Spring 4-20-2020

Advanced Techniques to Detect Complex Android Malware

Zhiqiang Li
University of Nebraska - Lincoln, zhiqiangleeusa@gmail.com

ADVANCED TECHNIQUES TO DETECT COMPLEX ANDROID MALWARE

by

Zhiqiang Li

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professors Qiben Yan and Witawas Srisa-an

Lincoln, Nebraska

May, 2020

ADVANCED TECHNIQUES TO DETECT COMPLEX ANDROID MALWARE

Zhiqiang Li, Ph.D.

University of Nebraska, 2020

Adviser: Qiben Yan and Witawas Srisa-an

Android is currently the most popular operating system for mobile devices in the world. However, its openness is the main reason for the majority of malware to be targeting Android devices. Various approaches have been developed to detect malware.

Unfortunately, new breeds of malware utilize sophisticated techniques to defeat malware detectors. For example, to defeat signature-based detectors, malware authors change the malware's signatures to avoid detection. As such, a more effective approach to detect malware is by leveraging malware's behavioral characteristics. However, if a behavior-based detector is based on static analysis, its reported results may contain a large number of false positives. In real-world usage, completing static analysis within a short time budget can also be challenging.

Because of the time constraint, analysts adopt approaches based on dynamic analyses to detect malware. However, dynamic analysis is inherently unsound as it only reports analysis results of the executed paths. Besides, recently discovered malware also employs structure-changing obfuscation techniques to evade detection by state-of-the-art systems. Obfuscation allows malware authors to redistribute known malware samples by changing their structures. These factors motivate a need for malware detection systems that are efficient, effective, and resilient when faced with such evasive tactics.

In this dissertation, we describe the developments of three malware detection systems to detect complex malware: DroidClassifier, GranDroid, and Obfusifier.

DroidClassifier is a systematic framework for classifying network traffic generated by mobile malware. GranDroid is a graph-based malware detection system that combines dynamic analysis, incremental and partial static analysis, and machine learning to provide time-sensitive malicious network behavior detection with high accuracy. Obfusifier is a highly effective machine-learning-based malware detection system that can sustain its effectiveness even when malware authors obfuscate these malicious apps using complex and composite techniques.

Our empirical evaluations reveal that DroidClassifier can successfully identify different families of malware with 94.33% accuracy on average. We have also shown GranDroid is quite effective in detecting network-related malware. It achieves 93.0% accuracy, which outperforms other related systems. Lastly, we demonstrate that Obfusifier can achieve 95% precision, recall, and F-measure, collaborating its resilience to complex obfuscation techniques.

ACKNOWLEDGMENTS

The completion of this dissertation would not be possible without the financial support from my advisors: Professors Witawas Srisa-an and Qiben Yan. I want to thank them both for the efforts they put forth throughout my Ph.D. study.

Dr. Srisa-an has been mentoring me since the beginning of my Ph.D. career. He guides me well throughout my research work. His motivation and patience have given me massive power to make progress in my research. He is my mentor and a better advisor for my Ph.D. study beyond any imagination. I thank him for encouraging me and sharing insightful suggestions. He has given me the freedom to pursue various projects without any objection. He plays a significant role in polishing my research writing skills. I will never forget his guidance and kindness.

Dr. Yan has a significant influence on me, and it has been an honor to be his student. I learned a lot from him in the field of cybersecurity. I appreciate his generous contributions of time and ideas to make my Ph.D. experience productive. He is very enthusiastic and energetic. He has provided insightful information about my research, which is very inspirational for me.

Apart from my advisors, I would like to express the gratitude to my supervisory committee: Prof. Lisong Xu, Prof. Song Ci, and lastly, Prof. David Swanson for the encouragement, valuable comments, and many insightful suggestions.

I also thank Jun Sun and Lichao Sun, who have been collaborating with me during my study. I will remember the inspirational discussions we had, and the moments of ecstasy after we caught the deadline. I would never forget my fellow labmates for the fun time we spent together. They are Supat Rattanasuksun, Zhen Hu, Shakthi Bachala, Yutaka Tsutano, and Jackson Dinh. Every time I talk to Supat, I always feel relaxed, and he gives me a lot of valuable advice in life. I always enjoy the Thai

massage he gave me. Zhen is the only girl in our lab and we are from the same country. To me, she is similar to some of my friends in college in China. Getting along with her reminds me of my precious time in college. Shakthi has a cheerful personality and is a person who can bring joy to others. I feel relaxed with him. Yutaka is an excellent and proud man, and he pursues things to the extreme. I can always learn from him. Jackson is quick-minded and can solve problems quickly. He is a good helper for me. I want to say thank you to Thammasak Thianniwet for helping me to improve my writing skills and all of my friends from the University of Nebraska for talking and hanging out with me. Their immense support and critiques help me rectify numerous issues and strengthen my papers.

Most importantly, I want to say thank you to the most influential people in my life: my parents and my sister. They gave me endless support, encouragement, and motivation to achieve my goals.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android is currently the most popular smart-mobile device operating system in the world, holding about 80% of worldwide 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 [1]. On the contrary, only one new threat family on iOS was reported.

Among these malware families, one type of attack involves sending sensitive and private user's information to external sites. Because these malicious actions generate trails of network traffic, it is possible to detect apps that perform such malicious actions by observing network activities. In the past, security analysts have used network connectivity analysis to identify mobile applications to facilitate network management tasks [2]. Because cybercriminals have also exploited Android's network connectivity to glean sensitive information or launch devastating network-level attacks [3,4,5], studying network traffic going into or coming out of Android devices can yield unique insights about the attack origination and patterns. Therefore, researchers have statically or dynamically analyzed network information to detect malicious Android apps.

Static analysis approaches [6, 7, 8, 9, 10, 11] perform sound analysis in an offline

manner and thus incur no runtime overhead. However, static analysis can result in excessive false positives. Moreover, they are often ineffective when various forms of obfuscation and encryption techniques are applied to the program codes. Dynamic analysis approaches, on the other hand, are more precise but incur additional runtime overhead [12, 13, 14]. as they need to incorporate the apps' actual runtime behaviors that may be triggered by dynamically downloaded codes from remote servers. However, the analysis results are unsound. Furthermore, recent reports indicate that dynamic analysis can be easily defeated if an app under analysis can discover that it is being observed (e.g., running in an emulator), and as a result, it behaves as a benign app [15, 16, 17].

