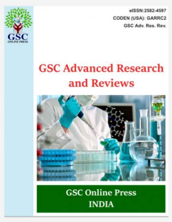
GSC Advanced Research and Reviews

eISSN: 2582-4597 CODEN (USA): GARRC2

Cross Ref DOI: 10.30574/gscarr

Journal homepage: https://gsconlinepress.com/journals/gscarr/



(Research Article)

Clone detection to prevent software piracy in android play store

Md Fahim Ahammed *

College of Engineering and Business, Gannon University, 109 University Square, Erie, Pennsylvania, 16541, United States of America.

GSC Advanced Research and Reviews, 2024, 21(03), 108–131

Publication history: Received on 18 October 2024; revised on 02 December 2024; accepted on 04 December 2024

Article DOI: https://doi.org/10.30574/gscarr.2024.21.3.0487

Abstract

Android is currently one of the most popular smartphone operating systems. Many application developers have been enticed by the enormous demand for mobile smartphone devices. The availability of reverse engineering tools for Android applications, however, also attracted virus authors' and plagiarists' attention. Cloning of applications has been a significant challenge to the Android market in recent years. However, Android accounts for the lion's share of all mobile malware globally, and its security vulnerabilities have garnered a lot of public attention. In this research, we look into how to recognize known Android malware using a clone detector. We compile a set of Android programs known to be malicious as well as a set of good programs. NiCad, a near-miss clone detector, is used to locate the classes of clones in a small fraction of the malicious programs after we retrieve the Java source code from the dex (Dalvik Executable file) file of the applications. The remaining malicious programs' source files are then searched for using these clone classes as a signature. As a control group, the non-harmful collection is utilized. In our analysis, we were able to decompile over 100 potentially harmful programs from 19 different malware families. According to our findings, a small sample of malicious programs can be used as a training set to detect 95% of known malware with a 96.88% accuracy rate and extremely few false positives. Our technique can successfully and consistently identify detrimental programs that are part of specific malware families. Moreover, illegal distribution is another part of software piracy, which is also a prevalent issue in the IT industry. We described an algorithm for the Android library licensing improvement to give insights to a developer on enhancing security measures to prevent the illegal distribution of individual Android applications.

Keywords: Android; Piracy; Clone; Malware; Security.

1. Introduction

After the growth and revolution of the use of Android phones, software piracy has rapidly grown up simultaneously. Android has dominated the smartphone market and a huge variety of Android apps have been created as smartphone usage has grown quickly. The most recent estimate of the number of apps in the Google Play Store was 2.6 million in December 2023 [1], according to data from the statistics portal Statista. Android applications continue to be the main target of attackers because they are widely used and manage highly sensitive data via interacting with IoT devices and cloud servers. Due to the high-level but straightforward bytecode language utilized, reverse engineering Android apps, for instance, is significantly simpler than on other mobile platforms, and several reverse engineering tools are available. Android apps are therefore simple to hack, repurpose, and copy.

Some programmers may steal code from open-source programs or libraries, but they break the terms of the license when they use it in a different setting. Attackers or unauthorized developers can simply reverse-engineer legitimate programs in the Android environment, copy their code, repackage it with "purpose-added" features, and re-market it in the same or different markets. According to [2], an average developer lost 14% of their advertising revenue and 10% of

* Corresponding author: Md Fahim Ahammed.

their user base due to software clones. Some attackers may even put malicious code into legal apps while repacking in order to infect unsuspecting users. 1083 out of 1260 malware samples, or 86%, were app clones with harmful payloads, according to [3]'s research revealed that almost 25% of the material in Google Play Store apps was duplicated, including several forms of spam, app rebranding, and app cloning. As a result, repackaged apps or cloned apps may violate smartphone users' rights to privacy and security as well as the copyright of the original writers.

On May 13, counterfeit computer software was seen in a store in Dhaka, the capital city of my country Bangladesh a day after a report on software piracy revealed that Bangladesh had the highest percentage of counterfeit goods in the Asia-Pacific region [4]. Even Software piracy, which refers to the unlicensed copying or distribution of software, has become an increasingly serious criminal behavior issue in Bangladesh. Bangladesh has the highest rate of piracy in the Asia-Pacific, at 92% [4]. So as per the statistics, it is a significant concerning issue to work on the prevention of software piracy specifically for the Android app to secure individual's information, security, etc.

With the rapid growth of the Android platform and the widespread availability of mobile applications, software piracy has become a significant concern for developers and app store administrators. Software piracy refers to the unauthorized distribution, copying, or modification of software, including clone apps that mimic the functionalities and appearance of legitimate applications. These clone apps pose a threat not only to the revenues of developers but also to user privacy and data security. According to Richard Stallman [5], individuals shouldn't refer to copying and distribution as "piracy" because the term is used by copyright holders to claim that such copying is just as bad as piracy and that it involves attacking ships, killing, and stealing people. Therefore, effective clone detection techniques are crucial in preventing software piracy in the Android Play Store.

Clone detection is the process of identifying similar or identical portions of code within software applications. In the context of the Android Play Store, clone detection techniques are employed to identify cloned apps that infringe upon the intellectual property rights of legitimate developers.

By detecting and removing clone apps, software piracy can be mitigated, protecting the interests of both developers and users.

The Android Play Store, being one of the largest repositories of Android applications, faces the challenge of detecting clone apps among the vast number of available apps. These clone apps often attempt to deceive users by imitating the appearance, functionality, and even the package names of popular legitimate applications. Therefore, specialized clone detection techniques are required to distinguish between genuine apps and their clones.

The objective of clone detection in the Android Play Store is two-fold: first, to identify cloned apps that infringe upon the intellectual property rights of developers, and second, to ensure the safety and security of users by preventing the distribution of malicious or modified clones that may compromise user data or privacy.

Various techniques and approaches have been proposed to detect clones in software applications, and these can be adapted and extended to the unique context of the Android Play Store. These techniques include code-based analysis, behavior-based analysis, metadata analysis, and machine learning-based approaches. By employing a combination of these techniques, clone detection systems can effectively identify and categorize clone apps based on their similarity, type, and potential impact on software piracy. The identification of dangerous software using clone detection is a topic of active research [6] [7]. Researchers have created a number of techniques with varied degrees of effectiveness for finding clones within and between source files [8]. One such tool that has demonstrated success in locating close clones in source code is the NiCad [9] clone detection tool. In this study, we show how to use a static clone detection technique to find malware in Android applications. Our prediction stipulates that near-miss clone detection will give us a way to spot changes in known harmful code. We gather both harmful and good applications. The training set and evaluation set of harmful programs are separated.

Android applications continue to be the main target of attackers because they are widely used and manage highly sensitive data via interacting with IoT devices and cloud servers. Due to the high level but straightforward bytecode language utilized, reverse engineering Android apps, for instance, is significantly simpler than on other mobile platforms, and there are numerous reverse engineering tools available. As a result, Android apps are simple to copy, reuse, and crack. Some programmers may steal code from open-source programs or libraries, but they do so in violation of the license when they use it in a different context. Attackers or unauthorized developers can simply reverse-engineer legitimate programs in the Android environment, copy its code, repackage it with "purpose-added" features, and re-market it in the same or different markets. According to Gibler et al. [6], an average developer lost 14% of their revenues from advertising and 10% of their user base to app clones. Some attackers even have the capability of inserting unwanted code into legitimate apps during repacking in order to infect unsuspecting users [3], [7], [8]. 86% of samples containing malware were app clones with harmful payloads, according to Zhou and Jiang [8]. According to Viennot et al. [9], about 25% of the material in Google Play Store apps was duplicate, including several forms of spam, app restructuring, and app cloning. Repackaged or cloned apps, as a result, violate the copyright of their original creators and risk endangering smartphone users' security and privacy.

This literature review aims to explore and analyze the existing research papers, studies, and articles related to clone detection techniques in the context of preventing software piracy in the Android Play Store. By examining the effectiveness, limitations, and advancements of these techniques, we can gain insights into the current state-of-the-art in clone detection and identify areas that require further research and improvement. Overall, clone detection plays a vital role in safeguarding the Android Play Store ecosystem by preventing software piracy, protecting developers' intellectual property, and ensuring a secure and reliable experience for users. In this paper we presented android app clone detection result and analysis in order to detect detrimental apps which is then investigated how those apps pirate the user information or crack the paid app license to make a third-party access for illegal distribution and threat over internet.

2. Literature Review

Software piracy is a major concern for developers and businesses that rely on software sales to generate revenue. One way to prevent software piracy is through the use of clone detection techniques, which aim to identify and remove unauthorized copies of software from the market. In recent years, the use of clone detection in the context of the Android Play Store has become increasingly important, as the platform has become a popular target for software pirates.

Software piracy is a vital issue for software developers, especially those who develop apps for the Android platform. One approach to prevent software piracy is clone detection, which involves identifying and removing cloned versions of an app from the Android Play Store. In this literature review, we will examine the research on clone detection to prevent software piracy in the Android Play Store.

A study conducted by [10] proposed an automated clone detection approach for detecting clones of Android apps in the Play Store. The approach used a combination of code similarity analysis and app behavior analysis to identify clones of an app. The study evaluated the approach on a dataset of 50,000 Android apps and found that it was able to detect clones with an accuracy of 96%.

Another study by [11] proposed a clone detection approach that used a combination of static and dynamic analysis techniques to identify clones. The approach was evaluated on a dataset of 7,000 apps and was found to be effective in detecting clones with a precision of 95.3% and a recall of 98.7%.

In a more recent study, [12] proposed a deep learning-based approach for clone detection in Android apps. The approach used a convolutional neural network (CNN) to extract features from app code and used a Siamese network to compare the similarity between pairs of apps. The study evaluated the approach on a dataset of 1,000 apps and found that it was able to detect clones with an accuracy of 92.4%.

In addition to these studies, several tools have been developed for clone detection in Android apps. One such tool is Clonedigger, which is an open-source tool that uses code similarity analysis to identify clones of an app. Another tool is Androguard, which is a framework for analyzing Android apps that includes a clone detection module.

Repackaging detection has been the focus of numerous researchers from various sides. Finding applications with comparable execution flows has been one of the key tracks. The authors in [13] compute the Control Flow Graphs of each application's code and cluster them to locate such apps. By using Local Sensitive Hashing to extract features from the code of applications, Andarwin groups the applications based on a subset of these features [14].

Instead of the actual application's code or execution patterns, CloneSpot is a novel technique that focuses on meta-information of applications that is taken from Android application markets. CloneSpot thus outperforms code inspection methods in terms of time effectiveness at market scale. In order to determine whether a suspect application is actually a clone, a malware analyst should manually confirm CloneSpot's output sets with potential repackaged applications. These sets can then be supplied to additional sandboxing applications [15].

Applications' execution patterns have also been researched. The authors of suggest MIGDrodid [16], a system built upon comparing the Method Invocation Graphs of all applications that rates the threat level of each one individually. Guan approach for spotting commonalities between each app's input/output symbolic representations is suggested [15]. In addition, the authors of [17] suggest searching for repackaging indicators by spotting irregularities brought up by Smali decompilers in the dex code's data section.

Many writers have tackled the issue at the market scale by putting forth scalable approaches, such as Andradar [18], which records application removals and other changes while continuously monitoring many markets. In fact, several of the suggested [19] solutions are scalable, and capable to keep up with the malware generation's daily increase in speed. In [20], the author also conducts extensive research of 200,000 Android apps to identify those that have been copied.

In fact, numerous authors have investigated the possibility that sensitive applications like financial apps, messaging applications, or even Android Home devices may fall prey to repackaging [21]. However, some authors have made an effort to use repackaging to improve security or audit applications. For example, in [22], the authors propose adding a privacy reporter component to programs to audit the use of personal data, and in [23], they suggest putting a user-level sandbox through applications using repackaging.

According to certain studies [24], code copying could result in license violations and other legal repercussions. A bad developer could like to keep his cloning operations under wraps, and he might work to obscure the cloned program in an effort to fool clone detectors. To prevent licensing concerns and disguise illegal code reuse, obfuscation is utilized [25]. Therefore, it is important to take into account how obfuscated code affects the metrics of Android app cloning. For handling apps that are obfuscated, certain experiments have been done. Software Bertilonage [26] and other signature-based methods for identifying class cloning are more susceptible to obfuscation, particularly to changes like renaming and ordering (such as switching up the order of methods, for example).

Schulze and Meyer [25] presented a case study to assess the robustness of particular clone detectors regarding such obfuscations and provided a source-code level model for semi-automated code obfuscations. The four most popular clone detection methods—text-based, token-based, AST-based, and PDG-based techniques—were contrasted and their differences were discussed [27]. They took into account code obfuscation's impacts at the source code level rather than the executable code level. The primary goal of our method is to identify duplicated apps at the source code level.

