# Attack of the Clones: Detecting Cloned Applications on Android Markets

Jonathan Crussell[1,2], Clint Gibler[1], and Hao Chen[1]

[1] University of California, Davis
{jcrussell,cdgibler}@ucdavis.edu, hchen@cs.ucdavis.edu
[2] Sandia National Labs*, Livermore, CA
jcrusse@sandia.gov

**Abstract.** We present DNADroid, a tool that detects Android application copying, or "cloning", by robustly computing the similarity between two applications. DNADroid achieves this by comparing program dependency graphs between methods in candidate applications. Using DNADroid, we found at least 141 applications that have been the victims of cloning, some as many as seven times. DNADroid has a very low false positive rate — we manually confirmed that all the applications detected are indeed clones by either visual or behavioral similarity. We present several case studies that give insight into why applications are cloned, including localization and redirecting ad revenue. We describe a case of malware being added to an application and show how DNADroid was able to detect two variants of the same malware. Lastly, we offer examples of an open source cracking tool being used in the wild.

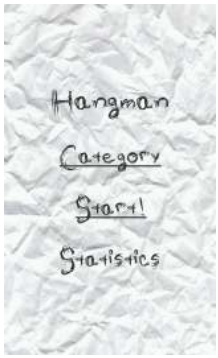
## 1 Introduction

In the past few years, mobile phones sales have grown explosively. As of November 2011, Android has dominant smart phone marketshare [9], with phone sales recently reaching 850,000 activations per day [24]. The Android operating system provides the core smartphone experience, but much of the user experience relies on third-party applications. To this end, Android has numerous marketplaces where users can download third-party applications that enable easy access to social networking, games, and more. As with traditional desktop applications, there is a need to protect users from malicious applications and developers from plagiarists who wish to benefit from a legitimate developer's hard work.

Developers can release applications on the official Android Market and/or on any one of a number of third-party markets. They can charge directly for their applications, but many choose to instead offer free applications that are ad-supported or contain in-game billing for additional content. Some applications have both a premium (paid) and free, ad-supported version.

* Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

S. Foresti, M. Yung, and F. Martinelli (Eds.): ESORICS 2012, LNCS 7459, pp. 37–54, 2012.




(a) `Google Market`
`br.com.passeionaweb...`
Ads: `googleads`


(b) `EoeMarket`
`com.ttmobilegame...`
Ads:: `wooboo`

**Fig. 1.** A pair of cloned applications. This paper detects cloning based on code similarity only, as an application's UI may be easily changed. Caption lists the market, package name and ad library associated with each application.

It is important to maintain a healthy market environment to encourage developers to continue creating applications. One important aspect of a healthy market is that developers are financially compensated for their work, an issue we investigate in this paper. There are several ways developers may lose potential revenue: a paid application may be "cracked" and released for free or a free application may be copied, or "cloned", and re-released with changes to the ad libraries that cause ad revenue to go to the plagiarist. In the latter case, the plagiarist may modify an existing library in an application, replacing the developer's client ID[1] with her own, or she may insert a new ad library that gives revenue to the plagiarist. Unfortunately, the openness of Android markets and the ease of repackaging Android applications contribute to the ability of plagiarists to clone applications and resubmit them to markets. Unlike Apple's App Store, where applications must pass a review process, applications on most Android markets are distributed without review. The official Android Market, however, recently added a service that scans new applications [35]. Although Google claims Bouncer drastically reduced the amount of malware installed by users, its effects on clones, which may not be malicious towards the user, is unknown.

Android application cloning has been reported by developers and the academic community [22, 23, 41]. An example we discovered, Fig. 1, shows the screenshots of two applications that are similar both in their UI and code, but were uploaded to different markets by different developers. Our analysis found the two to have significant code overlap, suggesting that at least one is a clone[2]. Since

[1] A client ID is a developer-unique string or number used by advertisers to determine who should be compensated when an ad is displayed or clicked.
[2] Potentially they could both be clones of an application we have not analyzed.

it is straightforward to detect directly copied code, we expect plagiarists to disguise their code to evade detection. To combat these disguises, we need robust techniques for detecting Android application cloning. We develop a technique based on program dependence graphs (PDGs) because it is has been shown to be effective in resisting many types of detection evasion techniques, such as statement reordering, insertion, and deletion [34]. Additionally, since it is uncommon for PDGs to be the same for independently developed code, our technique has a very low false positive rate.

Our contributions in this paper are as follows: (1) We have designed and implemented DNADroid, a tool for detecting cloned Android applications. DNADroid detects code clones based on PDGs and therefore resists common program transformations. (2) We ran DNADroid on applications downloaded from thirteen Android markets. DNADroid detected at least 141 applications that have been cloned. We show examples of applications being cloned multiple times by different developers, in one case up to seven times. (3) We demonstrate the very low false positive rate of DNADroid — we have manually verified, through UI or functionality comparisons, that all applications detected by DNADroid are in fact clones. (4) We present five case studies that illustrate different goals of mobile application plagiarists.

## 2 Background

**Android Markets.** As Android has increased in popularity, the number of applications has rapidly increased [21]. Developers can publish in the official Android market for a one-time $25 dollar fee, or use alternative markets such as SlideMe [14] and GoApk [10] which often only require an email address to publish applications. Unlike Apple's App Store, Android markets tend not to vet applications but rather rely on user feedback. This relaxed policy makes it easier for people to clone, modify, and redistribute applications. Finding these clones is important to protect developers' intellectual property and revenue streams and to alert users of potentially malicious clones.