Due to the limitations above, it is not a surprise that recently introduced malware detection approaches perform hybrid analysis, leveraging both static and dynamic information. In general, hybrid analysis approaches statically analyze various application components of an app, execute the app, and then record runtime information [18,19,20]. These approaches then use both static and dynamic information to detect malicious apps, which can lead to more in-depth and precise results. However, most of the existing Android malware analysis approaches detect Android malware by matching manually selected characteristics (e.g., permissions) [6, 11, 21, 22] or predefined programming patterns [8, 10]. *The existing approaches do not capture the programming logic that leads to malicious network behaviors.*

Our key observation about a typical hybrid analysis approach is that: a significant amount of efforts are spent on constructing various static analysis contexts (e.g., API calls, control-flow graphs, and data-flow graphs). Yet, the malicious network behaviors are only induced by specific programming logic, i.e., *the network-related paths or events* (e.g., distilling and sending information to a suspicious C&C server) that have been dynamically executed. This can lead to wasteful static analysis efforts.

Furthermore, running an instrumented app or modified runtime systems (e.g., Dalvik or ART) to log events can incur significant runtime overhead (e.g., memory to store runtime information, and network or USB bandwidth to transport logged information for processing). In the end, it is still challenging for hybrid analysis to be able to complete its analysis within a given time budget (e.g., five minutes) as statically analyzing an app can yield varying time depending on the size and complexity of the app under analysis. *Adhering to a time budget is an important criterion for real-world malware analysis and vetting systems.*

Code obfuscation, a common approach used by developers to protect the intellectual properties of their software [23] by making reverse-engineering more difficult, has also been used by malware authors as an anti-analysis tool to hide malicious code within an application. As such, it is not surprising that we have seen applications of various obfuscation techniques to malicious apps to evade the security analysis. These techniques are especially effective in defeating existing malware and virus scanners, which often rely on signature matching or program analysis. In this work, we applied various obfuscation techniques to known malware samples and evaluated them by VirusTotal [24]. The analysis results indicate that many existing techniques deployed by VirusTotal would misclassify known but obfuscated malware samples as benign.

Applying code obfuscation to malware can also defeat state-of-the-art machine learning-based malware detection systems [9, 11, 13, 25, 26, 27]. These existing systems extract unobfuscated features from benign and malware Android samples to build classifiers to detect malware. One recent work [28] has shown that when obfuscated Android malware samples are submitted to these classifiers, they can be miscategorized as features used by these classifiers are now more ambiguous due to obfuscation [29]. *Developing an obfuscation-resilient systems would prevent malware authors from simply obfuscating known malware for redistribution.*

1.1 Contributions

The contributions of this dissertation are as follows:

1. We implement DroidClassifier, which considers multiple dimensions of mobile traffic information from different families of mobile malware to establish distinguishable malicious patterns. Besides, 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.

2. We develop GranDroid based on system-level dynamic graphs to detect malicious network behaviors. GranDroid utilizes detailed network-related programming logic to automatically and precisely capture the malicious network behaviors. GranDroid enables partial static analysis to expand the analysis scope at runtime, and uncover malicious programming logic related to dynamically executed network paths. Doing so can make our analysis approach more sound than a traditional dynamic analysis approach. We perform an in-depth evaluation of GranDroid in terms of the runtime performance and the efficacy of malicious network behavior detection. We show that GranDroid can run on real devices efficiently, achieving a high accuracy in detecting malicious network behaviors.

3. We implement Obfusifier, a machine-learning-based malware detector that is constructed using features from unobfuscated samples but can provide accurate

and robust results when obfuscated samples are submitted for detection. OBFUSIFIER generates method call graphs using static analysis. It then simplifies method call graph by removing the user-defined methods, system-level methods and only keeping Android API methods. This simplification process enables us to reconstruct a graph that is obfuscation-resistant while preserving the structural and semantic information concerning Android API usage of the original graph. OBFUSIFIER then extracts machine learning features from simplified graphs and these features can resist against code obfuscation because of graph simplification. We evaluate the detection efficacy and runtime performance OBFUSIFIER using both unobfuscated and obfuscated samples. The results show that OBFUSIFIER can handle obfuscated Android malware with high efficiency and accuracy.

Next, we describe these approaches in turn. Note that we embed prior related work inside each approach so that we can compare and contrast their capabilities to those of our systems after our systems have been introduced.

Chapter 2

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

Portions of this material have previously appeared in the following publication:

Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen, "Droidclassifier: Efficient adaptive mining of application-layer header for classifying android malware," in International Conference on Security and Privacy in Communication Systems. Springer, 2016, pp. 597–616.

In this chapter, 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 a 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 vast 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. At the same time, it is also more efficient than a state-of-the-art malware classification and detection approach [30].

The rest of this chapter is organized as follows. Section 2.1 explains why we consider multidimensional network information to build our framework. Section 2.2 discusses the approach used in the design of DroidClassifier, and the tuning of important parameters in the system. DroidClassifier is evaluated in Section 2.3. Section 2.4 discusses limitations and future work. Section 2.5 describes the related work, followed by the conclusion in Section 2.6.

2.1 Motivation

A recent report indicates that close to 5,000 Android malicious apps are created each day [31]. 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 [32]. 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 [30, 33, 34, 35]. 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. [30] 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 [30]. 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 2.3, 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. [30]. 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. [30], 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 [30]. As will be shown in Section 2.3, DroidClassifier is both effective and efficient in malware classification and detection.

2.2 System Design

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 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, and our system classifies the HTTP traffic into the malware family. It then evaluates whether the HTTP traffic requests are from a particular 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 detail.

2.2.1 Model Training

The training process requires four steps, as shown in Figure 2.1. The first step is collecting network traffic information of applications that can be used for training, classification, and detection. Concerning 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 a malware database. Lastly, we compute the scores that can be used for classification and detection. Next, we describe each of these steps in turn.

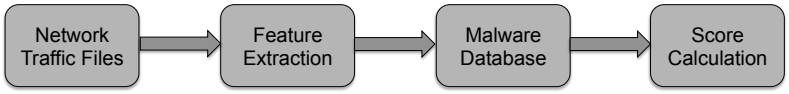

Figure 2.1: Steps taken by DroidClassifier to perform training

Collecting Network Traffic. To collect network traffic, we locate malware samples previously classified into families. We use the real-world malware samples provided by the Android Malware Genome Project [36] and Drebin [9] 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 a 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. [35] 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 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 typical ad-library traffic, and the traffic can also add 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 [37]. However, we also observe that the generated traffic uses fewer than 12 fields. 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 [34]. 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. [34], 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 is 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 2.1: 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 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 evade our detection at the 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 samples 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). We then extract all values or common longest substring patterns, in the case of Request-URI fields, of the five relevant features. Next, we 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 several 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 [38] to calculate the similarity between the testing URI and each pattern. Levenshtein 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.