The geometric characteristics (centroid) of dependent graphs is used by Chen [28] to determine how similar two apps' techniques are. It has been shown that their solution is accurate and scalable. But it has two shortcomings. As an example, if we filtering a library using the instruction count, which we can overlook methods that are nearly identical to the invocation count. The first thing to note is that the detection relies on the sorting order. Furthermore, they filter libraries from third parties using a whitelist, which can produce misleading results.

Researchers have proposed several methods to identify malware by utilizing static analysis, dynamic analysis, and signature-based techniques in order to combat software piracy in the field of Android clone detection. Similar to our method, signature-based malware detection is a common technique. A malware family can be identified using patterns that are taken from well-known malware. These patterns are typically collections of instructions in the form of bytes [29]. Previous research [30] considered

semantics-aware malware detection when detecting these syntactic patterns via semantic-preserving transformations. Our strategy might be thought of as a type of signature generation. However, we directly compare the pattern at the source code level and our signatures are at a higher level than templatized instruction sequences. The fundamental signature matching methods are also highly dissimilar.

Feature hashing detection is an illustration of a strategy that uses a signature-based technique. A code similarity identification approach for Android apps called Juxtapp [31] is based on feature hashing. From the DEX file's translated XML representation, basic blocks are produced and labeled. A moving window of size k is used to extract k-grams of opcodes from a code sequence within each basic block of an app as features, and a hash function is then used to feature hash the k-grams into bit vectors. To compare the similarity of two apps, the Jaccard similarity between two-bit vectors is produced. Juxtapp is proficient at detecting a range of Android security concerns, including as malware, pirated software, flawed code, and repackaging.

To find unauthorized app or malware, DroidMOSS [32] uses the fuzzy hashing approach. Instructions for the app and author information are regarded as features. The sequence of computer instructions is broken up into smaller chunks to compute a hash value for each one rather than processing the complete set at once. The final fingerprint of an app is created by combining all computed hash values. A similarity score, which is derived from the computation of the edit distance between two fingerprints, is used to compare two apps. DroidMOSS is effective at locating repackaged mobile apps.

Information flow tracing in mobile applications has been proposed using both static and dynamic taint analysis. By instrumenting the Dalivk virtual machine, TaintDroid [33] is an example of dynamic taint analysis that tracks threat information flow. A very accurate static taint analysis for Android applications is called FlowDroid [34].

To organize or categorize existing Android malware, Zhou and Jiang [35] gathered more than 1200 malware samples. The DEX file is extracted by Juxtapp [31], which then examines it for code similarity comparison between Android applications. For the purpose of identifying Android malware, Crowdroid [36] utilizes dynamic analysis to examine application behavior. Static analysis [37] is based on the inspection of source code or binary files looking for ominous trends.

In order to provide a static analyst paradigm for identifying Android malware, DroidMat [38] presents a static feature-based approach. Through the creation of a program dependence graph (PDG), DNADroid [39] is a tool that can identify malicious programs. Each operation in a program's dependency is represented by a PDG. In order to leverage WALA to create PDGs for each method, DNADroid uses dex2jar [30] to convert Dalvik byte codes to Java byte codes. Based on a comparison of matching PDG pairings, similarity between two apps is found. The PDG approach is a frequently employed tool for clone detection. Since it makes use of semantic data from the program, the outcome is more accurate.

AnDarwin [39] improves DNADroid to avoid app pairwise comparisons and makes use of the semantic data of Android apps to find related apps. AnDarwin gets started by calculating a PDG for the Android program. The application is then represented by the semantic vectors that were retrieved from the PDG. In order to increase scalability, AnDarwin locates related apps by clustering semantic vectors using a more effective approach called locality-sensitive hashing (LSH) [40]. Because it exclusively analyzes programs at the Java byte code level and is independent of other data, AnDarwin has the benefit of being dependable. We employ Java source code, a higher level of code, in our method. Using clone detection at the Java source code level, we locate comparable programs.

VUDDY is a method for finding susceptible code clones that was presented by Kim and Lee [41]. Four cutting-edge tools (SourcererCC, CCFinderX, Deckard, and ReDeBug) are used to compare it. Its adaptability, speed of execution, recall, and accuracy are assessed and contrasted with those of these tools. According to the results, it can scale 1BLOC with 100% precision and 82% recall in approximately 14 hours and 17 minutes. It features the best recall and scalability, lowest execution time, and maximum precision. VUDDY can recognize type1 and type2 clones and is supported by text-based methods. However, it can only detect type 3 and type 4 clones that fall outside of its purview and is restricted to C/C++.

A technique for cross-language clone identification called LICCA is proposed by Vislavski [42]. It can find type1, type2, and type3 clones and is facilitated by hybrid approaches (such as tokenbased, tree-based, and metric-based). However, as of right now, only semantically comparable segments may be detected using LICCA's clone detection feature, leaving out functionally similar segments. Another tool, CCSharp, a three-phase PGD-based clone detection method implemented as a tool, is presented by Wang [43]. Its experiment is run on a PostgreSQL program and an alternative program against three well-known clone detection tools (NICAD, Deckard, and SourcererCC). The setup time is six minutes and forty-two seconds for the less complex program, 0.95 seconds for the PostgreSQL program, and 33 minutes and 1 second for the more complex application. We can assume that semantics clones can be detected by CCSharp since it is a PDG-based clone detector. Nevertheless, it has certain procedural constraints that prevent it from processing some procedures and some configuration setting restrictions. CloneWorks, a tool for detecting type 3 code clones, was described by Svajlenko and Roy [44]. Svajlenko and Roy, who present two configurations—conservative and aggressive—for the detection of type 3 clones, provide additional details. Its accuracy is assessed and contrasted with cutting-edge tools (iClones, NICAD, SourerCC). Results reveal that its precision, which is comparable to other tools, is 83% for a cautious setup and 93% for an aggressive configuration. On a typical workstation, it can find type 3 clones as big as 250 MLOC within just 4 hours. Although type4 clones cannot currently be detected, type1 as well as type2 clones can CloneManager, a tool for method level code clone detection backed by a hybrid technique (text-based and metric-based approaches), is proposed by Kodhai and Kanmani [43]. By using dataset from several open-source systems and prominent clone detection tools (NICAD and CLAN), the experiment is run to assess its performance. Results showed that it can identify type 1 clones with 97% precision and 95% recall, type 2 clones with 88% precision and 98% recall, type 3 clones with 100% precision and 95% recall, and type 4 clones with 100% accuracy and 100% recall. However, it might encounter problems with language dependence, uses a little more memory, and relies on manual analysis, which is prone to human error. A technique called Vincent is presented by Ragkhitwetsagul [45] for the detection of image-based code clones. This tool uses the Jeccard and EMD similarity measurements and is backed by metrics-based techniques. The experiment is carried out to evaluate its effectiveness. According to the results of the trial, Vincent is capable of scaling 241924 LOC in approximately 5 hours and 31 minutes. Its precision has been assessed and contrasted with tools that are freely accessible to the public (CCFinderX, Deckard, iClones, NICAD, and Simian).

A tool called srcSlice is introduced by Newman [46] and supported by text-based methods. On the basis of specific factors (such as execution time and scalability), its performance is assessed. It can grow a 13,000,000 LOC Linux kernel within just 7 minutes, according to the experiment's findings, but as of right now, its capabilities are just C/C++-based. AnDarwin is a tool for discovering applications with similar code on a large scale that is backed using semantic clone detection techniques, according to (Crussell, Gibler, & Chen, 2013). To assess its performance, the experiment is run. According to the results, it can find 36,106 rebranded apps and at least 4295 copied applications in just 10 hours.

A recent two-step method for identifying repackaged apps was proposed by Glanz [47] and consists of the library detection method LibDetect and the app matching tool CodeMatch. Using LibDetect, the first stage locates and gets rid of library code in obfuscated apps. To be resistant to some obfuscation techniques, it depends on code representations that abstract over a number of the actual bytecode components. The best conceptual model of the rest app code is fuzzy hashed after the initial step to ensure that repackaged apps may be recognized using CodeMatch. They retrieved about 200 sample library APKs, obfuscated the apps with DexGuard, then attempted to re-identify the originally created library methods to test the resilience of their method. Their method uses bytecode to extract feature data from a DEX file.

For the purpose of identifying Android malware, certain unique semantic-based techniques have been created. One of the newest methods, Pegasus [48], uses model checking to check the policies set by users on a new program representation called Permission Event Graph (PEG). PEG displays the API/permission level behaviors of the Android event dependencies. To represent the consequences of the event system, it captures the semantic data about an app. However, Pegasus requires users to employ formulas for temporal logic when defining an app's behavior policies. Another semantics-based technique, Apposcopy [48], creates an Inter-Component Call Graph (ICCG), a new type of program representation with particular control and dataflow attributes, as part of an analysis for malware identification code. Our approach, which is solely syntactic, is based on pattern matching. In this research, we present a clone detection method to identify malicious and unapproved Android apps.

Based on the meta data used to provide an overview of the app, DNADroid [49] initially identifies probable comparable apps. The program dependency graph (PDG), which serves to act as identifier for every application that is to be compared, is created by DNADroid in the second stage. Before comparing the remaining PDG pairs that passed the test using a subgraph isomorphism, they use a filter to prune improbable clones.

Androguard is a tool for measuring similarity that supports a number of widely used similarity metrics. By comparing identical methods that are within the dex code across the apps, the similarity is calculated. Androguard is designed for comparing two apps on a short collection of data, not for spotting cross-market app clones [50].

In order to determine the centroid, Chen [51] retrieved methods from the applications and built a 3D-control vector graph (3D-CFG). They use the centroid to gauge method level-similarity across various markets after that. Last but not least, comparable apps are grouped together using the method level similarity result.

The challenge of identifying "piggybacked" programs, which are copies with an additional malicious payload added, was the focus of Zhou et al.'s study [52]. They initially divide the code among primary and non-primary modules using the module-decoupling technique. For each core module, they then obtain a semantic feature fingerprint, and then utilize a linear arithmic search technique to find apps that are similar to each other.

Juxtapp, a technique to identify code reuse among Android apps, was proposed by Hanna [31]. Juxtapp extracts the characteristics of the apps using feature hashing and k-grams of the opcode sequences.

Juxtapp has the ability to spot instances of known malware, vulnerable code reuse, and pirated versions of the original programs.

In contrast to techniques that rely on code similarities, FSquaDRA [53] identifies Android app copies by comparing the resource files required to produce the APK. They make use of the hashes which were generated and kept in the package throughout the app signing procedure. This method is resistant to code obfuscation, although certain minor adjustments to the resources will change how similar they are.

The use of clone detection techniques to find dangerous or unlicensed software has been studied. By contrasting one malicious software family with another, Walenstein and Lakhotia [54] shown that it is possible to identify evidence where components of one software system resemble components of another. Clone detection techniques were used by Bruschi [55] to show how to identify self-mutating malware, a specific type of code obfuscation.

In a nutshell, clone detection is an effective approach for preventing software piracy in the Android Play Store. The research on clone detection has demonstrated the effectiveness of various approaches and tools for identifying clones of an app. However, further research is needed to improve the accuracy and efficiency of clone detection approaches and to develop better strategies for preventing software piracy.

Overall, these studies demonstrate the effectiveness of clone detection techniques in preventing software piracy in the Android Play Store. However, there are still challenges to be addressed, such as the large number of apps in the Play Store and the constantly evolving nature of clone apps. Future research in this area will need to address these challenges in order to develop more effective clone detection techniques.

3. Methodology

In terms of Android application there are several techniques for finding clone detection which assume static, dynamic, or mixed program analysis. In our method, malware or unauthorized app in Android applications is found via static analysis. Our primary goal is to come up with a method that might detect any harmful or pirated apps in order to obtain high recall and precision and after the detection of the harmful or pirated app we further proposed an algorithm to enhance the security measures of the android library license.

The approach for clone detection in order to prevent software piracy is illustrated in Figure 1.