**Android Application Structure.** Applications are distributed in Android Packages (APKs). These packages contain everything that the application needs to run- from resources like images and XML files specifying UI layouts to the application code. APKs also include a *manifest* XML that specifies a number of aspects about the application, including its name, version information, the package (or namespace) of the code, the permissions it requires to execute, and much more. Android applications are primarily developed in Java, though native code may be used. The Java source code is compiled to Java byte code and then converted into the Dalvik executable (DEX) format. Although similar to Java byte code, DEX byte code is incompatible with the Java virtual machine and instead runs on the Dalvik virtual machine. The conversion of Java byte code to DEX byte code is largely reversible and there are several tools that handle this conversion. We analyze only the DEX byte code and leave native code analysis for future work.

## 3 Threat Model

Our goal is to find cloned Android applications. We assume that the plagiarist has access to the compiled APK file that has been uploaded to an Android market. We also assume that the plagiarist will change some part of the file in order to change its cryptographic hash, as detecting identical applications is trivial.

**Definition of "Clone".** Clones occur when two applications (1) have similar code but (2) have different ownership. Therefore, clone detection differs from code reuse detection because the latter is concerned with only the first criterion. Because of the second criterion, DNADroid ignores (1) third-party libraries (for advertising, additional functionality, etc.), since they are intended to be reused and (2) multiple versions of the same application if they have the same ownership. Every Android application is signed by the owner's private key before being uploaded to a market. We determine two applications to have the same owner if they are signed by the same key.

We use the term owner rather than developer to describe the entity which published the application because a plagiarist illegitimately claims ownership of an application by publishing it under her own name without having developed the core functionality. Additionally, it is the owner that receives the revenue generated by the application, not the original developer.

**Resistance to Evasion Techniques.** A plagiarist will most likely modify the cloned code to evade detection. We design DNADroid to resist all the following evasion techniques: (1) *High level modifications*: Modify package, class, method and variable names as well as add or delete classes and methods. Create, change, or delete constants. (2) *Method Restructurings*: Move methods between classes, split a large method into multiple smaller methods, or combine multiple methods into a larger one. (3) *Control Flow Alterations*: Swap the *if* and *else* branches after negating the truth value. Change *for* loops to infinite *while* loops with a *break* statement or vice versa. Rewrite loops using *goto* statements. *Switch* and *if/else* statements may be swapped and individual cases may be reordered, created or removed. (4) *Addition/Deletion*: Insert code that does not affect the value of computed results or delete existing code. (5) *Reordering*: Reorder any code segments that are data and control independent.

**Non Goals.** We do not attempt to find cloning in native code in an application. As only a small percentage (7%) of the 75,000 applications we analyzed include native code, this is currently acceptable. Additionally, it is significantly more difficult for a plagiarist to understand and modify native code than DEX byte code. If a plagiarist does copy native code from an application, there is a good probability that she will steal DEX byte code as well, which DNADroid would find.

DNADroid does not attempt to determine which applications are the victims and which are clones. Without external knowledge, this is difficult to do in general based on the code alone. Simple solutions like comparing application

release dates or file sizes do not work in all cases, for example when a plagiarist steals beta releases [23] or when a plagiarist replaces an advertising library with a different, smaller one.

## 4 Clone Detection Approaches and Related Work

We describe several approaches for statically detecting cloned code, explaining their strengths and weaknesses, and conclude with the method used by DNADroid. As Android applications are largely interactive, dynamically detecting cloned code would face the same scalability limitations as TaintDroid [28], where authors had to manually interact with each application. This eliminates techniques such as [32, 37] for detecting similar Android applications. We also list and categorize related work, motivating the need for DNADroid.

**Feature Based.** Feature based approaches analyze a program and extract a set of features. Plagiarism between two programs is detected by comparing the extracted features from the programs. The features chosen can vary significantly, from number or size of classes, methods, loops, or variables to included libraries. This approach is limited because it discards so much information about the structure of the programs. Feature based systems are highly susceptible to having a low detection rate or high false positive rate.

**Structure Based.** Structure based systems convert programs into a stream of tokens and then compare the streams between two programs. By converting programs into a stream of tokens and ignoring easily changed constructs such as comments, whitespace, and variable names, structure based systems detect plagiarism more robustly than feature based systems. Examples of this approach include JPLAG [38], Winnowing [40] and MOSS [18]. Comparing DEX byte code streams could be a quite quick and scalable method to find *exactly* or *near exactly* copied code.

Unfortunately, the byte code streams contain no higher level semantic knowledge about the code, making this approach vulnerable to code modifications. For example, structure based approaches cannot determine if one or more instructions in the stream have been spuriously added and do not contribute to the outcome of the program. Winnowing [40] attempts to find plagiarism with modifications using *k-grams*, by finding common token substrings of length $k$. If the differences between the programs are relatively infrequent or tend to be greater than $k$ tokens apart then the comparison will find many $k$-length token streams in common. However, a wily plagiarist could simply insert a random instruction every few instructions to utterly break the stream comparison.

**Program Dependency Graph (PDG) Based.** A Program Dependence Graph (PDG) represents a method in a program, where each node is a statement and each edge shows a dependency between statements. There are two types of dependencies: data and control. A data dependency edge between statements $s_1$ and $s_2$ exists if there is a variable in $s_2$ whose value depends on $s_1$. For example, if $s_1$ is an

assignment statement and $s_2$ references the variable assigned in $s_1$ then $s_2$ is data dependent on $s_1$. A control dependency between two statements exists if the truth value of the first statement controls whether the second statement executes.

