produced a complete set of common libraries to support in-depth analysis of Android apps, including the whitelist leveraged in the extension of this work. (2) As argued by literature work, libraries could be favoured by attackers to inject malicious payloads so as to repackage Android apps with the ability to reuse the same exploitation to other apps that have leveraged the same popular library. Removing libraries may also exclude the opportunity to pinpoint the malicious behaviours of repackaged malicious apps. Nonetheless, we believe that the consideration of libraries could be crucial to repackaged app detection approaches. We, therefore, have provided to our community a research-based prototype tool for generating library-free versions of given Android apps, aiming at encouraging the community to innovate in-depth analyses for better understanding the facts between libraries and app clones, including malicious ones.

Finally, despite our benchmark is carefully built following a strict definition of repackaged app pairs: (1) over 80% of code similarity and should be signed by different developers, our benchmark may lead to both under-approximate (obfuscation is not considered) and over-approximate (libraries are considered) results. Indeed, under-approximate could be reported if obfuscation (especially method signatures are manipulated) is not considered while over-approximate could be yielded if common libraries are taken into account. Additionally, the definition by itself may not be always true. For example, two apps could be independently implemented by two developers (i.e., different signatures) via cloning from the same app. Because the changes made by the cloning process can be small, the two apps may still remain over 80% of code similarity (mainly contributed by the app that the two apps cloned from). Based on our definition, we could still flag this two apps as a repackaged app pair. Nonetheless, although the repackaged app version is not directly modified from another app, we believe it could still be considered (to some extent) as a repackaged pair. Moreover, the current construction process of the benchmark is based on the extracted properties at the method level, which might introduce biases to repackaged app detectors that are implemented based on other means rather than the static analysis of methods. Nevertheless, it is non-trivial and probably time-intensive to manually verify the validity of all the app pairs in our benchmark. Hence, we commit to continuously improve the validity of the benchmark and eventually provide an oracle for evaluating repackage detection tools.

## 8 Related Work

Repackaging is an important issue in the Android ecosystem that must be continuously dealt with by the research community. We argue that the research around repackaged app detection is blocked by state-of-the-art work which record high performance rates in the literature while hindering comparative assessments. Related to our work are (1) studies that provide constructive discussions on the value of contributions in a research domain, (2) general research on clone detection and (3) frameworks for assessing repackaged detection approaches.

Critical review of research: Recently, Blackburn et al. present to our community a pragmatic guide to assessing empirical evaluations [99]. They state that an unsound claim can misdirect an entire field. In this work, we attempt to follow their guidelines and thus to avoid potentially unsound claims. Although we have not focused on finding fallacies in evaluation of state-of-the-art work, our motivation is similar to that of Monperrus [100] and his critical review of a state-of-the-art automated repair work. We have discussed the challenges that researchers in this domain must keep in mind and further provide new data, approach and ideas for potential research directions.

Code clone & software plagiarism detection: App repackaged detection deals with similar concepts as in traditional code clone detection approaches [83], [101], [102], [103], [82], [104], which either work at a higher level where files are directly compared [104] or work at a lower level where fragments of code (or graphs/trees) are considered, their objectives were to measure the similarity of code fragments. Even if the notion of clone fragment, be it a method, file or package, could be very useful for app similarity measurements, it is not sufficient in the context of Android, since Android apps have intensively leveraged framework and library code. In other words, two apps with similar code fragments are not necessarily similar.

Closely related to our new proposed approach is Clonewise [105], which, to the best of our knowledge, is the first to consider clone detection as a classification problem. Our approach, also in contrast with most state-of-the-art work, considers repackaging detection as a classification problem to enable a practical use in real-world settings.

Nevertheless, machine learning techniques, by allowing sifting through large sets of applications to detect malicious apps based on measures of similarity of features, have been extensively leveraged to conduct large-scale malware detection [106], [107], [108], [109]. Unfortunately, through extensive evaluations, the community of ML-based malware detection has not yet shown that current malware detectors for Android are efficient in detecting malware in the wild. One among the candidate reasons to this situation is the fact that most malware are actually repackaged from benign apps, their ML-based features are probably similar to those extracted from benign apps, making them indistinguishable for ML-based malware detection. Indeed, as pointed out by Meng et al. [110], the current feature-based malware detection approaches are not enough because they cannot provide detailed information beyond their mere detection. They thus propose an alternative approach that leverages semantic features (based on deterministic symbolic automaton (DSA)) to comprehensive Android malware and thereby to detect and classify them. Therefore, we believe that the detection of repackaged Android apps contributes to also valuable ingredients for detecting malicious Android apps.

