AnDarwin: Scalable Detection of Semantically Similar Android Applications

Jonathan Crussell, Clint Gibler, and Hao Chen

University of California, Davis
{jcrussell, cdgibler, chen}@ucdavis.edu

Abstract. The popularity and utility of smartphones rely on their vibrant application markets; however, plagiarism threatens the long-term health of these markets. We present a scalable approach to detecting similar Android apps based on their semantic information. We implement our approach in a tool called AnDarwin and evaluate it on 265,359 apps collected from 17 markets including Google Play and numerous third-party markets. In contrast to earlier approaches, AnDarwin has four advantages: it avoids comparing apps pairwise, thus greatly improving its scalability; it analyzes only the app code and does not rely on other information — such as the app's market, signature, or description — thus greatly increasing its reliability; it can detect both full and partial app similarity; and it can automatically detect library code and remove it from the similarity analysis. We present two use cases for AnDarwin: finding similar apps by different developers ("clones") and similar apps from the same developer ("rebranded"). In ten hours, AnDarwin detected at least 4,295 apps that have been the victims of cloning and 36,106 apps that are rebranded. By analyzing the clusters found by AnDarwin, we found 88 new variants of malware and identified 169 malicious apps based on differences in the requested permissions. Our evaluation demonstrates AnDarwin's ability to accurately detect similar apps on a large scale.

1 Introduction

As of March 2012, Android has a majority smart phone marketshare in the United States [15]. The Android operating system provides the core smart phone experience, but much of the user experience relies on third-party apps. To this end, Android has an official market and numerous third-party markets where users can download apps for social networking, games, and more. In order to incentivize developers to continue creating apps, it is important to maintain a healthy market ecosystem.

One important aspect of a healthy market ecosystem is that developers are financially compensated for their work. Developers can charge directly for their apps, but many choose instead to offer free apps that are ad-supported or contain in-app billing for additional content. There are several ways developers may lose potential revenue: a paid app may be "cracked" and released for free or a free app may be copied, or "cloned", and re-released with changes to the ad libraries that cause ad revenue to go to the plagiarist [21]. App cloning has been widely reported by developers, smart phone security companies and the academic community [8, 10, 11, 16, 22, 34, 33]. Unfortunately, the openness of Android markets and the ease of repackaging apps contribute to the ability of plagiarists to clone apps and resubmit them to markets.

Another aspect of a healthy market ecosystem is the absence of low-quality spam apps which may pollute search results, detracting from hard-working developers. Of the 569,000 apps available on the official Android market, 23% are low-quality [7]. Oftentimes, spammers will submit the same app with minor changes as many different apps using one or more developer accounts.

To improve the health of the market ecosystem, a scalable approach is needed to detect similar app for use in finding clones and potential spam. As of November, 2012, there are over 569,000 Android apps on the official Android market. Including third-party markets and allowing for future growth, there are too many apps to be analyzed using existing tools.

To this end, we develop an approach for detecting similar apps on a unprecedented scale and implement it in a tool called AnDarwin. Unlike previous approaches that compare apps pair-wise, our approach uses multiple clusterings to handle large numbers of apps efficiently. Our efficiency allows us to avoid the need to pre-select potentially similar apps based on their market, name, or description, thus greatly increasing the detection reliability. Additionally, we can use the app clusters produced by AnDarwin to detect when apps have had similar code injected (e.g. the insertion of malware). We investigate two applications of AnDarwin: finding similar apps by different developers (cloned apps) and groups of apps by the same developer with high code reuse (rebranded apps). We demonstrate the utility of AnDarwin, including the detection of new variants of known malware and the detection of new malware.

2 Background

2.1 Android

Android users have access to many markets where they can download apps such as the official Android market – Google Play [2], and other, third-party markets such as GoApk [1] and SlideME [3].

Developers must sign an app with their developer key before uploading it to a market. Most markets are designed to self-regulate through ratings and have no vetting process which has allowed numerous malicious apps onto the markets [35]. Google Play has developed a Bouncer service [27] to automatically analyze new apps. However, its effectiveness for finding similar apps, such as spam and clones, which may not be malicious, has not been studied.

2.2 Program Dependence Graphs

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.

2.3 Code Clones and Reuse Detection

Many approaches have been developed over the years to detect code clones [20, 23, 25, 26]. A code clone is two or more segments of code that have the same semantics but come from different sources. Finding and eliminating code clones has many software engineering benefits such as increasing maintainability and improving security, as vulnerabilities in clones only need to be found and patched once. Plagiarism and code clone detection share the same common goal: detecting reused code. However, code clone detection is largely focused on intra-app reuse, while plagiarism detection focuses on inter-app reuse, where the apps have separate code bases and have been identified as having different authors.

Tools that detect code clones generally fall into one of four categories: string-based, token-based, tree-based and semantics-based with semantics-based detection being potentially the most robust and often the most time consuming. Early approaches considered code as a collection of strings, usually based on lines, and reported code clones based on identical lines [9]. More recently, DECKARD [23] and its successor [20] use the abstract syntax tree of a code base to create vectors which are then clustered to find similar subtrees.

3 Threat Model

Our goal is to find Android apps that share a nontrivial amount of code, published by either the same or different developers. We determine similarity based on code alone and do not use meta data such as market, developer, package or description for any purpose other than analyzing the results of AnDarwin's clusters of similar apps. We consider only similarities between the DEX code of apps. We choose to leave native code to future work as only a small percentage (7%) of the 265,359 apps we analyzed include native code.