The evasion techniques discussed in our threat model (Sect. 3) hardly change a method's PDG. If the copied parts of the program behave the same as their original counterparts, they should have the same dependencies between the input and output variables. Since these dependencies do not change even after significant disguises have been applied to the copied code, PDG-based plagiarism detection is much more robust than structure based systems [34]. As we expect plagiarists to actively try to hide their work to various extents, this robustness is essential.

**Android Clone Detection.** There have been several recent papers which apply some of the above techniques to Android, here we briefly describe their approaches. We note that all these approaches are structure based or structure based approximations (using hashing).

Androguard [19] supports several standard similarity metrics including normal compression distance (NCD) and the comparison of the SHA256 hash of methods and basic blocks. NCD utilizes compressibility as a measure of similarity as two similar strings are more compressible than each on its own. DEXCD [27] tokenizes the opcodes in a decompiled APK and then attempts to find similar streams of opcodes between applications. DroidMOSS [41] computes fuzzy hashes of each method in the APK and combines them to form a hash for the entire APK. It then compares the fuzzy hashes of APKs to detect similarity based on the individual method hashes that both APKs share.

None of these tools use any semantic information to aid in detecting plagiarism. This makes them susceptible to evasion techniques discussed in Sect. 3. As such, we created DNADroid to more robustly detect the plagiarism of Android applications.

## 5 Methodology

DNADroid, as depicted in Fig. 2, proceeds in two stages. First, pairs of potentially cloned applications are selected based on their attributes. Then, the code of each pair of applications is examined to determine similarity.

### 5.1 Selecting Potentially Cloned Applications

The goal of an application plagiarist is to entice unwary users to choose her cloned application instead of the original. Since users find most applications through search, the plagiarist wishes to construct the name and description of her cloned application to resemble those of the original application so that both applications appear together in queries. Based on this observation, the first step of DNADroid is to select similar applications based on their attributes. As mentioned in Section 3, we do not consider pairs of applications signed by the same key, as they share a developer.

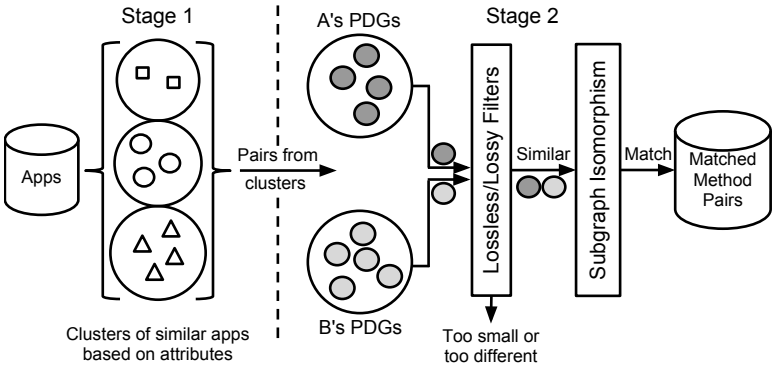



**Fig. 2.** Overview of DNADroid

**Determining Application Similarity Based on Attributes.** A plagiarist's goal is to have users install her clone so she will often use meta information that is similar to the victim application to describe the clone. By using similar meta information, a clone is more likely to appear in search queries with the victim. To mimic the search engines on Android markets, we use Solr [20], an open source enterprise-grade search platform from the Apache Lucene project, to index all the attributes of the applications, including name, package, market, owner, and description. In order to find clone candidates, we use Solr's fuzzy search on the meta information of one application to determine which applications are similar. These similar applications are fed into the second stage of DNADroid.

Although we found Solr effective in finding similar applications, DNADroid could use other tools for the same purpose, including using the markets' search and recommendation features directly.

### 5.2 Detecting Code Clones

The second stage of DNADroid determines the code similarity of a pair of applications.

**Constructing PDGs.** We convert both applications' code from the DEX format to a JAR using *dex2jar* [39]. [3] We then utilize WALA [25] to construct PDGs for each method in every class of the applications. We create the PDGs with only data dependency edges so that our detection is more robust against statement reordering, insertion and deletion

**Comparing PDGs.** DNADroid detects similarity between two applications by finding semantically similar code at the method level.

`Excluding Common Libraries` Many applications include third-party libraries, such as the ad library Admob or the Facebook API. As these libraries

[3] There are other tools available to convert from DEX to JAR, however, we found that *dex2jar* worked the best in practice. If a better tool became available, we could easily replace *dex2jar* with it.

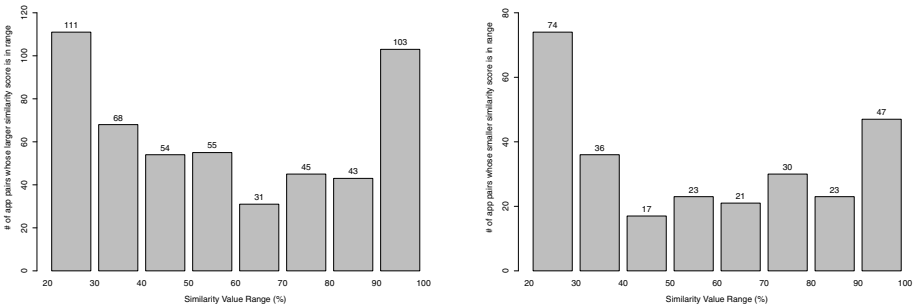
are not written by the owners of the applications, they should not be included in the clone detection. Libraries tend to have a common package name, like *com.admob.android* or *com.facebook.android*. However, we cannot simply filter classes based on package name alone, as a malicious owner could reuse a popular package name for her code or could insert malicious functionality into the library itself. We dumped both the package name and SHA-1 hash of known library files for thousands of applications and recorded the most frequent SHA-1 hashes for each library. This allows us to exclude common library code from analysis while remaining resistant to tampering.

