DroidClassifier: Efficient Adaptive Mining of Application-Layer Header for Classifying Android Malware

Zhiqiang Li[1], Lichao Sun[1], Qiben Yan[1], Witawas Srisa-an[1], Zhenxiang Chen[2]

[1] University of Nebraska–Lincoln, Lincoln NE 68588, USA
{zli,lsun,qyan,witty}@cse.unl.edu
[2] University of Jinan, Jinan, Shandong, 250022, China
czx.ujn@gmail.com

Abstract. A recent report has shown that there are more than 5,000 malicious applications created for Android devices each day. This creates a need for researchers to develop effective and efficient malware classification and detection approaches. To address this need, we introduce DroidClassifier: a systematic framework for classifying network traffic generated by mobile malware. Our approach utilizes network traffic analysis to construct multiple models in an automated fashion using a supervised method over a set of labeled malware network traffic (the training dataset). Each model is built by extracting common identifiers from multiple HTTP header fields. Adaptive thresholds are designed to capture the disparate characteristics of different malware families. Clustering is then used to improve the classification efficiency. Finally, we aggregate the multiple models to construct a holistic model to conduct cluster-level malware classification. We then perform a comprehensive evaluation of DroidClassifier by using 706 malware samples as the training set and 657 malware samples and 5,215 benign apps as the testing set. Collectively, these malicious and benign apps generate 17,949 network flows. The results show that DroidClassifier successfully identifies over 90% of different families of malware with more than 90% accuracy with accessible computational cost. Thus, DroidClassifier can facilitate network management in a large network, and enable unobtrusive detection of mobile malware. By focusing on analyzing network behaviors, we expect DroidClassifier to work with reasonable accuracy for other mobile platforms such as iOS and Windows Mobile as well.

Keywords: Mobile Security, Android Malware Detection, Malware Classification, HTTP Network Traffic

1 Introduction

Android is currently the most popular smart-mobile device operating system in the world, holding about 80% of world-wide market share. Due to their popularity and platform openness, Android devices, unfortunately, have also been

subjected to a marked increase in the number of malware and vulnerability exploits targeting them. According to a recent study from F-Secure Labs, there are at least 275 new families (or new variants of known families) of malware that currently target Android [8]. On the contrary, only one new threat family on iOS was reported.

As smart-mobile devices gradually become the preferred end-hosts for accessing the Internet, network traffic of mobile apps has been utilized to identify mobile applications to facilitate network management tasks [33]. However, the methods of identifying benign mobile applications fall short when dealing with mobile malware, due to the unique traffic characteristics of malicious applications. From our observation, malicious attacks by mobile malware often involve network connectivity. Network connection has been utilized to launch attack activities or steal sensitive personal information. As a result, studying network traffic going into or coming out of Android devices can yield unique insights about the attack origination and patterns.

In this paper, we present DroidClassifier, a systematic framework for classifying and detecting malicious network traffic produced by Android malicious apps. Our work attempts to aggregate additional application traffic header information (e.g., method, user agent, referrer, cookies and protocol) to derive at more meaningful and accurate malware analysis results. As such, DroidClassifier has been designed and constructed to consider multiple dimensions of malicious traffic information to establish malicious network patterns. First, it uses the traffic information to create clusters of applications. It then analyzes these application clusters (i) to identify whether the apps in each cluster are malicious or benign and (ii) to classify which family the malicious apps belong to.

DroidClassifier is designed to be efficient and lightweight, and it can be integrated into network IDS/IPS to perform mobile malware classification and detection in a large network. We evaluate DroidClassifier using more than six thousand Android benign apps and malware samples; each with the corresponding collected network traffic. In total, these malicious and benign apps generate 17,949 traffic flows. We then use DroidClassifier to identify the malicious portions of the network traffic, and to extract the multi-field contents of the HTTP headers generated by the mobile malware to build extensive and concrete identifiers for classifying different types of mobile malware. Our results show that DroidClassifier can accurately classify malicious traffic and distinguish malicious traffic from benign traffic using HTTP header information. Experiments indicate that our framework can achieve more than 90% classification rate and detection accuracy while it is also more efficient than a state-of-the-art malware classification and detection approach [2].

In summary, the contributions of our work are mainly two-fold. First, we develop DroidClassifier, which considers multiple dimensions of mobile traffic information from different families of mobile malware to establish distinguishable malicious patterns. Second, we design a novel weighted score-based metric for malware classification, and we further optimize the performance of our classifier using a novel combination of supervised learning (score-based classification)

and unsupervised learning (malware clustering). The clustering step makes our detection phase more efficient than prior efforts, since the subsequent malware classification can be performed over clustered malware requests instead of individual requests from malware samples.

The rest of this paper is organized as follows. Section 2 explains why we consider multidimensional network information to build our framework. Section 3 provides overview of prior work related to the proposed DroidClassifier. Section 4 discusses the approach used in the design of DroidClassifier, and the tuning of important parameters in the system. DroidClassifier is evaluated in Section 5. Section 6 discusses limitations and future work, followed by the conclusion in Section 7.

2 Motivation

A recent report indicates that close to 5,000 Android malicious apps are created each day [6]. The majority of these apps also use various forms of obfuscation to avoid detection by security analysts. However, a recent report by Symantec indicates that Android malware authors tend to improve upon existing malware instead of creating new ones. In fact, the study finds that more than three quarters of all Android malware reported during the first three months of 2014 can be categorized into just 10 families [26]. As such, while malware samples belonging to a family appear to be different in terms of source code and program structures due to obfuscation, they tend to exhibit similar runtime behaviors.

This observation motivates the adoption of network traffic analysis to detect malware [2, 5, 20, 31]. The initial approach is to match requested URIs or hostnames with known malicious URIs or hostnames. However, as malware authors increase malware complexities (e.g., making subtle changes to the behaviors or using multiple servers as destinations to send sensitive information), the results produced by hostname analysis tend to be inaccurate.