4 Methodology

AnDarwin consists of four stages as depicted in Figure 1. First, it represents each app as a set of vectors computed over the app's Program Dependence Graphs (Section 4.1). Second, it finds similar code segments by clustering all the vectors of all apps (Section 4.2). Third, it eliminates library code based on the frequency of the clusters (Section 4.3). Finally, it detects apps that are similar, considering both full and partial app similarity (Section 4.4).

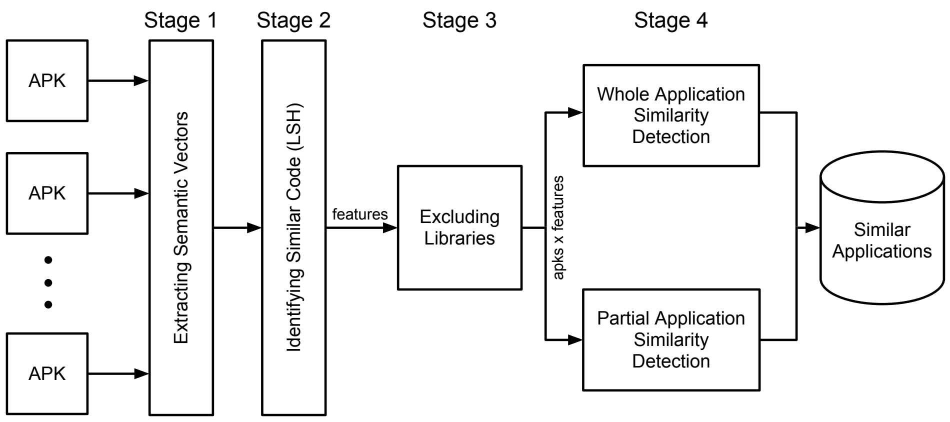

Stage 1
Stage 2
Stage 3
Stage 4
APK
APK
APK
Extracting Semantic Vectors
Identifying Similar Code (LSH)
features
Excluding Libraries
apks x features
Whole Application Similarity Detection
Partial Application Similarity Detection
Similar Applications

Fig. 1: Overview of AnDarwin.

We base the first two stages of AnDarwin on the approaches of Jiang et al. [23] and Gabel et al. [20] to find code clones in a scalable manner. AnDarwin uses these results to detect library code and, ultimately, to detect similar apps.

4.1 Extracting Semantic Vectors

The first stage of AnDarwin represents each app as a set of semantic vectors as follows. First, AnDarwin computes an undirected PDG of each method in the app using only data dependencies for the edges (as control dependencies edges may be easier to modify). Each PDG is then split into connected components as multiple data-independent computations may occur within the same method. We call these connected components *semantic blocks* since each captures a building block of the method and represents semantic information stored in the PDG. Finally, AnDarwin computes a *semantic vector* to represent each semantic block. Each node in the semantic block represents a statement in the method and has a type corresponding to that statement. For example, a node representing an *add* might have the type *binary operation*. To capture this information, semantic vectors are calculated by counting the frequency of nodes of each type in the semantic block. Continuing the above example, a semantic block with just x adds would have an x in the dimension corresponding to binary operations. AnDarwin uses a total of 20 node types, however, we could easily use more node information such as which binary operation is being performed to increase the precision of our vectors without dramatically increasing the complexity (Section 4.5). Semantic blocks with fewer than 10 nodes are discarded because they usually represent trivial and uncharacteristic code.

4.2 Identifying Similar Code

When two semantic blocks are code clones, they share the majority of their nodes and, thus, their semantic vectors will be similar. Therefore, we can identify code clones by finding near-neighbors of semantic vectors. While not all near-neighbors will be code clones, this technique works well in practice (Section 5).

To determine all the near-neighbors, we could attempt to compute similarity pairwise between all the semantic vectors. However, this approach is quadratic in the number of vectors which is computationally prohibitive given that there can easily be millions of vectors. Instead, we leverage Locality Sensitive Hashing (LSH), which is an algorithm to efficiently find approximate near-neighbors in a large number of vectors [5]. LSH achieves this by hashing vectors using many hash functions from a special family that have a high probability of collision if the vectors are similar. To identify near-neighbors, LSH first hashes all the vectors with the special hash functions and then looks for near-neighbors among the hash collisions. This allows LSH to identify approximate clusters of similar vectors (code clones) which AnDarwin will use to detect similar apps.

Since semantic blocks of vastly different sizes are unlikely to be code clones, we can improve the scalability further by grouping the vectors based on their magnitudes [23]. To ensure that code clones near the group boundaries are not missed, we compute groups such that they overlap slightly. LSH can then cluster each group quickly as each individual group is much smaller than the set of all vectors. Moreover, each LSH computation is independent which allows all the groups to be run in parallel. This also has the added benefit that we can tailor the clustering radius for each group to the magnitude of the vectors within the group — potentially allowing us to detect more code clones.

4.3 Excluding Library Code

A library is a collection of code that is designed to be shared between many apps. In Android, libraries are embedded in apps which makes it difficult to distinguish app code from library code. This is problematic because app similarity detection tools should not consider library code when analyzing apps for similarity. Prior approaches [16, 33] identified libraries using white lists and manual efforts; however, these approaches are inherently not scalable and prone to omission. In contrast, AnDarwin automatically detects libraries by leveraging the results of its clustering of similar code (Section 4.2).