`Lossless and Lossy Filters` Once we have constructed the PDGs for each method in $A$ and $B$, we apply two fast filters to exclude method pairs that are unlikely to be clones [34]. We first apply the *lossless* filter, which removes PDGs from consideration that are smaller than a specified size ($< 10$ nodes). Small matches between methods are more likely to occur by chance and these matches are often from trivial, boilerplate code.

Next we apply the *lossy* filter, which discards method pairs that are unlikely to match due to a difference in the distribution of types of nodes in the two PDGs. For example, a PDG that contains many method invocation nodes is unlikely to match one with none. First, we calculate a frequency vector for each of the methods in the pair. This vector counts how many times a specific node type occurs in the PDG. A method with five arithmetic operations would have a five in the dimension of the vector corresponding to arithmetic operations. We then compare these two vectors using hypothesis testing which calculates how likely one distribution is an observation from the first. Specifically, the hypothesis test we use is the G-test, which is a log likelihood ratio test. If the likelihood is below some significance threshold, $\alpha$, then we exclude the pair because the graphs have a low probability of being similar. Even though this filter may exclude similar PDGs in theory (hence the name *lossy*), we demonstrate experimentally that these cases are rare in practice with an $\alpha$ value of 0.05 (Sect. 6.4).

`Subgraph Isomorphism` If a pair of PDGs survives the above filters, the final test for similarity is subgraph isomorphism, which attempts to find a mapping between nodes in $PDG_A$ and nodes in $PDG_B$. Subgraph isomorphism is NP-Complete; however, when used for comparing PDGs, empirical evidence shows that it is often efficient because a PDG represents a single method, which developers tend to keep within a maintainable size. Additionally, PDGs are comprised of different statement types, which greatly reduces the possible mappings between two PDGs, as only nodes of the same statement type will match. We use the VF2 algorithm to compute subgraph isomorphisms, which is a backtracking algorithm geared towards matching large graphs [26]. VF2 takes advantage of the fact that PDGs contain a variety of node types, which restricts the total number of possible pairs of nodes for testing.

`Computing Similarity Scores` We determine the similarity of a pair of applications based on their matched PDG pairs. For each method $f$ (excluding the methods in known libraries) in application $A$, let $|f|$ be the number of nodes in this method's PDG. Find the best match of this PDG in $B$'s PDGs and denote




(a) Distribution computed based on the larger similarity score, $\max(sim_{A(B)}, sim_{B(A)})$, in each application pair

(b) Distribution computed based on the smaller similarity score,$\min(sim_{A(B)}, sim_{B(A)})$, in each application pair

**Fig. 3.** Distribution of similarity scores among application pairs. Each bar represents the number of application pairs whose similarity scores are in the range on the x-axis

it as $m(f)$. Our metric, *similarity score*, is the ratio between the sums of the $|f|$ values and the $|m(f)|$ values:

$$sim_{A(B)} = \frac{\sum_{f \in A} |m(f)|}{\sum_{f \in A} |f|} \tag{1}$$

Equation 1 shows the portion of application $A$ that is matched by code in application $B$.

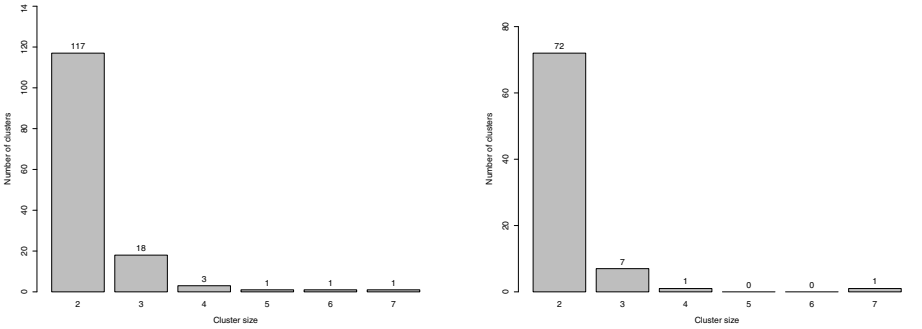
## 6 Evaluation

We collected 75,000 free applications from thirteen Android markets: the official Android market [31] and a number of third party markets [1, 3, 2, 5–7, 10–14]. From these applications, we randomly selected 9,400 pairs from the potential clones identified by the first stage of DNADroid based on their attributes (Sect. 5.1). The second stage of DNADroid determined which of these pairs were indeed clones based on code similarity (Sect. 5.2).

We used the Hadoop [4] MapReduce framework to parallelize DNADroid and HDFS to share data across a small cluster of one server-class and three desktop machines. The average throughput of DNADroid on this small cluster is 0.71 application pairs per minute.