To overcome these subtle changes made by malware authors to avoid detection, Aresu et al. [2] apply clustering as part of network traffic analysis to determine malware families. Once these clusters have been identified, they extract features from these clusters and use the extracted information to detect malware [2]. Their experimental results indicate that their approach can yield 60% to 100% malware detection rate. The main benefit of this approach is that it handles these subtle changing malware behaviors as part of training by clustering the malware traffic. However, the detection is done by analyzing each request to identify network signatures and then matching signatures. This can be inefficient when dealing with a large traffic amount. In addition, as these changes attempted by malware authors occur frequently, the training process may also need to be performed frequently. As will be shown in Section 5, this training process, which includes clustering, can be very costly.

We see an opportunity to deal with these changes effectively while streamlining the classification and detection process to make it more efficient than the approach introduced by Aresu et al. [2]. Our proposed approach, DroidClassifier, relies on two important insights. First, most newly created malware belongs

to previously known families. Second, clustering, as shown by Aresu et al., can effectively deal with subtle changes made by malware authors to avoid detection. We construct DroidClassifier to exploit previously known information about a malware sample and the family it belongs to. This information can be easily obtained from existing security reports as well as malware classifications provided by various malware research archives including Android Malware Genome Project [36]. Our approach uses this information to perform training by analyzing traffic generated by malware samples belonging to the same family to extract most relevant features.

To deal with variations within a malware family and to improve testing efficiency, we perform clustering of the testing traffic data and compare features of each resulting cluster to those of each family as part of classification and detection process. Note that the purpose of our clustering mechanism is different from the clustering mechanism used by Aresu et al. [2], in which they apply clustering to extract useful malware signatures. Our approach does not rely on the clustering mechanism to extract malware traffic features. Instead, we apply clustering in the detection phase to improve the detection efficiency by classifying and detecting malware at the cluster granularity instead of at each individual request granularity, resulting in much less classification and detection efforts. By relying on previously known and precise classification information, we only extract the most relevant features from each family. This allows us to use fewer features than the prior approach [2]. As will be shown in Section 5, DroidClassifier is both effective and efficient in malware classification and detection.

3 Related Work

Network Traffic Analysis has been used to monitor runtime behaviors by exercising targeted applications to observe app activities and collect relevant data to help with analysis of runtime behaviors [9, 15, 22, 28, 35]. Information can be gathered at ISP level or by employing proxy servers and emulators. Our approach also collects network traffic by executing apps in device emulators. The collected traffic information can be analyzed for leakage of sensitive information [7, 10], used for classification based on network behaviors [20], or exploited to automatically detect malware [3, 5, 31].

Supervised and unsupervised learning approaches are then used to help with detecting [14, 30, 34] and classifying desktop malware [17, 20] based on collected network traffic. Recently, there have been several efforts that use network traffic analysis and machine learning to detect mobile malware. Shabtai et al. [25] present a Host-based Android machine learning malware detection system to target the repackaging attacks. They conclude that deviations of some benign behaviors can be regarded as malicious ones. Narudin et al. [18] come up with a TCP/HTTP based malware detection system. They extracted basic information, (e.g. IP address), content based, time based and connection based features to build the detection system. Their approach can only determine if an app is malicious or not, and they cannot classify malware to different families.

FIRMA [21] is a tool that clusters unlabeled malware samples according to network traces. It produces network signatures for each malware family for detection. Anshul et al. [3] propose a malware detection system using network traffic. They extract statistical features of malware traffic, and select decision trees as a classifier to build their system. Their system can only judge whether an app is malicious or not. Our system, however, can identify the family of malware.

Aresu et al. [2] create malware clusters using traffic and extract signatures from clusters to detect malware. Our work is different from their approach in that we extract malware patterns from existing families by analyzing HTTP traffic and determining scores to help with malware classification and detection. To make our system more efficient, we then form clusters of testing traffics to reduce the number of test cases (each cluster is a test case) that must be evaluated. This allows our approach to be more efficient than the prior effort that analyzes each testing traffic trace.

4 Introducing DroidClassifier

Our proposed system, DroidClassifier, is designed to achieve two objectives: (i) to distinguish between benign and malicious traffic; and (ii) to automatically classify malware into families based on HTTP traffic information. To accomplish these objectives, the system employs three major components: *training module*, *clustering module*, and *malware classification* and *detection module*.

The *training module* has three major functions: feature extraction, malware database construction, and family threshold decision based on scores. After extracting features from a collection of HTTP network traffic of malicious apps inside the training set, the module produces a database of network patterns per family and the z_{score} threshold that can be used to evaluate the maliciousness of the network traffic from malware samples and classify them into corresponding malware families. To address subtle behavioral changes among malware samples and to improve detection efficiency, the *clustering module* is followed to collect a set of network traffic and gather similar HTTP traffic into the same group so as to classify network traffic as groups.

Finally, the *malware classification and detection module* computes the scores and the corresponding z_{score} based on HTTP traffic information of a particular traffic cluster. If this absolute value of z_{score} is less than the threshold of one family, our system classifies the HTTP traffic into the malware family. It then evaluates whether the HTTP traffic requests are from a certain malware family or from benign apps, the strategy of which is similar to that of the classification module. Our Training and Scoring mechanisms provide a quantitative measurement for malware classification and detection. Next, we describe the training, traffic clustering, malware classification, and malware detection process in details.

4.1 Model Training

The training process requires four steps as shown in Figure 1. The first step is collecting network traffic information of applications that can be used for

training, classification, and detection. With respect to training, the network traffic data set that we focus on is collected from malicious apps. The second step is extracting relevant features that can be used for training and testing. The third step is building malware database. Lastly, we compute the scores that can be used for classification and detection. Next, we describe each of these steps in turn.