A library consists of many semantic blocks which are mapped to semantic vectors by AnDarwin. When an app includes a library it inherits all the semantic vectors derived from library code. Therefore, when the semantic vectors are clustered and AnDarwin maps features to apps, features from library code will appear in many more apps. This is also the case for boilerplate code and any common compiler constructs which tend to occur in many apps. To exclude these uncharacteristic features, AnDarwin ignores any feature that appears in more than a threshold number of apps.

4.4 Detecting Similar Apps

The previous sections describe how AnDarwin creates features by clustering semantic vectors and how characteristic features are selected. AnDarwin determines app similarity based on these characteristic features using two approaches, one for full app similarity and the other for partial app similarity.

Full App Similarity Detection For full app similarity detection, AnDarwin represents each app as a set of features. In the simplest case, two very similar apps will have mostly or completely overlapping feature sets. Dissimilar apps' feature sets, on the other hand, should have little to no overlap. This is captured in the Jaccard Index of their two feature sets F_A and F_B, which reduces the problem of finding similar app to that of finding similar sets.

$$J(A, B) = \frac{|F_A \cap F_B|}{|F_A \cup F_B|} \tag{1}$$

Partial App Similarity Detection The above approach successfully finds apps that share most of their code but it is not robust enough to find clones that share only a part of their code. For example, consider an app and a copy of it that has added many methods and also removed many original methods to maintain a similar size. Although the app feature sets of these two apps agree on many features, their Jaccard Index may be low. To detect partial similarity, for each feature not excluded in the previous section, AnDarwin computes the set of apps that contain the feature. If two features have similar app sets, as determined by the Jaccard Index, these two features are shared by the same set of apps. If enough features share the same set of apps, AnDarwin has discovered a non-trivial amount of code sharing of non-library code. Therefore, by creating clusters of features based on their app sets, AnDarwin can detect partial app similarity by finding similar sets.

Finding Similar Sets Both full and partial app similarity detection require finding similar sets. As in Section 4.2, we could attempt to compute similarity pairwise between all the sets, however, this is again computationally prohibitive. Fortunately, this can be approximated efficiently using MinHash [12, 13].

MinHash was originally developed at Alta Vista to detect similar websites when represented as a set of features. To understand how MinHash works, first consider the binary matrix representation of the sets for full app similarity detection where columns are apps and rows are features. Let $h(A)$ be the MinHash of an app, A, and let it be defined as the first row of the matrix (going top-to-bottom) that is a one for the column corresponding to A. Then, if we were to create a random permutation of the rows of the binary matrix, for two apps, A and B, the probability that $h(A) = h(B)$ is the same as the Jaccard Index of the two app feature sets [30]. Rather than using just one permutation which may not find that two similar sets have the same MinHash value, many permutations and MinHash values can be calculated — creating a MinHash signature vector. These signature vectors are calculated for each app and can be clustered using

LSH (see Section 4.2). Therefore, MinHash allows AnDarwin to efficiently detect both full and partial app similarity.

The output of MinHash is a list of pairs of sets that are similar which we combine to create clusters of similar sets. To do so, we initialize a union-find data structure, which enables fast cluster merging and element lookup, with each set in a cluster by itself. We then process each pair, (X, Y) and merge the two clusters that contain X and Y if they are not already in the same cluster. By merging clusters in this way, the average similarity of sets within each cluster is decreasing with each pair processed. For example A may be similar to B, B to C, and C to D but this does not mean that A must be similar to D. We believe this is an acceptable trade off and leave alternative approaches to future work.

4.5 Time Complexity

In this section, we examine the total time complexity of AnDarwin. Let N be the number of apps analyzed. Then, the complexity of extracting semantic vectors is trivially $O(N * m)$, where m is the average number of methods per app (m is independent of N). The complexity of identifying similar code with LSH is: $O(d \sum_{g \in G} |g|^{\rho} \log |g|)$ [23]. Where d is the dimension of the semantic vectors (20), G is the set of vector groups, $|g|$ is the size of the vector group ($|g| <= N * m$) and $0 < \rho < 1$. This produces at most $O(N * m)$ clusters when there are no code clones at all. Finally, the complexity of MinHash is: $O(n \log n)$ where n is the number of sets. For full app similarity detection where there is one set per app, $n = N$, and for partial app similarity detection where there is one set per code clone, $n <= N * m$. Therefore, the total time complexity of AnDarwin is linearithmic, $O(N \log N)$, in the number of apps analyzed.

5 Evaluation

We have implemented our approach in a tool called AnDarwin. AnDarwin uses dex2jar [29] version 0.9.8 to convert DEX byte code to Java byte code. To build the PDGs required to represent apps as a set of semantic vectors, AnDarwin uses the T. J. Watson Libraries for Analysis (WALA) [14]. WALA supports building PDGs from Java byte code, eliminating the need for decompilation. Once AnDarwin has converted all the apps and represented them as sets of semantic vectors, AnDarwin uses the LSH code from [5] to cluster the semantic vectors to create features. These clustering results are then used to create the feature sets and app sets described in Section 4.4. Finally, to detect full and partial app similarity, AnDarwin uses MinHash, which we implemented based on [30].

