SAND2011-5889C

# Attack of the Clones: Detecting Plagiarism on Android Markets

Jonathan Crussell Clint Gibler Hao Chen
{jcrussell, cdgibler}@ucdavis.edu, hchen@cs.ucdavis.edu

August 17, 2011

**Abstract**

The number of applications developed for the Android smart phone operating system has exploded since its release in 2009. These applications have been developed largely by independent developers and are distributed through the official Android Market or one of many independent markets. Unlike Apple's App store, where applications must pass a review process before being published, applications on most Android markets are distributed without review. Unfortunately, due to the open nature of Android markets and the ease with which Android applications can be repackaged and redistributed, a number of Android applications have been plagiarized. These plagiarists create clones of applications that may make a paid application freely available, add or modify an advertising library for their monetary gain or insert malicious functionality.

We present DNADroid, a tool that detects potential plagiarism by robustly computing the similarity between two applications. DNADroid achieves this by comparing data dependency graphs between methods within the applications. We describe clone finding criteria that we apply to our dataset of 29,000 applications to find groups of potential clones. Among these clone candidate groups, we find the similarities and differences between multiple version of the same application, applications that have likely been stolen by another developer, and applications that have had malware inserted into them. We discuss four case studies: one case of seemingly benign plagiarism, one of an application being stolen, one in which malware has been added and lastly two variants of the same malware. Finally, we present our overall findings, including statistics on the number of likely plagiarized applications we encountered, giving the first large-scale quantitative look into the state of mobile application plagiarism.

## 1 Introduction

As Android has gained dominant smart phone market share, the monetary incentive to develop Android applications has increased. Developing a complex Android application takes a significant amount of time and effort, but copying an existing successful application can be done comparatively quickly. The open nature of Android markets and the ease with which Android applications can be repackaged and redistributed lowers the cost of copying applications, and a number of have already been plagiarized [5, 6]. There is a large monetary incentive for the unscrupulous to plagiarize an existing application and insert an advertising library or change an existing ad library's Client Id, sending them the ad revenue rather than the original developer. To continue profiting from their stolen applications, plagiarists are likely to attempt to disguise their copied code to prevent their applications from being pulled from markets.

Robust techniques are required for detecting Android application plagiarism, as the plagiarists may be talented developers with a large amount of time on their hands to mask their work. We chose to use program dependence graph (PDG) comparison, proposed by [16], to detect plagiarism because of its effectiveness in resisting many types of plagiarism evasion techniques, including statement reordering, insertion, and deletion. We utilize *lossless* and *lossy* filters that allow us to check for copied code on a large scale.

To find potential plagiarized applications, we describe a clone finding criteria which finds sets of applications that may be cloned. We present DNADroid, a tool that uses a robust plagiarism detection approach to detect similarities between applications.

Our contributions in this paper are as follows:

- We describe a clone finding criteria that can be applied to a set of applications in order to determine the most likely clones.
- We present DNADroid, a plagiarism detection tool which reports the similarity between two applications of interest. DNADroid also generates a "diff" between two applications which we automatically cluster to determine common plagiarism patterns.
- We present four case studies which describe specific examples of application cloning. These case studies explore the intent of the plagiarist, and the general reasons why plagiarism occurs on Android.
- Applying our clone criteria to our dataset of over 29,000 applications yielded 5074 clone candidate pairs to test. We give statistics about the number and types of clones found, giving the first large-scale insight into the current state of mobile application plagiarism.

## 2 Background

**Android Markets** As Android has increased in popularity, the number of applications developed for it has exploded [4]. The Android operating system provides the core functionality, however, most of the user experience is left to application developers. To encourage third-party developers, Android freely distributes its SDK, allowing anyone to create applications. Developers can publish in the official Android market for a one-time $25 dollar fee, or use alternative markets such as SlideMe[1] and GoApk[2]. Unlike Apple's App Store, Android markets tend not to vet applications that are submitted. Unfortunately, this relaxed policy makes it easier for plagiarists to clone applications and redistribute them, oftentimes with modifications. Finding these clones is important to protect developers' intellectual property and revenue streams, and to protect users from potentially malicious clones.

**Android Application Structure** Applications are distributed in packages called Android Packages (APKs). These packages contain everything 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 (such as INTERNET or READ_CONTACTS), and much more. The manifest also informs the Android operating system about the components of an application and how they interact with the rest of the system.

[1] http://slideme.org
[2] http://goapk.com

Android applications are primarily developed in Java, though native code may be used. The Java application 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 use *dex2jar* [18] as we found it converts DEX to Java byte code the most consistently.

**Program Dependence Graph** A Program Dependence Graph (PDG) is a graph where each node represents a statement in a method and edges represent 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 is executed.

# 3 Plagiarism Detection

In this section we discuss plagiarism detection methods that do not analyze PDGs. We also describe detection evasion techniques that may be applied by a plagiarist.

## 3.1 Detection Methods

Plagiarism detection systems have existed for decades, usually to catch students who turn in copied code [2]. These systems use a variety of mechanisms to detect similar code. Previous to GPLAG, there were two approaches to plagiarism detection according to [17]:

**Feature Based** These systems analyze a program and extract from it a set of features. In order to detect plagiarism between two programs, the features extracted are compared rather than the programs themselves. This approach is limited since so much information about the programs is discarded. Feature based systems are also susceptible to having a low detection rate or high false positive rate, depending on the features used.