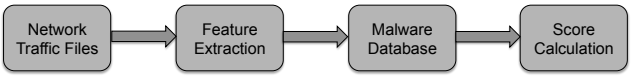


Fig. 1. Steps taken by DroidClassifier to perform training

Collecting Network Traffic. To collect network traffic, we locate malware samples that *have already been classified into families*. We use the real-world malware samples provided by Android Malware Genome Project [36] and Drebin [4] project, which classify 1,363 malware samples, making a total of 2,689 HTTP requests, into 10 families. We randomly choose 706 samples to build the training model, and the remaining 657 samples as malware evaluation set. We also use 5,215 benign apps, generating 15,260 HTTP requests, to evaluate the detection phase. These benign apps are from the Google Play store.

The first step of traffic collection is installing samples belonging to a family into an Android device or a device emulator (as used in this study). We use 50% of malware samples for training; i.e., 30% for database building and 20% for threshold calculation. We also use 20% of benign apps for threshold calculation.

To exercise these samples, we use *Monkey* to randomly generate event sequences to run each of these samples for 5 minutes to generate network traffic. We choose this duration because a prior work by Chen et al. [5] shows that most malware would generate malicious traffic in the first 5 minutes.

In the third step, we use *Wireshark* or *tcpdump*, a network protocol analyzer, to collect the network traffic information. In the last step, we generate the network traffic traces as PCAP files. After we have collected the network traffic information from a family of malware, we repeat the process for the next family.

It is worth noting that our dataset contains several repackaged Android malware samples. Though most of the traffic patterns generated by repackaged malware apps and carrier apps are similar, we find that these repackaged malware samples do generate malicious traffic. Furthermore, our samples also generate some common ad-library traffic, and the traffic can also bring noise to our training phase. In our implementation, we establish a "white-list" request library containing requests sending to benign URLs and common ad-libraries. We filter out white-listed requests and use only the remaining potential malicious traffic to train the model and perform the detection.

Extracting Features for Model Building. We limit our investigation to HTTP traffic because it is a commonly used protocol for network communication. There are four types of HTTP message headers: General Header, Request

Header, Response Header and Entity Header. Collectively, these four types of header result in 80 header fields [27]. However, we also observe that fewer than 12 fields are regularly used in the generated traffic. We manually analyze these header fields and choose five of them as our features. Note that we do not rank them. If more useful headers can be obtained from a different dataset, we may need to retrain the system.

Also note that we utilize these features differently from the prior work [20]. In the training phase, we make use of multiple fields, and come up with a new weighted score-based mechanism to classify HTTP traffic. Perdisci et al. [20], on the other hand, use clustering to generate malware signatures. In our approach, clustering is used as an optimization to reduce the complexity of the detection/classification phase. As such, our approach can be regarded as a combination of both supervised and unsupervised learning.

By using different fields of HTTP traffic information, we, in effect, increase the dimension of our training and testing datasets. If one of these fields is inadequate in determining malware family, e.g., malware authors deliberately tamper one or more fields to avoid analysis, other fields can often be used to help determine malware family, leading to better clustering/classification results. Next, we discuss the rationale of selecting these features and the relative importance of them.

Table 1. Features Extracted

Field Name	Description
Host	This field specifies the Internet host and port number of the resource.
Referer	This field contains URL of a page from which HTTP request originated.
Request-URI	The URI from the request source.
User-Agent	This field contains information about the user agent originating the request.
Content-Type	This field indicates the media type of the entity-body sent to the recipient.

• *Host* can be effective in detecting and classifying certain types of malware with clear and relatively stabilized hostname fields in their HTTP traffic. Based on our observation, most of the malware families generate HTTP traffic with only a small number of disparate host fields.

• *Referrer* identifies the origination of a request. This information can introduce privacy concerns as IMEI, SDK version and device model, device brand can be sent through this field as demonstrated by *DroidKungFu* and *FakeInstaller* families.

• *Request-URI* can also leak sensitive information. We observe that *Gappusin* family can use this field to leak device information, such as IMEI, IMSI, and OS Version.

• *User-Agent* contains a text sequence containing information such as device manufacturer, version, plugins, and toolbars installed on the browser. We observe that malware can use this field to send information to the Command & Control (C&C) server.

• *Content-Type* can be unique for some malware families. For example, *Opfake* has a unique "multipart/form-data; boundary=AaB03x" Content-Type field, which can also be included to elevate the successful rate of malware detection.

Request-URI and Referrer are the two most important features because they contain rich contextual information. Host and User-Agent serve as additional discernible features to identify certain types of malware. Content-Type is the least important in terms of identifiable capability; however, we also observe that this feature is capable of recognizing some specific families of malware.

Although dedicated adversaries can dynamically tamper these fields to evade the detection, such adaptive behaviors may incur additional operational costs, which we suspect is the reason why the level of adaptation is low, according to our experiments. We defer the investigation of malware's adaptive behaviors to future work. In addition, employing multiple hosts can possibly evade our detection at a cost of higher maintenance expenses. In our current dataset, we have seen that some families use multiple hosts to receive information and we are still able to detect and classify them by using multiple network features.

We also notice that these malware samples utilize C&C servers to receive leaked information and control malicious actions. In our data set, many C&C servers are still fully or partially functional. For fully functional servers, we observe their responses. We notice that these responses are mainly simple acknowledgments (e.g., "200 OK"). For the partially functional servers, we can still observe information sent by malware sample to these servers.

Building Malware Database. Once we have identified relevant features, we extract values for each field in each request. As an example, to build a database for the *DroidKungFu* malware family, we search all traffic trace files (PCAPs) of the all samples belonging to this family (100 samples in this case), extract all values or common longest substring patterns, in the case of Request-URI fields, of the five relevant features, put them into lists with no duplicated values, and build a map between each key and its values.

