AnDarwin: Scalable Detection of Android Application Clones Based on Semantics

Jonathan Crussell, Clint Gibler, and Hao Chen

Abstract—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 10 hours, AnDarwin detected at least 4,295 apps that are the victims of cloning and 36,106 rebranded apps. Additionally, AnDarwin detects similar code that is injected into many apps, which may indicate the spread of malware. Our evaluation demonstrates AnDarwin's ability to accurately detect similar apps on a large scale.

Index Terms—Program analysis, clustering, plagiarism detection, mobile applications

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 [19]. App cloning has been widely reported by developers, smart phone security companies and the academic community [8], [10], [11], [16], [20], [31], [32]. 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 percent 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

• *The authors are with the Computer Science, University of California, Davis, Davis, CA 95616 USA.*
E-mail: {jcrussell, cdgibler, chen}@ucdavis.edu.

Manuscript received 20 Dec. 2013; revised 5 Nov. 2014; accepted 20 Nov. 2014. Date of publication 17 Dec. 2014; date of current version 31 Aug. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2014.2381212

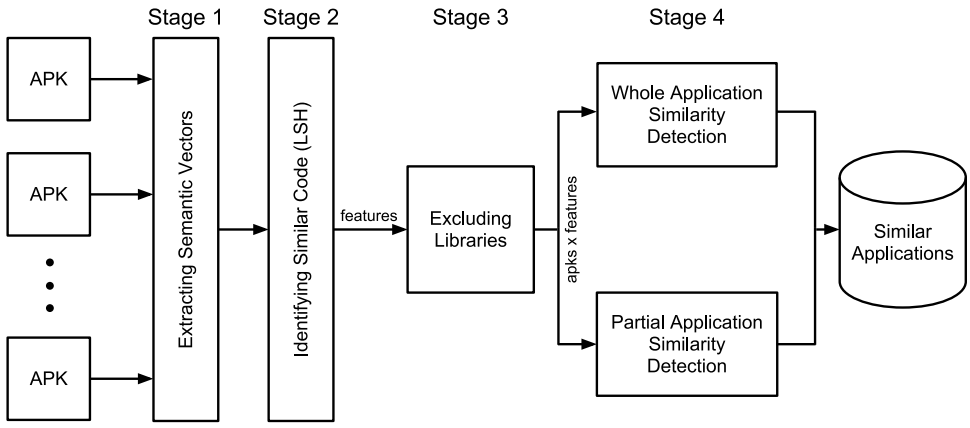


Fig. 1. Overview of AnDarwin.

vetting process which has allowed numerous malicious apps onto the markets [33]. Google Play has developed a Bouncer service [25] 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 [18], [21], [23], [24]. 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 [21] and its successor [18] 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. Our adversaries include plagiarists, who clone other developers' code, and spammers, who release many copies of similar apps. These adversaries may try to obfuscate their code to subvert clone detection. 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 percent) of the 265,359 apps we analyzed include native code.

4 Methodology

AnDarwin consists of four stages as depicted in Fig. 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). Additionally, we describe a post-processing methodology for detecting apps that have had similar code injected (Section 4.5) and analyze the total time complexity of AnDarwin (Section 4.6).

We base the first two stages of AnDarwin on the approaches of Jiang et al. [21] and Gabel et al. [18] 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 the T. J. Watson Libraries for Analysis (WALA) [14] to construct PDGs. WALA's PDGs contain a total of 20 node types so AnDarwin's semantic vectors have 20 dimensions. To increase the precision of the semantic vectors, we could use more information that WALA stores for each node, such as the specific binary operation performed. Doubling or even tripling the number of dimensions would not dramatically increase the overall time complexity of AnDarwin (Section 4.6). AnDarwin discards semantic blocks with fewer than 10 nodes as small semantic blocks typically represent trivial and uncharacteristic code. We determined this threshold through manually analyzing apps and discovering that even a few lines of code can result in fairly large semantic blocks once the code is translated into static single assignment form, a requirement for the construction of PDGs. In Section 5.10, we explore ways to capture edge information from the PDGs in the semantic vectors.

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, by finding the nearest neighbors of a semantic vector, we can identify potential code clones. Not all of the nearest neighbors will be code clone as two completely unrelated semantic blocks could have the same frequency of node types. This is due to the lossy nature of our semantic vector characterization: many PDGs may generate the same semantic vector. Specifically, the number of PDGs that generates a given semantic vector increases combinatorially with the magnitude of the semantic vector. Fortunately, our evaluation shows that these collisions are infrequent in practice (Section 5.10).

To determine all the nearest 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 nearest 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 nearest neighbors, LSH first hashes all the vectors with the special hash functions and then looks for nearest 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 [21]. 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.

From LSH, AnDarwin identifies a number of clusters of semantic vectors representing semantic blocks that could be code clones. However, AnDarwin's goal is not to identify code clones—we wish to identify app clones. Therefore, we must determine a representation of apps in feature space. Fortunately, we can use the code clones to represent apps. Specifically, we can assign each code clone an index and then represent each app as a binary vector where the ith dimension is 1 if the app contains a semantic block whose semantic vector is a member of the ith code clone and 0 otherwise. This binary vector has many dimensions, one for each code clone; however, it will be very sparse as each app will only have a relatively small number of semantic blocks. Therefore, to save space, we can store this vector for each app using a set containing the indices of the non-zero dimensions. We refer to this set as the app's *feature vector*.

