DroidLegacy: Automated Familial Classification of Android Malware

Luke Deshotels Vivek Notani Arun Lakhotia
University of Louisiana at Lafayette
alecdeshotels@gmail.com vivek200690@gmail.com arun@louisiana.edu

Abstract

We present an automated method for extracting familial signatures for Android malware, i.e., signatures that identify malware produced by piggybacking potentially different benign applications with the same (or similar) malicious code. The APK classes that constitute malware code in a repackaged application are separated from the benign code and the Android API calls used by the malicious modules are extracted to create a signature. A piggybacked malicious app can be detected by first decomposing it into loosely coupled modules and then matching the Android API calls called by each of the modules against the signatures of the known malware families. Since the signatures are based on Android API calls, they are related to the core malware behavior, and thus are more resilient to obfuscations.

In triage, AV companies need to automatically classify large number of samples so as to optimize assignment of human analysts. They need a system that gives low false negatives even if it is at the cost of higher false positives. Keeping this goal in mind, we fine tuned our system and used standard 10 fold cross validation over a dataset of 1,052 malicious APKs and 48 benign APKs to verify our algorithm. Results show that we have 94% accuracy, 97% precision, and 93% recall when separating benign from malware. We successfully classified our entire malware dataset into 11 families with 98% accuracy, 87% precision, and 94% recall.

Categories and Subject Descriptors Security and privacy [*Intrusion/anomaly detection and malware mitigation*]: Malware and its mitigation

Keywords Android malware, piggybacked malware, malware detection, familial classification, signature generation, module generation, static analysis, class dependence graphs

1. Introduction

Even though Android incorporates many security principles, such as, privilege separation, application permissions, component encapsulation, and signing applications [24, 25] there are a few key design decisions that make Android easy to exploit. Three such decisions are: self-signed apps, the open structure of Android Application Package (APK), and multiple, independent points-of-entry in an application. Using reverse engineering tools, such as those provided by Winiewski and Tumbleson [28], one can disassemble an APK file, add or modify code in it, re-assemble the file, attach the code to be invoked on some events, repackage and sign it, and publish it in the marketplace. This makes it quite straightforward to inject malicious code in a legitimate app. This technique, called piggybacking [33], is used by malware authors to create malicious versions of popular applications. These piggybacked applications are then released on the various Android marketplaces, quite often made available as free versions of popular apps to lure victims.

According to a study by Zhou and Jiang [34] almost 86% of the Android malware is piggybacked. There are clearly other malware as well, but piggybacking offers a very low cost option for injecting malicious code to run on an Android device. Piggybacking may also be used to gain an initial foothold on the victim machine, and then exploit other vulnerabilities to escalate privilege and install native code. Thus, any significant improvement in detection of piggybacked malware can increase the level of effort needed by malware authors to exploit an Android device.

Industry studies show that Android malware is following the same trajectory as Windows malware in using techniques to defeat detection [5, 21]. Malware authors employ classic code obfuscation techniques to defeat the APK hash or class hash based methods for detection, such as those used by Google Play. The obfuscation techniques may be as simple as renaming the packages and classes, or as complex as renaming registers or transforming the dex code. Ballano reports that in May 2012 there were about 159 polymorphic variants for each malicious APK family [5, 21].

In this paper we present a method for creating obfuscation resilient signatures for families of piggybacked malware and a method for scanning an APK for a particular malware. We achieve the resiliency by constructing signatures using Android API calls. Since an application must use a certain set of API calls to perform a certain behavior, the API calls remain invariant under polymorphic and renaming obfuscations. Our method has analogue in the use of Windows API calls to detect Windows malware [12, 30]. Piggybacked malware, however, introduce another challenge—separating the injected rider code from the legitimate application. We take advantage of the fact that piggybacked malware is loosely coupled with the code of the host application. We present a method of partitioning an APK into loosely coupled modules, and a method for identifying the rider modules, given a collection of piggybacked malware from the same family. When scanning an unlabeled APK for malware it is first partitioned into loosely coupled modules. Then the Android API calls made by each module are compared to the signature of each family. We present the results of two controlled tests to estimate the expected performance of our method if deployed in the real-world. To ensure that test results are statistically significant estimators of the method's performance, and

PPREW'14, January 25 2014, San Diego, CA, USA.
Copyright © 2014 ACM 978-1-4503-2649-0/14/01…$15.00.
http://dx.doi.org/10.1145/2556464.2556467

not a random fluke due to choice of data, we use a classic 10-fold cross-validation design.

Our method complements Zhou et al.'s [33] work on detecting piggybacked applications. They start with known benign applications to create signatures for benign apps, and then use those signatures to identify piggybacked apps. To be applied on a market place, this method uses vantage point trees for comparing a massive amount of Android applications. For example, The Google Play application market has about 800,000 applications and is predicted to have over one million by the second quarter of 2013 [2]. The use of familial signatures should be more efficient than comparing applications since there are fewer malware families than Android applications. However, the method does provide a way to identify and even classify piggybacked malware into families. Having identified malware families using their method, our method may be used to create signatures for malicious apps and locate any piggybacked application. Our method, like Zhou et al.'s, is amenable to parallelization as well as vantage point trees [31]. Once signatures have been generated for known malware, untested APKs can be analyzed in parallel with comparisons to signatures calculated efficiently via vantage point trees.

In summary, our paper makes the following contributions:

- A method for partitioning an Android APK into loosely coupled modules.
- A linear method to identify the malware module in a collection of piggybacked APKs from the same malware family.
- A method to extract Android API based signatures from modules that may be polymorphic or renamed variants.
- A method to scan an APK for the existence of signatures of an individual family.
- Results of a controlled experiment to evaluate the performance of the methods presented.

A piggybacked malware is analogous to a parasitic virus in the Windows domain in that it is hidden within, i.e., infects, other legitimate application. Though there has been significant body of work in classifying standalone Windows malware, such as, Trojans, downloaders, worms, etc., we do not know of any prior work in automated classification of parasitic malware. To the best of our knowledge ours is the first work providing an automated method for identifying parasitic malware code from infected samples and providing a method for classifying such infected samples.

The rest of the paper is organized as follows. The next section discusses related works. Our method is described in Section 3. Section 4 presents the results of our experiment, which is followed by sections presenting future works and our conclusions.

2. Related Works

While our focus is on detection of malicious Android applications, our work relates to the broader area of finding similar codes or software clones and creating software phylogenies [22, 27]. These works have been motivated for finding code clones to aid maintenance [8], to find plagiarism [10], and to detect malicious programs [26], and they have operated on source code as well as binaries. It would not be possible to do justice to the entire breadth and depth of work in clone detection that relates to our work.

Anti-malware systems in the classic desktop or server domain operate on the end-points, a strategy that cannot be emulated on Android because, unlike in Windows or Linux, in Android each app runs in its own sandbox. Android's security constraints do not permit an app to have access to the API trace of another app (with a different user id). It is more conducive to scan an app in the marketplace, and at that instant, it could be run in a sandbox outside a device, as is done by Google Play. However, it is quite conceivable for a malware to alter its behavior when run outside of a device, much the same way a Windows malware may evade analysis in a sandbox.

It is very easy to disassemble and statically extract the API calls from Android APK. Winiewski and Tumbleson provide a collection of tools for operating on APK files [28]. Using these tools one can easily disassemble and access the Dalvik instructions for each class. While it is indeed possible to obfuscate the Dalvik code [23], it is not easy to create obfuscations that can hide the Android APIs called by Dalvik instructions. Though, Android does not permit modification of instructions in memory, one can at runtime modify the instructions in a class file or create new class files at runtime, and use such mechanisms to hide the API calls [23]. While the use of such obfuscations will indeed negate the methods presented in this paper, the obfuscations themselves may form a signature that may serve as an alert. Besides accurate malware detection is undecidable [13], any method for malware detection must be an approximation and thus can be defeated. Our method is useful since it would raise the bar for the Android malware author, forcing them to use more expensive obfuscations.

Zhou et al. [34] thoroughly analyzed and labeled over 1,000 Android malware to form the Android Malware Genome Project. They discussed repackaging in detail and noticed that most of the malware in their collection had been repackaged. This malware classified into families allowed us to train our system to recognize known Android malware families.