Scoring of Malware Traffic Requests. In the training process, we assign scores to malware traffic requests to compute the classification/detection threshold, which we termed as *training* z_{score} *computation*. We need to calculate the malware z_{score} range for each malware family. We use traffic from 20% of malware samples belonging to each family for training z_{score} computation. For each malware family, we assign a weight to each HTTP field to quantify different contributions of each field according to the number of patterns the field entails, since the number of patterns of a field indicates the uncertainty of extracted patterns. For example, the field with a single pattern is deemed as a unique field, thus it is considered to be a field with high contributions. In contrast, the field with a number of patterns would be weighted lower. As such, we compute the total number of patterns of each field from the malware databases to determine the weight. The following formula illustrates the weight computation for each field: $w_i = \frac{1}{t_i} \times 100$, where w_i stands for the weight for ith field, and t_i is the number of patterns for the ith field for each family in malware databases. For instance, there are 30 patterns for field User-Agent of one malware family in malware databases, so the weight of User-Agent is $\frac{1}{30} \times 100$.

In terms of the Request URI field, we use a different strategy because this filed usually contains a long string. We use the Levenshtein distance [16] to calculate the similarity between the testing URI and each pattern. Levenshtein

Algorithm 1: Calculating Request Scores From One PCAP

```
dataBase[ ] ← Database built from the previous phrase
pcapFile ← Each PCAP file from 20% of malware families
fieldNames[ ] ← Name list for all the extracted fields
tempScore ← 0
sumScore ← 0
avgScore ← 0
for each httpRequest in pcapFile do
    for each name in fieldNames do
        if httpRequest.name ≠ NULL then
            if name ≠ "requestURI" then
                if httpRequest.name in dataBase(name) then
                    tempScore ← 100 {The default weight is 100}
                else
                    tempScore ← 0
                end if
            else
                similarity ←
                similarityFunction(httpRequest.requestURI, dataBase("requestURI"))
                tempScore ← 100 × similarity
            end if
        end if
        sumScore ← sumScore + tempScore
    end for
    avgScore ← sumScore ÷ Size of fieldNames
    record avgScore as the original score of each httpRequest
end for
```

distance measures the minimum number of substitutions required to change one string into the other. After comparing with each pattern, we choose the greatest similarity as a target value, for example, if the similarity value is 0.76, the weight will be 0.76×100 or 76 for the URI field. The score can be calculated using the following equation: $score = \frac{1}{N}\sum_{i=1}^{N} w_i \times m_i$, where w_i is weight for i_{th} field, and m_i indicates whether there is a pattern in the database that matches the field value. If there is, m_i is 1, otherwise, it is 0. Note that m_i is always 1 for the URI field.

After obtaining all the field values and calculating the summation of these values, we then divide it by the total number of fields (i.e., 5 in this case). The result is the original score of this HTTP request.

Then we need to calculate the malware z_{score} range for each family. we calculate the average score and standard derivation of those original scores which are mentioned above. Next, we calculate the absolute value of the z_{score}, which represents the distance between the original score (x) and the mean score ($\bar{x}$) divided by the standard deviation (s) for each request: $|z_{score}| = \left|\frac{x-\bar{x}}{s}\right|$.

Once we get the range of absolute value of z_{score} from all malware training requests of each family, it is used to determine the threshold for classification and detection. We will illustrate the threshold decision process in the following section. Algorithm 1 outlines the steps of calculating original scores from PCAP files. Note that in the testing process, the same z_{score} computation is conducted to evaluate the scores of the testing traffic requests, which we termed as *testing z_{score} computation* to avoid confusion.

4.2 Malware Clustering during Testing

We automatically apply clustering analysis to all of our testing requests. We use hierarchical clustering [24], which can build either a top-down or bottom-up tree to determine malware clusters. The advantage of hierarchical clustering is that it is flexible on the proximity measure, and is able to visualize the clustering results using dendrogram that can be used to choose the optimal number of clusters.

In our framework, we use the single-linkage [24] clustering, which is an agglomerative or bottom-up approach. According to Perdisci et al. [20], single-linkage hierarchical clustering has the best performance compared to X-means [19] and complete-linkage [12] hierarchical clustering.

Feature Extraction for Clustering. First, we need to compute distance measures to represent similarities among HTTP requests. We extract features from URLs and define a distance between two requests according to an algorithm proposed in [20], except that we reduce the number of features to make our algorithm much more efficient. In the end, we extract three types of features to perform clustering: domain name and port number, path to the file, and Jaccard's distance [11] between parameter keys. As an example, consider the following request:

```
http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2
```

The field, `www.example.com:80`, represents the first feature. The field, `/path/to/myfile.html`, represents the second feature. The field, `key1=value1&key2=value2`, represents the parameters, each is a key-value pair, of this request. To compute the third feature, we calculate the Jaccard's distance [11] between the keys. We do not use the parameter values here because these values can be very long, and the comparison between a large number of long strings will consume a large amount of time.

Note that in work by Perdisci et al. [20], they also use the same three features with an addition of the fourth with is the concatenation of parameter values to calculate the similarity of requests for desktop applications. According to [2], the length of URL is larger for the Android malware than the desktop malware, and from our tests, we find the time to calculate the similarity using the fourth feature is much longer than with just three features. We also find that we can get comparable clustering accuracy with just using the three features. As such, we exclude the fourth feature to make our system more efficient but without sacrificing accuracy. In Section 5, we show that our system is as effective as using four features [2], but is also significantly faster.

Recall that we extract five HTTP features (see Table 1) to perform training. Since these features are strings, we use the Levenshtein Distance [16] between two strings to measure their similarity. For parameter keys, Jaccard's distance [11] is applied to measure the similarity. Suppose the number of HTTP requests is N, we can get three $N \times N$ matrices based on three clustering feature sets. We calculate the average value of the three matrices, and regard this average matrix as the similarity matrix used by the clustering algorithm.

After the clustering, we calculate the average of the $|z_{score}|$ of each cluster. We consider requests from the same cluster as one group and use the average value to classify this cluster.