### 6.1 Similarity between Applications

We define application clones as a pair of applications that have similar code but different ownership. The comparison of each pair of applications $A$ and $B$




(a) Similarity score threshold=70% (b) Similarity score threshold=90%

**Fig. 4.** Distribution of clone cluster sizes

produces two similarity scores, $sim_{A(B)}$ and $sim_{B(A)}$, as defined in Equation 1. $sim_{A(B)}$ is the percentage of code in $A$ that is matched by code in $B$. A high similarity score shows that a substantial portion of one application is present in another, providing evidence of code cloning.

Figure 3 show the distributions of the similarity scores among all the pairs of applications analyzed: Fig. 3a uses the larger similarity score in each application pair while Fig. 3b uses the smaller score. Figure 3a shows that 103 application pairs have similarity scores above 90%, 43 application pairs between 80% and 90%, and 45 application pairs between 70% and 80%.

In this paper, we define two applications to be clones when at least one of the applications has a similarity score over 70% ($\max(sim_{A(B)}, sim_{B(A)}) \geq 70\%$). We choose to use the max similarity score of the pair to avoid the following problem: a malicious developer may add a significant amount of code to the cloned application, causing the original application code to match a small percent of the cloned application. However, she cannot influence the content of the original application, which has already been released. The original application will still be highly matched by the cloned application, causing DNADroid to identify the clone pair. Using a 70% similarity score threshold, DNADroid found at least 191 application pairs in which one or both of the applications are clones.

### 6.2 Clustering Cloned Applications

Are many Android applications cloned a small number of times or are a relatively few cloned many times? We attempt to gain insight into this question by clustering applications based on their computed similarities. Clusters are computed using the following algorithm: for each pair of applications, $A$ and $B$, if either $sim_{A(B)}$ or $sim_{B(A)}$ is above the threshold, then $A$ and $B$ are in the same cluster. After running this algorithm over all pairs of applications, we have a set of clusters, each of which contains at least 2 applications.

Figure 4 shows the distribution of the sizes of clone clusters at two different similarity score thresholds. The majority of the clone clusters have just two applications; however, there are larger clusters with the largest having seven

applications. Figure 4a illustrates that, at a 70% threshold, DNADroid found at least 141 applications that are victims of cloning. As each clone cluster of applications has at least one victim application, the number of clusters is a lower bound on the number of victim applications.[4]

It is instructive to examine the clone clusters. Figure 5 shows two clusters. Figure 5a shows a cluster of six applications. The bottom three applications ($21ad$, $aa87$, and $f59d$) are signed by the same private key (i.e., written by the same author) and have the same package name (`com.bwx`) but have different version numbers. The top application ($714a$) has a different package name (`com.zhanghuisns`) and is signed by a different private key. Finally, the middle two applications are signed by the same private key and have the same package name (`com.mybooft`). Based on the key signatures, we can split the graph into three families - top, middle and bottom. Using the similarity scores and the version information, it appears that the middle family most likely cloned from an ancestor of the bottom family and that the top family may have cloned from the middle or bottom family.

Figure 5b shows similar relationships between different families of applications, where a seed family appears to have been cloned multiple times by different developers. These figures demonstrate that clustering is an effective tool in analyzing relationships between cloned applications.

### 6.3 Visual and Behavioral Verification

To confirm that the application pairs identified by DNADroid are indeed similar, we examined their GUI and user interactions. Figure 7 shows the screenshots of some of the application pairs that were detected by DNADroid as clones. It takes only a quick glance to determine that both screenshots in each application pair are indeed very similar. For application pairs whose initial screen shots are drastically different, we manually ran and interacted with them to verify that they have similar functionality. Manual verification confirmed that every application pair found by DNADroid were in fact clones, yielding an experimental false positive rate of 0%.

### 6.4 Filter Performance

**Filter Effectiveness.** DNADroid uses several filters to improve its speed and scalability by excluding method pairs that are unlikely to match. A naive approach would require $O(n * m)$ method comparisons, where $n$ and $m$ are the number of methods in each application. To reduce the number of method pair comparisons, DNADroid uses three filters (Sect. 5.2). The library class filter excludes on average 27.16% of each application's classes. The *lossless* and *lossy* filters on average exclude 33.88% and 2.62% of the methods in an application,

[4] The victim application may or may not be a member of the clone cluster. The latter case arises if we downloaded only the clones of the victim application but not the victim itself.