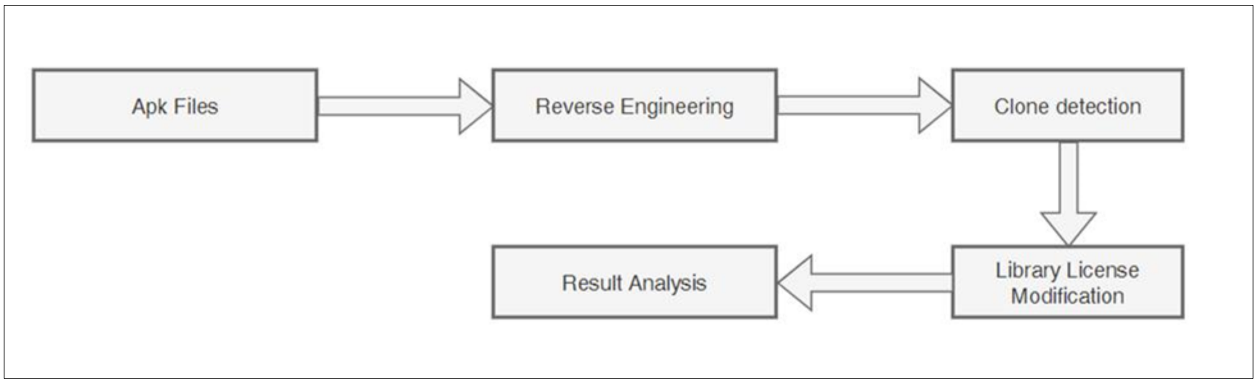


Figure 1 Procedure and method of clone detection and prevention of software piracy.

Android application distribution files, or APK files, are used as the starting point for a reverse engineering process to retrieve Java source code files. The clone detection phase, which consists of the signature generation and signature matching phases, receives the source files after the disassembling and decompiling of the android apk. In the clone detection phase, we used the NiCad clone detector tool for the two category phases that we mentioned, which task is to extrapolate or determine the clone classes of the Java files. To understand it more elaborately, we describe the basic concepts of the procedure in the following section.

3.1. Apk Collection and Inspection

We collected several Android apps from various sources such as websites, Bluestacks an Android emulator, etc. According to statista nearly 97% of the apps available on Google Play, the official Android market, are in the free category and may be downloaded for free [1]. In a similar vein, we found that the proportion of free apps is substantially higher than that of paid apps in third-party markets. As a result, we only include free apps in our dataset. Github, ApkPure, and Apkmirror, three other third-party Android stores, as well as Google Play, were used to obtain the dataset. In other markets, the apps are typically classed differently. We think that popular apps have higher repackaging values from the standpoint of copycats. We downloaded the most downloaded free programs from each market.

We used the apkpure.com and apkmirror.com website to collect third party Android apps. However, in most cases, we downloaded Android apps from the Bluestacks Android emulator. From the Bluestacks we have used Apk Extractor app from the play store. By using Apk Extractor app we extracted the specific Android app for further introspection and testing. We mostly followed this method to collect apk file of an Android app.

3.2. Reverse Engineering

Reverse engineering is the technique of examining an existing piece of software or code in order to check for any flaws or faults. The capacity to produce the source code from an executable is known as reverse engineering. This method is employed to test a program's functionality, get around security measures, etc. Therefore, reverse engineering can be defined as a technique or a process of altering a program to make it behave as the reverse engineer wants it to.

Joany Boutet has quoted Shwartz, saying, “whether it's rebuilding a car engine or diagramming a sentence, people can learn about many things simply by taking them apart and putting them back together again. That, in a nutshell, is the concept behind reverse-engineering - breaking something down in order to understand it, build a copy or improve it [56].”

Researchers [57] started coming up with strategies to decode the Dalvik Bytecode in the beginning of 2009. Marc Schonefeld released his tool "undx" in CanSecWest 2009. With the use of tools like JAD and JD-GUI, his tool could create a JAR file from an Android APK file, which could then be further translated to JAVA. The "undx" tool was effective when dealing with simple apps, but it had numerous issues with complicated Dalvik Bytecode.

That's when the Dex2Jar utility was created. Dex2Jar performs similar functions as undx, but it also has certain drawbacks when dealing with the intricate Dalvik Bytecode [58]. Since the application is delivered in pre-compiled binary format, it is impossible to directly debug the source code. Disassemblers, on the other hand, can reverse or transform the Dalvik Bytecode into a readable format. The Dalvik Virtual Machine binaries are in the .dex file format. In Dalvik VM, .dex files are disassembled using Backsmali. APKtool is used by programmers to modify the source code and repackage it. Many programmers have reversed different Android applications in order to research any vulnerabilities and carefully examine the code.

In terms of this decompiling, we implemented the method of reverse engineering on Android apps in order to get Java source code. To do so, first and foremost, we installed several apps in Bluestacks as well as installed the Apk Extractor app. Bluestacks is an Android emulator which is installed in the computer. Apk Extractor is an android app which is used for extract an apk file from an Android emulator to computer. So, by using Apk Extractor in Bluestacks, we exported several Android apps in the computer.

After we get the apk file of specific Android applications in the computer whether we get it from a website or via Bluestacks, we approach to the Java Decompiler Technique [59]. A particular kind of decompiler called a Java decompiler transforms class files into Java source code. Decompilation is the exact opposite of the compilation process. The decompiler does not create a copy of the source code as a result. It is a result of certain source code information being lost during program compilation. But there are many problems, for instance, byte code is not organized, but Java code is. Furthermore, the transformation is not one to one; two different Java programs may produce the same byte code. Actually, the Java decompiling part is the most crucial part of the reverse engineering technique.

It is known that each Android application executes on its own instance of the Dalvik Virtual Machine. The DVM executes files in the dex format, also known as the Dalvik executable format. The dex format is considered to be a very proficient binary format of machine instructions for the Dalvik Machine [60]. The main components and program logic of an Android application lie within the .classes .dex file which the user is unable to view. Hence, the Dex2Jar tool was developed in order to convert dex files into class format. With the help of this tool, it is now possible to view the source code of an application as a java code [30].

The Android malware testing app is placed in the same location where dex2jar is extracted. A packed or zipped bundle of files known as an Android application can be unzipped or extracted using WinZip. The malware is extracted to the same directory as Dex2Jar in .apk format. After execute the below command we get the apk java class files.

Command: d2j-dex2jar.bat “Tic_Tac_Toe_base.apk”(Windows)

dex2jar-2.0/d2j-dex2jar.sh classes.dex(Linux)

The process of Java decompiling is illustrated below in Figure 2 [61]:

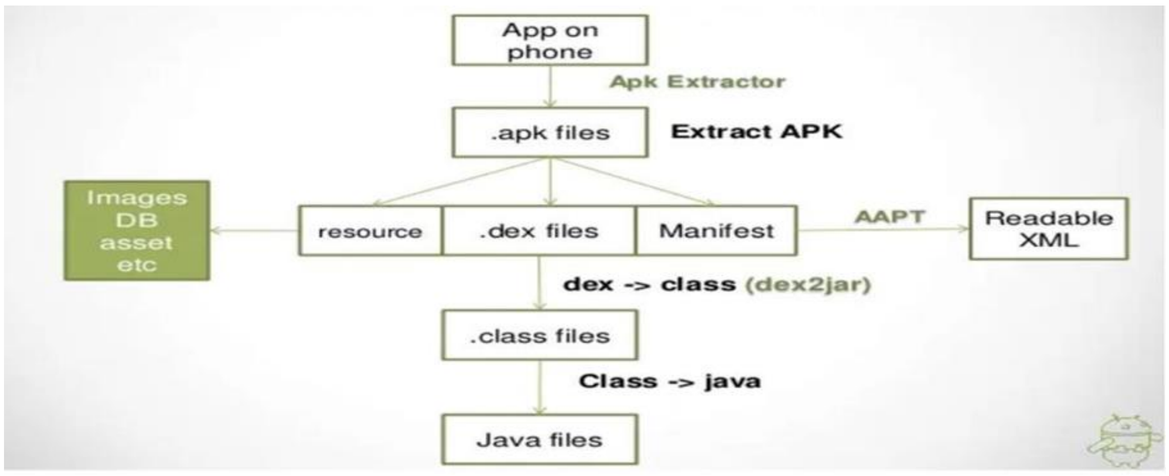


Figure 2 Java decompiling process

Furthermore, we used another tool named JD GUI so that we could be able to open up the executable jar file and see the inner structured code of the Android app, including java files, xml files, Manifest file etc. JD-GUI is a Java decompiler tool that is freely available to download. It is a distinct graphical tool that enables users to view the source code of .class files that contain Java. Its features include supporting Drag & Drop, JAR files, using the cross-platform wxWidgets toolkit, allowing users to navigate the hierarchy of class files, displaying Java code in color codes, showing log files, and allowing users to decompile files in the Java Stack traces.

3.3. Clone Detection

To prevent software piracy and clone apps in the Android Play Store, several techniques had been employed for clone detection. Here are some commonly used methods:

Package name analysis: Each Android app has a unique package name, which serves as its identifier. Clone detection techniques can analyze package names to identify apps that have similar or identical names. This can help identify potential clone apps that try to mimic the original application.

Hash value comparison: Hashing algorithms can be used to generate unique hash values for APK (Android application package) files. By comparing the hash values of different APK files, it's possible to detect identical or similar apps. If multiple apps have the same or similar hash values, it could indicate the presence of clones [62].

Metadata analysis: Clone detection techniques can analyze metadata associated with apps, such as the app title, description, developer name, and screenshots. Similarities in metadata across multiple apps could indicate the presence of clones [63].

Code analysis: Clone detection can involve analyzing the code structure and implementation details of Android apps. This can be done through static code analysis techniques. By comparing the source code or bytecode of different apps, similarities or identical code segments can be identified, suggesting the presence of cloned apps [64].

Behavioral analysis: Clone detection techniques can also analyze the behavior of apps at runtime. By monitoring various runtime characteristics, such as network traffic, API calls, or system interactions, similarities in behavior across different apps can be detected. This can help identify cloned apps that exhibit similar malicious or unwanted activities.
User feedback and reviews: User feedback and reviews play a vital role in detecting clone apps. Analyzing user reviews and ratings can help identify patterns or similarities in the experiences reported by users, which may indicate the presence of cloned apps [65].

Machine learning-based techniques: Machine learning algorithms can be trained on large datasets of known legitimate and clone apps. These algorithms can learn to recognize patterns, similarities, and features indicative of clone apps. By using features such as metadata, code structure, and behavior, machine learning models can assist in automating the detection of clone apps [66].

It's important to note that clone detection is an ongoing process, and new techniques are continually being developed as piracy methods evolve. App store administrators and developers work together to employ various mechanisms and tools to detect and prevent software piracy in the Android Play Store. However, in this paper, we used NiCad Clone Detector as our formal method, and we also found the clone report of the Android app. Moreover, we also explore some flaws of NiCad and suggest some descriptive mechanisms to improve the efficiency of NiCad in the result and discussion sections.

NiCad (Nicad is Closely Analyzing Duplicates) is a clone detector specifically designed for detecting code clones in software projects [67]. It is not tailored specifically for clone detection in the Android Play Store, but rather for general code clone detection in programming languages like Java, C, and C++.

Basically, two stages of clone detection are used:

Signature generation and signature matching. In the first stage, NiCad is evaluated to find clone classes in the android java training set. NiCad can calculate duplicates at various levels of granularity levels. For our approach, we look for function-level and block level. This allows partial concatenation of layers in two ways. The first is based on similar thresholds, the methods may be slightly different. But also, our classes with additional methods also fit based on a subset of similar methods. This are similar malware suites. Various malware will be grouped into different cloning classes. Then we take one of each layer cloned to act as a signature for this class. This set of copies is called signature set. In the second stage, NiCad is used in incremental mode to find duplicates of members of signatures identified in copyrights and benign assessments set. NiCad give us copy report, what can we do continue to investigate malware analysis. Before we set up the Clone detector tool 'NiCad' in our computer, we installed VirtualBox so that we could use Ubuntu operating system in order to run NiCad clone detector via command line. However, after installing this tool, we followed these steps to detect clone using NiCad clone detector. Firstly, I installed TXL and Nicad from the txl.ca website. Secondly, we then optimize NiCad by precompiling TXL programs and by running specific command in ubuntu terminal. Thirdly, we copied the source code of the android apk to the ./systems directory followed by the NiCad location and we had make an analysis directory to hold the source systems and results of NiCad. Lastly, we run the NiCad command on the java files by specifying cross clone, the language and analysis granularity like functions by using specific command in ubuntu terminal at the specific directory.