We crawled 265,359 apps from 17 Android markets including the official market and numerous third-party markets (Table 1).

Market	Apps	Market	Apps	Market	Apps
Google Play	224,108	SlideME	16,479	m360	15,248
Brothersoft	14,749	Android Online	10,381	1Mobile	9,777
Gfan	7,229	Eoemarket	5,515	GoApk	3,243
Freeware Lovers	1,428	AndAppStore	1,301	SoftPortal	1,017
Androidsoft	613	AppChina	404	ProAndroid	370
AndroidDownloadz	245	PocketGear	227		

Table 1: Market origins of the apps analyzed by AnDarwin. Since some apps appear on multiple markets, the total apps in the table is slightly more than the total 265,359 apps analyzed.

5.1 Semantic Vectors

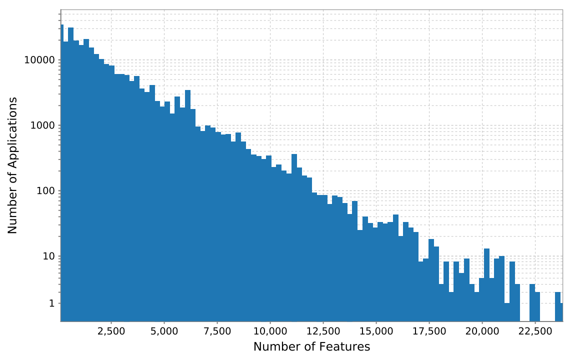
There are a total of 87,386,000 methods included in the 265,359 apps. These methods produced a total of 90,144,000 semantic vectors, meaning that on average a method has 1.03 connected components. Among the 90,144,000 semantic vectors, there are 4,825,000 distinct vectors. The average size of these 4,825,000 vectors is 77.87 nodes. The largest has 17,116 nodes. When we manually investigated the largest method, we found that the app builds a massive 5-dimensional array using hard coded values depending on different flags. Although perhaps not the best coding style, this large semantic vector does represent valid code that could be copied.

5.2 Code Features

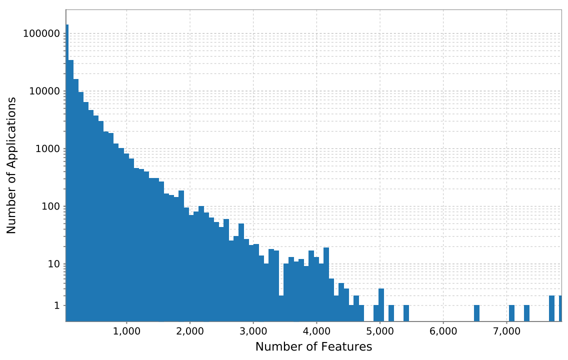
In total, AnDarwin found 87,386,000 methods included in the 265,359 apps that are clustered into 3,085,998 distinct features by LSH. 133,753 (4.3%) of these features are present in more than 250 apps and thus are not used in either full or partial app similarity detection. We selected this threshold based on the following insight: only features from library code tend to map to methods that share the same method signatures. Therefore, if the ratio of the number of apps a feature appears in to the number of distinct method signatures for that feature is large, it is highly likely that the feature represents library code. To select a library code threshold, we select a value and then count the number of excluded features for which this ratio is large and evaluate whether the threshold is acceptable. Using a ratio of four, we selected the threshold such that at least 50% of the excluded features exhibit this trait. We note that this threshold may be easily tweaked depending on false positive and false negative requirements.

5.3 App Complexity

Overall, AnDarwin found that a large number of apps are not very complex. Figure 2a shows the number of features per apps for the 265,359 apps before common feature exclusion. On average, apps have 2,045 features and the largest



(a) Before common feature exclusion



(b) After common feature exclusion

Fig. 2: Distribution of the number of features per app on logarithmic scale

app has 23,918 features. Once libraries are excluded, the number of apps with at least one feature drops to 231,184. Figure 2b shows that the average complexity drops dramatically once common features are excluded. The average number of features for these apps is 148, with the largest app having 7,908 features.

This is interesting from a software development point of view because it suggests that through libraries and good API design, most Android apps don't have to be very complex in order to perform their function.

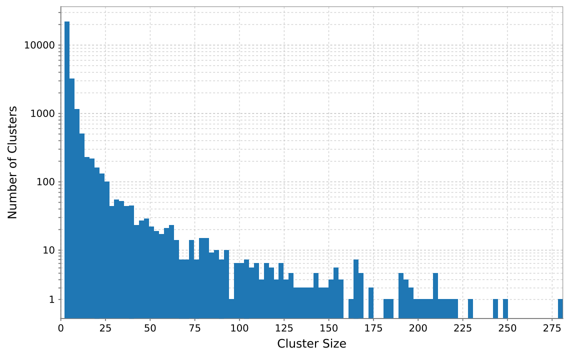
5.4 Full App Similarity Detection

Using full app similarity detection (Section 4.4), AnDarwin found 28,495 clusters consisting of a total of 150,846 distinct apps. Figure 3a shows the sizes of the clusters. As expected, the majority of clusters consist of just two apps. Surprisingly, some clusters are much larger, the largest of which consists of 281 apps. We will investigate these clusters in Section 6.2.