**Structure Based** These 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 things such as comments and whitespace, structure based systems provide a much more robust detection of plagiarism over feature based systems. Structure based systems, such as JPLAG[17] and MOSS[2], are commonly used in academic institutions to detect students who are turning in similar code and they perform well in these settings. However, these systems can be tricked by a plagiarist who utilizes the evasion techniques discussed in the following section.

## 3.2 Evasion Techniques

In order to evade detection, a plagiarist may employ many tricks to modify the copied copy. The techniques discussed in this section are directed towards evading Structure Based plagiarism detection, as Feature Based detection is fundamentally limited because it discards too much information, no matter how many features are used [17].

1. *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. Rewrite loops using *goto* statements or recursion.
2. *Statement Addition*: Insert extra statements into the plagiarized code that do not affect the value of computed results.
3. *Statement Reordering*: Reorder any statements that are not dependent on each other.

While some of these techniques may be time consuming for a plagiarist, it's important to note that these are largely simple alterations and could be partially or fully automated by a frequent plagiarist. These evasion techniques quickly throw off Structure Based systems; however, they 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. The fact that these dependencies do not change, even with significant disguises applied to the copied code makes PDG based plagiarism a very appealing option over Structure Based systems, as put forth by [16].

# 4 Threat Model

Given two Android applications, our goal is to determine the likelihood and the extent to which one may have plagiarized the other. It's likely that the plagiarist will only have access to the compiled APK file she wishes to copy, but it is straightforward to obtain a readable, though low-level version of the source[12, 18]. Our approach is unaffected by the plagiarist having access to the original source or to just the application itself.

We assume the plagiarist is freely able to modify package, class, method or variable names. Classes and methods may be added or deleted. Methods can be moved between classes. A large method can be split into multiple methods and multiple methods may be combined into one. In the following descriptions we define a statement to be any one of a number of possible code constructs, such as variable declarations or assignments, *if/else* conditions or *for/while* loops. Our approach is robust against the following alterations within a method. Existing statements may be reordered, new ones inserted or original ones deleted. Constants may be changed, deleted or created. *If/else* statements may have their conditions inverted and switched, *for* loops may be replaced by infinite *while* statements with *break*s or vice versa. Similarly, *switch* statements and *if/else* statements may replace each other and individual cases may be reordered, created or removed.

We assume plagiarists are unlikely to rewrite a large percentage of an existing application, as the time needed to understand an application's design well enough to significantly rewrite it may be similar to simply recreating the application from scratch.

We do not attempt to find plagiarism in native code included in an application. Converting assembly to higher level code or just trying to grasp its purpose is significantly more difficult than for DEX or Java byte code. Thus it is less likely for plagiarists to take the time to reverse engineer native code unless it performs some critical functionality, such as managing passwords or enforcing DRM. If a plagiarist is copying native code from an application there is a good probability that Java code is being stolen as well, which DNADroid would find. Finally, only a small percentage of applications include native code, only 6% of 18,000 according to [14].

# 5 Methodology

In this section we discuss the architecture and implementation of DNADroid. First, we describe the criteria that we used to select clones from our dataset of applications.

## 5.1 Clone Criteria

A key part of our work is being able to pick out likely groups of clones from our dataset. Due to the algorithmic complexity and CPU usage required by clone detection algorithms, running clone detection on every pair of applications we have is infeasible given our current resources. Analyzing every possible clone pair would require $O(n^2)$ comparisons, as one would have to compare each application with every other application. With the current number and the rate at which new applications become published, doing this would likely take a server farm. We go further into the computation power and time needed to do this in Section 7. Thus we must have a criteria for selecting groups of likely clones. We select clone groups based on a number of application attributes, including:

1. *Name*: The name of the application displayed to users.
2. *Package name*: A string that identifies the application code's namespace, assumed by phones to be globally unique[3].
3. *Version code*: An attribute specified in the manifest that is used by markets to determine the latest version of an application.
4. *Certificate fingerprints*: An application is signed by the developer[1] before it's published. We compute the SHA1 fingerprint of the certificate used to verify the application signature.
5. *Permissions*: The permissions an application require defines its capabilities.
6. *Market*: The market from which we acquired the application.

A plagiarist wanting to piggyback on the success of a popular application is likely to use an identical or similar name to the original, increasing the likelihood of being returned in search results or tricking users and thus downloaded. The package is used to uniquely identify an application on a phone, and is in theory unique within each market, since two applications with the same package cannot both be installed. However, since we have applications from multiple markets, there are possible collisions between package names. These collisions are significant because package names are usually created based on the developer's website URL which should be unique. Though Android applications should maintain the same name, package and public key across versions, we found several cases in which at least one of these changed between versions of an application we believe from the same developer. Determining authorship on Android is difficult in general and will be discussed further in Section 7.3. The version code yields the chronological ordering of multiple versions of the same application. This can help determine which version of an application a plagiarist most likely copied. Applications by the same developer, and especially multiple versions of the same application, should be signed by the same private key. We assume applications signed by the same private key are produced by the same developer, though the opposite is not always true. We take special note of cloned applications which require different sets of permissions since the addition of sensitive permissions may indicate that malicious code has been inserted.

[3]The official Android Market also enforces package uniqueness.