Assessment of repackaged detection algorithms: Complementary to our work, Huang et al. [81] have early proposed a framework to comprehensively evaluate the obfuscation resilience of repackaging detection algorithms. They demonstrate the obfuscation problem for repackaged detection algorithm by experimenting on Androguard. Following state-of-the-art work now regularly report on their performance with this framework. With our work, we aim for the same momentum of using a common dataset for evaluating approaches.

## 9 Conclusion

We proposed to review the challenges of repackaged app detection in the Android ecosystem. We then performed a review of state-of-the-art work and highlighted the necessity to put new life into the research on repackaged app detection. We contribute in this direction by building a comprehensive dataset of repackaging pairs, aiming at supporting replications of existing approaches and implications of new research directions.

## Acknowledgment

This work was supported by the Monash-Warwick Alliance Catalyst Fund (2018/2019), by the European Union, under the SPARTA project, by the Fonds National de la Recherche (FNR), Luxembourg, under projects CHARACTERIZE C17/IS/11693861 and Recommend C15/IS/10449467, and by the University of Luxembourg, under the VulFix project. The authors would like to thank the anonymous reviewers who have provided insightful and constructive comments that have led to important improvements in several parts of the manuscript. The authors also appreciate the help received from Timothée Riom who have helped to verify the literature review results.

## References

[1] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 431–444, New York, NY, USA, 2013. ACM.

[2] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: detecting cloned applications on android markets. In *ESORICS*, 2012.

[3] Li Li, Daoyuan Li, Tegawendé F Bissyandé, David Lo, Jacques Klein, and Yves Le Traon. Ungrafting malicious code from piggybacked android apps. *Technical Report*, 2016.

[4] Jason Ankeny. Feds seize android app marketplaces applanet, appbucket in piracy sting, August 2012. http://www.fiercemobilecontent.com/story/feds-seize-android-app-marketplaces-applanet-appbucket-piracy-sting/2012-08-22.

[5] Ustwo games. https://goo.gl/TuZnz4. Accessed: 2016-08-25.

[6] Pokemon go: New tampered apps & what you can do. https://blog.lookout.com/blog/2016/07/15/pokemon-go/. Accessed: 2016-08-25.

[7] Jin-Hyuk Jung, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. Repackaging attack on android banking applications and its countermeasures. *Wireless Personal Communications*, 73(4):1421–1437, 2013.

[8] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

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

[10] Haoyu Wang, Hao Li, Li Li, Yao Guo, and Guoai Xu. Why are android apps removed from google play? a large-scale empirical study. In *The 15th International Conference on Mining Software Repositories (MSR 2018)*, 2018.

[11] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *MSR*, 2016.

[12] Li Li, Jun Gao, Médéric Hurier, Pingfan Kong, Tegawendé F Bissyandé, Alexandre Bartel, Jacques Klein, and Yves Le Traon. Androzoo++: Collecting millions of android apps and their metadata for the research community. *arXiv preprint arXiv:1709.05281*, 2017.

[13] Staffs Keele. Guidelines for performing systematic literature reviews in software engineering. 2007.

[14] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[15] 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. Technical report, SnT, 2016.

[16] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 2018.

[17] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.

[18] Ayush Kohli. Decisiondroid: a supervised learning-based system to identify cloned android applications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 1059–1061. ACM, 2017.

[19] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 638–648. ACM, 2017.

[20] Ke Tian, Danfeng Daphne Yao, Barbara G Ryder, G Tan, and Guojun Peng. Detection of repackaged android malware with code-heterogeneity features. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[21] Ming Fan, Jun Liu, Wei Wang, Haifei Li, Zhenzhou Tian, and Ting Liu. Dapasa: detecting android piggybacked apps through sensitive subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 12(8):1772–1785, 2017.