2.2.2 Malware Clustering during Testing

We automatically apply clustering analysis to all of our testing requests. We use hierarchical clustering [39], which can build either a top-down or bottom-up tree to determine malware clusters. The advantage of hierarchical clustering is that it is

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
```

flexible on the proximity measure and can visualize the clustering results using a dendrogram to help with choosing the optimal number of clusters.

In our framework, we use the single-linkage [39] clustering, which is an agglomerative or bottom-up approach. According to Perdisci et al. [34], single-linkage hierarchical clustering has the best performance compared to X-means [40] and complete-linkage [41] 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 [34], 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: the domain name and port number, a path to the file, and Jaccard's distance [42] 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 [42] 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 would consume a large amount of time.

Note that in work by Perdisci et al. [34], 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 [30], 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 by just using the three features. As such, we exclude the fourth feature to make our system more efficient but without sacrificing accuracy. In Section 2.3, we show that our system is as effective as using four features [30], but is also significantly faster.

Recall that we extract five HTTP features (see Table 2.1) to perform training. Since these features are strings, we use the Levenshtein Distance [38] between two strings to measure their similarity. For parameter keys, Jaccard's distance [42] 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.

2.2.3 Malware Classification

We use the remaining 50% of malware samples in each family as the testing set. 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 is classified into this BaseBridge family.

2.2.4 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, and if the traffic's absolute value of z_{score} is higher 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 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* ($DetectionAccuracy = \frac{TP+TN}{TP+TN+FN+FP}$) and *malware detection rate* ($Malware\ Detection\ Rate = \frac{SUM(TP)}{SUM(FN)+SUM(TP)}$) of each family.

2.3 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 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 [30].

Our dataset consists of 1,363 malicious apps, and our benign apps are downloaded from multiple popular app markets by app crawler. Each downloaded app 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. We also collect a large amount of traffic data by an automatic mobile traffic collection system, similar to the system described in [35] 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 purposes. Similarly, we collect 2.15 GB of data generated by normal apps for model training and testing.

2.3.1 Malware Classification Effectiveness Across Different Cluster Numbers

In our experiment, we 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

Table 2.2: Classification Result with Different Number of Clusters (TPR=TP/(TP+FN); TNR=TN/(TN+FP))

Number of Clusters	TPR	TNR	Detection_Accuracy
200	73.90%	46.59%	46.95%
400	60.70%	66.45%	66.34%
600	60.70%	66.61%	66.52%
800	70.24%	91.39%	91.12%
1000	92.39%	94.80%	94.66%
1200	90.70%	94.45%	94.30%
1400	90.76%	94.42%	94.28%
2000	90.76%	93.79%	93.64%
3000	89.08%	93.15%	93.01%
4000	89.08%	93.11%	92.97%
5000	89.08%	93.06%	92.92%
6000	88.75%	92.45%	92.30%
7000	88.12%	93.02%	92.79%

different numbers of clusters for the whole dataset, ranging from 200 to 7000 clusters. Table 2.2 shows the classification results using 13 different numbers of clusters. When we increase the number of 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 which are previously known to be hard to detect and classify [9], 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.

2.3.2 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 2.3, in four 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,

Table 2.3: Malware Classification Performance with 1000 Clusters (F_Measure = 2TP / (2TP + FP + FN))

FamilyName	TP	FN	TN	FP	TPR (%)	TNR (%)	Detection Accuracy (%)	F_Measure (%)
BaseBridge	351	104	11994	44	77.14	99.63	98.82	82.59
DroidKungFu	286	74	7306	4827	79.44	60.22	60.77	10.45
FakeDoc	229	1	12263	0	99.57	100.00	99.99	99.78
FakeInstaller	73	1	11968	451	98.65	96.37	96.38	24.41
FakeRun	70	6	11890	527	92.11	95.76	95.73	20.8
Gappusin	66	16	7170	5241	80.49	57.77	57.92	2.45
Iconosys	17	4	8465	4007	80.95	67.87	67.89	0.84
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	91.56
AVG Results					89.90	87.64	87.60	53.06
AVG Results w/o DroidKungFu & Gappusin					92.39	94.80	94.66	64.71

FakeInstaller, FakeRun, MobileTx, Opfake, and Plankton. Specifically, FakeDoc and MobileTx show above 99% in F-measure, which means it almost detect everything correctly in these two families. However, DroidKungFu, FakeInstaller, FakeRun, Gappusin, and Iconosys show very low 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 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 and can lead to ambiguity. We also cross-reference our results with those reported by Drebin [9]. 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 2.5 by excluding DroidKungFu and Gappusin.

Table 2.4: 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	5.4
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	21.68
Gappusin	66	16	2914	9497	80.49	23.48	23.85	1.37
Iconosys	20	1	11304	1168	95.24	90.64	90.64	3.31
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	93.66
AVG Results					93.18	82.68	82.86	61.51
AVG Results w/o DroidKungFu & Gappusin					96.48	98.16	98.11	76.04

2.3.3 Comparing Detection Effectiveness of Clustering versus Non-Clustering

In Table 2.4, 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 is 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 the 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 the 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 2.3 and 2.4) 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.

2.3.4 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. [30], and Afonso et al. [43].

- Drebin [9] 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 [44] 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 [30] 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. [43] develop a machine learning system that detects Android malicious apps by using 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 2.5 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 [9]: AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos. We report the minimum, maximum, and average detection rate of these 10 virus scanners in columns 5 to 7 (AV1 – AV10).

Table 2.5: 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%

The most time-consuming part of the hierarchical clustering is the calculation of the similarity matrix. Aresu et. al [30] use one more feature, the aggregation of values in the 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 with that of DroidClassifier and report the result in Table 2.6. 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 because those families do not generate traffic with the Request-URI field.

Drebin and PermissionClassifier are the state-of-the-art malware detection system with high detection accuracy. Our approach is a 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 against them in terms of malware detection rate to prove the detection accuracy of

Table 2.6: 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

DroidClassifier. As our proposed classifier is a network-traffic based classifier, the main advantage of our classifier is that we can deploy our system on gateway routers instead of 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. [43] 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 essential, 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 specific malware samples from prematurely terminating execution to avoid analysis.

2.4 Discussion

In this chapter, 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 before encryption, or use on-device software such as Haystack [45] 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 classify malware and benign samples incorrectly. Furthermore, to generate network traffic information, our approach, similar to work by Afonso et al. [43], 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. [46]. 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 change their features without affecting their malicious behaviors dynamically. Currently, machine-learning-based detection systems suffer from this problem [47]. We need to consider how adversaries 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 evaluate our system further. 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 detect malicious apps automatically. 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.

2.5 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 [21, 48, 49, 50, 51]. 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 [12, 52], used for classification based on network behaviors [34], or exploited to detect malware automatically [33, 35, 53].

Supervised and unsupervised learning approaches are then used to help with detecting [54,55,56] and classifying desktop malware [34,57] 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. [58] 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. [59] 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 [60] 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. [53] 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. [30] 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.

2.6 Conclusion

In this chapter, we introduce DroidClassifier, a malware classification and detection approach that utilizes multidimensional application-layer data from network traffic information. DroidClassifier integrates clustering and classification frame 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 vast network.

Chapter 3

GranDroid: Graph-based Detection of Malicious Network Behaviors in Android Applications

Portions of this material have previously appeared in the following publication:

Z. Li, J. Sun, Q. Yan, W. Srisa-an, and S. Bachala, "Grandroid: Graph-based detection of malicious network behaviors in android applications," in International Conference on Security and Privacy in Communication Systems. Springer, 2018, pp. 264–280.

In this chapter, we set our research goal to enhance the capability of hybrid analysis and evaluate if it can provide sufficiently rich context information in detecting malware's malicious network behaviors on real devices within a specific time budget. Analyzing apps on real devices mitigates the evasion attacks by sophisticated malware that determines its attacking strategy based on its running environment. However, the challenge lies in need of lowering the analysis overhead incurred on resource-constrained mobile devices. Also, we aim at capturing additional relevant network-related programming logic by using dynamic analysis, so that we can avoid any wasteful efforts in distilling information from apps. We then evaluate the effectiveness

of the dynamically generated information in detecting malicious network behaviors of mobile malware.

To achieve this research goal, we introduce GranDroid, a graph-based malicious network behavior detection system. GranDroid has been implemented as a tool built on Jitana, a high-performance hybrid program analysis framework [61]. We extract four network-related features from the network-related paths and subpaths that incorporate network methods, statistic features of each subpath, and statistic features on the sizes of newly-generated files during the dynamic analysis. These features uniquely capture the programming logic that leads to malicious network behaviors. We then apply different types of machine learning algorithms to build models for detecting malicious network behaviors.

We evaluate GranDroid using $1,500$ benign and $1,500$ malicious apps collected recently, and run these apps on real devices (i.e., Asus Nexus 7 tablets) using event sequences generated by UIAutomator[1]. Our evaluation results indicate that GranDroid can achieve high detection performance with 93.2% F-measure.

The rest of the chapter is organized as follows. We provide a motivating example for this work in Section 3.1. We present system design and implementation in Section 3.2. We report our evaluation results in Section 3.3 and discuss the ramifications of the reported results in Section 3.4. We describe related work in Section 3.5 and conclude this chapter in Section 3.6.

3.1 Motivation

Bouncer, the vetting system used by Google, can be bypassed by either delaying enacting the malicious behaviors or not enacting the malicious behaviors when the app

[1]available from: https://developer.android.com/training/testing/ui-automator.html

is running on an emulator instead of a real device. Figure 3.1 illustrates a code snippet from Android.Feiwo adware [62], a malicious advertisement library that leaks user's private information including device information (e.g., IMEI) and device location. The Malcode method checks fake device ID or fake model to determine whether the app is running on an emulator.