We have written a configurable script to generate candidates using the specified criteria. The script finds potential clones based on any number of above listed attributes being the same or different. For example, searching for applications with the same name, package and public key will find multiple versions of an application by the same developer. Applications with the same name but different package, public key and different permissions are likely to be by different developers with potentially interesting extra code included, such as new ad libraries or malicious functionality. For our results and cases studies, we used the following two criteria to generate candidates: applications with the same name or package but different certificate fingerprints.

### 5.2 Clone Detection

Now we describe our clone detection process on a pair of Android applications, $A$ and $B$. First, we convert both applications' code from DEX format to a JAR using *dex2jar*[18]. We then utilize WALA to construct PDGs for each of the methods in every class of the applications. We create the PDGs with only data dependency edges in order to make the clone detection more robust against statement reordering. The PDGs we generate may not be fully connected graphs because some statements may only be connected by control dependency edges. As a fully connected graph is required for our comparison algorithm, we add edges from the method entry node to all other nodes in the method's PDG.

An optimization we use to reduce the search space is to compare the SHA1 hashes of each Java class file generated by *dex2jar*. If we find an overlap between the class files in the two applications, we exclude all the PDGs within those overlapping classes from analysis. This is very useful for comparing applications with the same library code, which is usually separated into its own set of classes. While this optimization may be fragile, we found it worked well in practice.

Once we have constructed the PDGs for each method in $A$ and $B$, we apply two filters proposed by GPLAG [16] to reduce the search space of the methods we must compare for potential matches- a *lossless* and *lossy* filter. We first apply the *lossless* filter, which removes PDGs from consideration that are smaller than a specified size. [16] states that small matches (we use $< 10$ nodes) between methods are relatively common and often trivial, so no interesting matches are lost with this filter.

We then begin a pairwise comparison between the remaining methods in $A$ with methods in $B$. Before running the comparison algorithm, we apply the *lossy* filter. The *lossy* filter 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. This may discard potential real matches, hence *lossy* filter. The hypothesis test we use is the G-test, which is a log likelyhood ratio test [16]. Given two distributions of nodes, the test calculates how likely it is that the second distribution is an observation from then first. The result is a test statistic, which asymptotically conforms to a chi-squared distribution. Using the chi-squared distribution, we can calculate the significance of the test statistic. If the result is significant at a given level $\alpha$, then we excluded the pair. This constant $\alpha$ can be tuned in order to adjust the false positive/negative rate of filtered pairs.

If a potential method pair passes both filters, we then compare the two PDGs with subgraph isomorphism testing, attempting to find a mapping between nodes in $PDG_A$ and nodes in $PDG_B$. Subgraph isomorphism is in general NP-Complete, however, for the specialized use of comparing PDGs, it is often efficient. PDGs are generally small because they only represent a single method, which developers tend to keep to a maintainable size. Also, PDGs contain a variety of node types which restricts the total number of possible pairs of nodes for testing [16]. For the subgraph

isomorphism algorithm, we used the VF2 algorithm, which is a backtracking algorithm geared towards matching large graphs [7]. We add a 30 second timeout for each method comparison to help our analysis find as many matches as possible under time constraints. While 30 seconds may seem like a very small amount of time to give a single method comparison, note that two extremely small applications with 100 methods each could in theory take 83 hours if every comparison took 30 seconds. Clearly, aggressive filtering and search space pruning is essential to being able to analyze any nontrivial number of potential clone pairs. The result of running the VF2 algorithm is a mapping of nodes in $PDG_A$ to nodes in $PDG_B$, if one is found. We compute the amount of each PDG the match covered and store the match for further analysis.

Once all the PDG pairs have been compared, we analyze the resulting matches to determine the similarity of the applications. One metric we use is the total PDG node coverage for an application, which is computed by summing the number of nodes covered in the best match for every method in $A$ and dividing it by the total size of the PDGs in $A$.

$$cov_A = \frac{\sum_{x_i \in A} |y_j|}{\sum_{x_i \in A} |x_i|} \quad \text{where } y_j \text{ is the best mapping for } x_i \tag{1}$$

This percentage tells us what portion of $A$ is in common with $B$. Another metric we use is the distribution of PDG match percentages. This is calculated by finding the best match for each method in $A$ and determining the percentage of total nodes in that method the match covers. Methods with a high percent coverage are more likely to be near exact matches with little functional difference while low match percentage means there could be a significant functionality different between the two. We calculate these metrics from both $A$ and $B$'s perspective. This means the for every comparison of two applications, there will be two coverage values and two matching distributions.

### 5.3 Diff Generation

After all PDGs have been compared and we have generated a mapping between PDGs in $A$ and $B$, we determine the difference between the two applications, henceforth referred to as the "diff." We currently choose a fairly basic diff generation scheme but we discuss possible enhancements in Section 10.

For each PDG in $A$, we determine the best matching PDG in $B$ by the number of statements matched. If even the number of nodes in the best match is significantly smaller than the PDG in $A$, we consider it to be a new method and generate a signature for it. We use the alphabetically sorted set of API methods that are accessed directly, or indirectly through other application methods, by the method as its signature. The set of API methods called is indicative of the functionality of the new method and is sufficient for the clustering discussed below. We generate these diffs from both $A$ and $B$'s perspective, so there will be two new methods signature files per comparison.