[22] Fang Lyu, Yaping Lin, and Junfeng Yang. An efficient and packing-resilient two-phase android cloned application detection approach. *Mobile Information Systems*, 2017, 2017.

[23] Yuta Ishii, Takuya Watanabe, Mitsuaki Akiyama, and Tatsuya Mori. Appraiser: A large scale analysis of android clone apps. *IEICE TRANSACTIONS on Information and Systems*, 100(8):1703–1713, 2017.

[24] Shengtao Yue, Weizan Feng, Jun Ma, Yanyan Jiang, Xianping Tao, Chang Xu, and Jian Lu. Repdroid: an automated tool for android application repackaging detection. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*, pages 132–142. IEEE, 2017.

[25] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Simidroid: Identifying and explaining similarities in android apps. In *The 16th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom 2017)*, 2017.

[26] Niccolò Marastoni, Andrea Continella, Davide Quarta, Stefano Zanero, and Mila Dalla Preda. Groupdroid: Automatically grouping mobile malware by extracting code similarities. 2017.

[27] Mario Linares-Vásquez, Andrew Holtzhauer, and Denys Poshyvanyk. On automatically detecting similar android apps. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[28] 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.

[29] Shahid Alam, Ryan Riley, Ibrahim Sogukpinar, and Necmeddin Carkaci. Droidclone: Detecting android malware variants by exposing code clones. In *Digital Information and Communication Technology and its Applications (DICTAP), 2016 Sixth International Conference on*, pages 79–84. IEEE, 2016.

[30] Olga Gadyatskaya, Andra-Lidia Lezza, and Yury Zhauniarovich. Evaluation of Resource-based App Repackaging Detection in Android. In *Proceedings of the 21st Nordic Conference on Secure IT Systems*, NordSec 2016, pages 135–151, 2016.

[31] 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.

[32] Haofei Niu, Tianchang Yang, and Shaozhang Niu. Clone analysis and detection in android applications. In *Systems and Informatics (ICSAI), 2016 3rd International Conference on*, pages 520–525. IEEE, 2016.

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

[34] Fang Lyu, Yapin Lin, Junfeng Yang, and Junhai Zhou. Suidroid: An efficient hardening-resilient approach to android app clone detection. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 511–518. IEEE, 2016.

[35] Daeyoung Kim, Amruta Gokhale, Vinod Ganapathy, and Abhinav Srivastava. Detecting plagiarized mobile apps using api birthmarks. *Automated Software Engineering*, 23(4):591–618, 2015.

[36] Hugo Gonzalez, Andi A Kadir, Natalia Stakhanova, Abdullah J Alzahrani, and Ali A Ghorbani. Exploring reverse engineering symptoms in android apps. In *Proceedings of the Eighth European Workshop on System Security*, page 7. ACM, 2015.

[37] Parvez Faruki, Vijay Laxmi, Ammar Bharmal, Manoj Singh Gaur, and Vijay Ganmoor. Androsimilar: Robust signature for detecting variants of android malware. *Journal of Information Security and Applications*, 22:66–80, 2015.

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

[39] Mingshen Sun, Mengmeng Li, and John C.S. Lui. Droideagle: seamless detection of visually similar android apps. In *WiSec*, 2015.

[40] Sibei Jiao, Yao Cheng, Lingyun Ying, Purui Su, and Dengguo Feng. A rapid and scalable method for android application repackaging detection. In *ISPEC*, 2015.

[41] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Zou Wei, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *USENIX Security*, 2015.

[42] Alessandro Aldini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Detection of repackaged mobile applications through a collaborative approach. *CCPE*, 2015.

[43] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. Detecting clones in android applications through analyzing user interfaces. In *ICPC*, 2015.

[44] Xueping Wu, Dafang Zhang, Xin Su, and WenWei Li. Detect repackaged android application based on http traffic similarity. *SCN*, 2015.

[45] 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.

[46] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of android application clones based on semantics. *TMC*, 2014.