```
1: public static Malcode(android.content.Context c) {
2:    ...
3:    v0 = c.getSystemService("phone").getDeviceId();
4.    if (v0 == 0 || v0.equals("000000000000000") == 0) {
5.        if ((android.os.Build.MODEL.equals("sdk") == 0) &&
      (android.os.Build.MODEL.equals("google_sdk") == 0))  {
6:             server = http.connect (server A);}
7:        else{
8:             server = http.connect (server B); }}
9:    else{
10:     server = http.connect (server B);}
11:   // Send message to server through network interface
12:   ...}
```

Figure 3.1: Android.Feiwo Adware Example

In this example, if the app is being vetted through a system like BOUNCER, it would be running on an emulator that matches the conditions in Lines 4 and 5. As a result, it will then connect to a benign server, i.e., *server A*, which serves benign downloadable advertisement objects (i.e., Line 6). However, if the app is running on a real device, it will make a connection to a malicious server, i.e., *server B*, which serves malicious components disguised as advertisements (i.e., Lines 8 and 10). An emulator-based vetting system then classifies this app as benign since the application never exhibits any malicious network behaviors.

For static analysis approaches, the amount of time to analyze this app can vary based on the complexity of code. Furthermore, there are cases when static analysis cannot provide conclusive results as some of the input values may not be known at the

analysis time (e.g., the location of *server B* can be read in from an external file). This would require additional dynamic analysis to verify the analysis results. Therefore, using static analysis can be quite challenging for security analysts if each app must be vetted within a small time budget (e.g., a few minutes).

Our proposed approach attempts to achieve the best of both static and dynamic approaches. Specifically, we propose to find suspicious code locations by using dynamic analysis to identify executable components. It then supplements dynamic analysis results with static analysis of these executed components to uncover more execution paths. Finally, it uses a machine learning classifier to quickly determine if the app has malicious network behaviors.

For example, when we use our approach to analyze Malcode, it would first run the app for a fixed amount of time. While the app is running, our hybrid analysis engine pulls all the loaded classes (including any of its methods that have been executed and any classes loaded through the Java reflection mechanism) and incrementally analyzes all methods in each class to identify if there are paths in an app's call graph that contain targeted or suspicious network activities. Despite the malware's effort in hiding the malicious paths, our system would be able to identify the executed path that includes the network related API calls on Lines 6, 8 and 10. These paths are then decomposed into subpaths and submitted to our classifier for malicious pattern identification.

There are two notable points in this example. First, our approach can analyze more information within a given time budget than using dynamic analysis alone. This would allow vetting techniques including BOUNCER to achieve a higher precision without extending the analysis budget. Second, unlike existing approaches such as DROIDSIFT, which only considers APIs invoked in the application code [63], our approach also retrieves low level platform and system APIs that are necessary to perform the targeted

actions. This allows our approach to build longer and more comprehensive paths, leading to more relevant information that can further improve detection precision. In the following section, we describe the design and implementation of GranDroid in detail.

3.2 System Design

We now describe the architectural overview of our proposed system, which operates in three phases: *graph generation*, *feature extraction*, and *malicious network behavior detection*, as shown in Figure 3.2. Next, we describe each phase in turn.

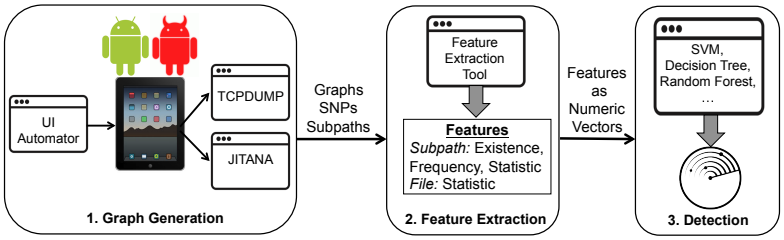

Figure 3.2: System Architecture

3.2.1 Graph Generation

GranDroid detects malicious network behaviors by analyzing program contexts based on system-level graphs. As illustrated in Figure 3.2, the process to generate the necessary graphs involves three existing tools and an actual device or an emulator (we used an actual device in this case). First, we install both malicious and benign apps with known networking capability on several Nexus 7 tablets. Next, we select malware samples and benign apps that can be *exercised* and can *produce network traffic*. We discard incomplete malware samples and the ones that produce zero network traffic, as GranDroid currently focuses on detecting malicious network behaviors. However, GranDroid can be extended to cover other types of malware (e.g., those that corrupt

files). For future work, we plan to show that our graph-based approach is also effective for detecting other types of malicious behaviors.

Next, we use UIAutomator to generate event sequences to exercise these apps. The tablet is also connected to a workstation running TCPDump to capture network traffic information and Jitana [61], a high-performance hybrid program analysis tool to perform on-the-fly program analysis. Because it is possible that UIAutomator cannot generate the necessary event sequences to exercise components in an app that generates network traffic, we also use TCPDump to verify that the apps we investigate indeed generate network traffic. If UIAutomator fails to generate event sequences for an app that is known to produce network traffic, that particular app is subsequently discarded.

While UIAutomator exercises these apps installed on a tablet, we use Jitana to *concurrently* analyze loaded classes to generate three types of graphs: classloader, class, and method call graphs that our technique utilizes. Jitana can analyze application code, third party library code, framework code (including implementations of various Android APIs), and underlying system code. Jitana performs analysis by off-loading its dynamic analysis effort to a workstation to save the runtime overhead. It periodically communicates with the tablet to pull classes that have been loaded as a program runs. Once these classes have been pulled, Jitana analyzes these classes to uncover all methods and then generates the method call graph for the app. As such, we are able to run Jitana and TCPDump simultaneously, allowing the data collection process to be completed within one run. For the apps that we cannot observe network traffic, we also discard their generated graphs. Next, we provide the basic description of the three types of graphs used in GranDroid.