Other similar works include DNADroid [14]and PiggyApp. Zhou et al. [33] created PiggyApp to detect piggybacked apps. They extract a primary module from a legitimate application consisting of the code representing the main functionality of an application. Non-primary modules would include related libraries, advertisements, injected malware, etc. They create a signature from the primary module by extracting features from it. This signature was then scalably searched for in multiple Android marketplaces with the use of vantage point trees[31]. Any applications containing the primary module as well as new non-primary modules were considered piggybacked. These new non-primary modules were labeled as rider code injected into the legitimate application.

Our work is significantly similar to PiggyApp, so it is worth noting the differences between the two methods. First, the two methods begin with different inputs. Our method takes a labeled malware family as input while PiggyApp takes an arbitrary APK. Second, our method seeks to create a signature that represents a malware family, and PiggyApp seeks to detect piggybacking by matching primary modules. Third, malicious rider code is identified using different procedures. Our method identifies malicious modules by finding shared code within a malware family. PiggyApp creates a collection of "rider" modules not present in legitimate versions of applications. These "rider" modules are then clustered, and the clusters are manually analyzed to determine if they are malware and which family they belong to. With 850,000 apps in the android marketplace[1] compared to 150-300 malware families, our blacklisting approach seems more practical than PiggyApp's whitelisting strategy. However, blacklisting has it's own challenges. Malware code might vary in order to evade detection. Also, malware code is usually just a small part of the entire app. Our novel solutions to these challenges are described further in section3.2 and section3.3. This also motivated us to choose class-level granularity, since malware classes can easily be added to benign packages.

Alazab and colleagues [3] analyzed the familial classification of Android malware by various anti-virus products. Their results indicate that traditional anti-virus mechanisms are not able to correctly identify malicious Android applications. These results expose the need for automated familial classifiers.

Our method is based on module decoupling to separate injected code from legitimate code. There is a significant amount of literature on grouping units of code into modules [4, 19]. Elish et al. proposed a user-centric dependence analysis system for identifying malicious mobile apps [16].

vxClass [7] finds common modules to generate signatures in binaries. However, they assume the entire binary is a malware, while we go through the APK to find malware modules.

Kinable and Kostakis [18] researched malware classification based on class dependence graph clustering.

Batyuk et al. [6] developed a static analysis system for detecting and mitigating malicious activities. They are not trying to remove malware from the markets. Instead they seek to make malicious applications harmless by patching the dangerous code. A better understanding of Android malware phylogeny should provide insight into improving these mitigation techniques.

Wu et al. [29] provide a feature-based mechanism called DroidMat for detecting Android malware. DroidMat uses a number of features including requested permissions, intent passing, manifest files, and system calls. It uses these features as input for clustering and classification algorithms to efficiently label applications as malicious or benign. DroidMat gathers features statically, so it does not need to run applications being analyzed. While this system analyzes several features, it does not consider code modules or familial signatures. The results of our malware or benign test are comparable to the performance of DroidMat as shown in Table 4.5.

MAST [11] is a triage system that prioritizes suspicious applications over those unlikely to contain malware. MAST's rankings allowed detection of 95% of malware at the cost of analyzing 13% of the non-malicious applications on average across multiple markets.

DroidAnalytics [32] is a comprehensive Android malware analysis suite that uses multiple levels of features. It hashes API calls at method level to create method signatures and then hashes classes and APk to generate multilevel features.

3. Method

In this section we present a method to extract a signature to identify a repackaged malware family. The method takes as input a collection of repackaged malware prescreened by some other process as belonging to a family. Its output is a signature, as described later, that may be used to determine whether a new, unseen, APK belongs to the same repackaged malware family.

Our method is based on the following assumptions.

A1. The malware code does not interact with the code of its host application.

A2. To perform its malicious behavior the malware must make Android API calls.

A3. Two repackaged malware belonging to the same family perform similar malicious behavior.

A4. Each APK in the prescreened collection of repackaged malware was created by infecting different applications,(potentially, providing very different capabilities).

Assumption A1 follows directly from how repackaged malware are created. Assumption A2 is true for any application, for a program cannot perform anything significant without interacting with the OS, which on Android is accessed by Android API calls. Assumption A3 states our definition of what would constitute a malware family. It essentially states that if one repackaged malware abused premium services and the other stole mobile data they ought to be in different families. Assumption A4 may appear a bit strong, but, in practice, is not difficult to ensure. In essence, if all the APKs in a prescreened family are a result of repackaging different versions of the same legitimate application, the APKs will have a lot of similar code, besides the malware. Even though our method may separate all the similar modules, it would not be able to identify the malicious module. Thus, for best results, our method requires that the prescreened samples classified as belonging to the same malware family be from a variety of different applications. This would increase the probability that a module that is common to all (or most) of them belongs to the malware.

It's important to highlight one caveat of our assumptions. When the prescreening of repackaged APKs into families is done by a human analyst, as is the case with the APKs made available by NCSU's Malware Phylogeny [34], it may not be possible to ensure Assumption A3. The human analyst may place APKs in the same family based on similarity of their runtime behavior, or similarity of package names, or existence of some string in the code, or any other peculiar characteristics. Its our expectation that these peculiar characteristics that lead a malware analyst to group APKs in the same family would have materialized due to some similarity in their external behavior.

Our method of extracting signatures to recognize an Android malware family from a collection of repackaged malware families has the following steps:

1. **Partition an APK into modules.** Partition the classes of each APK of the prescreened collection into modules, such that there is high cohesion between classes in the same module, and low coupling between classes in different modules.
2. **Locate malware modules.** Identify a combination of modules, one from each APK, such that the selected combination has the highest inter-module similarity.
3. **Extract malware signature.** Use the selected combination of modules to create a signature for the family.

The following subsections describe each of the the above steps, and are followed by a subsection describing how the signatures may be used to scan a new APK. For clarity of presentation we also provide figures to illustrate the key steps.

3.1 Partition an APK into modules

We follow Zhou et al.'s [33] method to partition an APK into modules. Following their method, we create a Class Dependence Graph (CDG) from an APK, and then use their clustering to partition the classes. Our CDG, however, is different from that created by Zhou et al. The key difference is that a node in our CDG represents an Android class, whereas in Zhou et al.'s graph it represents a package. This difference is significant since it changes the level of granularity, and thus the level at which obfuscations are ignored. The package level granularity implies that the Zhou et al. method would work well when the benign code and the rider code stay in distinct packages. Whereas our class level granularity would yield good results when the classes remain distinct.

Figure 1 shows the process of partitioning an APK into modules. Given an APK, the first step is to disassemble it into smali, an assembly language for the dex format used by Dalvik, the Android Java VM implementation [28]. As is Java convention, there is one smali file per Java class. We use the function $\mathcal{C}$ to represent the operation of extracting classes from an APK.

$$\mathcal{C} : APK \rightarrow \mathcal{P}(Class)$$

The next step is to analyze each smali file of the APK to extract dependencies between its classes. Each class contains Dalvik instructions to implement the corresponding Java class, and hence contains information such as the name of the class, name of its super class, field names and their types, method names, their param-

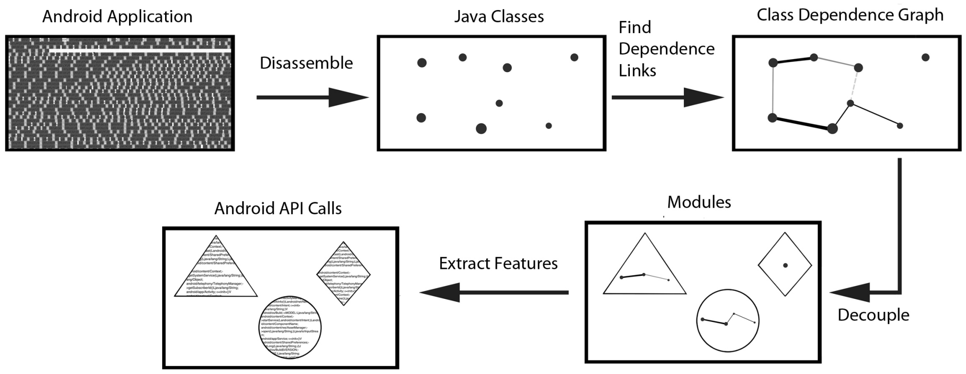


Figure 1. Creating Modules: The process of identifying modules through dependence relationships. Produces a set of Android API calls to represent each module.

eters, and their instructions. The smali language contains instructions to create objects of a class, to access the fields of an object, and to call methods.