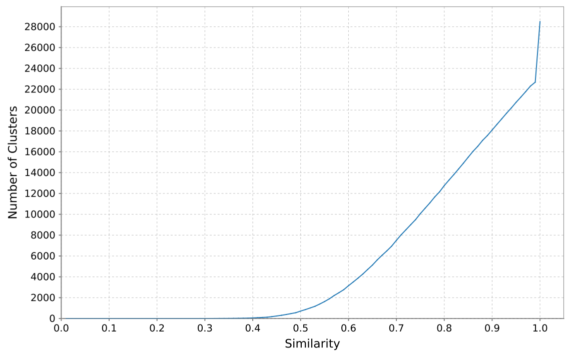
To evaluate the quality of the clusters, we compute intra-cluster app similarity based on the average Jaccard Index (Equation 1) between each pair of apps. For each cluster C, we compute the similarity score, $Sim(C)$, as:

$$Sim(C) = avg\{(A, B) \in C : J(A, B)\} \tag{2}$$

The similarity scores are between 0 and 1, where a score close to 1 indicates that all apps in the cluster have almost identical feature sets. Figure 3b shows the cumulative distribution of the similarity scores of the 28,495 clusters. It shows that almost no clusters have similarity scores below 0.5, and more than half of the clusters have similarity scores of over 0.80. This demonstrates the effectiveness of AnDarwin in clustering highly similar apps.



(a) Histogram of the cluster sizes on logarithmic scale



(b) Cumulative distribution function of $Sim(C)$

Fig. 3: Full App Similarity Detection.

5.5 Partial App Similarity Detection

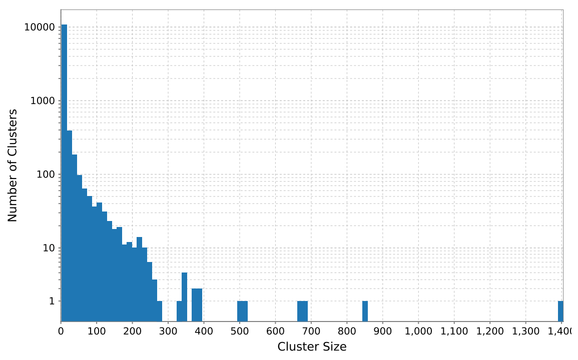
Using partial app similarity detection, AnDarwin found 11,848 clusters consisting of 88,464 distinct apps. Figures 4a and 4b show the sizes and similarity of these clusters, respectively. As partial app similarity is designed to detect app pairs that share only a portion of their code, we cannot measure them with Equation 1. Consider the scenario where an attacker copies an app but adds an arbitrarily large amount of code. In this case, Equation 1 will be small even though the original and clone share all of the original app's features. Therefore, for each cluster C, we compute the similarity score, $Sim_p(C)$, as:

$$Sim_p(C) = avg\{(A, B) \in C : \frac{|F_A \cap F_B|}{\min(|F_A|, |F_B|)}\} \tag{3}$$

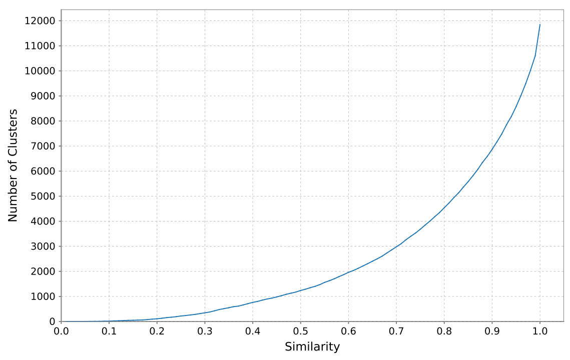
Figure 4b shows the cumulative distribution function of $Sim_p(C)$ for the partial app similarity detection clusters. Comparing Figure 3b to Figure 4b, we observe that some clusters based on partial app similarity have low intra-cluster similarity scores while almost no cluster based on full app similarity has similarity scores below 0.5. On the surface, this might suggest that partial app similarity produces lower quality clusters. However, this in fact shows the power of partial app similarity. When a cluster has a low similarity score, it indicates that the common features among the apps in this cluster are relatively small compared to the app sizes, so full app similarity detection cannot identify these common features.

5.6 Performance

We evaluated AnDarwin's performance on a server with quad Intel Xeon E7-4850 CPUs (80 logical cores with hyper threading) and 256GB DDR3 memory.



(a) Histogram of the cluster sizes on logarithmic scale



(b) Cumulative distribution function of $Sim_p(C)$

Fig. 4: Partial App Similarity Detection.

Using 75 threads, it took 4.47 days to extract semantic vectors (Stage 1) from all 265,359 apps (only 109 seconds per thread to process each app). We note that this stage only occurs once for each app, regardless of changes to subsequent stages and can be parallelized to any number of servers to reduce the total time.

The next most expensive stages are the LSH clustering in Stage 2 (Section 4.2) and the two MinHash-based clusterings in Stage 4 (Section 4.4). LSH clusters all 4,825,000 distinct vectors in just over 49 minutes. This time could be reduced to seven minutes if we were to run all the groups in parallel, rather than serially (as done in our current implementation). Full app similarity detection runs in just over 35 minutes. In total, it takes under ten hours to complete full app similarity detection including all the database operations and data transformations. On its own, partial app similarity detection took seven hours but this is expected as it clusters 2,952,245 sets whereas full app similarity detection only clusters 265,359. Interestingly, this time estimates how long it would take to run MinHash for full app similarity detection on 2,952,245 apps. Both MinHash times could be improved by using more than our single server.