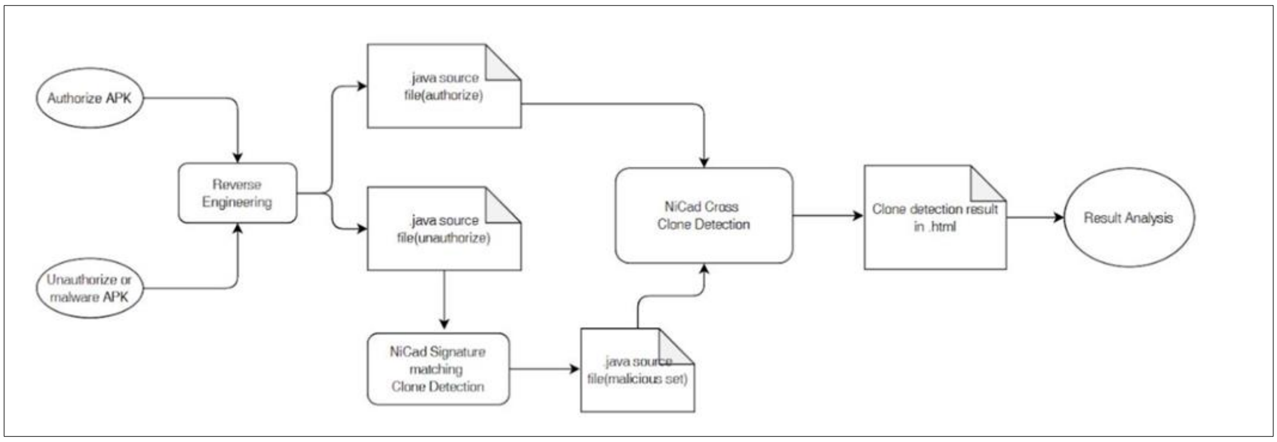

Authorize APK
Unauthorize or malware APK
Reverse Engineering
java source file(authorize)
java source file(unauthorize)
NiCad Signature matching Clone Detection
java source file(malicious set)
NiCad Cross Clone Detection
Clone detection result in .html
Result Analysis

Figure 3 Clone detection approach

NiCad follows a multi-step process to identify code clones such as:

Tokenization: The source code files to be analyzed are first tokenized, breaking down the code into smaller meaningful units called tokens. Tokens can be identifiers, keywords, literals, operators, or other syntactic elements [68].

Parsing: The tokenized code is then parsed into an abstract syntax tree (AST). The AST represents the syntactic structure of the code and captures the relationships between different elements in the code [69].

Clone Detection: NiCad employs a detection algorithm to compare the ASTs of different code fragments and identify similarities [70]. The algorithm identifies clones by comparing the subtrees of the ASTs, searching for similar patterns or structures. It uses metrics such as the size of the clone, the number of statements, and the similarity threshold to determine whether code fragments are considered clones.

Clone Classification: Once the clones are detected, NiCad categorizes them based on their similarity and type. It distinguishes between Type 1 clones (identical code fragments), Type 2 clones (structurally similar but not identical), and Type 3 clones (functionally similar but structurally different) [67].

Clone Visualization: NiCad provides visualization tools to present the detected clones in a graphical format. The visualization helps developers analyze the clones, understand their distribution, and make informed decisions on how to address them [71].

NiCad incorporates several optimization techniques to improve its performance and scalability, such as token hashing and clone indexing. These techniques help reduce the time and memory required for clone detection, allowing it to handle large codebases efficiently. It's important to note that NiCad is primarily designed for code clone detection within software projects, and its application to clone detection in the Android Play Store or APK files may require additional adaptations and integration with other tools or techniques for instances TXL tools also need to optimize NiCad by precompiling TXL programs with the specific commands run in Ubuntu.

3.4. Library License Modification

The last technique we consider for this research is that modifying android application library license security faculty. This is a very important approach to preventing software piracy in a different way and it is a way more difficult for a pirate to copy or clone authorize android app and distribute it illegally. For instance, after clone detection of any Android app we can modify or find the bug inside the infrastructure of the code and make the necessary security modifications to make the app more stubborn to pirate it according to the individual preferences. A crucial element is the License Verification Library (LVL). The service can eventually be circumvented by a determined attacker who is prepared to disassemble and reassemble code, but application developers can make the hacker's effort extremely challenging, possibly to the point where it is just not worth their time.

The LVL guards against casual piracy—users attempting to copy APKs directly from one device to another without paying for the app—right out of the box. There are some methods to make it difficult for attackers that are technically proficient to decompile any application and eliminate or disable LVL-related code.

We have described a technique in this paper in order to enhance the security of the individual android app so that it prevents software from being illegally distributed as well as we designed an algorithm that could give insights of how we can develop the android security segment over new challenges and new pirates. Although modifying the Android license library is not a recommended technique to prevent software piracy but it may enhance the security level and resist the unauthorized distribution. The Android license library, such as Google Play Licensing, is designed to enforce licensing terms and restrictions for the usage and distribution of individual app through the Google Play Store [72]. It primarily ensures that users have obtained a valid license to use the app. To prevent software piracy and protect apps from unauthorized distribution and usage, there are several techniques, including the following:

Code Obfuscation: Using code obfuscation tools like ProGuard or DexGuard to obfuscate the app's code. Obfuscation makes it harder for potential attackers to understand and reverse engineer any app, thus deterring unauthorized modifications and cloning [25].

Licensing and Authorization Mechanisms: Implement robust licensing and authorization mechanisms within the app. This can include verifying the validity of license keys, implementing device binding, and enforcing restrictions on multiple installations or usage across different devices [73].

Server-Side Verification: Consider implementing server-side verification to validate the authenticity of the app installation and license. This can involve making requests to the app server to verify the license key or perform additional checks to ensure the app is being used legitimately [74].

In-App Purchases and Subscriptions: Utilize in-app purchases and subscription models to monetize any app and restrict access to premium features or content. By linking the app's functionality to a valid purchase or subscription, we can prevent unauthorized usage [75].

Digital Rights Management (DRM): For apps that involve media content, consider employing DRM technologies to protect against unauthorized copying or distribution of the content. DRM can encrypt media files and implement usage restrictions, providing an additional layer of protection against piracy [76].

App Integrity and Tampering Checks: Implement integrity checks within the app to detect modifications or tampering attempts. These checks can include verifying the app's digital signature, checking for unauthorized modifications to critical files, or using checksums to ensure the app's integrity [77].

Regular Updates and Patching: Continuously monitor and address security vulnerabilities in individual app. Release regular updates that include bug fixes, security patches, and new features. Promptly address any reported vulnerabilities to mitigate the risk of exploitation by potential attackers [74].

App Store Policies and Reviews: Comply with the guidelines and policies set by the official app stores, such as the Google Play Store. Regularly monitor user reviews and reports to identify and address instances of unauthorized usage or distribution. Remember that no technique can provide 100% protection against software piracy. However, by implementing a combination of these measures, we can significantly reduce the risk of unauthorized usage and protect our app's intellectual property [78].

One way to prevent the distribution of paid Android applications for free on unlicensed websites is to use the Android License Verification Library. After adding licensing verification, the app will query Google Play to determine whether the license is currently valid each time it is launched. Therefore, if a user purchases an app through the Google Play Store, the license is legal; however, if the user installs the app from any other website, the result is invalid.

A trustworthy Google Play licensing server can be contacted via the network-based service known as Google Play Licensing to check if an application is currently licensed for the user of the device in question. The capacity of the Google Play licensing server to ascertain whether a specific user has a valid license to use a specific application is the foundation of the licensing service. If a user is a recorded purchaser of the application, Google Play considers them to be licensed.

When your application sends a request to a service that is hosted by the Google Play client application, the request is said to have begun. After that, the Google Play application queries the license server and receives a response. Your application receives the outcome from the Google Play application, which it can then use to decide whether to allow or deny future usage of the application.

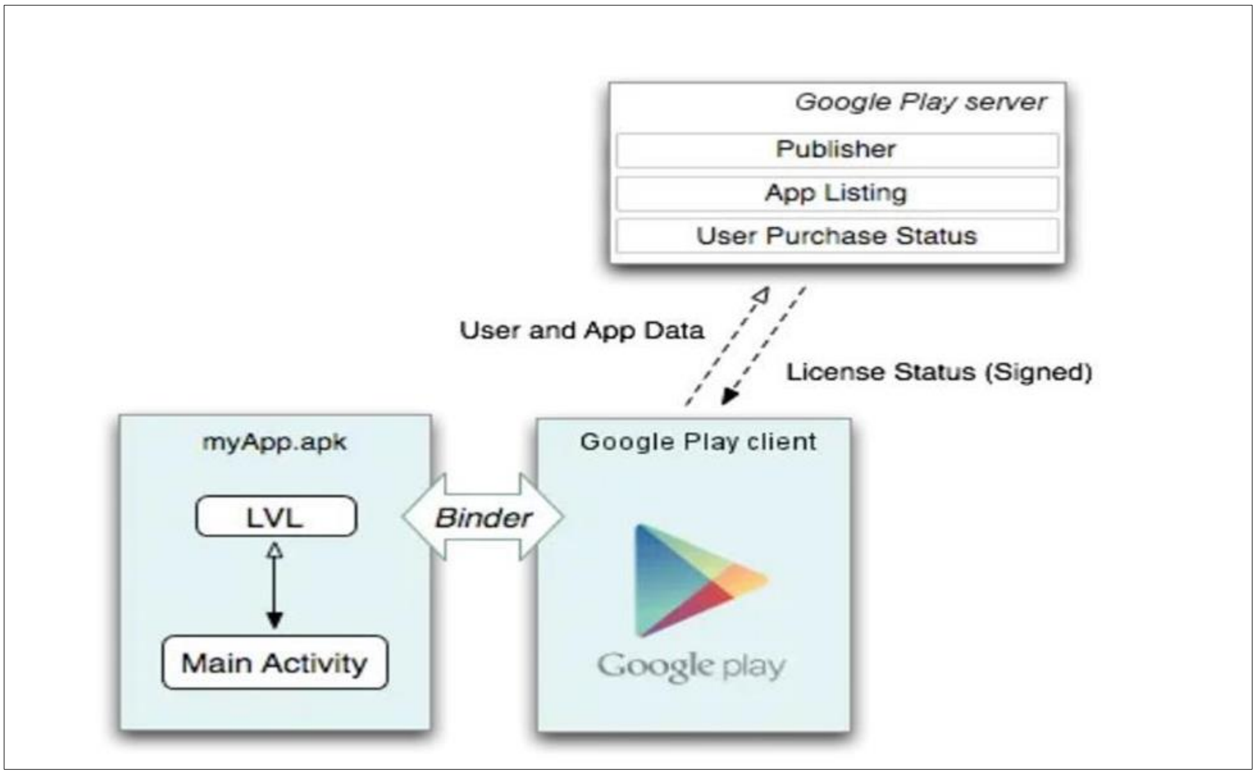
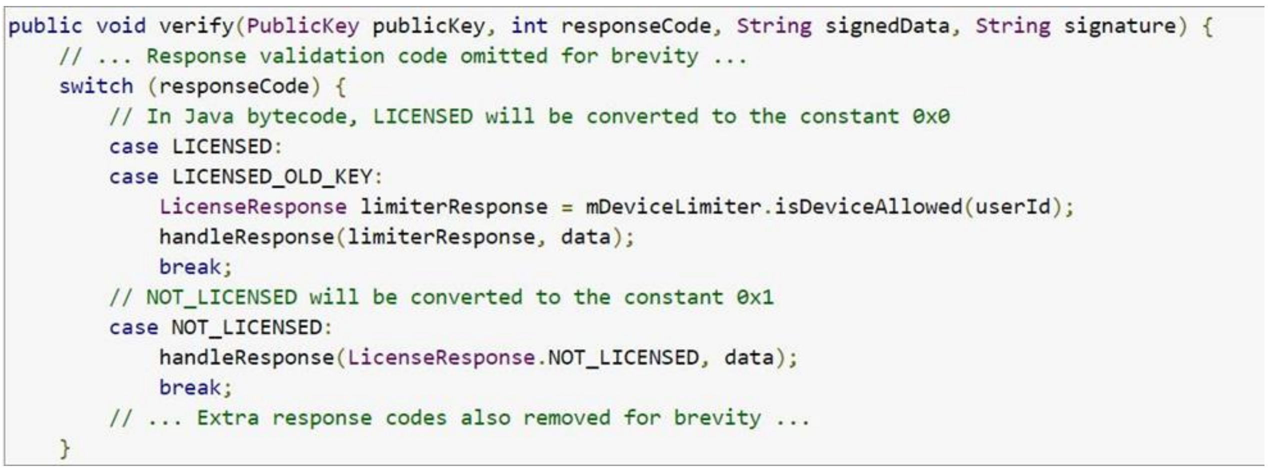


Figure 4 Library license response process

The License Verification Library (LVL), a library we can add or modify in our application, handles all licensing-related interactions with the Google Play licensing service. It is part of our SDK for adding licensing to applications. We recommend modifying the licensing verification library so that it would be challenging for an attacker to change the disassembled code and obtain a successful license check as a result. This actually offers defense against two separate kinds of attacks: it guards against hackers attempting to crack licensed software, but it also hinders the ease with which attacks intended for other software (or even the standard LVL distribution itself) can be readily moved over to any software. The objective should be to both make the application's bytecode more sophisticated and to provide it a distinctive LVL implementation.