The function $\mathcal{D}$ represents the dependency information extracted from a class.

$$\mathcal{D} : Class \times Class \times DepType \to Num$$

where $Dep\,Type$ is the set of dependence {inheritance, methodcall, fieldaccess}. For classes c_1 and c_2, and dependence type d, $\mathcal{D}(c_1, c_2, d)$ gives a count n with the following meaning. When d is $methodcall$, n gives the number of instructions in the smali file of class c_1 making calls to a method of class c_2. Similarly, when d is $fieldaccess$, n gives the number of instructions in the smali file of class c_1 accessing data members of class c_2. When d is $inheritance$, n is at most one, and it says whether c_1 is inherited from c_2.

The dependencies between classes are represented as a CDG. Each node of the CDG represents a single class of that APK. An edge in the graph represents dependence between two classes. The edge is annotated with a number representing the strength of the dependence computed using a weighted sum of all the dependencies computed by $\mathcal{D}$. The following expression gives the computation of strength of an edge from class c_1 to c_2.

$$S(c_1, c_2) = \sum_{d \in Dep\ Type} (\mathcal{D}(c_1, c_2, d) + \mathcal{D}(c_2, c_1, d)) \times w(d)$$

where $w : Dep\,Type \to Num$ is a weighting function.

The CDG of an APK A, which can also be represented as a matrix, is defined as follows:

$$CDG : APK \to Class \times Class \to Num$$

$$CDG(A) = \{(c_1, c_2) \mapsto S(c_1, c_2) | c_1, c_2 \in \mathcal{C}(A)\}$$

The CDG of an APK is used to partition its set of classes into modules. Since the Android development paradigm encourages development of classes around the individual behaviors exposed by a program, an Android program naturally partitions into modules. The programmer intended modules, however, are not always explicit in the class/directory hierarchy. We discover these modules from the CDG. An ideal partition would be one in which there is a strong dependence between classes within the same module and weak dependence between classes in different modules.

The general agglomerative clustering algorithm is a bottom up algorithm [17]. It starts with placing each class in a cluster by itself, and iteratively selects and merges a pair of clusters, until a stopping condition is reached. The algorithm requires two significant decisions: a) how to compute the similarity between a newly merged cluster and the other clusters and b) the stopping condition. Following Zhou et al.'s [33] algorithm, we compute the similarity between a cluster created by merging two clusters, say, x, y and z, by summing the similarities between the clusters being merged, i.e., sum of similarity between x and z and similarity between y and z. Our algorithm stops merging when no pair of clusters have similarity greater than some threshold.

Let $Agg : (Class \times Class \to Num) \to \mathcal{P}(\mathcal{P}(Class))$ represent the agglomerative clustering algorithm described above. Our algorithm for partitioning an APK into modules can then be described as follows:

$$Part : APK \to \mathcal{P}(Class)$$

$$Part(A) = Agg(CDG(A))$$

The clusters generated by Zhou et al.'s algorithm have the property that the cumulative sum of dependencies between the classes of two clusters is less than the selected threshold. With appropriately choosing the weighting function to create the CDG and the threshold, and under Assumption A1, stated earlier, the algorithm partitions the classes of an APK into modules with very weak intermodular coupling.

Though we use Zhou et al.'s [33] algorithm for identifying modules in an APK, we diverge from them on how the modules identified are used. Zhou et al.'s goal is to find a primary module, a module that provides the main functionality of the APK. In contrast, our goal is to find, in an APK, the module that provides the malware functionality, the subject of the next section.

3.2 Locate malware modules

In the previous step each APK is partitioned into modules, where a module is a set classes. We now describe our algorithm to identify which of the module(s) of an APK implements a malware functionality.

Figure 2 shows the steps involved in locating the malware modules. As stated earlier, the input to our algorithm is a collection of APKs, classified by some other means to belong to the same family. The output of the algorithm to locate malware modules is a vector consisting of one module from each APK. The vector contains modules that are most like each other in the various APKs. Under Assumption A4—that the APKs represent repackaged malware from the same family and that the original APKs implement different capabilities—there is a very high likelihood that the resulting vector selects a malicious module from each APK.

Let $\mathbf{F} = \{A_1, \ldots, A_n\}, n > 2$ be the set of APKs representing some repackaged malware family, where A_i represent the i^{th} APK in the ordered set. We use $\mathbf{P}(\mathbf{F})$ to denote the cross-product of the

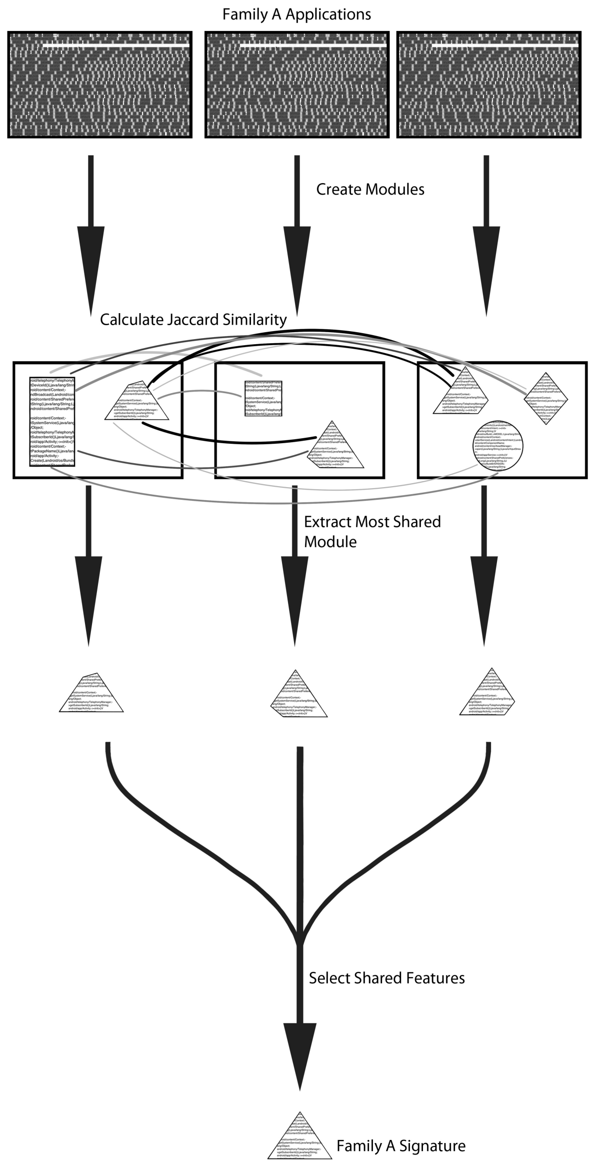


Figure 2. Familial Signature Extraction: Signature extraction is a three step process. First, modules for each application in a family are found. Second, a module most shared throughout the family is extracted from each application. Third, the features well represented in these modules are combined to form a familial signature.

modules of all the APKs in the family.

$$\mathbf{P}(\mathbf{F}) = Part(A_1) \times \ldots \times Part(A_n), \text{ where } A_i \in \mathbf{F}$$

Our goal is to select an element of $\mathbf{P}(\mathbf{F})$, a tuple consisting of one module from each APK, such that the modules selected represent a malicious module.

The first step is to create a representation of modules such that two modules from different APKs can be compared. If the modules represented versions of the same benign program we could compare two modules using the names of their classes. Since malware can (and do) rename their classes before repackaging, that two modules have classes with different names is not an indicator that they do not belong to the same malware. However, as per Assumption A3, two repackaged malware belonging to the same family perform the same malicious behavior, and to perform any significant behavior, malicious or otherwise, an APK must make calls to the Android API. Following this insight, we map a module to the set of API calls made by its classes, and use this set for further comparison.

Let $\pi : Class \rightarrow \mathcal{P}(Android\ API)$ represent the function that extracts the set of Android API calls made by a smali class file, and $[\Pi : \mathcal{P}(Class) \rightarrow \mathcal{P}(Android\ API)$ be the corresponding function lifted to apply on a set of classes. The similarity between two modules may be measured using the Jaccard Similarity of their set of API calls, as defined below:

$$jaccardSim : \mathcal{P}(API\ calls) \times \mathcal{P}(API\ calls) \rightarrow Num$$

$$jaccardSim(X,Y) = \mid X \cap Y \mid / \mid X \cup Y \mid$$

Since the APKs represent a repackaged malware family, we expect that all of the APKs will share at least one module. Since a malware may be modified before being repackaged, it is not necessary that the shared modules have the identical set of API calls, though, and hence the use of Jaccard Similarity.