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], [31] 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 we compute the feature vectors for apps, features from library code will appear in many more feature vectors than non-library code (recall that a feature represents a code clone identified by LSH). Additionally, features representing boilerplate code and compiler-generated constructs will appear much more frequently. 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 Similarity of their two feature sets F_A and F_B, which reduces the problem of finding similar app to that of finding similar sets

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

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 Similarity 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 Similarity, 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 is another Locality Sensitive Hashing algorithm that was originally developed at Alta Vista to detect similar websites when represented as a set of features. As with the LSH algorithm described in Section 4.2, the MinHash algorithm starts by computing a signature for each set based on hashing and random permutations of the feature space. It then identifies nearest neighbors among sets that have hash collisions. Using Minhash, AnDarwin can 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 Detecting Commonly Injected Code

Once AnDarwin has clustered apps, it would be useful to automatically detect if similar code has been injected into many apps. For example, a malware author may download many innocuous apps, inject malware and then upload these spiked apps. We introduce two concepts: *cores* and *diffs*. Each app cluster computed by AnDarwin has a *core* which represents the code common to all apps in the cluster. AnDarwin computes the core by calculating the intersection of the feature sets of the apps in the cluster. Using this core as a point of reference, AnDarwin calculates a diff for each app in the cluster by subtracting the cluster's core from the app's feature set. This new set represents a superset of the code that is modified in the app.

AnDarwin calculates these cluster cores and corresponding app diffs for each app cluster for both full and partial similarity detection. Then, it clusters the diffs with MinHash to find apps that have had similar code injected.

4.6 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: $O(d \sum_{g \in G} |g|^{\rho} \log |g|)$ [21]. 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.

4.7 Online Clone Detection

So far we have described AnDarwin as an offline tool, because we must collect all the apps before running AnDarwin. More likely in the real-world, however, new apps are uploaded or crawled continuously. To meet this need, we extend AnDarwin to have an online mode, which processes apps as they come in.

Online clone detection works in three steps. First, it determines an app's feature vector. Recall that this is based on the clusters of semantic vectors (Section 4.2). Each new app has a number of semantic blocks and the semantic vector for each block may or may not have already been seen before in another app. If a semantic vector has been seen before, we need no further processing for that vector as we already know the feature index. For new semantic vectors, we must run LSH to identify the nearest neighbors for the vector. We then either merge the semantic vector into an existing cluster based on its neighbors and thus assign it to a feature, or, if it has no close neighbors, create a new feature. Applying this process to each of the semantic vectors for a new app produces a feature vector for the app.

Once AnDarwin computes the app's feature vector, it then excludes features previously identified as library features. There is a chance that this new app might help to identify a new library. As we expect this to happen rarely, we rerun the library detection component periodically rather than when every new app is added.

Finally, we identify similar apps based on full and partial app similarities. Finding full app similarity matches is straightforward: we compute the MinHash signature for the

TABLE 1
Market Origins of the Apps Analyzed by AnDarwin

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		

Since some apps appear on multiple markets, the total apps in the table is slightly more than the total 265,359 apps analyzed.

app's feature set and then check for collisions with previously analyzed apps. Partial app similarity detection is more difficult—the introduction of one new app with n features causes n updates to the feature-to-app sets used by partial app similarity. For this reason, we run the partial app similarity detection periodically rather than when each each new app is added.

5 Evaluation

We have implemented our approach in a tool called AnDarwin. AnDarwin uses dex2jar [27] 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 [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 [28].

AnDarwin has many parameters, each of which can be tuned based on the dataset AnDarwin is applied to. Many of the parameters, such as those used for the LSH-based clustering of the semantic vectors, were determined based on trial-and-error. This involved running the analysis with different parameters and manually investigating the results. For some of the parameters, such as those used for MinHash, there are equations that can be used select the appropriate parameters. For full app similarity, we wished to find apps that share at least 50 percent of their features using at most 200 permutations. Using these values and the equation in Section 3.4.3 of [28] we could select the appropriate number of bands to break the MinHash signature into. An interesting direction for future work would be to find a way to automatically tune the parameters of AnDarwin to reach a desired false positive or false negative rate. This is inherently difficult as clustering is unsupervised—there is no ground truth about what is the correct clustering.

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

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 five-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 percent) 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 percent of the excluded features exhibit this trait. We note that this threshold may be easily tweaked depending on false positive and false negative requirements.