Class Loader Graph and Class Graph. A Class Loader Graph of an app includes all class loaders called when running an app. Direct edges show the inheritance

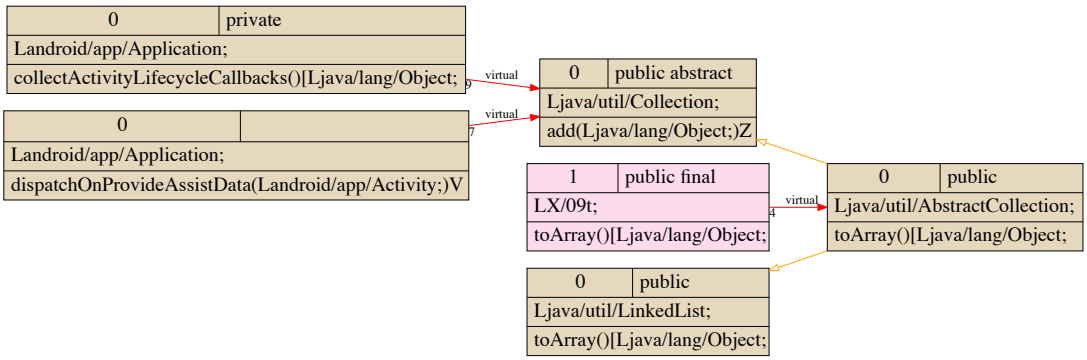

Figure 3.3: Method Graph

relationship between two class loaders. System class loaders are parents of application class loaders. A Class Graph shows relationships among all classes. The important information that these graphs provide includes the ownership, relationship between methods, classes, and the app that these classes belong to (based on the class loader information). Such information is particularly useful for identifying paths and subpaths as it can help resolving ambiguity when multiple methods belonging to different classes share the same name and method's signature. Both Class Loader Graph and Class Graph are used to generate size information feature for machine learning classification.

Method Graph. Our system detects malicious network behaviors by exploring the invoking relationship of methods in the Method Graph. As shown in Figure 3.3, blocks represent methods, and edges indicate invoking relationships among methods. Each block contains the name of the method, its modifiers, and the class name which this method belongs to. *Sensitive Network Paths* (*SNPs*) are defined as paths that contain network-related APIs. We generate SNPs from the method graph of each app.

Note that these dynamically generated graphs are determined by the event sequences that exercise each app. As such, they reflect the runtime behavior of an app. Another useful information contained in these graphs includes the specific Android

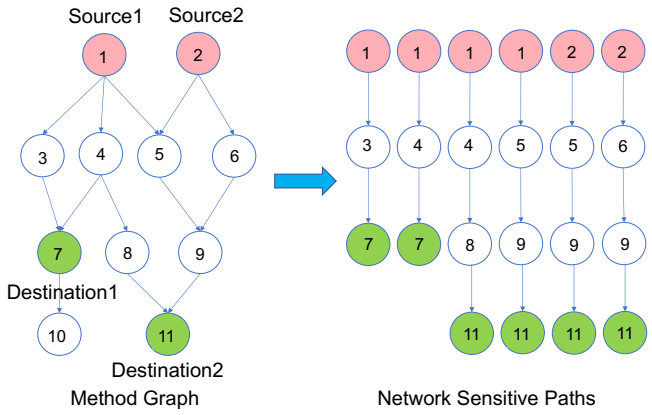

Figure 3.4: Path Generation

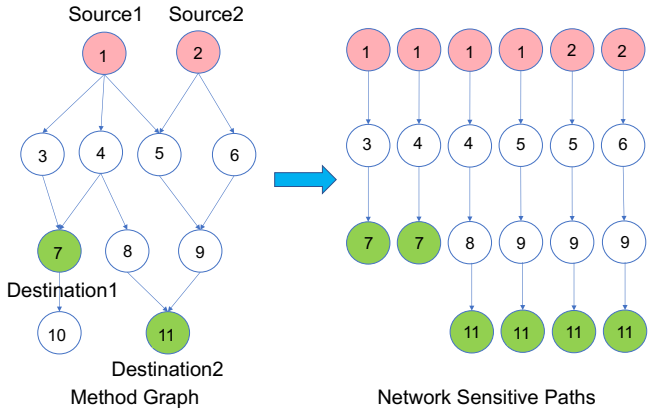

Figure 3.5: Subpath Generation

APIs provided by Google and used by each app. We observe that detecting an actual malicious act often boils down to detecting critical Android APIs that enable malicious behaviors. For example, if a malicious app tries to steal users' private information by sending it through the Internet, network-related APIs must be used to commit this malicious act. In addition to network-related APIs, there are also other system-level and user-defined methods that can be exploited by malware authors. JITANA can capture the invocations of these APIs and any lower-level APIs that can help with identifying SNPs and their subpaths formed by these sensitive method invocations.

The information can be extracted from the Method Graph of each app. Next, we describe the process of generating SNPs and the corresponding subpaths.

Sensitive Network Path (SNP) Generation. An SNP (a path related to network behavior) can be used to determine if an app exhibits malicious network behaviors. To generate SNPs, we extract all the network-related Android APIs provided by Google, and network-related APIs from third-party HTTP libraries, such as Volley [64], Okhttp [65], Picasso [66], and Android Asynchronous Http Client [67]. In the Method Graph, we consider all nodes whose in-degree is zero as sources and all network-related method nodes as destinations. GranDroid generates SNPs from sources to destinations via depth-first search (DFS). Each SNP contains all the methods (nodes) from the program entry points to network-related destinations.

Figure 3.4 illustrates the SNP Generation. There are two sources (Node 1 and Node 2, marked as red) and two destinations (Node 7 and Node 11, marked as green) in the graph. Node 1 and Node 2 are sources as no edges are flowing into them. Node 7 and Node 11 are destinations as they are network-related methods. Starting from Node 1, Node 2, and ending with Node 7, Node 11, six SNPs can be identified: 1→3→7, 1→4→7, 1→4→8→11, 1→5→9→11, 2→5→9→11 and 2→6→9→11.

SNP preserves the order of methods, and we believe that paths from malware have different patterns compared to those from benign apps. In the following section, we will explain our strategies in extracting features from SNP.

Sensitive Network Subpath (SNS) Generation. To extract features, we also need to extract all the subpaths from each SNP. These subpaths are regarded as patterns for machine learning classification. In our system, we only use the starting node and the ending node to indicate subpath and ignore all the nodes between them. Figure 3.5 shows the process of generating subpaths. For instance, there is a SNP: 1→4→8→11, and all the subpaths are 1→4, 4→8, 8→11, 1→8, 4→11, 1→11. We

ignore the intermediate nodes because these subpaths imply the intermediate nodes. For example, 1→4, 4→8, and 8→11 can imply that 4 and 8 are between 1 and 11. These subpaths are then converted into numeric vectors in the Feature Extraction phase.

3.2.2 Feature Extraction

We now describe the features that our system extracts from the information generated by the Graph Generation phase. Our features come from the generated graphs, paths, and subpaths. We also consider the amount of the generated features for each malware sample as another feature. To quantify this, we use the size of the file that stores the feature of each app. File size provides a good approximation of the volume of each generated feature.

Subpath Existence Feature (F1). We extract all the SNSs for each malicious app in the training set and build a database to store them. We order these subpaths by their names and form a Boolean vector from these subpaths. For each sample in the testing set, GranDroid generates the SNSs for each app, and we check whether these subpaths match any paths stored in the database. A matching subpath indicates a malicious pattern, and the corresponding bit in the Boolean vector is set to 1. Otherwise, the corresponding bit remains at 0. Even though our training set contains more than 20,000 subpaths, the vectorization process can be efficient when a database management system (e.g., SQLite) is used. This subpath vector provides an enhanced feature for classification. The subpaths reflect the programming logic of malware, and therefore, GranDroid inherently captures the relationship among methods in the network-related paths.