Let $\overrightarrow{X}$ be a vector of modules, and $\overrightarrow{X}(i)$ denote the i^{th} element of $\overrightarrow{X}$. We define $JS(\overrightarrow{X})$ to represent the mutual pairwise similarity between all elements of $\overrightarrow{X}$, as follows:

$$JS(\overrightarrow{X}) = \sum_{i=1}^{|\overrightarrow{X}|} \sum_{j=1}^{|\overrightarrow{X}|} jaccardSim(\Pi(\overrightarrow{X}(i)), \Pi(\overrightarrow{X}(j)))$$

We wish to find a vector of modules $\overrightarrow{X} \in \mathbf{P}(\mathbf{F})$ with the highest pairwise similarity. There may, however, be more than one vector with the same, highest pairwise similarity. The answer we desire then belongs to the following set.

$$\mathbf{M}(\mathbf{F}) = \left\{ \overrightarrow{X} \mid \overrightarrow{X} \in \mathbf{P}(\mathbf{F}) \wedge JS(\overrightarrow{X}) = \max_{\overrightarrow{Y} \in \mathbf{P}(\mathbf{F})} JS(\overrightarrow{Y}) \right\}$$

The above specifies the set we wish to compute. An naïve implementation of this specification would be exponential since it explores all permutations of vectors representing modules of all APKs. Procedure $algM$ of Figure 3 presents an efficient implementation of an “approximation” of the above. We compute an approximation, and not the exact answer, because computing JS is expensive as well. Let there be n APKs, and a module of an APK has up to c API calls, then the cost of computing each JS is $O(n^2c)$, assuming computation of Jaccard similarity is linear on c. Instead we compute $\tilde{JS}$, an approximation of JS, described below.

$$\tilde{JS}(\overrightarrow{X}) = \sum_{j=2}^{|\overrightarrow{X}|} jaccardSim(\Pi(\overrightarrow{X}(1)), \Pi(\overrightarrow{X}(j)))$$

Whereas JS computes the pairwise Jaccard similarity between all the elements of a vector, $\tilde{JS}$ computes Jaccard similarity of only its first element with each of the other elements, and hence can be computed in $O(cn)$. Thus, the answer computed by procedure $algM$, Figure 3, belongs to the following set:

$$\tilde{\mathbf{M}}(\mathbf{F}) = \left\{ \overrightarrow{X} \mid \overrightarrow{X} \in \mathbf{P}(\mathbf{F}) \wedge \tilde{JS}(\overrightarrow{X}) = \max_{\overrightarrow{Y} \in \mathbf{P}(\mathbf{F})} \tilde{JS}(\overrightarrow{Y}) \right\}$$

The function $\tilde{\mathbf{M}}$ describes the set of modules that may belong to the malware family $\mathbf{F}$. This set will be non-empty since there will always be a vector with the maximal inter-module similarity.

The procedure $algM$, Figure 3, takes an APK A_1 and a list of APKs $Alist$ and returns a list of sets of classes (or $\perp$ if $Alist$ is

```
algM: $APK \times APK^* \rightarrow \mathcal{P}(Class)^*_\perp$

algM($A_1$, Alist) =
  let fn bestmatch($m_1$, $A_j$) =
    let $(\xi', m') := (0, \perp)$
    in
      foreach $m_j \in Part(A_j)$ do
        let $\xi$ = jaccardSim($\Pi(m_1), \Pi(m_j)$)
        in
          if $\xi > \xi'$ then
            $(\xi', m') := (\xi, m_j)$
    $(\xi', m')$
  fn algM'($m_1$, Alist) =
    if Alist = <> then $(0, <>)$
    else
      let $A_j$::Arest = Alist
        $(\xi_{rest}, M_{rest})$ = algM'($m_1$, Arest)
        $(\xi_j, m_j)$ = bestmatch($m_1$, $A_j$)
      in
        $(\xi_j + \xi_{rest}, m_j :: M_{rest})$
  ($max_\xi$, best_modules) := $(0, \perp)$
  in
    foreach $m_1 \in Part(A_1)$ do
      let ($\xi_i$, modules) = algM'($m_1$, Alist)
      in
        if $\xi_i > max_\xi$ then
          ($max_\xi$, best_modules) := ($\xi_i$, modules)
    best_modules
```

Figure 3. Algorithm for finding a malware module in a collection of APKs

empty) with the following property.

$$algM(A_1, (A_2, A_3, \dots, A_n)) \in \tilde{\mathbf{M}}((A_1, A_2, A_3, \dots, A_n))$$

Figure 3 presents the algorithm in a functional notation extended with the overloaded operator ':=' to provide an imperative declaration and an imperative assignment. The procedure $algM$ contains two internal procedures $bestMatch$ and $algM'$. Procedure $bestMatch$ finds the module of A_j that best matches module m_1 (of A_1). The match is found by enumerating all of the modules m_j of A_j and computing the Jaccard similarity between the set of API calls of m_1 and of module m_j. Besides returning the best matching module, Procedure $bestMatch$ also returns the corresponding Jaccard similarity. Procedure $algM'$ applies $bestMatch$ on each APK of $Alist$ and returns a list of modules that best match module m_1 of A_1 and the sum of the Jaccard similarities of m_1 with each of the matched modules. Finally, the main body of $algM$ iterates over each module m_1 of A_1 to find the best modules. It then returns the m_1 with the highest similarity along with the corresponding best matched modules.

That $algM$ finds a collection of modules that best match some module in A_1, an arbitrary APK chosen as the first one, is obvious because the module returned from each APK is the best matching module in that APK. The collection of modules with the highest sum of similarities would thus be the best such collection.

3.3 Extract malware signature

Once we have identified the corresponding malware modules in each repackaged malware APK, we are ready to extract a signature to represent that family. We characterize a malware family by a set of API calls. Any APK containing a module with API calls similar to those of the signature could be suspected as a member of that family.

The following equations present one method of generating the signature. Given an API call, x, and a vector of modules, $\overrightarrow{X}$, the function $DF(x, \overrightarrow{X})$ computes the 'document frequency' of x, that is, the fraction of modules in $\overrightarrow{X}$ that call the API x.

$$DF(x, \overrightarrow{X}) = \frac{\sum_{i \in |\overrightarrow{X}|} \left| \{x\} \cap \Pi(\overrightarrow{X}(i)) \right|}{\left| \overrightarrow{X} \right|}$$

Since $\left| \overrightarrow{X} \right| = n$, the number of APKs, $DF(x, \overrightarrow{X})$ gives the fraction of APKs whose module in $\overrightarrow{X}$ call the API x. The following function then defines the signature, a set of API calls, that represent the malware family. Function Sig_1 selects the set of API calls that are used in more than t percent of APKs' malware modules.

$$Sig_1(\mathbf{F}) = \{x | x \in \Pi(\bigcup \tilde{\mathbf{M}}(\mathbf{F})), DF(x, \tilde{\mathbf{M}}(\mathbf{F})) > t\}$$

where $\bigcup \tilde{\mathbf{M}}(\mathbf{F})$ represents the union of all the modules in all the vectors computed by $\tilde{\mathbf{M}}(\mathbf{F})$.

Though the above captures the necessary intuition, it is not reliable when the number of APKs in a family is small because the presence or absence of the API even in a single module has proportionally greater effect. We therefore use maximum term-frequency frequency normalization [20] by normalizing the count of occurrence of each API call in a vector by the largest number of occurrences of any API call in that vector. The resulting function, presented below, produces more stable signatures.

$$SDF(x, \overrightarrow{X}) = \frac{DF(x, \overrightarrow{X})}{\max_{y \in \bigcup \overrightarrow{X}} DF(y, \overrightarrow{X})}$$

$$Sig_2(\mathbf{F}) = \{x | x \in \Pi(\bigcup \tilde{\mathbf{M}}(\mathbf{F})), SDF(x, \tilde{\mathbf{M}}(\mathbf{F})) > t\}$$

The function Sig_2 summarizes our algorithm for computing the signature of an Android malware family represented by the repackaged malware APKs $\mathbf{F}$.