[47] Martina Lindorfer, Stamatis Volanis, Alessandro Sisto, Matthias Neugschwandtner, Elias Athanasopoulos, Federico Maggi, Christian Platzer, Stefano Zanero, and Sotiris Ioannidis. Andradar: Fast discovery of android applications in alternative markets. In *DIMVA*, 2014.

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

[49] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. Divilar: diversifying intermediate language for anti-repackaging on android platform. In *CODASPY*, 2014.

[50] Hugo Gonzalez, Natalia Stakhanova, and Ali A Ghorbani. Droidkin: Lightweight detection of android apps similarity. In *International Conference on Security and Privacy in Communication Systems*, pages 436–453. Springer, 2014.

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

[52] Xin Sun, Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. Detecting code reuse in android applications using component-based control flow graph. In *IFIP SEC*, 2014.

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

[54] Su Mon Kywe, Yingjiu Li, Robert H. Deng, and Jason Hong. Detecting camouflaged applications on mobile application markets. In *ICISC*, 2014.

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

[56] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *SIGMETRICS*, 2014.

[57] 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.

[58] 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.

[59] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *WiSec*, 2014.

[60] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *MobiSys*, 2013.

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

[62] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, Manoj Singh Gaur, and Ammar Bharmal. Androsimilar: robust statistical feature signature for android malware detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, pages 152–159. ACM, 2013.

[63] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *AsiaCCS*, 2013.

[64] Timothy Vidas and Nicolas Christin. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *CODASPY*, 2013.

[65] Min Zheng, Mingshen Sun, and John C.S. Lui. Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware. In *TrustCom*, 2013.

[66] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *CODASPY*, 2013.

[67] 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.

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

[69] 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.

[70] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *CODASPY*, 2012.

[71] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: a scalable system for detecting code reuse among android applications. In *DIMVA*, 2012.

[72] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *ESSoS*, 2012.

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

[74] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101, 2011.

[75] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. Libd: Scalable and precise third-party library detection in android markets. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 335–346. IEEE, 2017.

[76] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: fast and accurate detection of third-party libraries in android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 653–656. ACM, 2016.

[77] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. In

*Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 356–367. ACM, 2016.

[78] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Moonlightbox: Mining android api histories for uncovering release-time inconsistencies. In *The 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*, 2018.

[79] Dexguard. https://www.guardsquare.com/dexguard. Accessed: 2016-08-25.

[80] Sandmark: A tool for the study of software protection algorithms. http://sandmark.cs.arizona.edu. Accessed: 2016-08-25.

[81] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In *International Conference on Trust and Trustworthy Computing*, pages 169–186. Springer, 2013.

[82] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD06*, pages 872–881. ACM Press, 2006.

[83] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical report, School fo Computing, TR 2007-541, Queen's University, 2007.

[84] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Droidra: Taming reflection to support whole-program analysis of android apps. In *The 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016.

[85] Wenbo Yang, Yuanyuan Zhang, Juanru Li, Junliang Shu, Bodong Li, Wenjun Hu, and Dawu Gu. Appspear: Bytecode decrypting and dex reassembling for packed android malware. In *18th International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID, pages 359–381, 2015.

[86] Yueqian Zhang, Xiapu Luo, and Haoyang Yin. Dexhunter: toward extracting hidden code from packed android applications. In *European Symposium on Research in Computer Security*, pages 293–311. Springer, 2015.

[87] Eric Bodden, Siegfried Rasthofer, Philipp Richter, and Alexander Roßnagel. Schutzmaßnahmen gegen datenschutzunfreundliche smartphone-apps. *Datenschutz und Datensicherheit-DuD*, 37(11):720–725, 2013.

[88] Siegfried Rasthofer, Steven Arzt, Max Kolhagen, Brian Pfretzschner, Stephan Huber, Eric Bodden, and Philipp Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. In *Science and Information Conference (SAI), 2015*, pages 247–256. IEEE, 2015.

[89] James R Cordy and Chanchal K Roy. The nicad clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 219–220. IEEE, 2011.

[90] Androguard: Reverse engineering, malware analysis of android applications. https://github.com/androguard. Accessed: 2016-08-25.