5.7 Accuracy

Full App Similarity Detection To measure the false positive rate of AnDarwin's full app similarity detection, we leverage DNADroid [16], a tool that robustly compares Android apps pairwise for code reuse. DNADroid uses subgraph isomorphism to detect similarity between the PDGs of two apps. In the author's evaluation of DNADroid, it had an experimental false positive rate of 0%, making it an ideal tool for evaluating AnDarwin's accuracy.

Unfortunately, DNADroid is too computationally expensive to apply to all the pairs of apps AnDarwin found. Instead, we randomly selected 6,000 of the 28,495 clusters and then randomly selected one app from each cluster to compare against all other apps in the cluster. This resulted in a total of 25,434 pairs which it took DNADroid 83 hours to analyze.

DNADroid assigns each app in a pair a coverage value which indicates how much of the app's PDG nodes appear in the other app. To assess AnDarwin, we use the maximum of these two coverage values for each pair. DNADroid found that 96.28% of the clusters had 70% of the max coverage values over 50%(equal to the Jaccard Index used by AnDarwin) and 95.50% of the clusters had 90% of them over the threshold. Using the 70% criteria, this gives full app similarity detection a false positive rate of just 3.72% at the cluster level.

We do not attempt to measure the false negative rate of AnDarwin as there is no feasible way to find ground truth, e.g., all the similar apps in our collection of 265,359 apps.

Partial App Similarity Detection Unfortunately, DNADroid and its coverage values are inappropriate for evaluating the accuracy of partial app similarity detection. DNADroid considers apps as a whole and calculates similarity based on the matched portion to the size of the whole app. If DNADroid were used to verifying partial app similarity detection, we would incorrectly report a false positive in the case where two apps share a part of their code but not a significant (over the DNADroid coverage threshold of 50%) amount of their total code. Again, due to the lack of ground truth, we do not attempt to measure the false positive or false negative rate of partial app similarity detection.

6 Findings

6.1 Clone Victims

One use case of AnDarwin is finding clones on a large scale. Clones are different apps (not different versions of the same app) that are highly similar but have different owners. We determine ownership using two identifiers associated with each app we crawl: 1) the developer account name plus the market name and 2) the public key fingerprint of the private key that digitally signed the app. Assuming that a developer's account and her private key are not compromised, no two apps with different owners can share both of these identifiers. Therefore, we assume apps have different owners if they do not share either identifier.

Definitively counting the number of clones is non-trivial as it requires knowing which apps are the originals. Instead, we estimate the number of apps that are the victims of cloning. Each app belongs to at most one cluster and each app in a cluster is similar to at least one other app in the cluster. Therefore, each cluster is a family of similar apps which must have a victim app, the original app, even if we have not crawled the victim app. Then, the number of victims is at least equal to the number of clusters where there is more than one owner, as determined by the two identifiers above. Using just the full app similarity clusters, which were vetted in Section 5.7, AnDarwin found that at least 4,295 apps have been the victims of cloning.

6.2 Rebranded Apps

Using full app similarity detection, AnDarwin found 764 clusters containing more than 25 apps. Our investigation of these large clusters found a trend that some developers rebrand their apps to cater to different markets. The idea of rebranding is not a new concept – it has been widely used on the web (e.g. WordPress blogs). For example, one cluster consists of weather apps each targeting a different city. Similarly, we found clusters for news, trivia, books, radio stations, wallpapers, puzzles, product updates and even mobile banking apps. Some of these rebrandings are as trivial as just swapping the embedded images.

To estimate the number of rebranded apps, we use the owner identifiers described in Section 6.1 to map each app to an owner. If at least 25 apps in a cluster have the same owner, we consider those apps to be rebranded. Using this metric, 599 of the 764 clusters with at least 25 apps include rebranded apps. In total, we found 36,106 rebranded apps.

A surprising example of app rebranding is a cluster of mobile banking apps. This cluster contains 109 distinct apps that share a common package name prefix. Searching by this prefix, we found 175 apps on the Google Play Store, which includes 80 of the 109 apps present in our clusters. Interestingly, several of the apps were available on both 1Mobile and Play, and two of the apps are signed by a different key than the other 107 apps.

6.3 New Variants of Known Malware

Once malware has been discovered, it is important to use this knowledge to identify variants of the malware in an automated way. We hypothesize that by analyzing the clusters produced by AnDarwin containing known malware we may automatically discover new variants of those malware. Using the malware dataset from [35], we found 333 apps were clustered with known malware and were not included in the malware dataset.

We uploaded these 333 apps to VirusTotal [4], a website for running a suite of anti-virus software on files. It recognized 136 as malware, with 88 never having been uploaded to VirusTotal before. Among the 136 malware, approximately 20 are variants of the DroidKungFu family [24]. Approximately another 20 are identified as belonging to various malware families described in [35]. The remaining apps are identified as adware that contains either AirPush or AdWo. These advertising libraries show ads even when the app is not running [31] and have been known to have misleading ad campaigns [32]. These results demonstrate AnDarwin's utility for discovering new variants of malware.