3.4 Analyzing Unlabeled APKs

Figure 4 illustrates the process of checking an unlabeled APK for matches to family signatures and is summarized by the following function:

$$Scan : APK \times \mathcal{P}(APK) \rightarrow Bool$$

$$Scan(A, (F)) = \vee_{x \in PartA} JaccardSim(\Pi(x), Sig_2(\mathbf{F})) > t$$

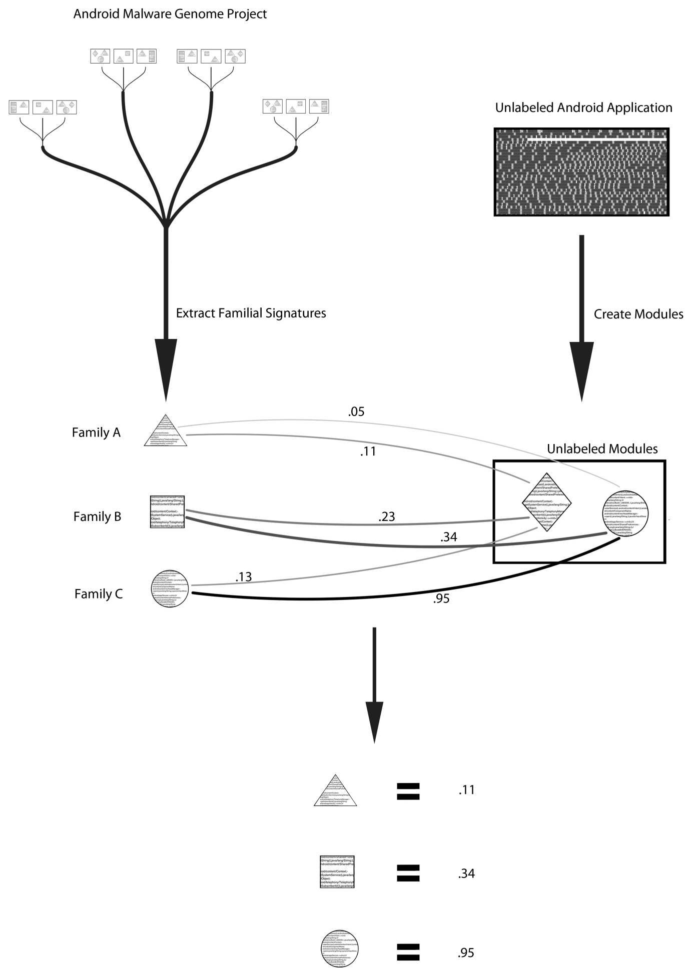
And APK A belongs to the malware family $\mathbf{F}$ if the Jaccard Similarity of the API called by any of its modules against the API calls in the signature of $\mathbf{F}$ is greater than some threshold t.

Besides scanning a new APK, our algorithm can also be used to verify whether the pre-scanned collection $\mathbf{F}$ indeed represents malware from the same family. This can be done by computing signatures using multiple random sub-samples of $\mathbf{F}$ and computing their Jaccard Similarity. A similarity value of 1 is a good indicator that the collection represents the same family.

Since our algorithm can non-trivially be extended to identify the malware module(s) in a scanned APK, it can also be used in triage to select the modules that ought to receive greater attention by a human analyst or undergo more expensive analysis.

4. Experiment

In this section we present the results of a controlled experiment designed to estimate the performance of the proposed method when deployed in the real world. We envision two applications of the method: 1) malware detection, i.e., identify whether a new, unknown app is malware or benign, and 2) familial classification, i.e.,



Android Malware Genome Project
Unlabeled Android Application
Extract Familial Signatures
Create Modules
Family A
Family B
Family C
Unlabeled Modules
.05
.11
.23
.34
.13
.95
.11
.34
.95

Figure 4. Analysis of an Unlabeled APK: Illustration of the entire method. Familial signatures are extracted and compared to the modules of an unlabeled APK. In this case, the unlabeled APK would be flagged by our method as a potential member of Family C.

determine the malware family of an app known to be malware. We formulate the following two tests to evaluate the expected effectiveness of our method for each of these applications.

T1. **Is Malware? test**: In this test we evaluate how well the signatures for individual malware families can be used to discriminate between a malware and benign Android app. According to the function $Scan$ in Section 3.4 an app belongs to a family if the similarity of any of its module exceeds a certain threshold. We consider an unknown app to be malware if it is determined to belong to any of the malware families. It is considered benign otherwise. In this test, classifying a malware as benign would be an error, as would the reverse classification.

T2. **Membership test**: In this test we evaluate how well a known malware APK is classified in the correct family. It uses only the first step from the previous test–determining the family of an Android APK known to be malware. In this experiment, assigning an app to the wrong family would be an error as would not assigning it to any family.

In the following sections, we describe the dataset used, the experiment procedure, and the results along with our discussion.

4.1 Dataset

The selection of dataset is an important and challenging issue in any controlled experiment. It is more challenging when performing any work in malware analysis. While it is easy to gather a large collection of malware from a variety of sources, such as, Virus Total, bittorrent, individual anti-malware researchers, anti-malware companies, or using your own honeypots, such a large collection of data is not sufficient for controlled experiments. For such experiments, besides the malware samples, we also need the relevant ground truth necessary to validate the results of a test. For instance, for Test T1 we need a collection of APKs that are definitely malicious and another collection that is definitely benign. Although one can download terabytes of APKs from a variety of sources, one cannot say for certain that an arbitrary APK is definitely benign, definitely malware, or that it definitely belongs to a certain malware family. In the absence of such certainty the results of any controlled experiment are difficult to use when projecting the expected performance when deployed.

Indeed in our lab we have a large collection of APKs—over 5,000 APKs acquired from a leading anti-malware company and over 1,000 APKs from bittorrent. While it would be reasonable to expect the collection from the anti-malware company to be malware, our experience says that it is not a certainty. When it comes to benign APKs the confidence dips lower. We cannot be certain that an APK downloaded from any marketplace, including Google Play, is indeed benign. We can have high confidence, but not certainty. This becomes all the more challenging for Test T2 for it also requires classification of malware into families, and malware APKs classified in families are still harder to find.

We err on the side of less data with higher confidence on its ground truth, as against more data with less confidence. Besides having higher confidence in the results of the experiment, such data also offers another benefit. When the confidence in the data is high it becomes easier to investigate the errors in classification to determine the root cause. This benefit becomes obvious from our discussions, below.

For the malware collection we use the data provided very graciously by the AMGP [34]. As of the time this study was performed the AMGP project had compiled 1,260 Android apps, all malware, and categorized them in 49 families. The apps in this collection have been compiled by researchers for the explicit purpose of creating a dataset for experimentation. Thus it also serves as a good benchmark to compare various works. The malware collection was gathered from a variety of markets and then painstakingly classified into various malware families by hand.

Of the AMGP collection, 86% of the apps are piggybacked malware and the others are stand-alone malware apps. Since our method is explicitly aimed at classifying piggybacked apps, we first removed all the stand-alone malware from the dataset. We next removed any family of malware from our set that contained less than 10 malware instances. This was necessary since we use 10-fold cross-validation for estimating performance, thus requiring at least 10 samples in a family. These culls left us with a set of 1,052 piggybacked Android malware from 14 families. Table 4.1 lists each family and the number of malware instances representing it in our experiment. As evident from the table our collection is not balanced. It would strengthen the results of the study if the distribution in the collection represents the distribution in the real-world. We, however, do not have any specific data that can be used to infer how the distribution of malware families in our dataset relates to the distribution in the real-world.

We took a very cautious approach in choosing the benign apps. Given our goal is to help marketplaces scan apps for malware it would not be appropriate to download arbitrary apps from any marketplace, because doing so would contaminate the results of the

Android Malware Family	No of APKs
ADRD	22
AnserverBot	187
BaseBridge	122
DroidDream	16
DroidDreamLight	46
DroidKungFu1	34
DroidKungFu2	30
DroidKungFu3	309
DroidKungFu4	96
Geinimi	69
GoldDream	35
JSMSHider	16
Pjapps	16
Zsone	12
Total	1,052

Table 1. Dataset Description

test. After exploring various options we chose to use APKs downloaded from the Android phone of a student who was actively using those apps. This method limited us to a small collection of 48 benign apps including those for Facebook, Skype, PocketCloud and some games amongst others. However, it did ensure that we had higher confidence that the collection was benign. Another alternative would be to get apps directly from some trusted developers, or download the apps from only trusted developers from Google Play. Since the apps in our collection were downloaded from multiple sources we believe the data, though small, is a better representation of benign apps in arbitrary marketplaces.