There are three areas where we concentrated our research and attention when modifying the license library: the licensing library's basic logic, the places of entry and exit for the license library, and the licensing library is called by the application, and it manages the license response. We have primarily concentrated on LicenseChecker and LicenseValidator, two classes that make up the heart of the LVL logic in the case of the core licensing library. Our objective is to alter these LVL classes as much as we can while preserving the application's original functionality. Consider the following code:



```java
public void verify(PublicKey publicKey, int responseCode, String signedData, String signature) {
    // ... Response validation code omitted for brevity ...
    switch (responseCode) {
        // In Java bytecode, LICENSED will be converted to the constant 0x0
        case LICENSED:
        case LICENSED_OLD_KEY:
            LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId);
            handleResponse(limiterResponse, data);
            break;
        // NOT_LICENSED will be converted to the constant 0x1
        case NOT_LICENSED:
            handleResponse(LicenseResponse.NOT_LICENSED, data);
            break;
        // ... Extra response codes also removed for brevity ...
    }
```

In most cases, we explore that an attacker might attempt to switch the code from the LICENSED and NOT_LICENSED cases in order to treat an unlicensed user as licensed. Even obfuscation makes it relatively simple to identify where this check is carried out in the application’s bytecode because an attacker will already be aware of the integer values for LICENSED (0x0) and NOT_LICENSED (0x1) by looking at the LVL source.

We consider the following algorithm and modifications to make this more challenging:

STEP 1 : A method PUBLIC VOID verify(PublicKey publicKey, int responseCode, String signedData, String signature) is created.

STEP 2 : An object of class java.util.zip.CRC32 crc32 is created with value java.util.zip.CRC32().

STEP 3 : The function update(responseCode) from object crc32 is called.

STEP 4 : INT variable transformedResponseCode is instantiated to value crc32.getValue().

STEP 5 : IF transformedResponse == 3523407757.

STEP 6 : An object of class LicenseResponse limiterResponse is instantiated to value mDeviceLimiter.isDeviceAllowed(userId).

STEP 7 : The member method handleResponse(limiterResponse, data) is called.

STEP 8 : IF transformedResponseCode == 1007455905.

STEP 9 : An object of class LicenseResponse limiterResponse is instantiated to value mDeviceLimiter.isDeviceAllowed(userId).

STEP 10 : The member method handleResponse(limiterResponse, data) is called.

STEP 11 : IF transformedResponseCode == 2768625435.

STEP 12 : The member method userIsntLicensed() is called

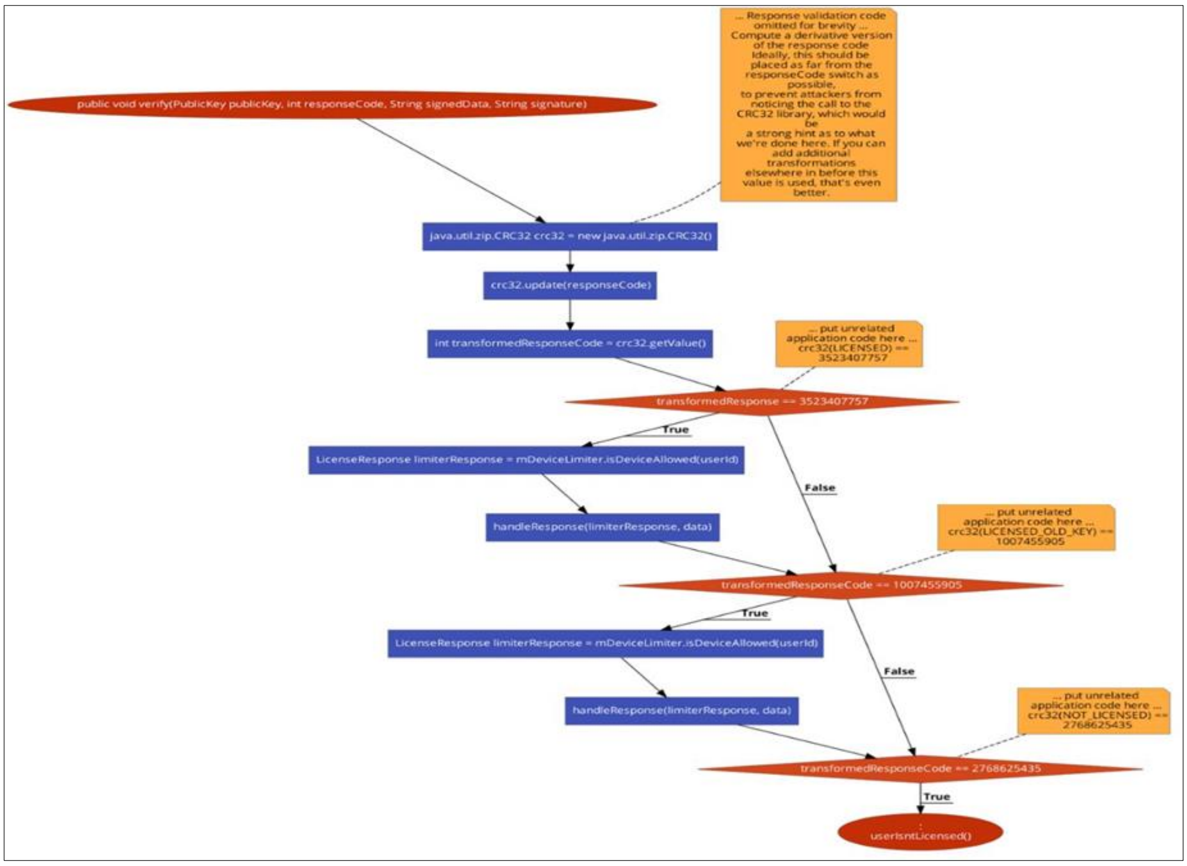

public void verify(PublicKey publicKey, int responseCode, String signedData, String signature)
... Response validation code omitted for brevity ... Compute a derivative version of the response code Ideally, this should be placed as far from the responseCode switch as possible, to prevent attackers from noticing the call to the CRC32 library, which would be a strong hint as to what we're done here. If you can add additional transformations elsewhere in before this value is used, that's even better.
java.util.zip.CRC32 crc32 = new java.util.zip.CRC32()
crc32.update(responseCode)
int transformedResponseCode = crc32.getValue()
... put unrelated application code here ... crc32(LICENSED) == 3523407757
transformedResponse == 3523407757
True
LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId)
handleResponse(limiterResponse, data)
False
... put unrelated application code here ... crc32(LICENSED_OLD_KEY) == 1007455905
transformedResponseCode == 1007455905
True
LicenseResponse limiterResponse = mDeviceLimiter.isDeviceAllowed(userId)
handleResponse(limiterResponse, data)
False
... put unrelated application code here ... crc32(NOT_LICENSED) == 2768625435
transformedResponseCode == 2768625435
True
userIsntLicensed()

Figure 5 Code flow of the library license modification algorithm

4. Results and discussion

By attempting to analyze the patterns, scope, and variations of known malware in the Android paradigm using code clone detection techniques, this paper undertakes an empirical analysis to comprehend the state of benign and unauthorized or malware Android application inner structure to deconstruct the app in order to prevent software piracy. The following provides responses to the research issues raised by this study.

4.1. RQ 1: Can we use the clone detection technique to extract malicious code frommalicious apps to create a malware signature set?

Motivation: Many researchers have looked into the prospect of using clone detection to find malware. Clone detection was utilized by Karademir [79] to locate JavaScript malware in Adobe Acrobat (PDF) files. At the binary level, [30] and [80] both recognize the code clone fragments. None of them are able to identify the malware in Java source code. If we can extract the malicious code from the malware, it will not only aid in locating the infection but also in its elimination.

Approach. In order to create an unwanted signature as a malware pattern, we must first extract the malicious code from the source code. We divide our data into three categories: testing, extraction of malicious code, and benign set. The malicious code extraction set, which still maintains the same directory structure as the testing set, is the primary emphasis of RQ1. They both fall under one of 24 malware families.

We use the malicious code extraction dataset for each malware family independently to use the clone detection technique to create the malware signature set. In this stage, the NiCad clone detector is utilized. The decompiled Java source code of each APK sample is contained in a number of sub-folders in each malware family folder, and each of these sub-folders should contain identical or similar malicious code fragments that are a part of the same malware family. As a result, NiCad makes it simple to group similar code into a single class. The clone classes in the malicious data extraction set are shown in the NiCad clone detection report. We can create a signature set for 24 separate malware families based on the information about the clone classes, and the code extracted from each malware family is saved as a new Java file with the name maleware familyname.java. These actions enable us to extract the detrimental code from the sample set.

Findings: Clone classes can be created by malicious code. To determine if a collection of malicious code fragments can form a clone class, we analyze the NiCad results for each malware family in the extraction set. Table 1 displays the initial findings from this signature generating step at a level of 100% similarity. The number of clone classes created from detrimental classes is shown in column 3. Only those clone classes from a single malware family are retained across all samples. We carry out sensitivity analysis in our experiment including the extraction of malware code. We do clone detection using three different clone kinds with similarity limits of 80%, 90%, and 100%. The majority of malware signature code is identical or similar. As a result, the barrier has little impact. The clone detection results for various similarity thresholds and clone kinds of these various settings do not significantly differ in terms of the malware code extraction. We only display the result in Table 1 at 100% similarity because the malware code extraction is carried out within a single piece of malware and should be present in the decompiled Java code of programs.

The creation of Android apps frequently makes use of open-source or third-party libraries. When we create the malware code signature from the decompiled code, we take this into consideration. Each directory name in the decompiled programs' explicit folder structure denotes the purpose of the underlying code, such as util, and sdk. If the code is located in a library folder like sdk, we skip it. Despite having the same code, which is detrimental, apps from different malware families are not the same programs. The likelihood of having the same library is slim. As a result, it is challenging to create a clone class that encompasses all malware family programs.

Table 1 Experimental results on the detrimental identical dataset

Malware Group	**APK Experimented**	**Clone Classes Finding**	**Similarity (%)**
RAT	8	12	100
Ghostpush	5	8	100
Comebot	10	3	100
Exodus	5	14	100
Fake_av_reader	5	24	100
Feabme	10	17	100
Fake_Bankers	10	21	100
Krep	12	11	100
Malbus	5	2	100
Rough_skype	1	12	100
Towelroot	10	66	100
Triada	10	37	100
Xbot	8	7	100
Zazdi	5	125	100
DroidKungfu2	20	72	100

As an illustration, Table 1's first row contains the malware family RAT. With the help of Android malware and a mobile Remote Administration Tool (RAT), an attacker can take control of victims' devices using Spymax. Once the virus has been placed on a phone, the attacker can carry out numerous attacks that have a significant negative impact on the privacy and confidentiality of the victim's data. It is strong, widely accessible, and doesn't require the victim's device to have root access. We consider this malware RATs out of 24 different malware types, and we find that as part of the malicious extraction set, we used 12 sample programs. As a result, the folder RAT has 12 subfolders. We give each subdirectory a sequence number so that we can quickly identify the original file for each clone class. The partial outcome of the phase 1 clone detection is shown in Fig. 6. In this example, we only display classes that contain malware as clone classes. Those classes are undoubtedly the malicious code clone classes because they contain 10 identical code snippets taken from several APK sample source code files.

Despite the fact that a dissimilar class is a clone class and does not include all sub-samples, we do not consider it to be detrimental.