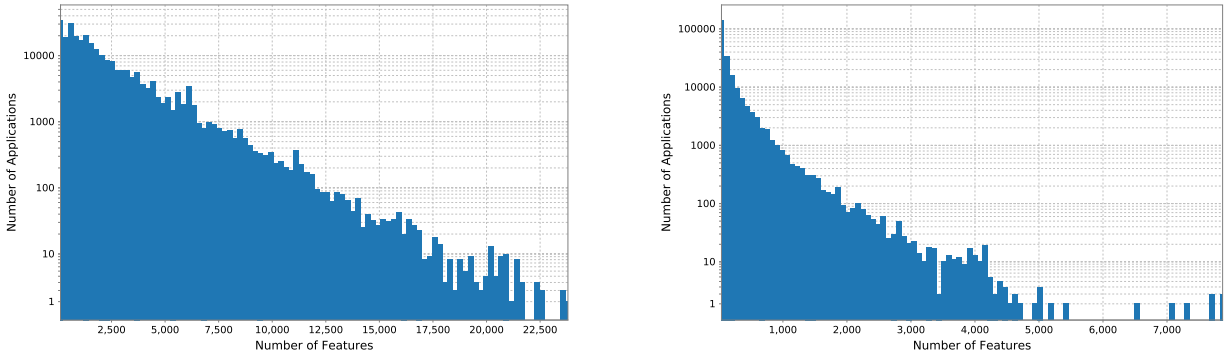
5.3 App Complexity

Overall, AnDarwin found that a large number of apps are not very complex. Fig. 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 app has 23,918 features. Once libraries are excluded, the number of apps with at least one feature drops to 231,184. Fig. 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 do not 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



(a) Before common feature exclusion (b) After common feature exclusion

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

apps. Fig. 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 Similarity (Equation (1)) between each pair of apps. For each cluster C, we compute the similarity score, $Sim(C)$, as

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

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. Fig. 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.

5.5 Partial App Similarity Detection

Using partial app similarity detection, AnDarwin found 11,848 clusters consisting of 88,464 distinct apps. Figs. 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

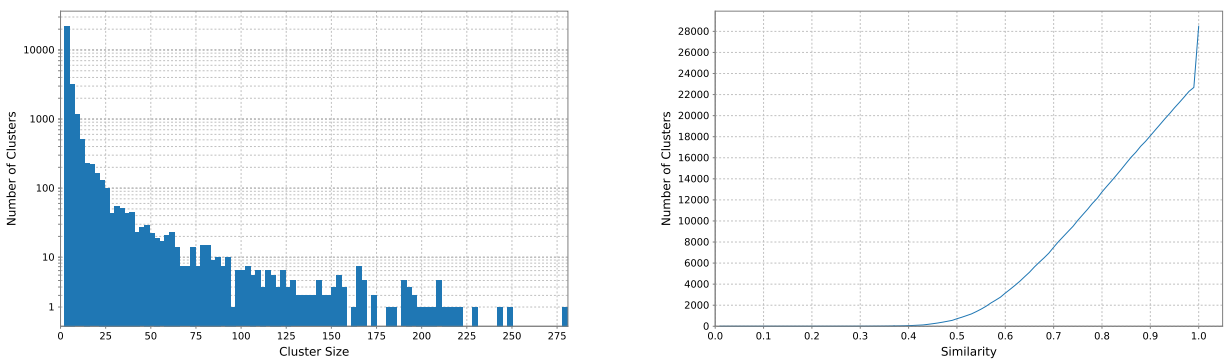
$$Sim_p(C) = avg\left\{(A, B) \in C : \frac{|F_A \cap F_B|}{\min(|F_A|, |F_B|)}\right\}. \quad (3)$$

Fig. 4b shows the cumulative distribution function of $Sim_p(C)$ for the partial app similarity detection clusters. Comparing Fig. 3b to Fig. 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

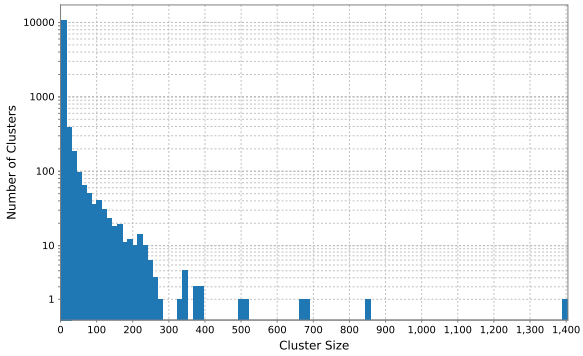
We evaluated AnDarwin's performance on a server with quad Intel Xeon E7-4850 CPUs (80 logical cores with hyper threading) and 256 GB DDR3 memory. Using 75 threads, it took 4.47 days to extract semantic vectors (Stage 1) from all 265,359 apps (only 109 seconds per thread to process each app). We note 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

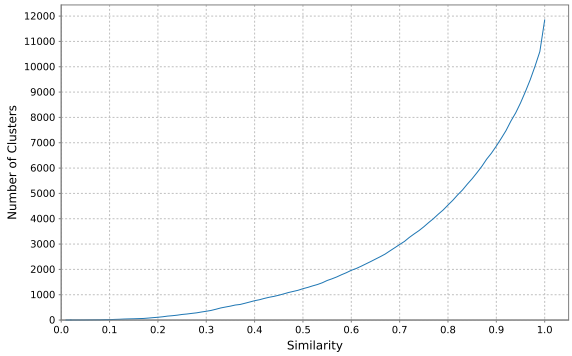


(a) Histogram of the cluster sizes on logarithmic scale (b) Cumulative distribution function of $Sim(C)$

Fig. 3. Full App Similarity Detection.