#### 5.3.1 Diff Clustering

Because manually analyzing all the diffs generated by DNADroid would be very time consuming, we decided to automatically cluster them. This allows us to analyze a small set of diffs in each cluster and then extrapolate similarities between them to the cluster as a whole.

Before clustering diffs, we first filter any application pairs that do not have at least 10% coverage of one of the applications. We are primarily interested in investigating common diffs in potentially plagiarized applications, and applications with such a low coverage are most likely not clones. This filter is acceptable because we found experimentally that in most cases at least one application will have at least 50% coverage, even when entire advertising libraries have been replaced. A plagiarizer may be able to add enough code to lower the clone's coverage, however, they cannot change the original which will still have a high coverage value- unless the plagiarizer heavily modifies the plagiarized code. As stated in Section 4, the amount of work to hide plagiarism from a PDG-based detection approach is almost as expensive as rewriting the application.

To compare the diffs, first we calculate the context triggered piecewise hash (CTPH)[4] on each set of API methods found in the diff. CTPHs, also known as fuzzy hashes, are used for detecting homologies between two sequences of bytes, and therefore can be used to detect similar sets of API methods. We then compare each hash found in $\text{diff}_1$ with each hash found in $\text{diff}_2$ and count the number that have significant overlap. The number matched and the number of methods in each diff allow us to calculate a percent dissimilarity between the two diffs, as a percentage. We calculate this dissimilarity value for each pair of diffs created by DNADroid, creating in a dissimilarity matrix.

Once we have generated the dissimilarity matrix, we use R [13] to create hierarchical clusters. Hierarchical clustering starts by assuming each diff is in it's own cluster and then iteratively combines clusters using an agglomeration method until there is a single cluster remaining. As it's merging, the algorithm keeps track of the merges it makes and this can be used to generate a graph, called a dendrogram, for visualizing the hierarchy. This behavior is appealing because we do not know the number of clusters to expect, so we were able to generate all possible and visualize them in a graph. After reviewing the graph, we select how many clusters to break the tree into and write the resulting clusters to file. The results for this clustering are presented in Section 7.

# 6 Case Studies

As several pairs of applications in our case studies share a name and/or package, we will refer to applications by the first 4 characters of their SHA1 hash. We give more detailed information about each application in the Appendix A.

## 6.1 "Benign" plagiarism

Several applications were reported by DNADroid as having both applications 100% covered by 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. These changes are interesting for two reasons. First, as part of Android design practices, all String are supposed to be externalized. String externalization separates code logic from String data and simplifies localization. Second, there seems to be no incentive for the plagiarism apart from providing an application to an otherwise excluded audience. For the latter reason, we believe these pairs to be cases of "benign" plagiarism, since there appears to be no benefit to the plagiarizer. In total we have 68 pairs that have 100% coverage, however, without manual review, we cannot confirm that they are all "benign". For example, the copied application may have a different client id, which changes which account gets revenue generated by clicks on advertisements.

[4] http://ssdeep.sourceforge.net

### 6.2 Changes to advertising libraries

Our next case study consists of two applications, dea9 and ff64, which have two different advertising libraries. These two applications are signed with different private keys, which leads us to believe that they're not from the same developer. They also have different client ids, but this is not surprising given that each advertiser has it's own format for client ids. Both applications have the INTERNET permission and ff64 additionally has the READ_PHONE_STATE, which allows access to device identifying information. The extra permission is most likely for the advertising library to access the device ID to uniquely identify the user and potentially track their clicks across applications. The number of classes is, 73 and 64, in dea9 and ff64 respectively. A total of 34 classes were filtered from analysis because they had identical SHA1 hashes. Of the remaining classes, DNADroid reported that dea9 has 54.40% coverage and ff64 has 44.74%.

When we compared these two applications, we noticed that dea9 has Youmi[5] advertising classes, while ff64 has WooBoo[6] advertising classes. Apart from these advertising libraries, the application code was practically identical. Of the 42 application classes, 34 were in the excluded set, with the remaining 8 having only minor changes such as different resource ids and advertising layout parameters. Since we don't have any additional information about the applications, it's impossible to tell which application is the original, and which is the clone. This case study shows that using coverage alone is an imperfect measurement for determining how similar two applications are because simply replacing a common library can decrease the coverage value substantially. It also show that coverage is a lower bound since the actual overlap may be great if excluded classes were taken into account.

In Section 7.2, we discuss the results of our diff clustering which discovered that this sort of plagiarism occurs frequently in the pairs that we tested.

### 6.3 Malware added to a application

"HippoSMS" (5c70) is a malicious application recently discovered by [15] which we downloaded and compared to our collection of applications. We found that it shares the same package name as d3ea, a Chinese video player, which we crawled from GoApk. Both applications require a surprising number of sensitive permissions; d3ea requires 11 permissions and 5c70 requires 10. According to STOWAWAY[10], a tool for detecting over privileged applications, d3ea requires 6 permissions that it doesn't use, where as 5c70, the malicious application, only requires 1 extra. 5c70 has 150 classes and d3ea has 142. DNADroid reports 90.45% coverage of 5c70 and 97.62% coverage of d3ea, with 107 classes excluded from analysis.

According to [15] and through our manual inspection of the two application, we have identified the malicious component of the application. The malicious component sends SMS messages to fixed numbers that cost the phone owner money. It also attempts to hide its tracks by deleting any messages from the users SMS history that appear to have come from a number that costs the owner money.