[91] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[92] Marcos Sebastián, Richard Rivera12, Platon Kotzias12, and Juan Caballero. Avclass: A tool for massive malware labeling. 2016.

[93] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[94] William G. Cochran. *Sampling Techniques*. Wiley Eastern Limited, 1977.

[95] Glenn D. Israel. Determining sample size. *University of Florida, series of the Program Evaluation and Organizational Development*, 1992.

[96] 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 (TIFS)*, 2017.

[97] Li Li, Tegawendé F Bissyandé, and Jacques Klein. Baseline Approach for Repackaged App Detection: A Supplement Document of the Research Paper Entitled "Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark". In *https://github.com/serval-snt-uni-lu/RePack/blob/master/RePack_Supplement.pdf*, 2019.

[98] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, May 2009.

[99] Stephen M Blackburn, Amer Diwan, Matthias Hauswirth, Peter F Sweeney, José Nelson Amaral, Tim Brecht, Lubomir Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, et al. The truth, the whole truth, and nothing but the truth: a pragmatic guide to assessing empirical evaluations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 38(4):15, 2016.

[100] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 234–242, New York, NY, USA, 2014. ACM.

[101] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95, Jul 1995.

[102] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377, Nov 1998.

[103] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[104] H.A. Basit and S. Jarzabek. A data mining approach for detecting higher-level clones in software. *Software Engineering, IEEE Transactions on*, 35(4):497–514, July 2009.

[105] Silvio Cesare, Yang Xiang, and Jun Zhang. Clonewise detecting package-level clones using machine learning. In Tanveer Zia, Albert Zomaya, Vijay Varadharajan, and Morley Mao, editors, *Security and Privacy in Communication Networks*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 197–215. Springer International Publishing, 2013.

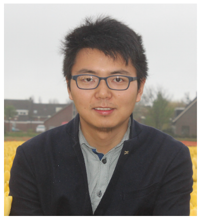
[106] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[107] Boyun Zhang, Jianping Yin, Jingbo Hao, Dingxing Zhang, and Shulin Wang. Malicious codes detection based on ensemble learning. In *Proceedings of the 4th international conference on Autonomic and Trusted Computing*, ATC'07, pages 468–477, Berlin, Heidelberg, 2007. Springer-Verlag.

[108] Justin Sahs and Latifur Khan. A machine learning approach to android malware detection. In *Intelligence and Security Informatics Conference (EISIC), 2012 European*, pages 141–147. IEEE, 2012.

[109] R. Perdisci, A. Lanzi, and Wenke Lee. Mcboost: Boosting scalability in malware collection and analysis using statistical classification of executables. In *Computer Security Applications Conference, 2008. ACSAC 2008. Annual*, pages 301–310, 2008.

[110] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan. Semantic modelling of android malware for effective malware comprehension, detection, and classification. In *ISSTA*, pages 306–317. ACM, 2016.



**Li Li** is a lecturer (a.k.a., Assistant Professor) and a PhD supervisor at Monash University, Australia. Prior to joining Monash University, he spent 1.5 years as a Research Associate at the Serval group, SnT, University of Luxembourg. He received his PhD degree in computer science from the University of 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, TSE, TIFS, TDSC, TOPS, EMSE, JSS, IST, etc. His personal website is http://lilicoding.github.io.

**Tegawendé F. Bissyandé** is a Research Scientist with SnT, University of Luxembourg. He received his PhD degree in Computer Science from the University of Bordeaux in 2013. His work is mainly related to Software Engineering, specifically empirical software engineering, reliability and debugging as well as mobile app analysis. His works 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, ICPC.

**Jacques Klein** is a Senior Research Scientist (faculty position) with SnT, University of Luxembourg. He leads a group of about 10 researchers focusing on Mobile Security and Software Engineering. Dr. Klein has standing experience and expertise on (1) successfully running industrial projects with impressive experience in data analytics, software engineering, information retrieval, etc., (2) Android security including both static analysis techniques for tracking privacy leaks and machine learning for identifying malware. Dr. Klein has been successful in publishing relevant results in top journals/conferences including TSE, TIFS, Empirical Software Engineering journal, Usenix Security, PLDI, ICSE, POPL, ISSTA, etc.