4.3 Malware Classification

We use the remaining 50% of malware samples in each family as the testing set. In order to determine the threshold for classification, we include traffic from 20% benign apps and 20% malware samples. We use the same method as depicted in the previous section to calculate the original score of each benign request. However, when we calculate the z_{score} range of benign apps, we use the mean score ($\bar{x}$) and standard derivation(s) of the 20% malware family we have in previous sections (i.e. $|z_{score}| = \left|\frac{x-\bar{x}(malware)}{s(malware)}\right|$). Then we use the malware z_{score} range and benign z_{score} range to determine the threshold for each malware family in an adaptive manner.

For instance, in the *BaseBridge* family, the absolute range of z_{score} varies from 1.0 to 1.3 using malicious traffic from 20% malware samples. Meanwhile, this value ranges from 1.5 to 10 for the 20% benign apps using the *BaseBridge* database. As a result, we can then set the threshold to be 1.4, which is computed by $(1.3 + 1.5)/2$. For the testing traffic, if the absolute value of z_{score} derived by testing z_{score} computation is less than the threshold, the app will be classified into this BaseBridge family.

4.4 Malware Detection

This detection process is very similar to the clustering process. However, the testing set has been expanded to include traffic from both malicious apps and 5,215 benign apps. The detection phase proceeds like the classification phase. We use BaseBridge family as an example. After extracting each HTTP request from PCAP files, we calculate the score based on BaseBridge training database, similar to classification phase, if the traffic's absolute value of z_{score} is greater than the BaseBridge threshold, we believe this traffic comes from BaseBridge family, and the traffic request is classified as malicious. Otherwise, the traffic does not belong to the BaseBridge family. In the end, if the traffic request is not assigned to any malware families, this request is deemed as benign.

Next, we illustrate how to calculate the detection accuracy for each malware family through an example using the BaseBridge family. If a request is from a BaseBridge family app, and it is also identified as belonging to it, then this is true positive (TP). Otherwise, it is false negative (FN). If the request is not from BaseBridge family app, but it is identified as belonging to it, then it is false positive (FP); otherwise, it is true negative (TN). We then calculate the *detection accuracy* ($Detection\ Accuracy = \frac{TP+TN}{TP+TN+FN+FP}$) and *malware detection rate* ($Malware\ Detection\ Rate = \frac{SUM(TP)}{SUM(FN)+SUM(TP)}$) of each family.

5 Evaluation

We evaluate the malware classification performance of DroidClassifier. We use 30% of the malware samples for database building, 20% of both malware and benign apps for threshold calculation. We set up the testing set to use the remaining 50% of the malware samples and 80% of benign apps. Specifically, we evaluate the following performance aspects of DroidClassifier system.

1. We evaluate *classification effectiveness* of DroidClassifier to classify malicious apps into different families of malware. We present the performance in terms of detection accuracy, TPR (True Positive Rate), TNR (True Negative Rate) and F-Measure. Our evaluation experiments with using different numbers of clusters to determine which one yields the most accurate classification result.
2. We evaluate the *malware detection effectiveness* of DroidClassifier using only malware samples as the training and testing sets. We only focus on how well DroidClassifier correctly detects malware. The detection performance is represented by detection accuracy.
3. We evaluate the *influence of clustering on malware detection effectiveness* by comparing the detection rates between the best case in DroidClassifier when the number of cluster is 1000, and DroidClassifier without clustering process.
4. We *compare our classification effectiveness with results of other approaches.* We also compare the efficiency of DroidClassifier with a similar clustering system [2].

Our dataset consists of 1,363 malicious apps, and our benign apps are downloaded from multiple popular app markets by app crawler. Each app downloaded from app market is sent to VirusTotal for initial screening. The app is added to our normal app set only if the test result is benign. Eventually, we get a normal app set of 5,215 samples belonging to 24 families. A large amount of traffic data are collected by an automatic mobile traffic collection system, similar to the system described in [5], in order to evaluate the classification/detection performance of DroidClassifier. In the end, we get 500.4 MB of network traffic data generated by malware samples in total, out of which we extract 18.1 MB of malicious behavior traffic for training purpose. In a similar manner, we collect 2.15 GB of data generated by normal apps for model training and testing.

5.1 Malware Classification Effectiveness Across Different Cluster Numbers

In our experiment, we perform an evaluation to investigate the sensitivity of our approach to the number of clusters. Therefore, we strategically adjust the number of clusters to find the optimal number that is used to classify malware in the testing data. To do so, we evaluate 13 different numbers of clusters for the whole dataset, ranging from 200 to 7000 clusters. Table 5 shows the classification results using 13 different numbers of clusters. When we increase the number of

Table 2. Classification Result with Different Number of Clusters (TPR=TP/(TP+FN); TNR=TN/(TN+FP); F_Measure = 2 * (TPR * TNR) / (TPR + TNR))

Number of Clusters	TPR	TNR	Detection_Accuracy	F_Measure
200	73.90%	46.59%	46.95%	57.15%
400	60.70%	66.45%	66.34%	63.44%
600	60.70%	66.61%	66.52%	63.52%
800	70.24%	91.39%	91.12%	79.43%
1000	92.39%	94.80%	94.66%	93.58%
1200	90.70%	94.45%	94.30%	92.54%
1400	90.76%	94.42%	94.28%	92.55%
2000	90.76%	93.79%	93.64%	92.25%
3000	89.08%	93.15%	93.01%	91.07%
4000	89.08%	93.11%	92.97%	91.05%
5000	89.08%	93.06%	92.92%	91.03%
6000	88.75%	92.45%	92.30%	90.56%
7000	88.12%	93.02%	92.79%	90.50%

clusters from 200 to 1000, the detection accuracy also improves from 46.95% to 94.66%, respectively. However, using more than 1000 clusters does not improve accuracy. As such, using 1000 clusters is optimal for our dataset. In this setting but without using *DroidKungfu* and *Gappusin*, the two families previously known to be hard to detect and classify [4] , DroidClassifier achieves TPR of 92.39% and TNR of 94.80%, respectively. With these two families, our TPR and TNR still yield 89.90% and 87.60%, respectively.