Although we cannot be certain, given the number of permissions d3ea requires, it's possible that the developer of d3ea intended to insert malware into the application at a later time, or that d3ea is a clone itself.

[5] http://youmi.net

[6] http://wooboo.com.cn

### 6.4 Two variants of the same malware

Our final study consists of two malicious applications, adc8 and 6fed, both of which are identified by VirusTotal[7] as being variants of "BaseBridge" family of malware. Both applications require numerous dangerous permissions, such as SEND_SMS and WRITE_CONTACTS. We list their complete permissions in the Appendix A. efed requires extra permissions RESTART_PACKAGES and WRITE_APN_SETTINGS. 6fed has 61 Java classes and adc8 has 39, between these, none are identical at the SHA1 level. Our coverage results are 37.48% and 30.98% for adc8 and efed, respectively.

Both applications have been stripped of meaningful class and method names. Classes that mapped to one letter in efed now map to a different letter in adc8. Automatic obfuscation tools, like ProGuard[8], make this process very simple.

"BaseBridge" has many similarities to "HippoSMS". "BaseBridge" also has functionality for sending text messages to premium numbers and hiding it's tracks from the user by deleting messages. In addition, it has a command and control structure that can instruct the application to send a text message or call a phone [9].

When these files are compared with Ssdeep, it finds a 30% similarity. Although DNADroid reports similar similarity values, 37% and 31%, DNADroid reports much more useful information, such as what methods are "new" in each application and which are similar. This case study demonstrates DNADroid's ability to detect code reuse, even with heavy modifications.

## 7 Evaluation

Our clone finding criteria produced 5074 potential clone pairs from our dataset of 29,115 applications. The analysis took close to 168 hours to run on a server-class machine. In order to ensure that our analysis finished, we set a time limit of 120 minutes on each pair comparison, which is well above the average completion time of 70 minutes. On average, the applications we tested had 97.84 classes and 541.02 total methods. We did not whitelist any classes from our analysis, such as advertising libraries, although our SHA1-based Java class filter often excluded library classes. Of the 5074 pairs, we had 1636 results where both coverage values were percentages. In a few cases, usually when the applications were practically identical, we had our filters completely eliminate the search space in one of both of the applications. This occurred in 17 and 227 comparisons respectively. We had a number of timeouts (3194), even with the 120 minute timeout, which may indicate that our criteria of same name or package but different certificate fingerprints was too broad or that we need more aggressive filtering. While this may be a large portion of the total tested, we note that we did still get a significant number of results with which to test the rest of our methodology and leave tuning the analysis to future work. We discuss several potential performance improvements in Section 10.

We found that on average 254.68 methods matched between applications, which is 55.17% of the average number of methods in an application. Table 1 presents the distribution of coverages we discovered among comparisons that completed. The first column shows the number of pairs where both coverage values were within the given range, and identifies comparisons with both applications having a similarly sized coverage. Columns two and three show the distribution of the best and

[7] http://virustotal.com

[8] http://proguard.sourceforge.net

| Coverage Value(%) | $c1 \wedge c2$ | $\max(c1, c2)$ | $\min(c1, c2)$ | $\lvert c1 - c2 \rvert$ |
|---|---|---|---|---|
| 100 | 68 | 122 | 68 | 0 |
| $75 \leq X \leq 99$ | 155 | 319 | 194 | 52 |
| $50 \leq X \leq 74$ | 114 | 266 | 190 | 58 |
| $25 \leq X \leq 49$ | 71 | 165 | 214 | 130 |
| $1 \leq X \leq 24$ | 427 | 614 | 625 | 1068 |
| 0 | 66 | 66 | 66 | 137 |

Table 1: Coverage is the percentage matched for each application in the comparison. Column one shows when both coverage are in a given range. Columns two and three show when the best and worst, respectively, are in a given range. Finally, column four shows how similar the two coverage values are.

worst coverage values in each result. The max column shows how many applications are a subset of the other and the min column shows the size of the smallest set of methods shared between the two applications. Finally, the last column shows the distribution of the difference between the two coverages which shows how often coverage values are similar between the two applications.

### 7.1 Filter Performance

Filtering is crucial in order to analyze applications in a reasonable amount of time. Given that there were on average 541.02 methods in an application, there are thus $541.02^2$ possible pairs for comparison. Comparing two methods is essentially the subgraph isomorphism problem, which is NP-Complete, so each comparison can be very costly. We limit the number of comparisons we must do with three filters: class exclusion at a SHA1 level, the *lossless* filter for small PDGs and the *lossy* filter which tests that methods are composed of similar node types. The average number of identical classes that are excluded from analysis is 43.99, or 34.21% of the average 97.84 classes in an application. This lends credence to the effectiveness of our clone finding criteria. As every excluded class contains on average 5.23, this is a substantial reduction in the number of comparisons. The *lossless* filter on average excludes 40.88% of the methods in an application with the *lossy* filter reducing an additional 8.34%. The combination of these filters reduced the average number of matches tested from an unfiltered search space size of 431704.67 comparisons to a much more manageable 17852.98, a 87.58% reduction.