Subpath Frequency Feature (F2). As mentioned above, Subpath Existence Feature is extracted to form a numeric vector based on network subpaths of malware

in the training set. To generate Subpath Existence Feature, we check if the identified subpath exists in the database or not. However, in generating Subpath Frequency Feature, we count how many times the subpath appears for each sample.

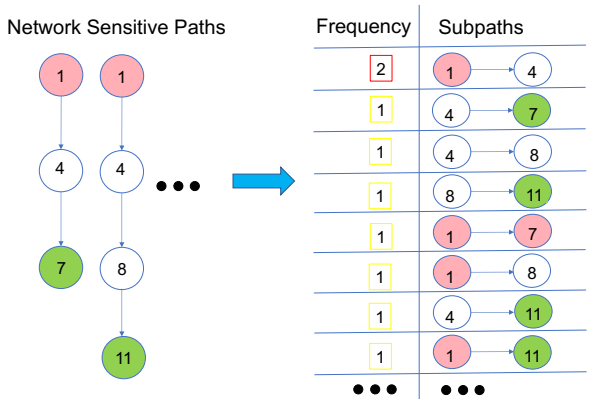

Figure 3.6: Subpath Frequency Feature

To do so, we use both SNP and SNS information. As shown in Figure 3.6, we have two SNPs of an app: 1→4→7 and 1→4→8→11. Subpath 1→4 appears in both sensitive paths, and therefore the frequency of this subpath is 2. Instead of marking 1/0 to build Subpath Existence Feature, we mark the frequency value in the vector position. Intuitively, the frequency of the subpaths can be useful in representing the usage pattern of malicious programming logic.

Path Statistic Feature (F3). We collect several statistic features for each Android app from its Network Sensitive Path. We use nine statistical features that include the lengths of the longest and short paths, the average path length, the number of paths, the number of classes and methods in all paths, the sum of lengths of all paths, and the average numbers of classes and methods per path. We observe that these statistical features can represent malicious network behaviors. For instance, we notice that malware that conducts malicious network behaviors tends to generate shorter and fewer paths than benign apps. These features form a numeric vector to reflect

the unique characteristics of malicious network behaviors that can further represent these paths in greater detail.

File Statistic Feature (F4). In the previous sections, we discuss Method Graph, Class Graph, ClassLoader Graph generated by JITANA. We also present our strategy to generate the SNP, along with methods to extract Subpath Existence Feature, Subpath Frequency Feature, and Path Statistic Feature based on Sensitive Network Path. For each app, we save all of these graphs, paths, and feature information into separate files. We hypothesize that the size of these files can be used to form another numeric feature vector for our machine-learning-based detection system. This is because the file size accurately reflects the amount of generated information that can provide some insight into the complexity of these network paths (e.g., the numbers of API calls and the number of paths). In the end, the attributes we use to form the File Statistic Feature for each app include the size of each graph (method graph, class graph, and class-loader graph) and each generated feature (SNPs, subpaths, subpath existence, subpath frequency, and path statistics).

3.2.3 Detection

In the Detection phase, we apply three well-recognized machine learning algorithms to determine if an Android app has malicious network behaviors automatically.

Our system utilizes four different features (F1 - F4), as previously mentioned. Intuitively, we consider that each of the four feature sets can reflect malicious network behaviors in some specific patterns. For example, in terms of Subpath Existence Feature, one subpath alone or several subpaths appear together might be the pattern to identify malicious behaviors. For Subpath Frequency Feature, we not only consider the existence of subpaths but also how many times each subpath appears in all the paths for each app. The frequency of subpaths might be helpful to construct more

meaningful program logic patterns because the subpaths with higher frequency might be critical to identify malicious behaviors. Regarding Path Statistic Feature, for example, we observed that paths from malware are usually shorter than paths from benign apps so that the path statistic information might improve the accuracy of the machine learning-based system. The size of files where we store graphs and features can also help build our detection system. For example, we observe that the size of the Method Graph from malware is usually smaller than benign apps, so we gather these file size information and form a File Statistic Feature.

To get the best detection result, we need to mine the dependencies of features within each feature set and the relationship between different feature sets. We discussed approaches to convert feature set F1, F2, F3, and F4 into a numeric vector in the previous section. We can simply unionize or aggregate different feature sets into a combined feature set. For example, we can define a new feature set by combining F3 and F4.

Even though there are many supervised learning algorithms to use, we only apply three widely adopted algorithms to build malware detectors. Support Vector Machine (SVM) is commonly used for binary classification based on hyperplane. Decision Tree uses a tree-like structure to make decisions. Random Forest performs classification based on multiple decision trees [68, 69, 70].

3.3 Evaluation

We present the results of our empirical evaluation of GranDroid. We first explain the process of collecting our experimental objects. Next, we report our detection results by using different sets of features. We also compare our methods with other related approaches. Lastly, we report the runtime performance of GranDroid.

3.3.1 Data Collection

Initially, our dataset consists of 20,795 apps from APKPure [71] collected from January 2017 to March 2017. We also downloaded 24,317 malware samples from VirusShare [72]. Note that these samples are newer than those from the Android Genome Project [36], an accessible malware repository that was also used by DroidMiner.