Table 3. Malware Classification Performance with 1000 Clusters

FamilyName	TP	FN	TN	FP	TPR (%)	TNR (%)	Detection Accuracy (%)	F_Measure (%)
BaseBridge	351	104	11994	44	77.14	99.63	98.82	86.96
DroidKungFu	286	74	7306	4827	79.44	60.22	60.77	68.51
FakeDoc	229	1	12263	0	99.57	100.00	99.99	99.78
FakeInstaller	73	1	11968	451	98.65	96.37	96.38	97.50
FakeRun	70	6	11890	527	92.11	95.76	95.73	93.90
Gappusin	66	16	7170	5241	80.49	57.77	57.92	67.26
Iconosys	17	4	8465	4007	80.95	67.87	67.89	73.84
MobileTx	227	1	12265	0	99.56	100.00	99.99	99.78
Opfake	93	4	12396	0	95.88	100.00	99.97	97.89
Plankton	1025	51	11279	138	95.26	98.79	98.49	96.99
AVG Results					89.90	87.64	87.60	88.76
AVG Results w/o DroidKungFu & Gappusin					92.39	94.80	94.66	93.58

5.2 Detection Effectiveness Per Family

Next, we further decompose our analysis to determine the effectiveness of DroidClassifier by evaluating our effectiveness metrics per malware family. As shown in Table 3, in six out of ten families, our system can achieve more than 90% in F-Measure, meaning that it can accurately classify malicious family as it detects more true positives and true negatives than false positives and false negatives. As

the table reports, our system yields accurate classification results in BaseBridge, FakeDoc, FakeInstaller, FakeRun, MobileTx, Opfake, and Plankton. Specifically, FakeDoc and MobileTx shows above 99% in F-measure, which means it almost detect everything correctly in these two families. However, DroidKungFu, Gappusin and Iconosys shows less than 80% F-measure.

Discussion. Our system cannot accurately classify these three families (i.e. DroidKungFu, Gappusin and Iconosys) due to two main reasons. First, the amounts of the network traffic for these families are too small. For example, we only have 38 applications in Iconosys family and among these, only 19 applications produce network traffic information. We plan to extend the traffic collection time to address this issue in future works.

Second, the malware samples in DroidKungFu and Gappusin families produce a large amount of traffic information that shares similar patterns with that of other families. This leads to ambiguity. We also cross-reference our results with those reported by DREBIN [4]. Their results also confirm our observation as their approach can only achieve less than 50% detection accuracy, which is even lower than that achieved by our system. This is the main reason why we report our result in Table 5 by excluding DroidKungFu and Gappusin.

Table 4. Classification Performance without Clustering Procedure

FamilyName	TP	FN	TN	FP	TPR (%)	TNR (%)	Detection Accuracy (%)	F_Measure (%)
BaseBridge	437	18	12038	0	96.04	100.00	99.86	97.98
DroidKungFu	286	74	2195	9938	79.44	18.09	19.86	29.47
FakeDoc	229	1	12263	0	99.57	100.00	99.99	99.78
FakeInstaller	73	1	12419	0	98.65	100.00	99.99	99.32
FakeRun	75	1	11876	541	98.68	95.64	95.66	97.14
Gappusin	66	16	2914	9497	80.49	23.48	23.85	36.35
Iconosys	20	1	11304	1168	95.24	90.64	90.64	92.88
MobileTx	227	1	12265	0	99.56	100.00	99.99	99.78
Opfake	84	13	12396	0	86.60	100.00	99.90	92.82
Plankton	1049	27	11302	115	97.49	98.99	98.86	98.24
AVG Results					93.18	82.68	82.86	87.62
AVG Results w/o DroidKungFu & Gappusin					96.48	98.16	98.11	97.31

5.3 Comparing Detection Effectiveness of Clustering versus Non-Clustering

In Table 4, we report the detection results when clustering is not performed (i.e., we configure our system to have a cluster for each request). As shown in the table, the detection accuracy without clustering are significantly worse than those with clustering for DroidKungFu and Gappusin. In DroidKungFu family, the detection accuracy decreases from 60.77% to 19.86% by eliminating clustering procedure. In Gappusin family, the detection accuracy decreases from 57.92% to 23.85%. However, after removing these two families, it shows better average detection accuracy than DROIDCLASSIFIER with clustering procedure. The detection accuracy of the Iconosys family increases from 67.89% to 90.64% by removing the clustering procedure.

Discussion. Upon further investigation of the network traffic information, we uncover that the network traffic generated by many benign applications and that of the Iconosys family are very similar. As such, many benign network traffic flows are included with malicious traffic flows as part of the clustering process. However, the overall detection rate including two worst cases (i.e. AVG results in Table 3 and 4) shows that DroidClassifier with clustering is more accurate than DroidClassifier without clustering. In addition, the clustering mechanism enables the cluster-level classification, which classifies malware as a group, while the mechanism without clustering classifies malware individually. This makes DroidClassifier with clustering much more efficient than the mechanism without clustering, in terms of system processing time.

5.4 Comparing Performance with Other Mobile Malware Detectors

In this section, we compare our detection results with other malware detection approaches, including Drebin, PermissionClassifier, Aresu et al. [2], and Afonso et al. [1].

- Drebin [4] is an approach that detects malware by combining static analysis of permissions and APIs with machine learning. It utilizes Support Vector Machine (SVM) algorithm to classify malware data set.
- PermissionClassifier, on the other hand, uses only permission as the features to perform malware detection. During the implementation, we use the same malicious applications used to evaluate Drebin. Then we use Apktool [29] to find the permissions called by each application. We randomly separate the data set as training and testing set. SVM classification approach is employed to perform malware classification.
- Aresu et. al [2] extract malware signatures by clustering HTTP traffic, and they use these signatures to detect malware. We implement their clustering method, and compare the result with that produced by our system.
- Afonso et al. [1] develop a machine learning system that detects Android malicious apps by using the dynamic information from system calls and Android API functions. They employ a different dynamic way to detect malware and also use Android Malware Genome Project [36] as the dataset.