### 7.2 Diff Clusters

As discussed in Section 5.3.1, we used R to create hierarchical clusters of the diffs generated by DNADroid. After reviewing the hierarchical clusters graph, we decided to split the diffs into 30 clusters. We then manually reviewed a random subset of 10 diffs in each cluster to determine what is similar between the diffs we reviewed. Based on this manual review, we labeled each cluster and present the results in Table 2. Most labels are the name of the advertising library that was introduced in each of the diffs in the cluster. Clusters labeled "Unknown" are clusters for which we could not determine the common changes to. Cluster 17, which is labelled "Amazon" consists of diffs generated when applications were compared with those in the Amazon Android Market; all applications on the Amazon market contain Amazon DRM libraries. Clusters labelled "Garbage"

| Cluster # | Size | Label |
|---|---|---|
| 1 | 25 | Youmi/Admob/InnerActive |
| 2 | 537 | Unknown |
| 3 | 489 | Unknown |
| 4 | 20 | Mobclick |
| 5 | 96 | WooBoo |
| 6 | 10 | Garbage |
| 7 | 26 | InnerActive |
| 8 | 17 | Casee |
| 9 | 51 | Google Ads |
| 10 | 49 | WooBoo |
| 11 | 43 | Youmi/Admob/InnerActive |
| 12 | 57 | Wooboo |
| 13 | 33 | Youmi |
| 14 | 21 | InnerActive/Youmi |
| 15 | 14 | Google Ads |

| Cluster # | Size | Label |
|---|---|---|
| 16 | 33 | Unknown |
| 17 | 7 | Amazon |
| 18 | 36 | Google Ads/Airpush |
| 19 | 42 | Google Ads |
| 20 | 24 | Garbage |
| 21 | 55 | Google Ads |
| 22 | 28 | Wiyun |
| 23 | 20 | Admob |
| 24 | 28 | Youmi |
| 25 | 30 | Unknown |
| 26 | 23 | Unknown |
| 27 | 51 | Admob |
| 28 | 19 | Sosceo/Madhouse |
| 29 | 23 | Unknown |
| 30 | 14 | Google Ads |

Table 2: Diff Clustering labelling

consist of diffs that are not useful, for example, diffs where the difference is a call to an Object constructor.

Interestingly, the majority of our clusters are centered around new advertising libraries added into applications. Of the 30 clusters, only 8 were not related to advertising libraries. In some cases, such as clusters 1 and 11, we found that there were changes to a combination of advertising libraries.

Another interesting result of our labeling is that there are often multiple clusters for a single advertising library. This could be because we selected too many clusters but it could also be because of multiple versions of these libraries. There are also multiple scenarios for the change in advertising libraries. First, the application may have an older version of the advertising and the possible clone replaces it with a newer version, or vice versa. Second, the application may not have the library at all and the possible clone has inserted it. These different versions and scenarios in which libraries create many different possible diffs that could have caused the repetition of advertising libraries in our clustering.

There were 2 clusters, clusters 2 and 3, that were an order of magnitude larger than the other clusters. We tried several things to break up these clusters, first, we increased the number of clusters but this only lead to the other clusters being further split with no significant change to these clusters. Next we tried different agglomeration methods, which decides how to merge clusters at each iterative step of the hierarchical clustering algorithm. We originally chose the Wards method for agglomeration, because its minimum variance method generates compact, spherical clusters which are easier to analyze. When we tried the other methods included with R, the number of clusters exploded beyond which we could manually analyze. In Section 10, we will describe our future plans to improve our clustering results.

### 7.3 Discussion

During our case studies and diff clustering, we did a lot of manual analysis of applications and the diffs that DNADroid generated. This section discusses some of the problems encountered which may have effected our evaluation.

Determining the original given a pair of application non-trivial in many cases. Given that most of our applications were crawled in a relatively short amount of time, we do not have time series data from which to draw information about which application came first. Furthermore, we found that some developers possibly change their signing keys between versions of an application, making it difficult in some cases to say whether an application is even from the same developer. We hypothesis that the Client Id, which is used to link the developers account with a advertising request, may be less volatile between application version since it is not generated by the developer and could be better for attributing an application to a developer. However, this too has challenges because each advertising library may have a different mechanism for specifying the Client Id, some of which may be difficult to determine statically.

When comparing two applications, it's important to compare the same versions of the application so that developer contributions do not get mixed in with plagiarizer contributions. Unfortunately, often times we did not have the exact same versions of the application and thus were forced to compare with an earlier or later version. In these cases, some of the changes are obviously developer contributions because they improve the functionality of the application, however, given the complexity of applications, we could often not tell without extensive manual review. In order to aid us in the future, we plan to crawl the markets for a much longer period of time.

## 8 Misc Findings

This sections describes a few interesting things that we found during our analysis.

**SmaliHook.class** - We uncovered this Java class in one of the applications we were manually reviewing. This class is designed to help disable license checks in Android applications and is part of the AntiLVL cracking tool[9]. The class includes methods to spoof the device id, make fake license checks which always return true, and hide modifications from the application itself by returning the expected file size, md5, signatures for the original application. We found 79 APKs containing this class. Given the nature of AntiLVL, it's almost certain that these applications are all clones.

**Created-By: 1.1.2 (AntiLVL)** - When we were investigating an application with a SmaliHook.class, we looked in the CERT.SF, which is the signature file listing the digital signatures of every file in the application, and we discovered that in the header it displays what created the signature file. In this case, it appears that the file has been created by AntiLVL, which as described in the previous section is an application cracking tool. Since AntiLVL modifies files in the application, it must resign the application itself which means that all the signatures need to be rewritten. Because there is no certificate chain to application signing, this modification will not be detected by an installer's phone. This header was included in 54 applications that are most likely all clones.