To prevent dataset pollution, we cross-check all our apps from APKPure with VirusTotal. The cross-checking process took 29 days. Some of these benign apps might be identified as malicious by some of the anti-virus scanners, such as AVG, BitDefender, F-Secure, and Kaspersky, and we remove those apps from benign dataset. This is done to ensure that the benign dataset is free of contaminants. After the scan process by VirusTotal, only 11,238 apps from APKPure are considered as benign apps. The malicious samples from the VirusShare have been identified as malicious, and these samples form our malicious dataset.

Next, we need to select apps with network behaviors. We use UIAutomator to build test cases so that each app can perform interactions with the system app. We then measure the code coverage by first statically determining the total number of SNPs. After that, we determine the number of SNPs that UIAutomator can execute. The ratio of dynamic SNPs and static SNPs represents the code coverage. In this study, our average code coverage for all the apps is 22%.

When we execute each app, we also run TCPDUMP packet analyzer in the table to capture the network traffic information and save it as a PCAP file. Usually, malware which conducts malicious network behaviors regularly sends and receives HTTP packets. As such, we only select apps by mainly focusing on their HTTP traffic in the PCAP files. Initially, we have 11,238 benign apps and 24,317 malicious apps. After removing apps without HTTP traffic, only 1,725 malicious apps and 1,625

benign apps remain. To have a balanced dataset, we randomly select 1,500 benign and 1,500 malicious apps to form our dataset.

3.3.2 Detection Result

For each experiment, we run the 10-fold cross-validation on the dataset. We generate different sets of features for these datasets by ways explained in previous sections and apply three different machine learning methods to build our detection system. For each case, we compute several performance metrics to evaluate our system: Accuracy, Precision, Recall, and F-measure. To compare the performance with other methods, we also implement two popular approaches based on our dataset.

Result Based on F1. We first implement our system based on Subpath Existence Feature (F1). Table 3.1:F1 shows the result of applying SVM, Decision Tree, and Random Forest on F1. F1 consists of 22,464 subpaths in total extracted from the training set; thus the numeric vector consists of 22,464 attributes. Building our classifier with the high-dimensional data costs a significant amount of time. To reduce the dimension of this feature set, we apply Principal Component Analysis (PCA) [73] on our dataset. After applying PCA, there are only 30 transformed attributes left to form a new numeric vector. We choose 30 components because more than 99% of the variance can be retained after applying PCA.

We compare four metrics for each classification method in Table 3.1. The accuracy for F1 when using SVM is 79.3%; however, Decision Tree achieves the highest accuracy at 84.3%, and Random Forest achieves the accuracy of 83.3%. It is also worth noting that F1 is similar to the modality feature used by DroidMiner. As such, we can also regard GranDroid's performance based on F1 as that of a reimplemented DroidMiner being applied to our dataset, i.e., the reported results for F1 are representative of the results of DroidMiner.

	F1			F2			F3			F4			F3 ∪ F4		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	79.3	84.3	83.3	60.3	82.7	83.0	88.7	86.3	87.7	50.3	91.0	91.7	50.3	89.0	92.3
II. Precision	71.6	95.6	94.6	55.9	74.7	91.6	92.6	85.2	86.5	50.2	87.7	91.9	50.2	88.7	92.1
III. Recall	97.3	72.0	70.7	97.3	98.7	72.7	84.0	88.0	89.3	100	95.3	91.3	100	89.3	92.7
IV. F-Measure	82.5	82.1	80.9	71.0	85.1	81.0	88.1	86.6	87.9	66.8	91.4	91.6	66.8	89.0	92.4

Table 3.1: The performance of GranDroid using five different features (F1 – F4, F3 & F4) and three different Machine Learning algorithms: Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF).

Result Based on F2. As explained in Section 3.2, Subpath Frequency Feature (F2) is based on F1. It builds a feature vector based on the frequency of each subpath. We also apply PCA to reduce the data dimension. Table 3.1:F2 shows the detection result. For F2, Decision Tree achieves the highest F-measure of 85.1%. It achieves an accuracy of 82.7% with 74.7% precision and 98.7% recall. It appears that F2 only slightly affects the overall performance of our system.

Result Based on F3. F1 and F2 are created by checking the existence and frequency of subpaths in the training set. In essence, these first two vectors can be classified as signature-based features as they correlate the existence of a subpath and its frequency to malware characteristics. For example, if many malware samples contain subpaths S1 and S2, we would regard apps that have both S1 and S2 as malicious. However, if only S1 appears in the training set, S2 may be ignored when generating features. This is a significant shortcoming of this signature-based method.

To overcome this shortcoming, we extract statistical information from SNP to construct Path Statistic Feature (F3). As illustrated in Table 3.1:F3, F3 achieves higher performance than F1 and F2 in terms of all four metrics. This indicates that statistical information related to paths is an essential factor that can improve detection performance. When we apply the three algorithms, we find that SVM performs slightly

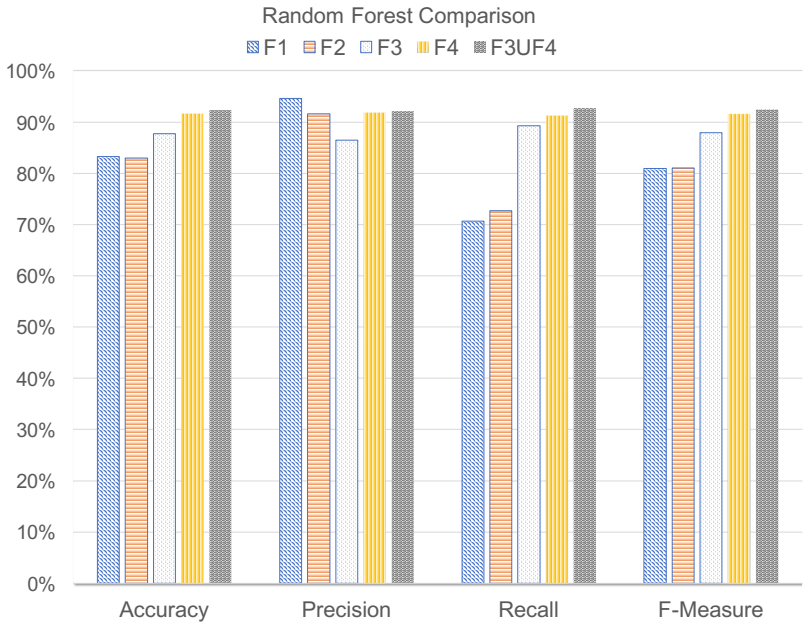

Figure 3.7: Performance of Random Forest

better than Decision Tree and Random Forest for F3 as it achieves an F-measure of 88.1%. In contrast, the other two approaches (Decision Tree and Random Forest) achieve 86.6% and 87.9%, respectively.

Result Based on F4. Besides the statistical feature from paths, we also convert the size of all the graph and feature files into numeric vectors. We refer to this feature as File Statistic Feature (F4). Table 3.1:F4 shows the result based on F4. F4 surprisingly outperforms F1, F2 and F3. When F4 is used with Random Forest, it can achieve an F-measure of 91.6%. This also indicates that the volume of generated features (represented as file sizes) is a strong differentiator between malicious network behaviors and benign ones.

Result Based on F3 $\bigcup$ F4. We have shown that statistical feature sets, F3 and F4, provide higher detection accuracy than F1 and F2. Intuitively, we hypothesize that we may be able to further improve performance by combining F3 and F4. To do so,