6.4 New Malware Detection in Clones

Zhou et al. [35] found that 86.0% of their malware samples were repackaged versions of legitimate apps with additional malicious code, aiming to increase their chances of being installed by providing useful functionality. Since malware often requires many more permissions than regular apps, we hypothesize that we may detect new malware by searching for apps that require more permissions than the others in the same cluster. Intuitively, apps that are clustered together have similar code and for some to require more permissions is suspicious. To investigate this hypothesis, we searched for apps that require excessive permissions as follows (using clusters from both full and partial app similarity detection). First, for each cluster, we compute the union of the permissions required by all its apps. Then, we identify apps that require at least 85% of the permission union. Finally, if the apps identified in the previous step are fewer than 15% of the total apps in the cluster, we mark these apps as suspicious. Using this criterion, we found 608 suspicious apps. 16 of these apps overlap with the malware dataset from [35] and 1 overlaps with the previous section.

As before, we uploaded these apps to VirusTotal and it identified 243 as malware. Furthermore, 169 of these had never been seen before. This represents a lower bound on the actual number of malware in the suspicious apps as we did not investigate the suspicious apps for new malware which may not be identified by VirusTotal. The identified malware is from known families such as DroidKungFu [24], BaseBridge [19] and Geinimi [28]. By searching for apps with excessive permissions, AnDarwin identified known malware as suspicious without prior knowledge of their existence. This result demonstrates that AnDarwin is an effective tool for identifying suspicious apps for more detailed analysis.

7 Discussion

7.1 Adversarial Response

A specific use case of AnDarwin is to find plagiarized apps in a scalable manner. Based on our implementation details, plagiarists may attempt to evade detection using obfuscation. Some of these obfuscation techniques are effective against AnDarwin, however, they are difficult to perform automatically.

Futile Obfuscations AnDarwin is robust against all transformations that do not alter methods' PDGs, which is the basis for our similarity detection. This includes, but is not limited to, (1) syntactical changes such as renaming packages, classes, methods and variables, (2) refactoring changes such as combining or splitting classes and moving methods between classes, and (3) method restructuring such as splitting methods with multiple connected components into separate methods and reordering code segments within a method that are data and control independent.

AnDarwin is also robust against code addition. A plagiarist may add a few methods or a new library to their plagiarized app. Since the original and the plagiarized app still share a core of similar code, AnDarwin would still detect them using partial app similarity detection.

Potentially Effective Obfuscations AnDarwin is less robust against obfuscations that dramatically alter methods' PDGs. For example, plagiarists may be able to alter app methods to mimic the semantic vectors of library code or use PDG node splitting to increase the distance between the original semantic vector and the plagiarized one. Additionally, plagiarists could artificially join connected components within methods using dead code to increase the distance between the semantic vectors or split each connected component into a set of very small methods that are too small to be considered by AnDarwin. Ultimately, plagiarists could reimplement the original app.

The subversions listed above are difficult for most similarity detection tools to detect, including AnDarwin. Fortunately, all these subversions require substantial effort on the part of the plagiarists as it would be difficult for tools to do this automatically. Further, such a tool would require intimate knowledge of the targeted app to ensure that the plagiarized app still functions correctly.

7.2 Probability of a False Positive

In this section, we examine the probability that two dissimilar apps are clustered together by full app similarity detection. Consider two similar apps that share n features. Assuming that features are independent, which is the case when library code is excluded, then:

$$Pr[\text{share n features}] = Pr[\text{share feature}]^n = Pr[\text{share close SV}]^n \tag{4}$$

Where "close SV" means two semantic vectors that will be clustered together by LSH or are identical. Now, consider the case where two apps are not similar, but are clustered together anyway. This means they must still agree on n features, where each of these n agreements is a false positive which we shall refer to as a *feature collision*. Feature collisions can occur in two ways: (1) semantic vector collision and (2) non-code clone semantic blocks generating "close" semantic vectors. Fortunately, even if the probability of a feature collision is very high, there has to be n feature collisions in order to have a false positive. We have found that, on average, apps contain 148 features after excluding common features. Therefore, in order for two unrelated apps to have a Jaccard Index above our threshold of 50%, there must be approximately 100 feature collisions. Even if the probability of a feature collision was 95%, the probability of a false positive with this many features would be less than one percent.

8 Related Work

There have been several approaches proposed recently to find similar Android apps. Closest to AnDarwin is [34]. They use a heuristic based on how tightly classes within the app are coupled (using its call graph) to split apps into primary and rider sections. Then, they represent the primary section as vectors which they cluster in linearithmic time. This heuristic allows [34] to detect some partial app similarity, however, it would be easy for a plagiarist to circumvent these heuristics by adding dead code to the call graph to artificially couple unrelated classes. In contrast, AnDarwin's partial app similarity does not rely on heuristics. Additionally, while AnDarwin's features represent the functionality of methods of an app and are thus difficult to change, [34]'s features include the app's permissions, the Android API calls used and several other features, all of which may be easily changed. [34] can also detect commonly injected code by clustering the rider sections, however, they use the same features and heuristics which are easily changed and circumvented, respectively. All other related work described below compares applications pairwise, yielding significant scalability problems. Additionally, neither [34] nor any other related work provides the ability to robustly find partial app similarity, as AnDarwin does.