In total we found 83 unique applications containing SmaliHook.class or the AntiLVL header.

[9] http://androidcracking.blogspot.com/p/antilvl.html

# 9 Limitations

This section describes limitations of our approach, possible evasion techniques for plagiarists and potential countermeasures to them.

**Evasion techniques** A plagiarist may exploit our filtering to influence our similarity analysis. For example, by splitting all methods into as few operations as possible, a plagiarist may be able to cause our lossless filter to filter these methods, missing the plagiarism. Alternatively, plagiarists may insert lots of dead code into each method to trick our lossy filter into excluding the method from being checked against the original method because the distribution of nodes is dissimilar.

**Possible solutions** The simplest solution to preventing these evasion techniques would be to disable filtering. However, this would make our analysis take exponentially longer and is thus unacceptable. A better solution would be to use more tamper proof filters. For example, our lossy filter that compares node distribution similarity could additionally use information from the control flow graph to detect dead code or statements whose effects do not impact the application state or method return value.

# 10 Future Work

Clone detection is a computationally expensive process, and given the number of potential clones we want to test, improving the performance of clone detection is important. In particular, we could improve our performance by refining our filters and creating new criteria to filter PDGs that could minimize the search space while excluding as few potential matching methods as possible. Additionally, we may consider using a suboptimal subgraph isomorphism algorithm which should be of polynomial complexity, since VF2 seems to be the most efficient optimal algorithm [11].

Our work in diff generation and clustering is still incomplete and we plan to pursue it as a major thrust of our future work. The diff generation is not very robust and this could potentially be exploited by a frequent plagiarist. For example, a plagiarist may increase the number of methods potentially callable from each method randomly which would prevent us from recognizing common diffs. The clustering produces mixed results, on the one hand, we were able to successfully label a large number of our diffs using it, on the other hand, there were many clusters that had the same label and two-thirds of our diffs were grouped into just two clusters. This could be improved by improving our diff generation mechanism, however, we will likely have to investigate other clustering strategies.

We were able to extract the ad Client Id's for many applications, identifying the user receiving ad revenue for an application. Though it is possible for one plagiarizer to have multiple accounts with a given advertiser, we found several cases in which the plagiarist used the same Client Id across multiple cloned apps. We would like to compare Client Id's across many stolen applications to determine if most plagiarisms are the result of several main parties. On a similar note, we would like to gain insight into the amount of revenue gained by plagiarizers by analyzing the number of clones they created and their respective popularities. Finally, we would like to determine if we can identify cloned applications by the same plagiarizer using a combination of ad Client Id, the certificate of the private key used to sign the applications and a common diff structure between her clones.

## 11 Related Work

**PL Work in Plagiarism Detection** JPlag [17] is a structure based plagiarism detection method. First it converts each program into a string of canonical tokens. Programs are then checked for plagiarism by attempting to cover one program's token string by substrings taken from the other. Due to it's reliance on the structure of the programs, JPLAG is vulnerable to statement reordering, insertion or deletion.

[19], the ideas behind MOSS [2], compares documents by dividing them into *k-grams*- contiguous substrings of length $k$. Each $k$-gram is then hashed and a subset of those hashes becomes the document's fingerprint. If two documents share one or more fingerprints then they must also share a $k$-gram, indicating potential plagiarism. As only a subset of the $k$-grams are used in a document's fingerprint, plagiarisms residing in unused segments may not be found. Being structure based makes this approach vulnerable to the same evasion techniques as JPlag.

GPLAG [16] proposed detecting plagiarism by analyzing PDGs which results in detection that is robust to statement reordering, insertion or deletion. As the structure of methods is analyzed rather than the contents, GPLAG is also unaffected by class, method or variable renaming. We leverage the idea of analyzing PDGs to gain similar robustness in our analysis and also adopt *lossy* and *lossless* filters to scale to large programs.

**Android Plagiarism Detection** Androguard [3] is a tool that can aid in the reverse engineering of Android applications, obfuscate applications to help protect developers from plagiarism and can determine the similarities between applications. Based on the documentation and a review of the source code, Androguard supports several similarity metrics including the normal compression distance (NCD). The entropy of an object may also be measured. Additionally Androguard compares the SHA256 hash of methods and basic blocks. While these methods may find relatively simple plagiarism, adding or changing statements will break the hash comparisons and moderate changes will likely hide plagiarism from the similarity metrics.

DEXCD [8] detects Android clones by comparing similarities in pcodes in Android DEX class files. First, DEXCD tokenizes the instructions in each application. After finding a matching pair of two distinct pcode instructions, the algorithm examines all pairings of subsequent pcode instructions seeking a further match. As DEXCD relies on matching sequences of pcodes it is highly vulnerable to code transformations that change the order of operations or add or remove instructions. We were unfortunately unable to compare our results with DEXCD due to difficulties in getting DEXCD to run.