4.2 Procedure

The dataset, described in Table 4.1, contains APK files. As described in Section 3, these APK files first need to be partitioned into modules, as described in Section 3.1. This process requires a sequence of operations as shown in Figure 1. For the purpose of the experiment, we used android-apktool in conjunction with custom scripts that condition the apktool output as suitable to our needs. These scripts extract the set of classes in each APK, representing the function $\mathcal{C}$, and the various relations between classes, representing the function $\mathcal{D}$. Next, we build the CDG, represented by the function S. Since the CDG computation is influenced by Zhou et al. [33], we also use their weighting function w, described below:

$$w = \{inheritance \mapsto 10, methodcall \mapsto 2, fieldaccess \mapsto 1\}$$

This is followed by the agglomerative clustering algorithm represented by $Agg : (Class \times Class \rightarrow Num) \rightarrow \mathcal{P}(\mathcal{P}(Class))$, using a minimum threshold of 5 for merging clusters.

The next step is to locate the malware modules of each Android malware family using algorithm $algM$ given in Figure 3. The malware modules for each each family are then used to extract a signature for that family, as described in Section 3.3 using threshold $t = 0.5$ in function Sig_2.

The real-world performance of the entire process was estimated using 10-fold cross validation [9]. Each of the malware families of Table 4.1 were partitioned into 10 (almost) equal size sets, called folds. Ten iterations of each test were performed, once for each fold. In each iteration, APKs in the selected fold were held out as testing data, and the remaining APKs were used to compute the signature of each family. Finally, instead of computing the function $Scan$ of Section 3.4, the highest Jaccard similarity between the modules of each of the held out APK with the signature of each family was computed and recorded. In this final step of each fold the highest Jaccard similarity of modules of each of the 48 benign apps with the signature of each family was also computed and recorded.

The above described 10-fold training and testing process yields a similarity matrix that has one column per malware family, one row per malware APK, and 10 rows per benign APK. The cell in the matrix contains the highest Jaccard similarity of a module of the corresponding the APK (row) and the family signature (column). There is only one row per malware APK because, as per K-fold cross-validation protocol, each APK is used only once for testing. There are 10 rows per benign APK because all of the benign APKs are used for testing in each of the 10 folds.

The similarity matrix is then used for evaluating Test T1 with varying thresholds for similarity value. Each row represents an 'unknown' APK, as it was not used in the training. If any of the similarity values in a row is above a given threshold that APK is assigned the label 'malware' (positive class), otherwise the label 'benign' (negative class) is assigned. If our assignment matches the actual classification, the match is categorized as true-positive (TP) or true-negative (TN). If a malware APK (positive) is incorrectly assigned the label benign (negative), the mismatch is categorized as false-negative (FN). On the other hand, if a benign APK is incorrectly assigned the label malware, the mismatch is categorized as false-positive (FP). Thus, given some threshold, each row in the similarity matrix is categorized into TP, FN, TN, and FP.

The similarity matrix, without the 480 rows for benign apps, is used for evaluating Test T2. The benign apps are not used since the purpose of the test is to determine accurate classification of known malware into a malware family. The interpretation of positive and negative samples in Test T2 is different from Test T1, but consistent with the general usage for binary classifiers. In this test each column represents a classifier for a single malware family. For each classifier (column), all the samples (rows) belonging to the same family as the classifier are considered positive, and all other samples are considered negative. Thus, the categories TP, FN, TN, and FP represent matches and mismatches accordingly.

We performed the two tests using the thresholds 0.5, 0.6, 0.7, 0.8, 0.9, and 0.98. For each threshold we categorize the results and then count the raw numbers of TPs, FPs, FNs, and FPs, and use these measures to compute the following classic summary measures:

Precision	TP / (TP + FP)
Recall	TP / (TP + FN)
True Negative Rate	TN / (TN + FP)
Accuracy	(TP + TN) / (TP + FN + TN + FP)
False Positive Rate	1 - TNR

Precision measures the probability that an APK in the test set labeled as positive by the algorithm is actually positive. Recall measures the probability that a positive sample in the test set is actually identified positive. True Negative Rate (TNR), also called specificity, measures the probability that a negative sample in the test set is actually identified as negative. Accuracy measures the chance that the label assigned by the method, whether positive or negative, is actually correct. False Positive Rate (FPR) is the converse of TNR. It measures the probability that negative sample is misclassified as positive. An ideal classifier has high recall and low FPR, and since we have high recall, our analysis of errors is focused on the FPR.

The interpretation of the above measures for Test T1 and T2 change depending on the meaning of the positive and negative labels. Their specific meaning with respect to a test are presented when the results are discussed.

	Similarity Thresholds					
	0. 5	0.6	0.7	0.8	0.9	0.98
Precision	0.7241	0.8127	0.9732	0.9726	0.9721	0.9700
Recall	0.9502	0.9462	0.9273	0.9071	0.8896	0.8277
True Negative Rate	0.4396	0.6625	0.9604	0.9604	0.9604	0.9604
Accuracy	0.7498	0.8348	0.9403	0.9280	0.9174	0.8798

Table 3. Results- Is Malware?

	Similarity Thresholds					
	0.5	0.6	0.7	0.8	0.9	0.98
True Positives	706	703	689	674	661	615
False Negatives	37	40	54	69	82	128
True Negatives	211	318	461	461	461	461
False Positives	269	162	19	19	19	19

Table 2. Observations- Is Malware?

4.3 Results and Discussion: Is Malware? test

As mentioned above, DroidLegacy is not able to analyze native code. DroidKungFu3 is known to be heavily using native code and hence we had to remove the 309 samples belonging to the family, from the dataset described in Table 4.1. Table 4.2 presents the raw counts of the matches and mismatches for varying values of similarity threshold for Test T1, and Table 4.3 presents the four summary measures. In our design the positive and negative samples are not balanced; there are 743 positives and only 480 negatives (10 folds $\times$ 48 benign). In such a situation recall and TNR (or FPR) are better measures since each uses counts of either the positive or the negative test data. This ensures that the biases do not interfere with the analysis.

To focus our discussion we first choose a threshold that has ideal trade-off between recall and FPR. For the data presented, 0.7 is the ideal threshold as the classification of negative samples stabilizes at that threshold. As evident from Table 4.2, the TN and FP counts do not change at thresholds higher than 0.7. At this threshold there are 19 FPs (4% FPR), which is slightly high. And this error jumps significantly at lower thresholds. Also at this threshold the recall is 0.9273, implying that 93% of the malware is still correctly classified. At the similarity threshold of 0.7, Table 4.3 shows a precision of 0.9732, implying that given an APK labeled as malware by the algorithm, there is a 97% chance that the APK was actually a malware. While a high precision implies higher certainty when a sample is labeled as malware, a high recall implies higher certainty that when a sample is marked benign, it actually is benign since it rarely misses a malware in the dataset. Finally, at this threshold the accuracy is 0.9403, indicating that the probability that the algorithm labels an arbitrary APK correctly is 94%, a performance that may be acceptable in scanning Android marketplaces for malicious apps.

Investigating the False Positives Rates

We now focus on the high FPR and present the result of investigating the reasons. The Table 4.3 shows that TNR, the converse of FPR, is 0.9604, implying that the probability the algorithm labels a benign APK as benign is 96%. This is a lower rate than expected and results due to the abnormally high false positives.

Per our investigation all the false positives in this test fall in two categories. The first class of errors contributing to false positives is in the use of advertisement libraries by both malware and benign apps. In our dataset the Zsone family references an advertisement also referenced by the benign application Wyse PocketCloud. As a result this app is classified as malware. Indeed if one were to consider the advertisement capability of Zsone as malicious, it may be appropriate to classify Wyse PocketCloud as malware too. But that may not be desired since the free versions of many legitimate applications serve advertisements. In our dataset of 48 benign apps, Wyse PocketCloud was the only app misclassified by the ZSone classifier and it was misclassified in each of the 10 folds causing 10 of the 19 total False Positives.

The second and final reason for false positive appear to be a statistical anomaly. In all machine learning systems, one must account for the possibility of poorly labeled training data. Fortunately, 10 fold cross validation, by it's very design, allows us to take that possibility into consideration. The classifier for the DroidKungFu-4 family misclassified 9 out of 48 benigns, and all the misclassifications were in the same fold. This accounts for the result of poorly classified training data. Since there are 10 folds, when deployed in the field there is 1 in 10 chance we would have deployed that classifier, thus reducing the expected probability of misclassifying a known benign due to DroidKungFu-4 to 0.01875 (=0.1*9/48).