(a) Histogram of the cluster sizes on logarithmic scale



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

Fig. 4. Partial App Similarity Detection.

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 10 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.

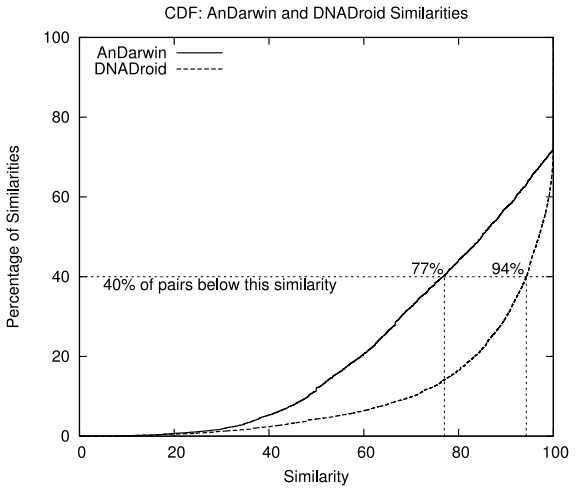
5.7 Accuracy

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 percent, making it an ideal tool for evaluating AnDarwin's accuracy.

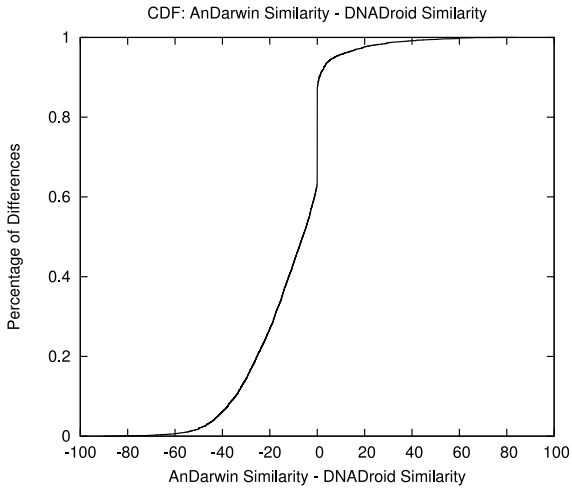
Unfortunately, DNADroid is too computationally expensive to apply to all the pairs of apps AnDarwin found. Instead, we randomly selected 6,000 of the 28,495 clusters and then randomly selected one app from each cluster to compare against all the other apps in the cluster. This resulted in a total of 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. When we compared the randomly selected app from each cluster to all the other apps in the cluster using DNADroid, we found that 96.28 percent of the clusters had at least 70 percent of the coverage values over 50 percent. This threshold is the same as the Jaccard Similarity used by AnDarwin. Furthermore, 95.50 percent of the clusters had 90 percent of the coverage values over the same threshold. Using the 70 percent criteria, only 3.72 percent of AnDarwin's full app similarity detection clusters were not verified by DNADroid and, therefore, are considered false positives.

In Fig. 5a, we plot the CDF of the individual similarities for AnDarwin and DNADroid for the 25,434 pairs of apps. Interestingly, AnDarwin's similarities are consistently lower than DNADroid's. There are two possible causes for this: AnDarwin's semantic vector approach fails to find method matches that are subgraph isomorphic and thus found by DNADroid or AnDarwin's and DNADroid's library exclusion methodologies produce different results. DNADroid



(a) Cumulative distribution of AnDarwin and DNADroid similarities



(b) Cumulative distribution function of AnDarwin similarity - DNADroid similarity

Fig. 5. Comparison of AnDarwin and DNADroid similarities.

TABLE 2
Results of Manual Analysis on App Pairs Where AnDarwin's and DNADroid's Similarities Have the Largest Differences

Similarity Score		Application Pairs			
Comparison	Average of (AnDarwin - DNADroid)	Total	Visually Similar	Behaviorally Similar	True Positives
AnDarwin << DNADroid	−84%	25	25	N/A	100%
AnDarwin >> DNADroid	68%	25	11	13	96%

"True positives" is the percentage of pairs that were visually or behaviorally similar.

excluded libraries using signatures calculated over the bytecode of known library package names. This means that DNADroid is limited to well-known libraries that have been available for a longer period of time. It also means that libraries that have had their package names changed are undetectable to DNADroid. AnDarwin's library exclusion, on the other hand, does not require the knowing the library package names in advance making it much more robust. Since AnDarwin's library exclusion is more complete and may also include trivial code that gets reused often, we would expect AnDarwin's similarities to be consistently lower than DNADroid's.

In Fig. 5b, we plot the CDF of the difference between AnDarwin's similarity and DNADroid's. In a majority of pairs (60 percent), for an individual pair of apps, AnDarwin's similarity is lower than DNADroid's. In a large number of pairs (30 percent), AnDarwin and DNADroid agree almost exactly, showing virtually no difference between the computed similarities. Finally, in a small number of pairs (10 percent), DNADroid's similarity is actually lower than AnDarwin's. With a margin of $+/-10$ percent, AnDarwin and DNADroid agree on over 50 percent of the app pairs.