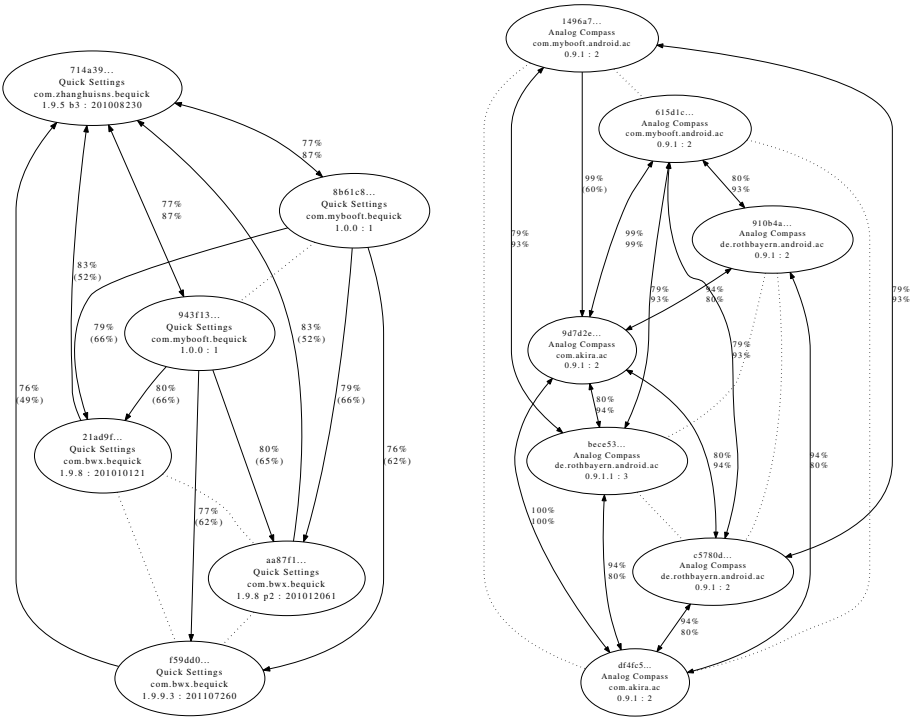


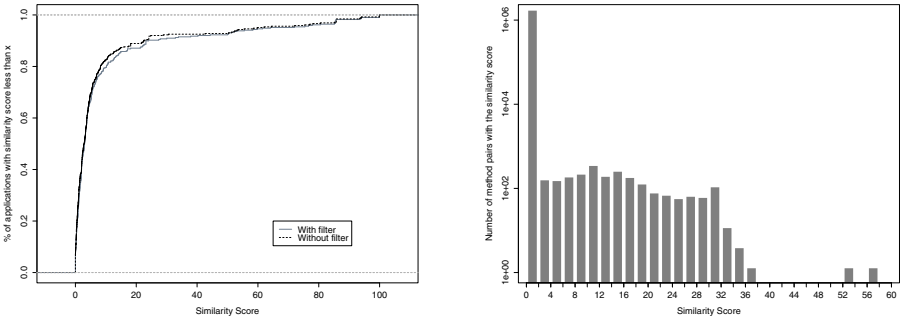

**Fig. 5. Application Clone Clusters.** Each node represents an application. The label in each node contains the SHA-1 hash prefix, name, package, version name, and version code of the application. Each solid edge from Application $A$ to Application $B$ means that a large percentage ($> 70\%$) of $A$ is found in $B$, where the top number on the edge is the similarity score of $A$ in $B$, and the bottom number (in parentheses) is the similarity score of $B$ in $A$. A dotted line links two applications by the same author (have the same public key signature).

respectively. Combined, these three filters reduce DNADroid's search space by 90.04%.

**Filter Accuracy.** Of the three filters, only the *lossy* filter may exclude interesting methods pairs that would have matched[5]. We wish to ensure that our *lossy* filter rarely rejects similar method pairs, as this would cause DNADroid to underreport the similarity of applications and potentially miss clone pairs.

To measure the accuracy of the *lossy* filter, we randomly selected 250 application pairs already examined by DNADroid and reran them without the *lossy* filter. Figure 6a is a CDF of the application similarity scores, both with and without the *lossy* filter. The figure shows that the *lossy* filter has negligible

[5] The class filter excludes known common libraries and the lossless filter excludes small methods, neither of which constitute interesting code reuse.




(a) Effect of *lossy* filter on application similarity scores.

(b) Distribution of similarity scores of method pairs excluded by the *lossy* filter in log-y scale.

**Fig. 6.** Examining *lossy* filter effectiveness

impact on similarity scores. Therefore, the *lossy* filter does not cause DNADroid to miss clone pairs it would otherwise have found.

Figure 6a explored the macro effects of the *lossy* filter, in Fig. 6b we examine its effect on individual method pairs. Figure 6b shows the histogram of the similarity scores of the method pairs excluded by the *lossy* filter on a *log-y* scale. As expected, more than 99.86% of these similarity scores are zero (The similarity score of A in B, $sim_{A(B)}$, is zero if the PDG A is not subgraph isomorphic to the PDG B). Only a few similarity scores exceed 40%, and no score exceeds 60%.

This experiment demonstrates that the *lossy* filter is highly accurate: it seldom excludes method pairs that are likely clones and it negligibly affects the similarity scores of application pairs.

## 7 Case Studies

**"Benign" Cloning.** DNADroid found 30 pairs that both have a 100% similarity score using our matching algorithm. For the few that we manually reviewed, we found that the applications were indeed identical, apart from having String values in the application translated. Since these strings are constants, changing them doesn't change the PDGs. There seems to be no incentive for the plagiarist apart from providing an application to an otherwise excluded audience. For the latter reason, we believe these pairs to be cases of "benign" cloning, since there appears to be no benefit to the plagiarist. However, without manual review, we cannot confirm that they are all "benign."

**Changes to Advertising Libraries.** A number of clone pairs involved applications that had changes to their advertising libraries. As stated in Sect. 5, DNADroid can discern application from library code in APKs. Using this and our coverage values, we can see when an application has most likely been cloned for monetary gain.

An example of such cloning is a download manager, XWind Downloader, which we have found on three different markets: the official Android Market [31], GoApk [10], and Freeware Lovers [8]. The versions available from the official Market and Freeware Lovers have the same SHA-1 hash and were both published by the same developer account name, which leads us to believe that the author has officially published his application in both markets. The GoApk version, however, has a different SHA-1 hash and is signed with a different developer key. The GoApk version has removed the Youmi [17] advertising library present in the application from two other markets and has replaced it with the WooBoo [16] advertising library. DNADroid found 99.9% of the official Android Market version within the GoApk version, an almost sure sign of cloning.