4.4 Results and Discussion: Membership test

For the Membership test the various measures are computed separately for the classifier for each family and also cumulatively for the entire corpus. The raw counts of TP & FN and TN & FP for familial classification for each classifier are presented graphically in Figures 5 and 6. Once again we notice that the data is unbalanced; skewed to the negative class. This happens because, with reference to the classifier for a family, all malware not in the family are negative samples. As per Table 4.1, the 14 families have from 12 to 309 positive samples. Thus, the negative samples range from 743 to 1,040.

Investigating the unexpectedly high False Positives

Figure 6 shows a high false positive count for the DroidKungFu* families. In depth analysis revealed that most of the samples belonging to one DroidKungFu* family would match positive to other DroidKungFu* family signature. DroidLegacy, thus, successfully brought forward the fact that these families share modules and, possibly, common origin. The names of these families provide a good indication that the person who assigned the names to these families also considered them to be related [34], reinforcing our findings. It is important to note here that AMGP [34] classified the malware based on functional similarities. Our classifier, however, look for module similarity. Thus, while it might make sense to classify DroidKungFu* families as separate families based on functionality, but our experiments suggest we treat them as variants of a single family.

Since the four DroidKungFu* families are really variants of a single family, it may be more appropriate to treat them as a single family. We ran the test again treating the four DroidKungFu* families as one DroidKungFu family, but with four signatures. A malware APK is classified to belong to the DroidKungFu family if it was classified to be in any of the four related families using their corresponding signature. Table 4.4 presents the summary measure after this change. Again, as can be seen from the summary measures for the entire corpus, presented in Table 4.4, the similarity

	Similarity Thresholds					
	0.5	0.6	0.7	0.8	0.9	0.98
True Positives	1000	1000	991	973	957	905
False Negatives	52	52	61	79	95	147
True Negatives	9916	9917	10376	10389	10389	10389
False Positives	604	603	144	131	131	131
Precision	0.6234	0.6238	0.8731	0.8813	0.8796	0.8736
Recall	0.9506	0.9506	0.9420	0.9249	0.9096	0.8603
True Negative Rate	0.9426	0.9427	0.9863	0.9875	0.9875	0.9875
Accuracy	0.9433	0.9434	0.9823	0.9818	0.9805	0.9760

Table 4. Results - Membership Test Summary after merging DroidKungFu

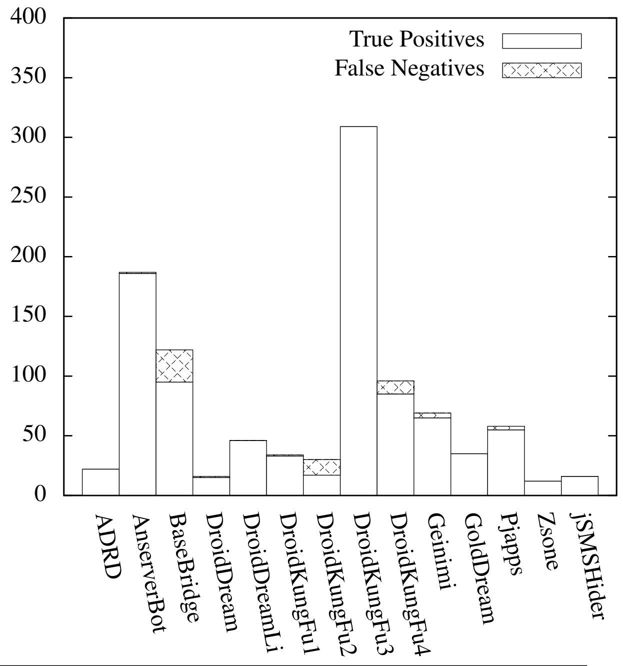


Figure 5. Results - Membership Test

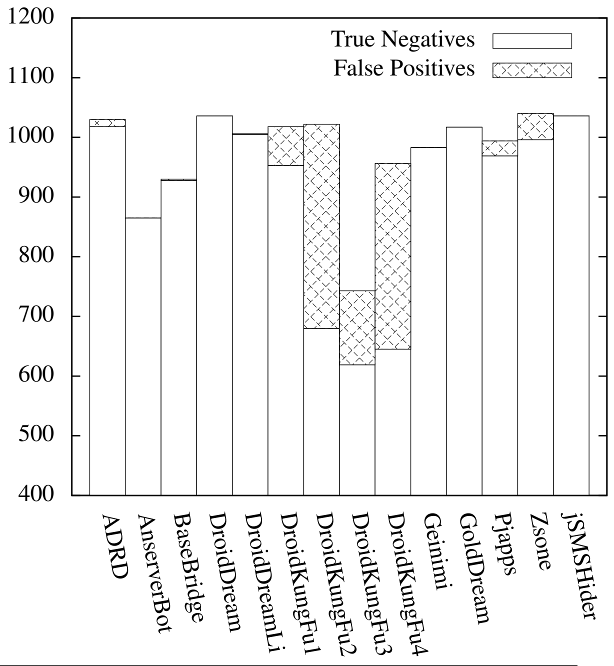


Figure 6. Results - Membership Test

threshold 0.7 strikes a good balance between recall and FPR. The recall of 0.9420 implies that there is a 94% certainty that a sample belonging to a family F is indeed a match to that family's signature. Thus, at the threshold 0.7 most samples are correctly picked up by the classifier of the family to which they belong. A false positive match in this test implies that malware of one family has a module that has high similarity with the signature of another family. According to the table the FPR is 0.0137 (i.e., 1-0.9863) which may be considered low enough.

The summary data in Table 4.4 shows that the precision for the entire corpus is 0.8731, at threshold 0.7. In Test T2, precision gives the probability (87%) that an arbitrary sample labeled by our algorithm to belong to a family F, actually belongs to that family F. This is clearly not desirable. This low precision is the result of the skewed dataset. Even though the false positive rate is low, the number of false positives (144) are relatively high. There is a logical explanation for this high number. Since there is an independent classifier for each family, its possible that the same sample is classified in multiple families. In such case, the sample may contribute to false positives to multiple families, inflating the error, discussed below.

The cumulative accuracy of familial classification of the entire corpus is 0.9823, indicating that there is a 98% chance that the algorithm correctly labels a randomly selected APK as belonging to or not belonging to a family.

4.5 Summary

The results are summarized in the ROC curves shown in Figure 7. It plots the Recall vs FPR (1-TNR) separately for each classifier for various thresholds. The the top left corner of an ROC curve is the desired region. The graph shows that most of the classifiers have points on the graph close to the top left corner, indicating that there is a threshold for which the classifier may be expected to perform with recall close to 1 and FPR close to 0. The classifier for DroidKungFu2 is an exception. It performed very poorly.

While our primary goal was to develop a familial classifier to classify malware samples into families, the signatures generated also allowed us to also weed out benign samples from the dataset.

It is instructive to compare the results of Test T1 with those reported in prior work on classifying Android apps as malware or benign. Table 4.5 compares our performance against Androguard [15] and DroidMat [29]. In general, Precision and Recall vary in-