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 percent) 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.

5.8 Manual Analysis

As an additional analysis of the false positive rate of AnDarwin's full app similarity detection, we manually investigate 50 app pairs. In order to select these app pairs, we leverage the results of the previous section to select pairs that have the largest reported differences between AnDarwin's and DNADroid's similarities. Specifically, we sort the pairs by the differences and take the top 25 pairs and the bottom 25 pairs. To manually investigate each app pair, we opened the two apps side-by-side in the Android emulator to determine if they were visually similar. In many cases clones will not bother changing the UI which makes such pairs easy to confirm. When the UIs are not similar, we manually interact with the apps to determine their behaviors given the same input. If the apps behave similarly, we also treat this as a true positive.

The results of our manual analysis are presented in Table 2. For pairs where DNADroid's similarity is much higher, we found all pairs to be visually similar. These pairs are potential false negatives of AnDarwin. Recall that these pairs were determined using the clusters produced by full app similarity detection and that for a pair to be analyzed, the two apps must have appeared in the same cluster. This means that AnDarwin correctly put these 25 pairs in the same cluster, despite reporting a low similarity. We call these pairs potential false negatives because these pairs were only placed in the same cluster after they were found similar to intermediate apps that caused the clusters containing each app in the pair to be merged. Therefore, if these intermediate apps were not included in our analysis, these pairs would have been false negatives of AnDarwin. In the other set of pairs where AnDarwin's similarity is much higher, we had to manually interact with many of the apps but were still able to determine that 96 percent were similar. These are confirmed false negatives of DNADroid. As discussed earlier in the previous section, we used DNADroid to verify the clusters produced by full app similarity detection and found a false positive rate of 3.72 percent. This includes the 24 pairs that are confirmed false negatives of DNADroid implying that the actual false positive rate of AnDarwin is even lower than 3.72 percent.

5.9 Commonly Injected Code

In this section, we describe the results of our commonly injected code detection. In this evaluation, we use the full app similarity detection clusters as input although the same methodology applies to the partial app similarity detection clusters. In total, we found 14,402 clusters using commonly injected code detection. Unsurprisingly, many of these clusters overlap with the full app similarity detection clusters. This occurs when there are at least three apps in a cluster, A, B and C and two of the apps, say B and C, have the same injected code that A does not. Then, the diffs for B and C will be identical, and they will be clustered together. Therefore, we filter the diff clusters using the original app clusters yielding a total of 694 clusters containing a total of 3,927 apps. This approach allows us to find apps with commonly injected code even when the injected code accounts for a very small percentage of the apps' code base. In one case, during full app similarity detection, AnDarwin placed app A and B in one cluster and app C and D in another one. Then, during commonly injected code detection, AnDarwin placed $A-B$ and $C-D$ in the same cluster. Our