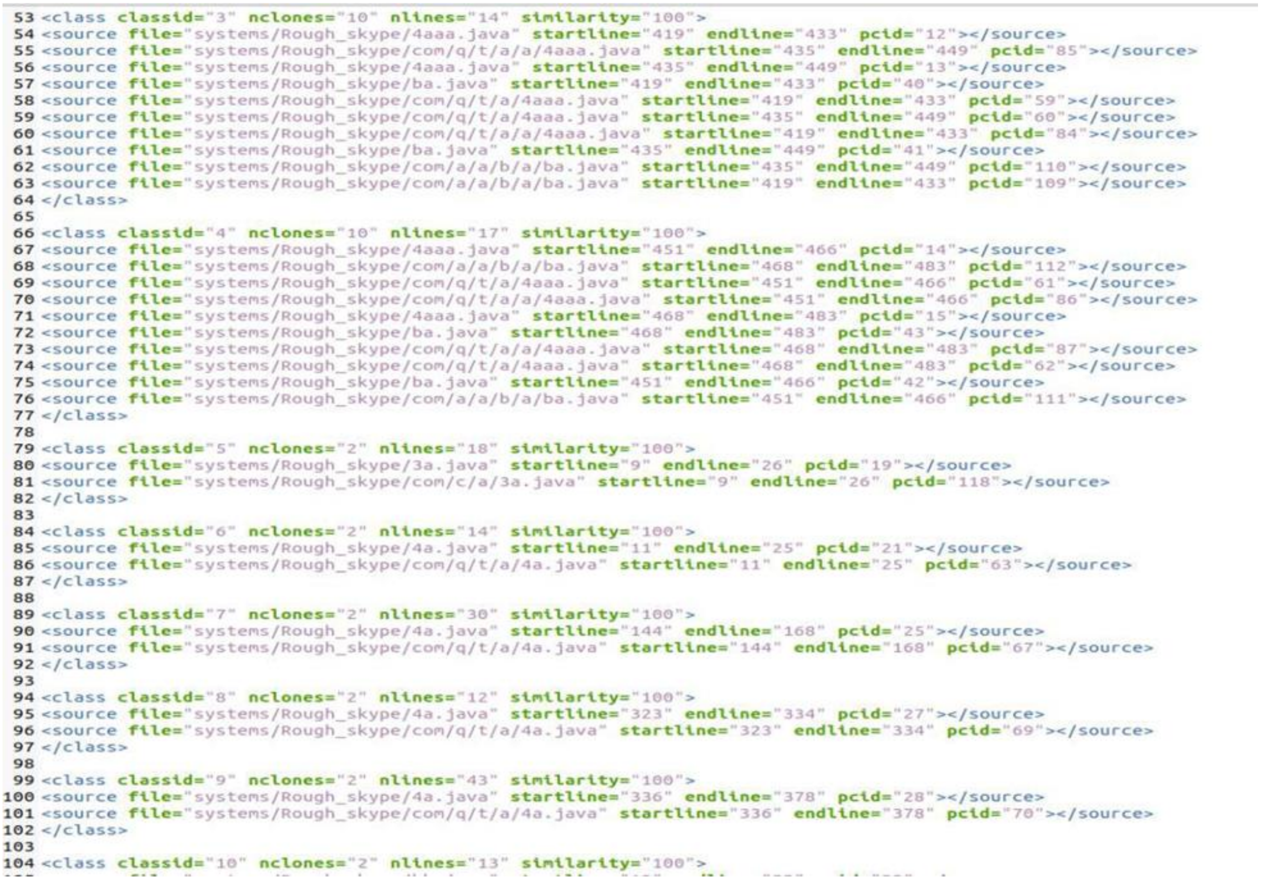
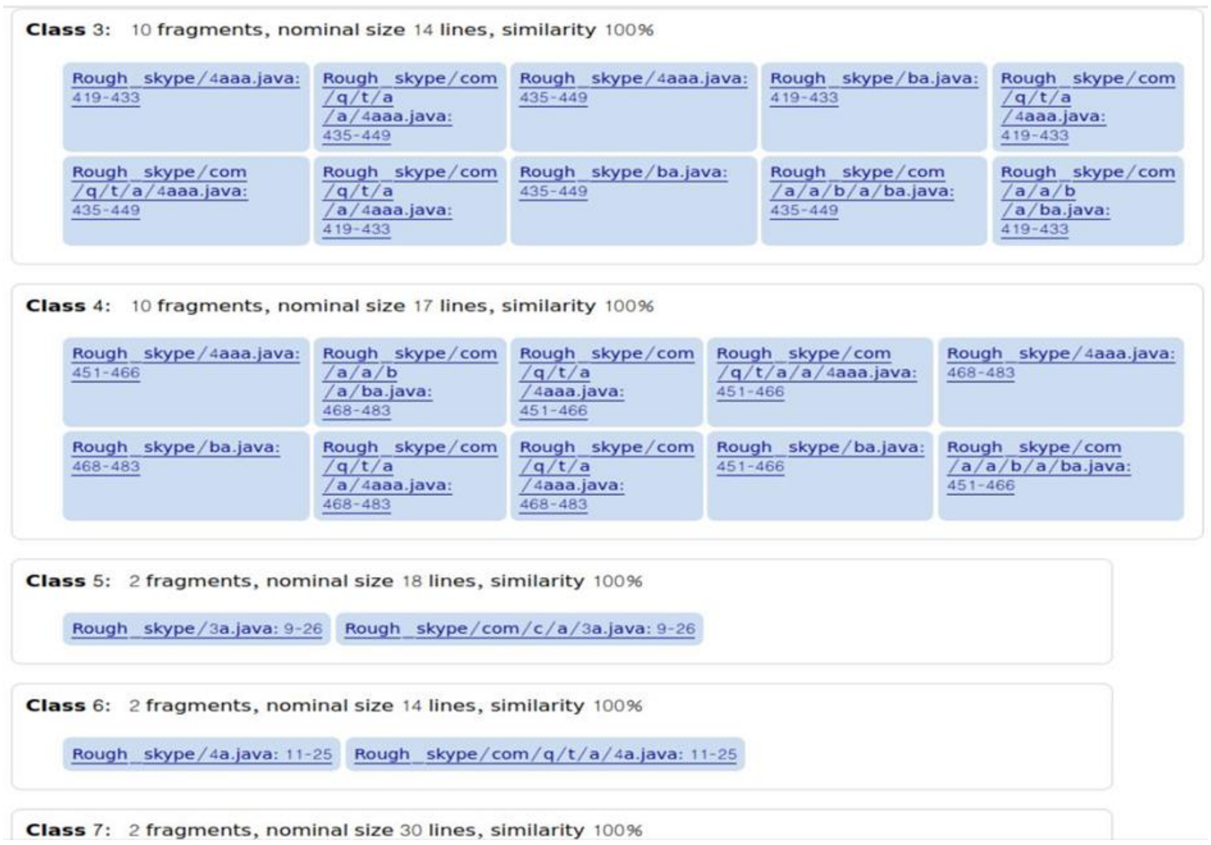

```
<class classid="3" nclones="10" nlines="14" similarity="100">
<source file="systems/Rough_skype/4aaa.java" startline="419" endline="433" pcid="12"></source>
<source file="systems/Rough_skype/com/q/t/a/a/4aaa.java" startline="435" endline="449" pcid="85"></source>
<source file="systems/Rough_skype/4aaa.java" startline="435" endline="449" pcid="13"></source>
<source file="systems/Rough_skype/ba.java" startline="419" endline="433" pcid="40"></source>
<source file="systems/Rough_skype/com/q/t/a/4aaa.java" startline="419" endline="433" pcid="59"></source>
<source file="systems/Rough_skype/com/q/t/a/4aaa.java" startline="435" endline="449" pcid="60"></source>
<source file="systems/Rough_skype/com/q/t/a/a/4aaa.java" startline="419" endline="433" pcid="84"></source>
<source file="systems/Rough_skype/ba.java" startline="435" endline="449" pcid="41"></source>
<source file="systems/Rough_skype/com/a/a/b/a/ba.java" startline="435" endline="449" pcid="110"></source>
<source file="systems/Rough_skype/com/a/a/b/a/ba.java" startline="419" endline="433" pcid="109"></source>
</class>

<class classid="4" nclones="10" nlines="17" similarity="100">
<source file="systems/Rough_skype/4aaa.java" startline="451" endline="466" pcid="14"></source>
<source file="systems/Rough_skype/com/a/a/b/a/ba.java" startline="468" endline="483" pcid="112"></source>
<source file="systems/Rough_skype/com/q/t/a/4aaa.java" startline="451" endline="466" pcid="61"></source>
<source file="systems/Rough_skype/com/q/t/a/a/4aaa.java" startline="451" endline="466" pcid="86"></source>
<source file="systems/Rough_skype/4aaa.java" startline="468" endline="483" pcid="15"></source>
<source file="systems/Rough_skype/ba.java" startline="468" endline="483" pcid="43"></source>
<source file="systems/Rough_skype/com/q/t/a/a/4aaa.java" startline="468" endline="483" pcid="87"></source>
<source file="systems/Rough_skype/com/q/t/a/4aaa.java" startline="468" endline="483" pcid="62"></source>
<source file="systems/Rough_skype/ba.java" startline="451" endline="466" pcid="42"></source>
<source file="systems/Rough_skype/com/a/a/b/a/ba.java" startline="451" endline="466" pcid="111"></source>
</class>

<class classid="5" nclones="2" nlines="18" similarity="100">
<source file="systems/Rough_skype/3a.java" startline="9" endline="26" pcid="19"></source>
<source file="systems/Rough_skype/com/c/a/3a.java" startline="9" endline="26" pcid="118"></source>
</class>

<class classid="6" nclones="2" nlines="14" similarity="100">
<source file="systems/Rough_skype/4a.java" startline="11" endline="25" pcid="21"></source>
<source file="systems/Rough_skype/com/q/t/a/4a.java" startline="11" endline="25" pcid="63"></source>
</class>

<class classid="7" nclones="2" nlines="30" similarity="100">
<source file="systems/Rough_skype/4a.java" startline="144" endline="168" pcid="25"></source>
<source file="systems/Rough_skype/com/q/t/a/4a.java" startline="144" endline="168" pcid="67"></source>
</class>

<class classid="8" nclones="2" nlines="12" similarity="100">
<source file="systems/Rough_skype/4a.java" startline="323" endline="334" pcid="27"></source>
<source file="systems/Rough_skype/com/q/t/a/4a.java" startline="323" endline="334" pcid="69"></source>
</class>

<class classid="9" nclones="2" nlines="43" similarity="100">
<source file="systems/Rough_skype/4a.java" startline="336" endline="378" pcid="28"></source>
<source file="systems/Rough_skype/com/q/t/a/4a.java" startline="336" endline="378" pcid="70"></source>
</class>

<class classid="10" nclones="2" nlines="13" similarity="100">
```

Figure 6 Example of phase 1 clone detection result in xml format of NiCad



Class 3: 10 fragments, nominal size 14 lines, similarity 100%

Rough_skype/4aaa.java: 419-433
Rough_skype/com/q/t/a/a/4aaa.java: 435-449
Rough_skype/4aaa.java: 435-449
Rough_skype/ba.java: 419-433
Rough_skype/com/q/t/a/4aaa.java: 419-433
Rough_skype/com/q/t/a/4aaa.java: 435-449
Rough_skype/com/q/t/a/a/4aaa.java: 419-433
Rough_skype/ba.java: 435-449
Rough_skype/com/a/a/b/a/ba.java: 435-449
Rough_skype/com/a/a/b/a/ba.java: 419-433

Class 4: 10 fragments, nominal size 17 lines, similarity 100%

Rough_skype/4aaa.java: 451-466
Rough_skype/com/a/a/b/a/ba.java: 468-483
Rough_skype/com/q/t/a/4aaa.java: 451-466
Rough_skype/com/q/t/a/a/4aaa.java: 451-466
Rough_skype/4aaa.java: 468-483
Rough_skype/ba.java: 468-483
Rough_skype/com/q/t/a/a/4aaa.java: 468-483
Rough_skype/com/q/t/a/4aaa.java: 468-483
Rough_skype/ba.java: 451-466
Rough_skype/com/a/a/b/a/ba.java: 451-466

Class 5: 2 fragments, nominal size 18 lines, similarity 100%

Rough_skype/3a.java: 9-26
Rough_skype/com/c/a/3a.java: 9-26

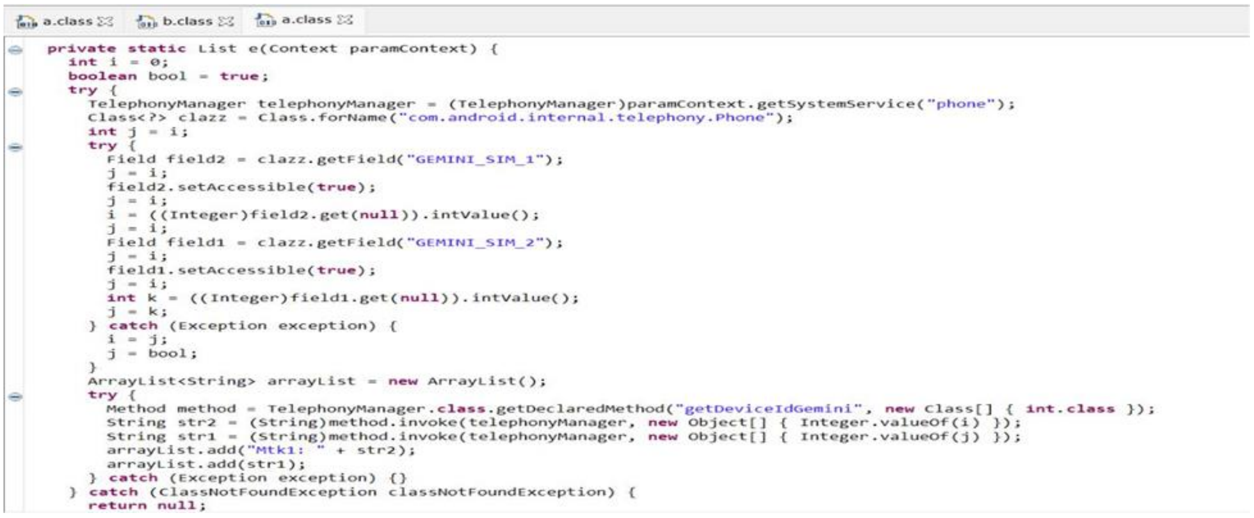
Class 6: 2 fragments, nominal size 14 lines, similarity 100%