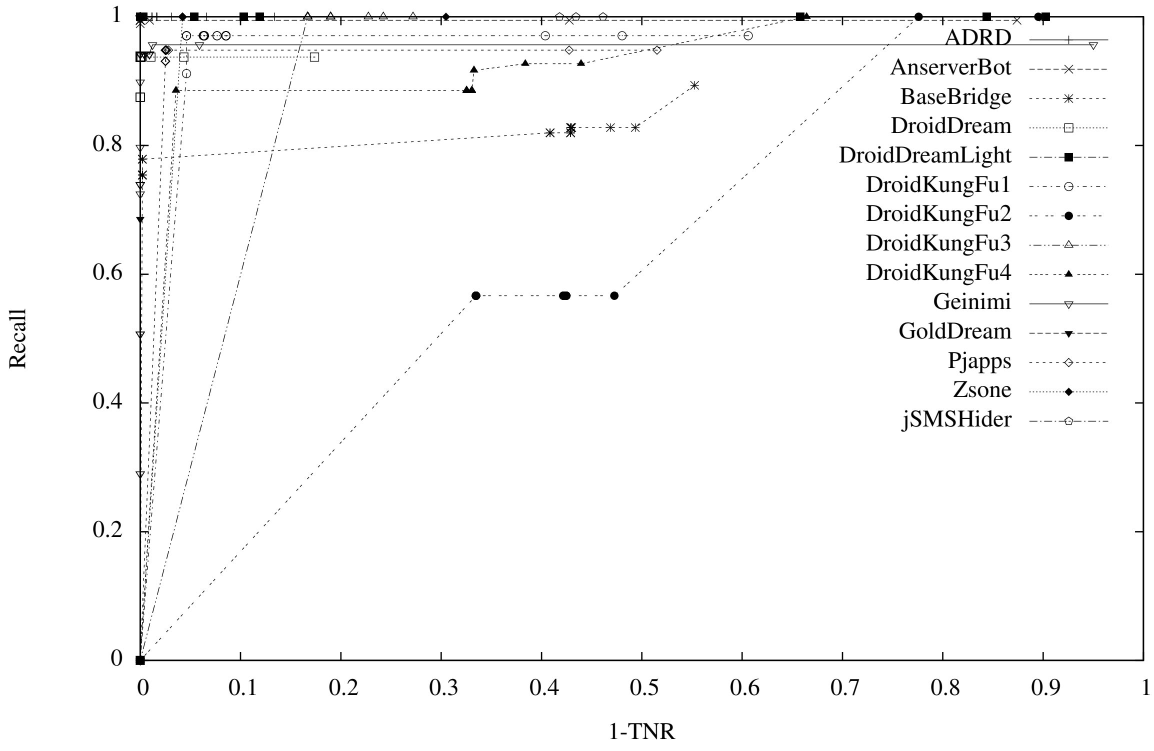


Figure 7. ROC Curve from the Membership Test

versely. The trade-off is weighed upon the intended goal of the system. DroidMat and Androguard favor Precision over Recall because they want to identify as many malware apps as possible without labeling any benign as malware. We favor Recall over Precision because we can accept a few benign apps labeled as malware but we do not want to miss out on any malware because we expect that all apps labeled as malware would go through further analysis, such as, by a human analyst. F-measure, the harmonic mean of Precision and Recall, is sometimes used as a single measure to compare performance. Table 4.5 shows our method outperforms the other two on the F-measure.

Since our malware distribution is not uniform across the various families, see Table 4.1, the signatures are also created using varying number of samples, ranging from 9 to 30 APKs. As the number of samples from a class increases, the intersection of APIs common in the malware modules is likely to decrease. It is indeed possible that some of the signatures may have overfit the data. A study of the effect of the number of samples in the training data on quality the signature generated would be worthy of future investigation.

Overall we conclude that

1. Our malware detection method provides very good performance in discriminating malware from benign for piggybacked malware families that do not use native code.
2. Our malware family classification method correctly places malware in families when the families are independent. On the other hand the method can also be used to find related families, and, by treating them all as a single family with multiple classifiers, the overall classification performance can be improved.

Target	Accuracy	Precision	Recall	F-measure
Androguard	0.9304	**0.9916**	0.4958	0.6611
DroidMat	**0.9787**	0.9674	0.8739	0.9183
DroidLegacy	0.9403	0.9732	**0.9273**	**0.9497**

Table 5. Comparing DroidLegacy to other systems

5. Conclusions

The ease with which one can introduce new code in an existing Android app has led to a proliferation of piggybacked Android malware, where malware code takes a free ride on benign code. While there has been prior work on finding piggybacked malware in plagiarized copies of a given benign Android app [33], there has not been any work in creating familial signatures for detecting Android malware. We have presented an automated method for extracting familial signatures for Android malware and a method to detect the malware in any Android app. Our results for detecting Android malware are comparable to other current works and, to the best of our knowledge, we introduce the first method for automated familial classification of Android APKs. Our method is insensitive to many obfuscations used by Android malware, such as, renaming of packages and classes, and polymorphic and metamorphic transformations. More work needs to be done for addressing other obfuscations, such as, dynamic generation of Dalvik code and the use of native code. In spite of these limitations, the method is still valuable since it raises the bar for malware authors, pushing them into using more difficult and expensive obfuscations.

One of our assumptions, A4, is that instances of malware families would infect different legitimate applications such that the most common code in a family would be malicious. However, we have discovered cases in which some legitimate advertiser or supporting library is found to be the most shared module within a malware family. While this is sometimes effective as a signature it is not a desirable result. We need to investigate the feasibility of a white list of common advertisers and supporting libraries.

Acknowledgments

This research work was sponsored in part by funds from Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract number FA8750-10-C-0171. We thank artist Jennifer Brown for figures 1, 2 and 4.

References

[1] Android statistics, 2013. URL `http://www.appbrain.com/stats/number-of-android-apps`.

[2] How many apps are in each app store?, 2013. URL `http://www.pureoxygenmobile.com/how-many-apps-in-each-app-store/`.

[3] M. Alazab, V. Monsamy, L. Batten, P. Lantz, and R. Tian. Analysis of malicious and benign android applications. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 608–616. IEEE, 2012.

[4] N. Anquetil, C. Fourrier, and T. C. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE'99: Proceedings of the Sixth Working Conference on Reverse Engineering (Washington, DC, USA),IEEE Computer Society*, page 235, 1999.

[5] M. Ballano. Android malware, Oct. 2012. URL `http://www.itu.int/ITU-D/eur/rf/cybersecurity/presentations/symantec-itu\_mobile.pdf`.

[6] L. Batyuk, M. Herpich, S. A. Camtepe, K. Raddatz, A. Schmidt, and S. Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*, pages 66–72. IEEE, 2011.

[7] C. Blichmann. Vxclass - clustering malwares, generating signatures, 2010. URL `http://www.zynamics.com/downloads/inbot10-vxclass.pdf`.

[8] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 36–43. IEEE, 2002.

[9] P. Burman. A comparative study of ordinary cross-validation, v-fold cross-validation and the repeated learning-testing methods. *Biometrika*, 76(3):503–514, 1989.

[10] S. Burrows, S. M. Tahaghoghi, and J. Zobel. Efficient plagiarism detection for large code repositories. *Software: Practice and Experience*, 37(2):151–175, 2007.

[11] S. Chakradeo, B. Reaves, P. Traynor, and W. Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[12] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46. IEEE, 2005.

[13] F. Cohen. Computer viruses: theory and experiments. *Computers & security*, 6(1):22–35, 1987.

[14] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security–ESORICS 2012*, pages 37–54. Springer, 2012.

[15] A. Desnos and G. Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 2011. URL `http://code.google.com/p/androguard/`.

[16] K. O. Elish, D. D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. *MOST 2012: MOBILE SECURITY TECHNOLOGIES 2012*, 2012.

[17] A. Jain and R. Dubes. *Algorithms for clustering data*. Prentice-Hall Advanced Reference Series. Prentice Hall PTR, 1988. ISBN 9780130222787. URL `http://books.google.com/books?id=7eBQAAAAMAAJ`.

[18] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 7(4):233–245, 2011.

[19] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Program Comprehension, 1998. IWPC'98. Proceedings., 6th International Workshop on*, pages 45–52. IEEE, 1998.

[20] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2008.

[21] S. Mody. 'i am not the D'r.0,1d you are looking for': an analysis of Android malware obfuscation. In *Proceedings of 23rd Virus Bulletin International Conference*. Virus Bulletin, Oct. 2013.

[22] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[23] P. Schulz. Code protection in android. Technical report, University of Bonn, 2012. URL `http://net.cs.uni-bonn.de/fileadmin/user\_upload/plohmann/2012-Schulz-Code\_Protection\_in\_Android.pdf`.

[24] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer. Google android: A comprehensive security assessment. *IEEE Security Privacy*, 8(2):35–44, 2010. ISSN 1540-7993. .

[25] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, page 1019, San Francisco, CA, 2011. USENIX Association.

[26] A. Walenstein and A. Lakhotia. The software similarity problem in malware analysis. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, 2007.

[27] A. Walenstein, A. Lakhotia, and R. Koschke. The second international workshop on detection of software clones: Workshop report. *ACM SIGSOFT Software Engineering Notes*, 29(2):1–5, 2004.

[28] R. Winiewski and C. Tumbleson. android-apktool - a tool for reverse engineering android APK files, 2013. URL `https://code.google.com/p/android-apktool/`.

[29] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE, 2012.

[30] Y. Ye, D. Wang, T. Li, and D. Ye. Imds: intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 1043–1047, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-609-7. .

[31] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[32] M. Zheng, M. Sun, and J. Lui. DroidAnalytics: a signature based analytic system to collect, extract, analyze and associate android malware. In *Proceedings of The 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 13), Melbourne, Australia, July 2013.*, 2013.

[33] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, page 185196, 2013.

[34] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, page 95109, 2012.