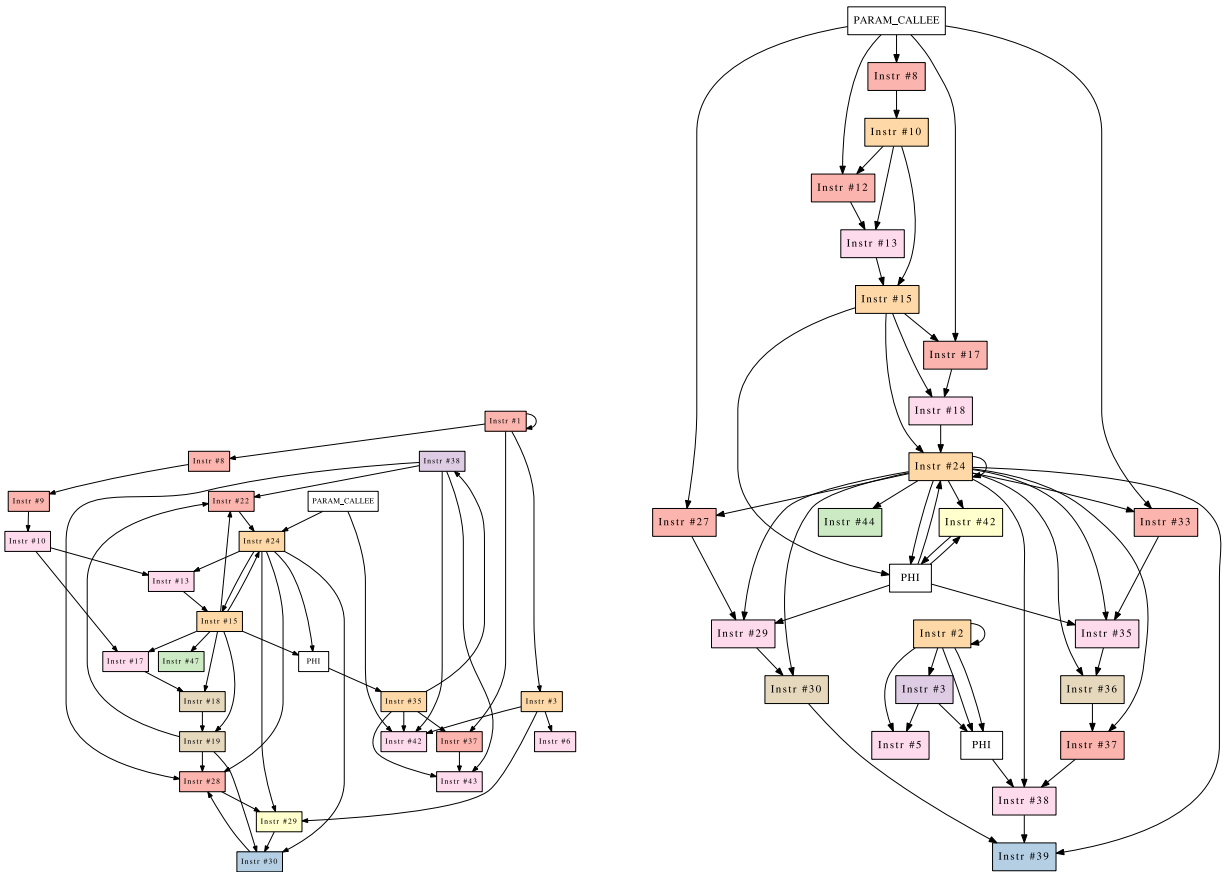


Fig. 6. Example of two PDGs that generate the same semantic vectors but are not subgraph isomorphic. Nodes in the graphs are colored according to their node types. PHI nodes are created when there are branches in the bytecode. Param Callee nodes show dependencies on method parameters. Neither PHI nor Param Callee nodes are used when building semantic vectors.

examination shows that app A and C have 366 and 308 features, respectively, while the common features among A and C ($A \cap C$) are only 8. Since the common features among A and C account for just 2 percent of all the features in either A and C, they cannot be typically considered clones. However, by extracting the diffs $A - B$ and $C - D$, our commonly injected code detection algorithm exposes the relationship between A and C, i.e., they contain similarly injected code.

5.10 Alternative Semantic Vectors

As stated in Section 4.1, we compute semantic vectors by creating a frequency vector over the different node types for a connected component in a PDG. This captures which node types are present in the PDG but does not capture any information about the structure of the graph. Without any structural information, AnDarwin may falsely say two methods are similar simply because they have the same number of nodes of each type. Therefore, we explore different ways in which we could capture the edge information to improve the semantic vectors to reduce the potential for false positives in the semantic vector clustering stage. For example, Fig. 6 shows two PDGs that generate the same semantic vector but have different structures, so they are not subgraph isomorphic (see Section 7.2 for a discussion of these *feature collisions*). Reducing false positives at this stage allows AnDarwin to form more precise features, which will improve the final quality of the clusterings that identify similar apps.

Total degree. A simple approach to add structural information to the semantic vectors is to include the total degree for nodes of each type. This is very similar to the approach taken with the nodes. This approach should perform well differentiating graphs with vastly different structures even if the node counts are the same; however, it may be less robust than other approaches. It gives nodes and edges equal weight in the semantic vector, which may reduce the work that a plagiarist has to do to modify the PDG such that it will not generate similar semantic vectors. This is because for every node added, multiple edges can be added.

Average In- or Out-Degree. As pointed out in the previous approach, edges and nodes should not be weighted equally in the semantic vector. To reduce the influence of the edges, we could normalize the total degrees using the number of nodes of the given type. Alternatively, we could normalize using the total number of edges in the graph. In this experiment, we choose to split degree counts for incoming edges and outgoing edges and include the normalized values using the number of nodes of the given type. This creates two approaches, one for the normalized degree of incoming edges for nodes of each type and the other for outgoing edges.

Max in- or out-degree. Our final approach uses the maximum degree of incoming or outgoing edges of each node type. We believe this approach will be the most robust against plagiarist modifications as it captures information about the most important nodes of each type in the vector. If a plagiarist wants to alter one of these new dimensions,

TABLE 3
Evaluation of the True Positive Rates for the Alternative Semantic Vectors

Approach	Features			
	Total	Singletons	Matched	True Positives
No edges	78,365	2,427	606	75.8%
Total degree	164,692	2,802	742	88.5%
Avg in-degree	112,874	2,496	675	79.3%
Avg out-degree	122,129	2,505	711	80.4%
Max in-degree	119,349	2,514	713	80.7%
Max out-degree	132,130	2,594	765	84.0%

For each approach, we randomly selected 4,000 features and determined whether a random pair of PDGs represented by the feature are subgraph isomorphic. No edges *is the currently used approach. The true positive rate is the percentage of pairs that were singletons (feature that represent a single PDG) or matched.*

she must add enough nodes and edges to create a new node that has more incoming or outgoing edges than the existing node in the graph. Depending on the size of the original graph, this may require significant additional code. One potential weakness of this approach is that some nodes may not be present in the original PDG, making the dimensions for the edge counts of those node types easier to change.