## 12 Conclusion

To combat the plagiarism of Android applications, markets need robust techniques to filter out these clones to protect developers and users. Application clones may harm developers by diverting ad revenue or reducing sales of non-free applications. Users of cloned applications may be unknowingly allowing trojans access to sensitive information stored on the phone. DNADroid is designed to find clones on a large scale. It achieves this by first selecting likely clones using a criteria and then analyzes the pairs to determine what methods, if any, have been plagiarized. To test DNADroid we applied it to 29,000 applications, which we have crawled from various Android markets. From these results we also describe four case studies which provide more insight into the motivations between the plagiarism.

## References


[1] Signing your applications. http://d.android.com/guide/publishing/app-signing.html.

[2] Alex Aiken. Moss (measure of software similarity). https://theory.stanford.edu/ aiken/moss/.

[3] Androguard. Androguard: Manipulation and protection of android apps and more... http://code.google.com/p/androguard/.

[4] AppBrain. Number of available android applications. Accessed August 15st, 2011. http://www.appbrain.com/stats/number-of-android-apps.

[5] BajaBob. Smalihook.java found on my hacked application. Accessed August 11st, 2011. http://stackoverflow.com/questions/5600143/android-game-keeps-getting-hacked.

[6] Scott Beard. Market shocker! iron soldiers xda beta published by alleged thief. Accessed August 1st, 2011. http://androidheadlines.com/2011/01/market-shocker-iron-soldiers-xda-beta-published-by-alleged-thief.html.

[7] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[8] Ian Davis. Dexcd. http://www.swag.uwaterloo.ca/dexcd/index.html.

[9] Stephen Doherty and Piotr Krysiuk. Android.basebridge. Accessed August 15st, 2011. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99.

[10] Adrienne Porter Felt, Ericka Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *To Appear in CCS*, 2011. http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-48.pdf.

[11] P. Foggia, C. Sansone, and M. Vento. A performance comparison of five algorithms for graph isomorphism.

[12] Jesus Freke. smali: An assembler/disassembler for android's dex format. https://code.google.com/p/smali/.

[13] Robert Gentleman and Ross Ihaka. The r project for statistical computing. http://www.r-project.org/.

[14] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Detecting privacy leaks in android applications. Accessed August 17st, 2011. http://www.cs.ucdavis.edu/research/tech-reports/2011/CSE-2011-10.pdf.

[15] Xuxian Jiang. Security alert: New android malware – hipposms – found in alternative android markets. Accessed August 14st, 2011. http://www.cs.ncsu.edu/faculty/jiang/HippoSMS/.

[16] C. Liu, C. Chen, J. Han, and P.S. Yu. 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*, pages 872–881. ACM, 2006.

[17] L. Prechelt, G. Malpohl, and M. Philippsen. Jplag: Finding plagiarisms among a set of programs. 2000.

[18] pxb1988. dex2jar: A tool for converting android's .dex format to java's .class format. https://code.google.com/p/dex2jar/.

[19] S. Schleimer, D.S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003.

# A Case Studies - Details

Permission Abbreviations:

| Permission | Abbreviation |
|---|---|
| ACCESS_NETWORK_STATE | ANS |
| CALL_PHONE | CP |
| DISABLE_KEYGUARD | DK |
| INSTALL_SHORTCUT | IS |
| INTERNET | I |
| MOUNT_UNMOUNT_FILESYSTEMS | MUF |
| READ_CONTACTS | RC |
| READ_PHONE_STATE | RPS |
| READ_SMS | RES |
| RECEIVE_BOOT_COMPLETED | RBC |
| RECEIVE_SMS | RCS |
| RESTART_PACKAGES | RP |
| SEND_SMS | SS |
| VIBRATE | V |
| WAKE_APN_SETTINGS | WAS |
| WAKE_LOCK | WL |
| WRITE_CONTACTS | WC |
| WRITE_EXTERNAL_STORAGE | WES |
| WRITE_SMS | WS |

A † next to a name indicates that the name has been translated.
**Case Study 2 (Section 6.2)**
dea93f55494cdb70ee0bbc63de2443d18204fc6f — ff645369bfae69f3235d2f1e236e6173aeb5725b Same:

- Package: com.cnllk
- Name: Traditional Lianliankan†
- Version: 1/1.0

| SHA1 | permissions |
|---|---|
| dea9 | I |
| ff64 | I,RPS |

**Case Study 3 (Section 6.3)**
d3eac47c7044a380f9d5770a080c86b67eb6ba9b — 6fedbfda043d6937a1769ad299e870224828b90b
Same:

- Package: com.ku6.android.videobrowser
- Name: Cool 6 Video†

| SHA1 | version | permissions |
|---|---|---|
| d3ea | 20/2.0.3 | ANS,I,IS,MUF,RPS,RES,RBC,RCS,SS,WES,WS |
| 5c70 | 20/2.0.0 | ANS,I,IS,MUF,RES,RBC,RCS,SS,WES,WS |

**Case Study 4 (Section 6.4)**
adc8ac7b72a1055438773e7f075028df681d987c — 6fedbfda043d6937a1769ad299e870224828b90b
Same:

- Package: com.android.battery
- Name: Android.Basebridge

| SHA1 | version | permissions |
|---|---|---|
| adc8 | 18/5.0.1 | ANS,CP,DK,I,RC,RPS,RES,RBC,RCS,SS,V,WL,WC,WS |
| 6fed | 24/5.1.3 | ANS,CP,DK,I,RC,RPS,RES,RBC,RCS,RP,SS,V,WL,WAS,WC,WS |