Rough_skype/4a.java: 11-25
Rough_skype/com/q/t/a/4a.java: 11-25

Class 7: 2 fragments, nominal size 30 lines, similarity 100%

Figure 7 Example of phase 1 clone detection result in html format of NiCad

The clone report includes information about where the identical code is located, including the file name, startline, and endline. So, for each member of each malicious clone class, such as the RAT we previously mentioned, we can extract a single harmful code fragment. With the help of Android malware and a mobile Remote Administration Tool (RAT), an attacker can take control of victims' devices using Spymax. Once the virus has been placed on a phone, the attacker can carry out numerous attacks that have a significant negative impact on the privacy and confidentiality of the victim's data. It is strong, widely accessible, and doesn't require the victim's device to have root access. Nevertheless, the Spymax RAT has a command and control server that enables the attacker to provide the malware instructions.



```java
private static List e(Context paramContext) {
  int i = 0;
  boolean bool = true;
  try {
    TelephonyManager telephonyManager = (TelephonyManager)paramContext.getSystemService("phone");
    Class<?> clazz = Class.forName("com.android.internal.telephony.Phone");
    int j = i;
    try {
      Field field2 = clazz.getField("GEMINI_SIM_1");
      j = i;
      field2.setAccessible(true);
      j = i;
      i = ((Integer)field2.get(null)).intValue();
      j = i;
      Field field1 = clazz.getField("GEMINI_SIM_2");
      j = i;
      field1.setAccessible(true);
      j = i;
      int k = ((Integer)field1.get(null)).intValue();
      j = k;
    } catch (Exception exception) {
      i = j;
      j = bool;
    }
    ArrayList<String> arrayList = new ArrayList();
    try {
      Method method = TelephonyManager.class.getDeclaredMethod("getDeviceIdGemini", new Class[] { int.class });
      String str2 = (String)method.invoke(telephonyManager, new Object[] { Integer.valueOf(i) });
      String str1 = (String)method.invoke(telephonyManager, new Object[] { Integer.valueOf(j) });
      arrayList.add("Mtk1: " + str2);
      arrayList.add(str1);
    } catch (Exception exception) {}
  } catch (ClassNotFoundException classNotFoundException) {
    return null;
```

Figure 8 A java class of RAT malware

One malware family's collection of malicious extraction sets may not contain identical or closely related malicious code. This is a very intriguing discovery because we would expect bad code to initially cross the whole extraction set inside a single malware family. However, we were only able to find a few clone classes from six out of ten extractions and additional clone classes from the remaining four extractions set for the DroidKungFu2 family. To put it another way, the signature set is made up of two different types of clone classes: one type is a clone class that has six pieces of code that are similar to or identical to another, and the other type is a clone class that has four pieces of code that are similar to or identical to another.We demonstrate the data from various sample clone classes in Table 1.

4.2. RQ 2: Can the Malware Signature Set be used to detect Malware Apps in the Rest ofthe Malicious Set?

Finding: The purpose of clone detection is to recognize code that is similar or identical. As a result, clone detection can be used as a pattern matching engine to locate comparable malware signature patterns in the evaluation set. NiCad detects clones in two ways: standard mode clone and incremental mode clone. The standard mode gives NiCad a single source folder to inspect, and all source files within that folder are checked for clones, as in RQ1. The malware file and the evaluation set can be clustered into a clone class using NiCad. In other words, the malware can be successfully detected using our clone detection technique. This outcome demonstrates the effectiveness of the clone detection technique in locating malware in Android. One file is from the malware signature set, and 17 files are grouped together into one clone class according to our findings. Table 1 informs us that a total of 15 malware app instances in the analysis set, and the clone detection result in the example is consistent with an effect of 15 malware files together with a malware signature file.

Therefore, in this example, we can identify all similar malware family apps using a single malware family signature. It demonstrates how a pattern-matching engine can use the clone detection method to find malware. Table 2 displays the outcomes of incremental mode clone detection comparing the evaluation set and the signature set.

Table 2 Results of malicious app detection comparing the evaluation set and the signature set.

Malware Group	**Evaluation**	**dataset**	**of**	**Number**	**of**	**Detected**	**Similarity**
	Malware Apps			Malicious App			(%)
RAT	8			8			100
Ghostpush	5			3			100
Comebot	10			9			100
Exodus	5			5			100
Fake_av_reader	5			5			100
Feabme	10			10			100
fake_Bankers	10			10			100
Krep	12			11			100
Malbus	5			2			100
Rough_skype	1			1			100
Towelroot	10			10			100
Triada	10			10			100
Xbot	8			8			100
Zazdi	5			5			100
DroidKungfu2	20			20			100

4.3. Limitation and future works

4.4. RQ 3: What potential drawbacks could our strategy have, and how can we improve them?

Finding: NiCad clone detector can detect similarity between code segments very precisely.

However, it has a few potential flaws or limitations. Some of these include:

False Positives: NiCad may sometimes produce false positive results, where code fragments are incorrectly identified as clones even though they are not. This can be due to the similarity of common programming patterns or code structures that may appear in unrelated code segments. False Negatives: NiCad may also have false negatives, where it fails to detect actual code clones. This can occur when clones have undergone significant modifications or when they exist in different forms or contexts.

Parameter Sensitivity: NiCad's clone detection results can be sensitive to the threshold and granularity settings. Different parameter values may lead to variations in the detected clones. Choosing appropriate values requires domain knowledge and experimentation.

Scalability: NiCad's performance may degrade when dealing with large codebases or projects with numerous files. The time and memory requirements may increase significantly, making it less efficient for large-scale code analysis.

To overcome these limitations, potential improvements and strategies for the future could include:

Refining Clone Detection Algorithms: Continuously improving the underlying clone detection algorithms can help reduce false positives and false negatives. Incorporating more advanced techniques and heuristics may enhance the accuracy of clone detection.

Parameter Optimization: Developing automated methods for selecting optimal threshold and granularity values can minimize the reliance on manual tuning. Machine learning or statistical approaches can be explored to find suitable parameter settings based on training data or historical clone detection results.

Advanced Context Analysis: Enhancing the context analysis capabilities of NiCad can help distinguish between legitimate code reuse and actual clones. Considering semantic information, variable names, and comments can provide better insights into code similarities and differences.

Incremental and Distributed Analysis: Enhancing NiCad's incremental analysis capabilities can improve performance and reduce resource requirements by focusing only on the modified parts of the code. Additionally, exploring distributed and parallel processing techniques can enable efficient analysis of large codebases.

Integration of Machine Learning: Leveraging machine learning algorithms can augment NiCad's clone detection capabilities. Training models on large-scale code repositories can improve clone detection accuracy and provide more context-aware results.

Collaboration and Community Input: Encouraging collaboration and involvement from the software engineering community can lead to valuable feedback, bug reports, and enhancements. Actively seeking input from users and incorporating their suggestions can help address known issues and improve NiCad's overall quality.

In terms of android library license modification, our proposed algorithm may not be applicable for all type of applications. We have designed this algorithm just to give an idea of how individual can improve the security of their own android app from the attackers to prevent the crack of the application and distribute it illegally. Moreover, Android library licensing can contribute to preventing attackers from cracking an application in some other ways as well which is given below:

Code Obfuscation: Applying a license to an Android library often involves code obfuscation techniques. Obfuscation makes the code more difficult to understand and reverse-engineer by obfuscating class and method names, removing debug information, and introducing other transformations. This makes it challenging for attackers to analyze the code and modify it to bypass licensing checks.

License Verification: Libraries can incorporate license verification mechanisms that validate the license key or token provided by the application. This verification process can include cryptographic checks or communication with a license server to ensure the license is valid and hasn't been tampered with. By enforcing license verification, unauthorized or cracked versions of the application that lack a valid license key can be detected and restricted.

Tamper Detection: Libraries can include mechanisms to detect tampering attempts on the application's code or resources. These mechanisms may employ checksums, digital signatures, or other integrity checks to verify the integrity of the application at runtime. If tampering is detected, the library can take appropriate actions, such as terminating the application or disabling certain features.

Anti-Debugging and Anti-Hooking Techniques: Android library licensing can employ anti-debugging and anti-hooking techniques to deter attackers from analyzing or modifying the application's behavior. These techniques can include detecting debuggers or hooking frameworks and responding accordingly, such as terminating the application or altering its behavior.

Secure Key Storage: Libraries can provide secure storage mechanisms for storing license keys or tokens, ensuring they are not easily accessible or modifiable by attackers. This prevents attackers from easily substituting or removing the license verification components.

Regular Updates and Patching: Library developers should actively monitor and address vulnerabilities or weaknesses in the licensing mechanism. Regular updates and patches can address any identified security flaws, improve resilience against attacks, and adapt to evolving cracking techniques.

It's important to note that while licensing measures can make it more challenging for attackers to crack an application, determined and skilled attackers may still find ways to bypass these protections. Therefore, it's crucial to employ multiple layers of security, including code obfuscation, encryption, integrity checks, and server-side validation, to enhance the overall security of the application. Additionally, a comprehensive security strategy should include monitoring, analysis, and response to new threats and vulnerabilities as they emerge.

5. Conclusion

In this study, we used a static analysis method called clone detection to find malware in the ecosystem of Android mobile apps in order to prevent the android application piracy. The NiCad clone detector that is used to cluster the characteristic code that distinguishes malware from one family from another. In our strategy, we used the clone detect technique in both standard mode and incremental mode. The clone signature on NiCad fulfilled the research goal of demonstrating the viability of clone detection methods in identifying Android malware. Our tests showed that our method has a high accuracy of 96.88% in detecting malware. Our technique can successfully and consistently identify detrimental programs that are a part of specific malware families. Moreover, we also present an algorithm in this paper to understand how we can enhance Android licensing security in order to prevent software piracy for individual app developers.

Compliance with ethical standards

Acknowledgments

In the tapestry of this research, I wish to express profound gratitude to the singular force that fueled its existence – myself. To the tireless dedication, unyielding passion, and countless hours invested in unraveling the intricacies of this pursuit, I acknowledge and appreciate the unwavering commitment. This solitary journey, though challenging, has been illuminated by the spark within, driving me to push boundaries and contribute to the realm of knowledge.

Disclosure of conflict of interest

The author declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Credit authorship contribution statement.

Md Fahim Ahammed : Methodology, Investigation, Funding acquisition, Formal analysis, Supervision.

References

[1] Statista, Number of available applications in the Google Play Store from December 2009 to December 2023, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store. [Accessed 20 December 2023].

[2] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang and H. 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, ACM (2013) 431–444.

[3] K. Chen, Y. Zhang and P. Liu, "Leveraging Information Asymmetry to Transform Android Apps into Self-Defending Code Against Repackaging Attacks," IEEE, 17 (2018) 1879 - 1893.

[4] The Daily Star, Bangladesh tops software piracy in Asia Pacific, https://www.thedailystar.net/news-detail-88279. [Accessed 15 May 2009].

[5] Wikipedia,Copyrightinfringement, https://en.wikipedia.org/wiki/Copyright_infringement. [Accessed 22 October 2023].

[6] D. Bruschi, L. Martignoni and M. Monga, "Code Normalization for Self-Mutating Malware," IEEE, 5 (2007) 46 - 54.

[7] A. Walenstein and A. Lakhotia, "The Software Similarity Problem in Malware Analysis," Dagstuhl, 6301 (2007) 1-10.

[8] C. K. Roy, J. R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," Elsevier, 74 (2009) 470-495.

[9] M. Mondai, C. K. Roy and K. A. Schneider, "Micro-clones in evolving software," in International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE(2018) 50-60.

[10] K. Chen, P. Liu and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in Proceedings of the 36th International Conference on Software Engineering, Association for Computing Machinery (2014) 175–186.

[11] "Semantic Clone Detection Using Machine Learning," in 15th IEEE International Conference on Machine Learning and Applications (ICMLA), IEEE (2016) 1024-1028.