For the 141 applications that we believe to be the victims of cloning, we compared the libraries that DNADroid detected in the victim with those in the clone. We found that 91 (65%) of these pairs had different libraries, all of which included changes to advertising libraries. This number suggests that plagiarists are often fiscally motivated, attempting to siphon ad revenue from popular applications.

**Malware Added to an Application.** "HippoSMS" is a malicious application recently discovered by [33] that we downloaded and compared to our collection of applications.

We found that it shares the same package name as a Chinese video player we crawled from GoApk. Both applications require a surprising number of sensitive permissions; the video player requires 11 permissions while the malware requires 10. According to Stowaway [29], a tool for detecting over privileged applications, the seemingly benign video player requires 6 permissions that it doesn't use, whereas the malware only requires 1 extra. Given the number of permissions the video player requires, we conjecture that its developer may have intended to insert malware into the application at a later time, or that the video player is a clone itself. When compared with DNADroid we discovered that 98.57% of the video player code is in the malicious application, a near certain indicator of cloning.

**Two Variants of the Same Malware.** This case study consists of two malicious applications that are identified by VirusTotal [15] as being variants of the "BaseBridge" malware family. Both applications have been stripped of meaningful class and method names. However, this obfuscation did not fool DNADroid — DNADroid found coverages of 35% and 28% between the two variants. Manual review confirmed that the methods matched between the applications perform the malware functionality. This demonstrates the potential of DNADroid to aid markets in automatically detecting similar variants of the same malware, though significant transformations could subvert DNADroid's current implementation.

**Use of Freeware Cracking Tool in the Wild.** During our exploration of public work in Android application cloning we encountered the cracking tool AntiLVL [36]. AntiLVL attempts to automatically subvert several types of license protection mechanisms used in Android applications including the Android

License Verification Library (LVL), Amazon Appstore DRM and Verizon DRM. We found applications cracked by AntiLVL hosted on several markets.

AntiLVL has several primary mechanisms for subverting license protections. After decompiling an application with *baksmali* [30], AntiLVL attempts to subvert common license enforcement checks by rewriting them to always return successfully. AntiLVL also inserts a new file, *SmaliHook.class* in the applications it rewrites. This class contains methods to spoof the device ID, make fake license checks which always return true, and hide AntiLVL's modifications from the application itself by returning the original applications file size, MD5, and signatures for the original application. We also found that the cracked applications occasionally show evidence of AntiLVL use in their CERT.SF, a signature file included in applications that lists the digital signatures of every file in the application.

We found 189 applications containing SmaliHook.class and 235 containing references to AntiLVL in their signature files for a total of 310 unique applications. Given the nature of AntiLVL, it's almost certain that these applications are clones of paid applications. Interestingly, even though only 8% of our total applications were acquired from Chinese markets, 88% of the applications including AntiLVL traces were from Chinese markets. Only four applications containing AntiLVL were obtained from the official Android market, despite it being the source of 65% of our applications. Two of the four applications were different versions of the same application which Google has since removed. Of the remaining two applications on the Android market, both are live and have "50,000 to 100,000" installs as of March 2012.

## 8 Discussion

**False Positive.** Since it is a serious allegation to claim an application is a clone, we design DNADroid to have a very low false positive rate. We manually verified that *all* the application pairs that DNADroid identified as clones are indeed similar, with either similar start-up screens or similar user interactions (Sect. 6.3).

**False Negative.** DNADroid may overlook cloned applications due to a few reasons. First, DNADroid uses Solr to select candidate cloned applications based on their attributes, such as name and description (Sect. 5.1). This is based on the observation that cloned applications often have similar attributes as the original so that they appear together in market search results. Therefore, if the plagiarist crafts the attributes of her application to avoid being identified as having similar attributes to the original application (e.g., by using a different language), it can avoid detection by DNADroid. However, by attempting to evade initial similarity detection, a plagiarist may jeopardize the chances of her clone being installed. This is not a fundamental limitation, as DNADroid would still find a high code similarity between the two if compared. If a better tool to identify similar applications becomes available, DNADroid could easily leverage it.

Another source of false negatives is program obfuscation. By using PDG-based clone detection, DNADroid can resist common program transformations (Sect. 3). However, there exist advanced program transformations that can evade PDG-based clone detection. This is a fundamental limitation of DNADroid. Even though these advanced transformations are feasible, they require much more effort by the plagiarist (ultimately, the plagiarist can reimplement the application, which is not cloning in the strict sense).

**Comparison to Other Approaches.** We ran Androguard [19] against the same 191 pairs that DNADroid identified as clones. Androguard performed well in some cases, but crashed on 24 pairs and found very low coverage values for 10 pairs, causing it to miss 18% of the pairs DNADroid found. We intended to compare DNADroid to DEXCD [27] and DroidMOSS [41] but DEXCD had problems running on the pairs DNADroid identified and DroidMOSS is not currently publicly available. We hope to compare results in the future.

**Performance.** There exist more efficient algorithms for detecting code clones, however, these algorithms trade robustness for speed. Robust techniques, such as those utilized by DNADroid are more expensive but result in fewer false positives and false negatives. Fortunately, we can take advantage of inexpensive meta information clustering and the inherent parallelism in clone detection to make DNADroid practical.