Table 5 reports the results of our evaluation. Drebin uses more features than PermissionClassifier, including API calls and network addresses. As a result, Drebin outperforms PermissionClassifier in detection accuracy. We also compare the results of our system against those of 10 existing anti-virus scanners [4]: AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos. We report the minimum, maximum, and average detection rate of these 10 virus scanner in columns 5 to 7 (AV1 – AV10).

The most time consuming part of the hierarchical clustering is the calculation of the similarity matrix. Aresu et. al [2] use one more feature, the aggregation of values in Request-URI field, to build their clustering system. We implement their method and evaluate the time to compute the similarity matrix. We then compare their time consumption for matrix computation of each malware family

Table 5. Detection Rates of DroidClassfier and Ten Anti-Virus Scanners

Method	Droid Classifier	Permission Classifier	Drebin	Aresu et al.	Afonso et al.	AV1 – AV10		
						Min	Max	Avg.
Full Dataset	94.33%	89.30%	93.90%	60% - 100%	96.82%	3.99%	96.41%	61.25%

with that of DroidClassifier and report the result in Table 6. For BaseBridge, DroidKungFu, FakeDoc and Gappusin, our approach incurs 60% to 100% less time than their approach while yielding over 94% detection rate. For other families, the time is about the same. This is due to the fact that those families do not generate traffic with Request-URI field.

Table 6. Time Comparison of Matrix Calculation (Experiments run on Apple MacBook Pro with 2.8GHz Intel Core i7 and 16G memory)

Family Name	Number of Requests	DroidClassifier (seconds)	Aresu et al. (seconds)
Plankton	1075	361	361
BaseBridge	454	**37**	**10230**
DroidKungFu	359	**86**	**3520**
FakeDoc	229	**9**	**820**
Opfake	96	8	8
FakeInstaller	73	9	9
FakeRun	75	10	10
Gappusin	81	**11**	**264**
MobileTx	227	61	61
Iconosys	20	9	9

Drebin and PermissionClassifier are the state-of-the-art malware detection system with high detection accuracy. Our approach is dynamic-analysis based approach. In the literature, as far as we know, there is a lack of comparative work using dynamic analysis on a large malware dataset to evaluate malware detection accuracy. Therefore, though Drebin and PermissionClassifier use static analysis features, we compare with them in terms of malware detection rate to prove the detection accuracy of DroidClassifier. As our proposed classifier is network-traffic based classifier, the main advantage of our classifier is that we can deploy our system on gateway routers instead on end user devices.

Work by Aresu et al. uses clustering to extract signatures to detect malware. We have emphasized the difference between our work and Aresu before. In terms of comparison, we compare the detection rate and time cost with them. Our work can achieve over 90% detection rate. Even though the purpose of our clustering is different, we can still compare the clustering efficiency. For BaseBridge, DroidKungFu, FakeDoc and Gappusin, our approach, in terms of clustering time, is more efficient than their approach by 60% to 100%.

Work by Afonso et al. [1] can achieve the average detection accuracy of 96.82%. So far, the preliminary investigation of detection effectiveness already indicates that our system can achieve nearly the same accuracy. Unlike their approach, our system can also classify samples into different families, which is important, as repackaging is a common form to develop malware. Their approach

still requires that a malware sample executes completely. In the case that it does not (e.g., interrupted connection with a C&C server or premature termination due to detection of malware analysis environments), their system cannot perform detection. However, our network traffic-based system can handle partial execution as long as the malware attempts to send sensitive information. The presence of our system is also harder to detect as it captures the traffic on the router side, preventing certain malware samples from prematurely terminating execution to avoid analysis.

6 Limitations and Future Work

In this paper, we use HTTP header information to help classify and detect malware. However, our current implementation does not handle encrypted requests through HTTPS protocol. To handle such type of requests in the future, we may need to work closely with runtime systems to capture information prior to encryption, or use on-device software such as Haystack [23] to decrypt HTTPs traffic.

Our system also expects a sufficient number of requests in the training set. As shown in families such as Iconosys, insufficient data used during training can cause the system to incorrectly classify malware and benign samples. Furthermore, to generate network traffic information, our approach, similar to work by Afonso et al. [1], relies on Monkey to generate sufficient traffic. However, events triggered by Monkey tool are random, and therefore, may not replicate real-world events especially in the case that complex event sequences are needed to trigger malicious behaviors. In such scenarios, malicious network traffic may not be generated. Creating complex event sequences is still a major research challenge in the area of testing GUI- and event-based applications. To address this issue in the future, we plan to use more sophisticated event sequence generation approaches to including GUI ripping and symbolic or concolic execution. [13]. We will also evaluate the minimum number of traffic requests that are required to induce good classification performance in future works.

Currently, our framework can only detect new samples from known families if they happen to share previously modeled behaviors. For sample requests from totally unknown malware samples, our framework can put all these similar requests into a cluster. This can help analysts to isolate these samples and simplify the manual analysis process. We also plan to extract other features beyond application-layer header information. For example, we may want to focus on the packet's payload that may contain more interesting information such as C&C instructions and sensitive data. We can also combine the network traffic information with other unique features including permission and program structures such as data-flow and control-flow information.

Similar to existing approaches, our approach can still fail against determined adversaries who try to avoid our classification approach. For example, an adversary can develop advanced techniques to dynamically change their features without affecting their malicious behaviors. Currently, machine-learning based detection systems suffer from this problem [32]. We need to consider how adver-

saries may adapt to our classifiers and develop better mobile malware classification and detection strategies.

We are in the process of collecting newer malware samples to further evaluate our system. We anticipate that newer malware samples may utilize more complex interactions with C&C servers. In this case, we expect more meaningful network behaviors that our system can exploit to detect and classify these emerging malware samples.