5.10.1 Experiment

To evaluate which approach is the most promising for improving our semantic vectors in future work, we performed the following experiment. We took 1,000 random APKs from our dataset and computed the semantic vectors using each of the above alternative approaches. We then clustered these semantic vectors to form six sets of features. To test the precision of each approach, we sampled 4,000 random features, looked up the semantic vectors represented by that feature, and then randomly selected a pair of methods whose PDGs were in that set of semantic vectors. We then compared each pair of PDGs using subgraph isomorphism provided by DNADroid [16].

Table 3 shows the results of this experiment. The second column shows the number of features that were found from the clusters of semantic vectors produced by LSH. The singletons column shows the number of sampled features for which there was only a single semantic vector in a single APK. These, by definition, cannot be considered false positives of the approach. Next, we report the number of sampled features whose random PDG pair were matched by subgraph isomorphism. Last, we calculate the true positive rate for each approach, counting both singletons and matches.

Table 3 shows all of the new approaches outperform the current approach, which includes no edge information. The *total degree* approach had the highest true positive rate, a 12.7 percent improvement over the current approach. While the total degree approach outperformed the other approaches, it is likely to be the least robust to plagiarist modification. Therefore, the max out-degree approach with a 8.2 percent higher true positive rate shows the most promise for replacing our current approach.

All of these suggestions double the size of the semantic vectors from 20 to 40, adding one new dimension for each node type. As stated in Section 4.6, the complexity of the LSH clustering stage of AnDarwin depends linearly on the size of the vectors. Therefore, none of these proposed semantic vectors would have a significant impact on the overall scalability of AnDarwin. In fact, in future work, we could explore combining multiple representations of the graph structure information such as max in-degree and max out-degree without significantly impacting performance.

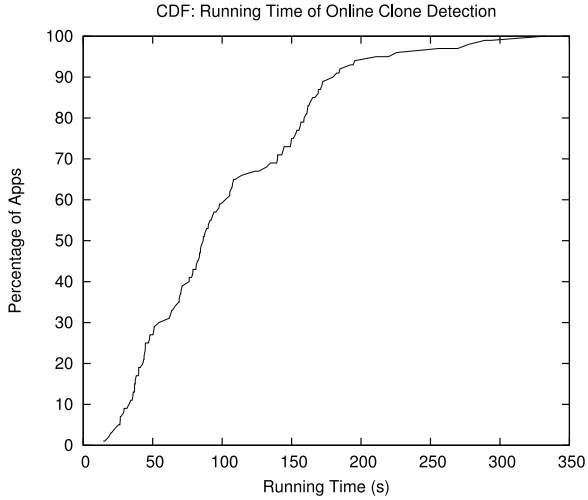


Fig. 7. CDF of running time of online clone detection.

5.11 Online Clone Detection

We evaluated the online clone detection described in Section 4.7. Starting with a database of semantic vectors and features computed from 265,359 apps used earlier, we selected 150 random new apps and ran them through online clone detection. Note that online clone detection is cumulative: after we run an app through, its semantic vectors and features become part of the database.

Fig. 7 shows the CDF of the time for running online clone detection on these apps. The times range from 14.4s to 332.0s with a median of 86.4s.

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 [33], 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 [22]. Approximately another 20 are identified as belonging to various malware families described in [33]. 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 [29] and have been known to have misleading ad campaigns [30]. These results demonstrate AnDarwin's utility for discovering new variants of malware.

6.4 New Malware Detection in Clones

Zhou and Jiang [33] found that 86.0 percent 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 percent of the permission union. Finally, if the apps identified in the previous step are fewer than 15 percent of the total apps in the cluster, we mark these apps as suspicious. Using this criterion, we found 608 suspicious apps. Sixteen of these apps overlap with the malware dataset from [33] and one 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 [22], BaseBridge [17] and Geinimi [26]. 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

$$Pr[\text{share n features}] = Pr[\text{share feature}]^n \quad (4)$$

$$= Pr[\text{share close SV}]^n, \quad (5)$$

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, n feature collisions must happen to create 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 Similarity above our threshold of 50 percent, there must be approximately 100 feature collisions. Even if the probability of a feature collision was 95 percent, 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 [32]. 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 [32] 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, [32]'s features include the app's permissions, the Android API calls used and several other features, all of which may be easily changed. [32] 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 [32] 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. DroidMOSS [31] 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 [20] compares apps based on sets of features created from k-grams of the opcodes inside the disassembled app's methods. 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 percent 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.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful feedback. For their help obtaining Android apps, they would like to thank Liang Cai, Dennis Xu, Ben Sanders, Justin Horton, and Jon Vronsky. For her ideas of alternative approaches to create semantic vectors, they would like to thank Julia Matsieva. Finally, they would like to thanks Sam Dawson for his work to manually investigate app similarity. This paper is based upon work supported by the US 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 US National Science Foundation.