## 9 Conclusion

The explosive growth of Android devices over the past few years has led to a booming mobile application community. Unfortunately, with increased incentives and low barriers to entry, plagiarists and clones have followed. To combat cloning, markets need robust techniques to identify these clones, as application clones harm the market ecosystem. We present DNADroid, a tool for finding clones on a large scale. DNADroid selects likely clone candidates based on their attributes and then robustly compares their code for significant overlap. We evaluated DNADroid on applications crawled from thirteen Android markets. DNADroid identified at least 141 applications that have been cloned and an additional 310 applications that were cracked with AntiLVL, an open source Android cracking tool. We describe five case studies which provide insight into different motivations for plagiarists. DNADroid has a very low false positive rate — we have confirmed that all the applications detected by DNADroid are indeed clones via visual or behavioral verification. Our findings indicate that DNADroid is an effective tool to aid in the fight against mobile application cloning.

**Acknowledgments.** The authors would like to thank Ben Sanders and Justin Horton for helping us obtain Android applications and our anonymous reviewers for their input. This paper is partially based upon work supported by the National Science Foundation (NSF) under Grant No. 0644450 and 1018964. Any opinions, findings, and conclusions or recommendations expressed in this

material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## References

1. Amazon appstore (May 2012), http://www.amazon.com/mobile-apps/
2. Android soft 4 u market (May 2012), http://www.androidsoft4u.com/
3. Androidonline market (May 2012), http://www.androidonline.net/
4. Apache hadoop (May 2012), http://hadoop.apache.org/
5. App china market (May 2012), http://www.appchina.com/
6. Brother soft market (May 2012), http://www.brothersoft.com/
7. Eoemarket (May 2012), http://www.eoemarket.com/
8. Freeware lovers market (May 2012), http://freewarelovers.com
9. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent (May 2012), http://www.gartner.com/it/page.jsp?id=1848514
10. Goapk market (May 2012), http://market.goapk.com
11. Handango market (May 2012), http://www.handango.com/
12. M360 market (May 2012), http://app.m.360.cn/
13. One mobile market (May 2012), http://www.1mobile.com/
14. Slideme: Android community and application marketplace (May 2012), http://slideme.org/
15. Virustotal (May 2012), http://virustotal.com
16. Wooboo advertising library (May 2012), http://www.wooboo.com.cn/
17. Youmi advertising library (May 2012), http://www.youmi.net
18. Aiken, A.: Moss (measure of software similarity) plagiarism detection system (1998)
19. Androguard: Androguard: Manipulation and protection of android apps and more... (May 2012), http://code.google.com/p/androguard/
20. Apache. Solr (May 2012), http://lucene.apache.org/solr/
21. AppBrain. Number of available android applications (May 2012), http://www.appbrain.com/stats/number-of-android-apps
22. BajaBob. Smalihook. java found on my hacked application (May 2012), http://stackoverflow.com/questions/5600143/android-game-keeps-getting-hacked
23. Beard, S.: Market shocker! iron soldiers xda beta published by alleged thief (May 2012), http://androidheadlines.com/2011/01/market-shocker-iron-soldiers-xda-beta-published-by-alleged-thief.html
24. Burns, M.: 850k daily android activations, 300m total devices, says andy rubin (May 2012), http://techcrunch.com/2012/02/27/850k-android-activations-daily-300m-total-devices-says-andy-rubin/
25. IBM T. J. Watson Research Center. Watson libraries for analysis (wala) (May 2012), http://wala.sourceforge.net/wiki/index.php/Main_Page
26. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10), 1367–1372 (2004)
27. Davis, I.: Dexcd (May 2012), http://www.swag.uwaterloo.ca/dexcd/index.html
28. Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, pp. 1–6. USENIX Association (2010)

29. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 627–638. ACM (2011)
30. Freke, J.: smali: An assembler/disassembler for android's dex format (May 2012), https://code.google.com/p/smali/
31. Google. Android market (May 2012), http://market.android.com
32. Jhi, Y.C., Wang, X., Jia, X., Zhu, S., Liu, P., Wu, D.: Value-based program characterization and its application to software plagiarism detection. In: Proceeding of the 33rd International Conference on Software Engineering, pp. 756–765. ACM (2011)
33. Jiang, X.: Security alert: New android malware – hipposms – found in alternative android markets (May 2012), http://www.cs.ncsu.edu/faculty/jiang/HippoSMS/
34. Liu, C., Chen, C., Han, J., Yu, P.S.: Gplag: detection of software plagiarism by program dependence graph analysis. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 872–881. ACM (2006)
35. Lockheimer, H.: Android and security (April 2012), http://googlemobile.blogspot.com/2012/02/android-and-security.html
36. lohan: Antilvl - android license verification library subversion (May 2012), http://androidcracking.blogspot.com/p/antilvl.html
37. Myles, G., Collberg, C.: Detecting software theft via whole program path birthmarks. In: Information Security, pp. 404–415 (2004)
38. Prechelt, L., Malpohl, G., Philippsen, M.: Finding plagiarisms among a set of programs with jplag. J. UCS 8(11), 1016 (2002)
39. pxb1988, dex2jar: A tool for converting android's .dex format to java's .class format (May 2012), https://code.google.com/p/dex2jar/
40. Schleimer, S., Wilkerson, D.S., Aiken, A.: Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, pp. 76–85. ACM (2003)
41. Zhou, W., Zhou, Y., Jiang, X., Ning, P.: Detecting repackaged smartphone applications in third-party android marketplaces. In: Proceedings of 2nd ACM Conference on Data and Application Security and Privacy, CODASPY 2012 (2012)

## A Screenshots


(a) (b)

**Fig. 7.** Screenshots of pairs of cloned applications