Androguard [6] currently supports two methods of similarity detection: comparing apps using the SHA256 hashes of methods and basic blocks and using the normal compression distance of pairs of methods between apps. DEXCD [18] detects Android clones by comparing similarities in streams of tokens from Android DEX files. DroidMOSS [33] computes a series of fingerprints for each app based on the fuzzy hashes of consecutive opcodes, ignoring operands. Apps are then compared pairwise for repackaging by calculating the edit distance between the overall fingerprint of each app. DNADroid [16] compares apps based on the PDGs of their methods. Juxtapp [22] disassembles each app and creates k-grams over the opcodes inside the app's methods. Next it hashes the k-grams to create features which are used to represent each app and then computes similarity by comparing sets of these features between pairs of apps. All of these approaches except DNADroid are vulnerable to plagiarism that involves moderate amounts of adding or modifying statements, though DNADroid's comparison is computationally expensive.

9 Conclusion

We present AnDarwin, a tool for finding apps with similar code on a large scale. In contrast with earlier approaches, AnDarwin does not compare apps pairwise, drastically increasing its scalability. AnDarwin accomplishes this using two stages of clustering: LSH to group semantic vectors into features and MinHash to detect apps with similar feature sets (full app) and features that often occur together (partial app). We evaluated AnDarwin on 265,359 apps crawled from 17 markets. AnDarwin identified at least 4,295 apps that have been cloned and an additional 36,106 apps that are rebranded. From the clusters discovered by AnDarwin, we found 88 new variants of malware and could have discovered 169 new malware. We also presented a cluster post-processing methodology for finding apps that have had similar code injected. AnDarwin has a low false positive rate — only 3.72% for full app similarity detection. Our findings indicate that AnDarwin is an effective tool to identify rebranded and cloned apps and thus could be used to improve the health of the market ecosystem.

Acknowledgements We would like to thank the anonymous reviewers for their insightful feedback as well as Liang Cai, Dennis Xu, Ben Sanders, Justin Horton, and Jon Vronsky for their assistance in obtaining Android applications. This paper is based upon work supported by the National Science Foundation under

Grant No. 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. Goapk market. http://market.goapk.com, April 2012.
2. Google play. https://play.google.com/store/apps, April 2012.
3. Slideme: Android community and application marketplace. http://slideme.org/, April 2012.
4. Virus total. https://www.virustotal.com, June 2012.
5. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. Ieee, 2006.
6. Androguard. Androguard: Manipulation and protection of android apps and more... http://code.google.com/p/androguard/, April 2012.
7. AppBrain. Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps, November 2012.
8. BajaBob. Smalihook.java found on my hacked application. http://stackoverflow.com/questions/5600143/android-game-keeps-getting-hacked, May 2012.
9. B.S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.
10. Scott Beard. Market shocker! iron soldiers xda beta published by alleged thief. http://androidheadlines.com/2011/01/market-shocker-iron-soldiers-xda-beta-published-by-alleged-thief.html, May 2012.
11. The Lookout Blog. Security alert: Gamex trojan hides in root-required apps - tricking users into downloads. http://blog.mylookout.com/blog/2012/04/27/security-alert-gamex-trojans-hides-in-root-required-apps-tricking-users-into-downloads/, April 2012.
12. A.Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.
13. A.Z. Broder, M. Charikar, A.M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 327–336. ACM, 1998.
14. IBM T.J. Watson Research Center. T.j. watson libraries for analysis (wala). http://wala.sourceforge.net, April 2012.
15. comScore. comscore reports march 2012 u.s. mobile subscriber market share. http://www.comscore.com/Press_Events/Press_Releases/2012/4/comScore_Reports_March_2012_U.S._Mobile_Subscriber_Market_Share, May 2012.
16. J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. *Computer Security–ESORICS 2012*, pages 37–54, 2012.
17. J. Crussell, C. Gibler, and H Chen. Scalable semantics-based detection of similar android applications. Technical report, University of California, Davis, 2013.
18. Ian Davis. Dexcd. http://www.swag.uwaterloo.ca/dexcd/index.html, April 2012.
19. S. Doherty and P. Krysiuk. Android.basebridge. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99, November 2012.

20. M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 321–330. IEEE, 2008.
21. C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. Adrob: Examining the landscape and impact of android application plagiarism. In *To Appear in the Proceedings of 11th International Conference on Mobile Systems, Applications and Services*, 2013.
22. S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.
23. L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.
24. X. Jiang. Droidkungfu. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html, November 2012.
25. R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis*, pages 40–56, 2001.
26. Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.
27. Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.com/2012/02/android-and-security.html, April 2012.
28. G. OGorman and H. Honda. Android.geinimi. http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99, November 2012.
29. pxb1988. dex2jar: A tool for converting android's .dex format to java's .class format. https://code.google.com/p/dex2jar/, April 2012.
30. A. Rajaraman, J. Leskovec, and J. Ullman. Mining of massive datasets. http://infolab.stanford.edu/ ullman/mmds/book.pdf, 2012.
31. T. Spring. Sneaky mobile ads invade android phones. http://www.pcworld.com/article/245305/sneaky_mobile_ads_invade_android_phones.html, June 2012.
32. Android Threats. Android/adwo. http://android-threats.org/androidadwo/, February 2013.
33. W. Zhou, Y. Zhou, X. Jiang, and P. Ning. 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.
34. Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM, 2013.
35. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of 33rd Symposium on Security and Privacy*. IEEE, 2012.