REFERENCES

[1] (2012, Apr.). Goapk market [Online]. Available: http://market.goapk.com

[2] (2012, Apr.). Google play [Online]. Available: https://play.google.com/store/apps

[3] (2012, Apr.). Slideme: Android community and application marketplace [Online]. Available: http://slideme.org/

[4] (2012, Jun.). Virus total [Online]. Available: https://www.virustotal.com

[5] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Proc. 47th Annu. IEEE Symp. Found. Comput. Sci.*, 2006, pp. 459–468.

[6] Androguard. (2012, Apr.). Androguard: Manipulation and protection of android apps and more... [Online]. Available: http://code.google.com/p/androguard/

[7] AppBrain. (2012, Nov.). Number of available android applications [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps

[8] BajaBob. (2012, May). Smalihook.java found on my hacked application [Online]. Available: http://stackoverflow.com/questions/5600143/android-game-keeps-getting-h acked

[9] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd Working Conf. Reverse Eng.*, 1995, pp. 86–95.

[10] S. Beard. (2012, May). Market shocker! iron soldiers xda beta published by alleged thief [Online]. Available: http://androidheadlines.com/2011/01/market-shocker-iron-soldiers-xda-be ta-published-by-alleged-thief.html

[11] The Lookout Blog. (2012, Apr.). Security alert: Gamex trojan hides in root-required apps—Tricking users into downloads [Online]. Available: http://blog.mylookout.com/blog/2012/04/27/

[12] A. Z. Broder, "On the resemblance and containment of documents," in *Proc. IEEE Compression Complexity Sequences*, 1997, pp. 21–29.

[13] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," in *Proc. 13th Annu. ACM Symp. Theory Comput.*, 1998, pp. 327–336.

[14] IBM T. J. Watson Research Center. (2012, Apr.). T. J. Watson libraries for analysis (WALA) [Online]. Available: http://wala.sourceforge.net

[15] comScore. (2012, May). Comscore reports march 2012 U.S. mobile subscriber market share [Online]. Available: http://www.comscore.com/Press_Events/Press_Releases/2012/4/comScore_Reports_March_2012_U.S._Mobile_Subscriber_Market_Share

[16] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. Comput. Security*, 2012, pp. 37–54.

[17] S. Doherty and P. Krysiuk. (2012, Nov.). Android.basebridge [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915 -4938-99

[18] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 321–330.

[19] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi, "Adrob: Examining the landscape and impact of android application plagiarism," in *Proc. 11th Int. Conf. Mobile Syst., Appl. Services*, 2013, pp. 431–444.

[20] 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 *Proc. 9th Conf. Detection Intrusions Malware Vulnerability Assessment*, 2012, pp. 62–81.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 96–105.

[22] X. Jiang. (2012, Nov.). Droidkungfu [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html

[23] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. 8th Int. Symp. Static Anal.*, 2001, pp. 40–56.

[24] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.

[25] H. Lockheimer. (2012, Apr.). Android and security [Online]. Available: http://googlemobile.blogspot.com/2012/02/android-and-security.html

[26] G. OGorman and H. Honda. (2012, Nov.). Android.geinimi [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111 -5403-99

[27] pxb1988. (2012, Apr.). dex2jar: A tool for converting android's .dex format to java's .class format [Online]. Available: https://code.google.com/p/dex2jar/

[28] A. Rajaraman, J. Leskovec, and J. Ullman. (2012). Mining of massive datasets [Online]. Available: http://infolab.stanford.edu/~ullman/mmds/book.pdf

[29] T. Spring. (2012, Jun.). Sneaky mobile ads invade android phones [Online]. Available: http://www.pcworld.com/article/245305/sneaky_mobile_ads_invade_android_ phones.html

[30] Android Threats. (2012, Feb.). Android/adwo [Online]. Available: http://android-threats.org/androidadwo/

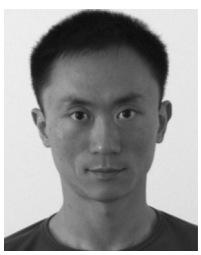
[31] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. 2nd ACM Conf. Data Appl. Security Privacy*, 2012, pp. 317–326.

[32] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of piggybacked mobile applications," in *Proc. 3rd ACM Conf. Data Appl. Security Privacy*, 2013, pp. 185–196.

[33] Y. Zhou, and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proc. 33rd Symp. Security Privacy*, 2012, pp. 95–109.

Jonathan Crussell received the BS, MS, and the PhD degrees in computer science from the University of California, Davis. He is a Senior Member of the Technical Staff at Sandia National Laboratories. His primary interests are in malware analysis and mobile systems.

Clint D. Gibler received the BS degree from Case Western Reserve University and the MS and PhD degrees in computer science at the University of California, Davis. He is a software security engineer at NCC Group Domain Services. His primary interests are in computer security and mobile computing. He received an US NSF Graduate Research Fellowship Honorable Mention in 2010.



Hao Chen received the BS and MS degrees from Southeast University and the PhD degree from the Computer Science Division, University of California, Berkeley. He is an associate professor in the Department of Computer Science at the University of California, Davis. His primary interests are computer security and mobile computing. He won the US National Science Foundation CAREER award in 2007, and UC Davis College of Engineering Faculty Award in 2010.