[12] F. Ullah, G. Srivastava and S. Ullah, "A malware detection system using a hybrid approach of multi-heads attention-based control flow traces and image visualization," Journal of Cloud Computing, 75 (2022) 1 - 21.

[13] W. Zhou, Y. Zhou, M. C. Grace, X. Jiang and S. Zou, "Fast, scalable detection of Piggybacked mobile applications," in Proceedings of the third ACM conference on Data and application security and privacy, ACM (2013) 185–196.

[14] J. Crussell, C. Gibler and H. Chen, AnDarwin: Scalable Detection of Semantically Similar Android Applications, European Symposium on Research in Computer Security, Berlin (2013).

[15] I. Martín and J. A. Hernández, "CloneSpot: Fast detection of Android repackages," Future Generation Computer Systems, 94 (2018) 740-748.

[16] UC3M, Applications of data analytics and machine learning tools to the enhanced design of modern communication networks and security applications, https://e-archivo.uc3m.es/handle/10016/30758. [Accessed 28 June 2019].

[17] D. Maiorca, D. Ariu, I. Corona, M. Aresu and G. Giacinto, "Davide MaiorcaStealth attacks: An extended insight into the obfuscation effects on Android malware," Computers & Security, Elsevier 51 (2015) 16-31.

[18] L. Li, T. F. Bissyandé and J. Klein, "Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark," IEEE Transactions on Software Engineering, IEEE 47 (2021) 676-693.

[19] A. Atzeni, A. Marcelli, A. Sánchez, G. Squillero, A. Tonda and F. Díaz, "Countering Android Malware: A Scalable Semi-Supervised Approach for Family-Signature Generation," IEEE Access, vol. 6, pp. 59540-59556, (2018).

[20] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek and C. Stransky, "You Get Where You're Looking for: The Impact of Information Sources on Code Security," in 2016 IEEE Symposium on Security and Privacy (SP), IEEE (2016) 289-305.

[21] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in 2012 IEEE Symposium on Security and Privacy, IEEE (2012) 95-109.

[22] K. A. Houser and D. Sanders, "The Use of Big Data Analytics by the IRS: Efficient Solutions or the End of Privacy as We Know It?," Vanderbilt Journal of Entertainment and Technology Law, JETLAW 19 (2020) 817.

[23] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang and H. Chen, "Detecting third-party libraries in Android applications with high precision and recall," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE (2018) 141-152.

[24] C. Vendome, D. German, M. D. Penta, G. Bavota, M. L. Vásquez and D. Poshyvanyk, "To Distribute or Not to Distribute? Why Licensing Bugs Matter," in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE (2018) 268-279.

[25] B. Kim, K. Lim, S.-J. Cho and M. Park, "RomaDroid: A Robust and Efficient Technique for Detecting Android App Clones Using a Tree Structure and Components of Each App's Manifest File," IEEE Access, IEEE 7 (2019) 72182-72196.

[26] J. Davies, D. M. Germán, M. W. Godfrey and A. Hindle, "Software bertillonage: finding the provenance of an entity," in Proceedings of the 8th Working Conference on Mining Software Repositories, ACM (2011) 183–192.

[27] P. Gautam and H. Saini, "Various Code Clone Detection Techniques and Tools: A Comprehensive Survey," in International Conference on Smart Trends for Information Technology and Computer Communications, Springer (2016) 655–667.

[28] K. Chen, P. Liu and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in Proceedings of the 36th International Conference on Software Engineering, ACM (2014) 175–186.

[29] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in Network and Distributed System Security Symposium (NDSS), NDSS (2014) 23-26.

[30] J. Chen, M. H. Alalfi, T. R. Dean and Y. Zou, "Detecting Android Malware Using Clone Detection," Journal of Computer Science and Technology, Springer 30 (2015) 942-956.

[31] S. Hanna, L. Huang, E. Wu, D. Song , S. Li and C. Chen, "Juxtapp: A Scalable System for Detecting Code Reuse among Android Applications," in Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer (2013) 62–81.

[32] W. Zhou, Y. Zhou, X. Jiang and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in Proceedings of the second ACM conference on Data and Application Security and Privacy, ACM (2012) 317–326.

[33] W. Enck , P. Gilbert, B.-G. Chun, L. P. Cox , J. Jung, P. McDaniel and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in 9th USENIX Symposium on Operating Systems Design and Implementation, ACM (2014) 99–106.

[34] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM (2014) 259–269.

[35] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in 2012 IEEE Symposium on Security and Privacy, IEEE (2012) 95-109.

[36] I. Burguera , U. Zurutuza and S. N. Tehrani, "Crowdroid: behavior-based malware detection system for Android," in Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM (2011) 15–26.

[37] A. Sadeghi , N. Esfahani and S. Malek, "Mining the Categorized Software Repositories to Improve the Analysis of Security Vulnerabilities," in International Conference on Fundamental Approaches to Software Engineering, Springer (2014) 155–169.

[38] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee and K.-P. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," in 2012 Seventh Asia Joint Conference on Information Security, IEEE (2012) 62-69

[39] J. Crussell, C. Gibler and H. Chen, "AnDarwin: Scalable Detection of Semantically Similar Android Applications," in 18th European Symposium on Research in Computer Security, Springer (2013) 182-199.

[40] S. Rastogi, K. Bhushan and B. B. Gupta, "Android Applications Repackaging Detection Techniques for Smartphone Devices," Procedia Computer Science, 78 (2016) 26-32.

[41] S. Rastogi, K. Bhushan and B. B. Gupta, "Android Applications Repackaging Detection Techniques for Smartphone Devices," Procedia Computer Science, 78 (2016) 26-32.

[42] T. Vislavski, G. Rakic, N. Cardozo and Z. Budimac, "LICCA: A tool for cross-language clone detection," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE (2018) 512-516.

[43] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam and B. Maqbool, "A Systematic Review on Code Clone Detection," in IEEE Access, IEEE (2019) 86121-86144.

[44] J. Svajlenko and C. K. Roy, "CloneWorks: A Fast and Flexible Large-Scale Near-Miss Clone Detection Tool," in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE (2017) 177-179.

[45] C. Ragkhitwetsagul, J. Krinke and B. Marnette, "A Picture Is Worth a Thousand Words: Code Clone Detection Based on Image Similarity," in 2018 IEEE 12th International Workshop on Software Clones (IWSC), IEEE (2018) 44-50.

[46] H. Alomari, C. Newman, T. Sage, J. I. Maletic and M. Collard, "srcSlice: A Tool for Efficient Static Forward Slicing," in The 38th International Conference on Software Engineering (ICSE 2016), ACM (2016) 621–624.

[47] L. Glanz, S. Amann, M. Eichberg, M. Reif, B. Hermann, J. Lerch and M. Mezini, "CodeMatch: obfuscation won't conceal your repackaged app," in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM (2017) 638–648.

[48] Y. Feng, S. Anand, I. Dillig and A. Aiken, "Apposcopy: semantics-based detection of Android malware through static analysis," in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM (2014) 576–587.

[49] X. Sun, Y. Zhongyang, Z. Xin, B. Mao and L. Xie, "Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph," in ICT Systems Security and Privacy Protection, Springer (2014) 142-155.

[50] Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. L. Spina and E. Moser, "FSquaDRA: Fast Detection of Repackaged Applications," in The 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy, Springer (2014) 30–145.

[51] C. Tang, S. Chen, L. Fan, L. Xu, Y. Liu , Z. Tang and L. Dou, "A large-scale empirical study on industrial fake apps," in Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ACM (2019) 183–192.

[52] M. C. Grace, W. Zhou, X. Jiang and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks, ACM (2012) 101–112.

[53] C. Soh, H. Beng, H. B. K. Tan, Y. Arnatovich and L. Wang, "Detecting Clones in Android Applications through Analyzing User Interfaces," in International Conference on Program Comprehension, IEEE (2015) 163-173.

[54] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. S. Bay, R. Hall and D. Scofield, "Malware Analysis and attribution using Genetic Information," in International Conference on Malicious and Unwanted Software, IEEE (2012) 39-45.

[55] D. Bruschi, L. Martignoni and M. Monga, "Code Normalization for Self-Mutating Malware," IEEE Security and Privacy Magazine, 5 (2007) 46-54.

[56] White Papers, SANS, https://www.sans.org/white-papers/33578/. [Accessed 22 March 2010].

[57] W. Zhou, Z. Wang, Y. Zhou and X. Jiang, "DIVILAR: Diversifying intermediate language for anti-repackaging on Android platform," in Proceedings of the 4th ACM conference on Data and application security and privacy, ACM (2014) 199–210.

[58] J.-H. Jung, J. Y. Kim, H.-C. Lee and J. H. Yi, "Repackaging Attack on Android Banking Applications and Its Countermeasures," Wireless Personal Communications, 73 (2013) 1421–1437.

[59] A. Desnos and G. Gueguen, "Android: From Reversing to Decompilation," in ESIEA: Operational Cryptology and Virology Laboratory, Semantic Scholar (2011) 1-24.

[60] M. Li, P. Wang, W. Wei, S. Wang, D. Wu, J. Liu, R. Xue, W. Huo and W. Zou, "Large-Scale Third-Party Library Detection in Android Markets," in IEEE Transactions on Software Engineering, 46 (2020) 981-1003.

[61] Slide Share, Reverse engineering android apps, https://www.slideshare.net/PranayAiran1/reverse-engineering-android-apps. [Accessed 14 April 2013].

[62] Z. Ma, H. Wang, Y. Guo and X. Chen, "LibRadar: fast and accurate detection of third-party libraries in Android apps," in Proceedings of the 38th International Conference on Software Engineering Companion, ACM (2016) 653–656.

[63] H. Wang, Y. Guo, Z. Ma and X. Chen, "WuKong: a scalable and accurate two-phase approach to Android app clone detection," in Proceedings of the 2015 International Symposium on Software Testing and Analysis, ACM (2015) 71–82.

[64] J. Akram, Z. Shi, M. Mumtaz and P. Luo, "DroidCC: A Scalable Clone Detection Approach for Android Applications to Detect Similarity at Source Code Level," in IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), IEEE (2018) 100-105.

[65] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in International Conference on Intelligent Communication and Computational Techniques (ICCT), IEEE (2017) 1-7.

[66] J. Svajlenko and C. K. Roy, "A Machine Learning Based Approach for Evaluating Clone Detection Tools for a Generalized and Accurate Precision," International Journal of Software Engineering and Knowledge Engineering, 29 (2016) 1399-1429.

[67] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with BigCloneBench," in IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE (2015) 131-140.

[68] L. Li, H. Feng, W. Zhuang, N. Meng and B. G. Ryder, "CCLearner: A Deep Learning-Based Clone Detection Approach," in IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE (2017) 249-260.

[69] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in IEEE 19th International Conference on Program Comprehension, IEEE (2011) 219-220.

[70] C. K. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in 16th IEEE International Conference on Program Comprehension, IEEE (2008) 172-181.

[71] M. H. Alalfi, J. R. Cordy and T. R. Dean, "Analysis and clustering of model clones: An automotive industrial experience," in IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE), IEEE (2014) 375-378.

[72] R. Duan, A. Bijlani, M. Xu, T. Kim and W. Lee, "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM (2017) 2169–2185.

[73] J. Herbert and A. Litchfield, "A Novel Method for Decentralised Peer-to-Peer Software License Validation Using Cryptocurrency Blockchain Technology," in Proceedings of the 38th Australasian Computer Science Conference (ACSC 2015), ACSC (2015) 27-33.

[74] S. Gunasekera, Android Apps Security, Apress Berkeley, CA (2012).

[75] K. A. Bamberger, S. Egelman, C. Han, A. Elazari and I. Reyes, "Can You Pay For Privacy? Consumer Expectations and the Behavior of Free and Paid Apps," Berkeley Technology Law Journal, 35 (2020) 327-366.

[76] Z. Wang, Z. Zhang, Y. Chang and M. Xu, An Approach to Mobile Multimedia Digital Rights Management Based on Android, Genetic and Evolutionary Computing, Switzerland (2014).

[77] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel and M. Backes, "The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators," in IEEE Symposium on Security and Privacy (SP), IEEE (2018) 634-647.

[78] B. Rashidi and C. Fung, "A Survey of Android Security Threats and Defenses," Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications, JOWUA 6 (2015) 3-35.

[79] S. Karademir, T. Dean and S. Leblanc, "Using clone detection to find malware in acrobat files," in Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research, IBM Corp. (2013) 70–80.

[80] M. R. Farhadi, B. C. Fung, P. Charland and M. Debbabi, "BinClone: Detecting Code Clones in Malware," in Eighth International Conference on Software Security and Reliability (SERE), IEEE (2014) 78-87.