Lastly, our system is lightweight because it can be installed on the router to automatically detect malicious apps. The system is efficient because our approach classifies and detects malware at the cluster granularity instead of at each individual request granularity, resulting in much less classification and detection efforts. As future work, we will experiment with deployments of DroidClassifier in a real-world setting.

7 Conclusion

In this paper, we introduce DroidClassifier, a malware classification and detection approach that utilizes multidimensional application-layer data from network traffic information. An integrated clustering and classification framework is developed to take into account disparate and unique characteristics of different mobile malware families. Our study includes over 1,300 malware samples and 5,000 benign apps. We find that DroidClassifier successfully identifies over 90% of different families of malware with 94.33% accuracy on average. Meanwhile, it is also more efficient than state-of-the-art approaches to perform Android malware classification and detection based on network traffic. We envision DroidClassifier to be applied in network management to control mobile malware infections in a large network.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Aziz Mohaisen, for their insightful feedback on our work. This work is supported in part by NSF grant CNS-1566388. This work is also based on research sponsored by DARPA and Maryland Procurement Office under agreement numbers FA8750-14-2-0053 and H98230-14-C-0140, respectively. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

References

1. V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus. Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, 11(1):9–17, 2015.
2. M. Aresu, D. Ariu, M. Ahmadi, D. Maiorca, and G. Giacinto. Clustering android malware families by http traffic. In *10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015.

3. A. Arora, S. Garg, and S. K. Peddoju. Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71. IEEE, 2014.
4. D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.
5. Z. Chen, H. Han, Q. Yan, B. Yang, L. Peng, L. Zhang, and J. Li. A first look at android malware traffic in first few minutes. In *IEEE TrustCom 2015*, Aug. 2015.
6. G. DATA. GDATA MOBILE MALWARE REPORT. `https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf`, July, 2015.
7. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.
8. F-Secure. F-secure mobile threat report. `https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf`, March, 2014.
9. P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, K. Papagiannaki, and P. Rodriguez. Best paper–follow the money: understanding economics of online aggregation and advertising. In *Conference on Internet measurement conference*, pages 141–148. ACM, 2013.
10. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *ACM conference on Computer and communications security*, pages 639–652. ACM, 2011.
11. P. Jaccard. *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.
12. A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM computing surveys (CSUR)*, 31(3):264–323, 1999.
13. C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, Lugano, Switzerland, 2013.
14. N. Kheir. Analyzing http user agent anomalies for malware detection. In *Data Privacy Management and Autonomous Spontaneous Security*, pages 187–200. Springer, 2013.
15. A. Le, J. Varmarken, S. Langhoff, A. Shuba, M. Gjoka, and A. Markopoulou. Antmonitor: A system for monitoring from mobile devices. In *SIGCOMM Workshop on Crowdsourcing and Crowdsharing of Big (Internet) Data*, pages 15–20. ACM, 2015.
16. V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Forschungsbericht.–707–710 S*, 1966.
17. S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 642–647. IEEE, 2013.
18. F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.
19. D. Pelleg, A. W. Moore, et al. X-means: Extending k-means with efficient estimation of the number of clusters. In *ICML*, volume 1, 2000.

20. R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, pages 391–404, 2010.
21. M. Z. Rafique and J. Caballero. Firma: Malware clustering and network signature generation with mixed network behaviors. In *Research in Attacks, Intrusions, and Defenses*, pages 144–163. Springer, 2013.
22. A. Rao, J. Sherry, A. Legout, A. Krishnamurthy, W. Dabbous, and D. Choffnes. Meddle: middleboxes for increased transparency and control of mobile traffic. In *ACM conference on CoNEXT student workshop*, pages 65–66. ACM, 2012.
23. A. Razaghpanah, N. Vallina-Rodriguez, S. Sundaresan, C. Kreibich, P. Gill, M. Allman, and V. Paxson. Haystack: In situ mobile traffic analysis in user space. In *arXiv preprint arXiv:1510.01419*, 2015.
24. L. Rokach and O. Maimon. Clustering methods. In *Data mining and knowledge discovery handbook*, pages 321–352. Springer, 2005.
25. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.
26. Symantec Corporation. Internet Security Threat Report 2014. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf`, Accessed on June 21, 2016.
27. Tutorialspoint. HTTP header. `http://www.tutorialspoint.com/http/http_header_fields.htm`, Accessed on June 21, 2016.
28. N. Vallina-Rodriguez, J. Shah, A. Finamore, Y. Grunenberger, K. Papagiannaki, H. Haddadi, and J. Crowcroft. Breaking for commercials: characterizing mobile advertising. In *ACM conference on Internet measurement conference*, pages 343–356. ACM, 2012.
29. R. Winsniewski. Android–apktool: A tool for reverse engineering android apk files. `http://ibotpeaches.github.io/Apktool/`, Accessed on June 21, 2016.
30. P. Wurzinger, L. Bilge, T. Holz, J. Goebel, C. Kruegel, and E. Kirda. Automatically generating models for botnet detection. In *European symposium on research in computer security*, pages 232–249. Springer, 2009.
31. Q. Xu, Y. Liao, S. Miskovic, Z. M. Mao, M. Baldi, A. Nucci, and T. Andrews. Automatic generation of mobile app signatures from traffic observations. In *IEEE INFOCOM*, April 2015.
32. W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers, a case study on pdf malware classifiers. In *Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.
33. H. Yao, G. Ranjan, A. Tongaonkar, Y. Liao, and Z. M. Mao. Samples: Self adaptive mining of persistent lexical snippets for classifying mobile application traffic. In *Annual International Conference on Mobile Computing and Networking*, 2015.
34. J. Zhang, S. Saha, G. Gu, S.-J. Lee, and M. Mellia. Systematic mining of associated server herds for malware campaign discovery. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 630–641. IEEE, 2015.
35. Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.
36. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.