we concatenate the feature vector of F3 with the feature vector of F4 and refer to the combined vector as $F3 \cup F4$.

Table 3.1:F3∪F4 validates our hypothesis. In this case, Random Forest achieves 92.3% detection accuracy, which is better than using either feature individually. Figure 3.7 graphically illustrates the comparison of different feature sets via Random Forest, which also shows that F3 ∪ F4 yields the best F-Measure.

3.3.3 Evaluating Aggregated Features

By concatenating F3 and F4, we can achieve better performance than using those two features individually. However, we hypothesize that the richness of path information contained in F1 and F2 may help us identify additional malicious apps not identified by using $F3 \cup F4$. As such, we first experiment with applying Random Forest on a new feature based on concatenating all features ($F1 \cup F2 \cup F3 \cup F4$). We find that the precision and F-measure are significantly worse than the results generated by just using $F3 \cup F4$ due to an increase of false positives.

Next, we take a two-layer approach to combine the *classified results* and not the features. In the first layer, we simply use Random Forest with features $F1$, $F2$, and $F3 \cup F4$, to produce three classification result sets (θ_{F1}, θ_{F2}, $\theta_{F3 \cup F4}$). As Table 3.1 shows that the results in θ_{F1} and θ_{F2} contain false positives, we combat this problem by only using results that appear in both result sets (i.e., $\theta_{F1} \cap \theta_{F2}$). We then add the intersected results to $\theta_{F3 \cup F4}$ to complete the combined result set ($\theta_{combined}$). $\theta_{combined}$ is then used to compare against the ground truth to determine the performance metrics. In summary, we perform the following operations on the three classification result sets produced by the first layer:

$$\theta_{combined} = \theta_{F3 \cup F4} \cup (\theta_{F1} \cap \theta_{F2})$$

Using this approach, we are able to achieve an accuracy of 93.0%, a precision of 92.9%, a recall of 93.5%, and a F-measure of 93.2%. This performance is higher than that of simply using $F3 \cup F4$ as the feature for classification (refer to Table 3.1).

3.3.4 Comparison with Related Approaches

Next, we compare the performance of GranDroid to two prior approaches that have been created to detect network-related malware. However, there are existing dynamic analysis techniques that use network traffic behaviors to detect malware and botnets [13, 74, 75]. These approaches try to achieve the same objective as ours but take a different approach. The major difference is that their works observe dynamic network traffic information while our approach focuses on programming logic that can lead to invocations of network-related methods. The benefit of their approaches is that the detection model is built on actual malicious traffic. If a malicious traffic behavior is detected by executing an app, the app is then classified as malware.

Our approach, on the other hand, does not consider network traffic. Instead, we identify executed network paths and break each path down into subpaths to achieve more precise results. Our work also considers additional paths and methods that are part of the executed component. So our detection model is built using information that is beyond the dynamically generated information via execution. In summary, their approaches use dynamically generated information to build detection models. In contrast, our approach uses the information to explore further related paths and methods that can be useful in detecting malware. Therefore, the amount of information used by our approach to building the detection models lies between the amount of information used to build dynamic analysis models and that of static analysis models. Next, we show how GranDroid performs against two of these purely dynamic analysis approaches.

Approach-1: HTTP Statistic Feature. Prior research efforts have used network traffic information to conduct the malware or botnet detection [74]. Their work mainly focuses on extracting the statistical information from PCAP files, converting such information into features, and then applying machine learning to construct the detection system.

Feature Description
The Number of HTTP Requests
The Number of HTTP Requests per Second
The Number of GET Requests
The Number of GET Requests per Second
The Number of POST Requests
The Number of POST Requests per Second
The Average Amount of Response Data
The Average Amount of Response Data per Second
The Average Amount of Post Data
The Average Amount of Post Data per Second
The Average Length of URL

Table 3.2: Utilized HTTP Statistic Features (Approach-1)

To facilitate a comparison with GranDroid, we reimplement their system. Table 3.2 lists all the extracted features. Table 3.3:Approach-1 shows the detection results. As shown, Random Forest achieves the best F-measure of 80.6%. This is significantly lower than our approach when F3 and F4 are used with Random Forest. As a reminder, our approach achieves the F-measure of 93.2%.

Approach-2: HTTP Header Feature. Next, we compare the performance of GranDroid to that of an approach that uses HTTP header information (four header fields) extracted from network traffic information as features [75]. For each malware sample, they check the corresponding traffic file generated by the sample and build the numeric vector by checking if its header information can be found in the training

set. The vector is a four-bit binary vector, such as <1, 1, 0, 1>. As reported, they build a classification system that can achieve more than 90% detection accuracy [75].

We reimplement their approach and apply it to our dataset. We use four features: host, request URI, request method, and user agent. Table 3.3:Approach-2 shows the detection result. Note that the results of SVM, Decision Tree, and Random Forest are correctly reported as being the same (i.e., F-measure of 78% and accuracy of 73.1%). One reason for this behavior might be that there are only four bits in the vector, indicating a simple structure, and therefore, all three ML methods generate the same result.

	Approach-1			Approach-2		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	57.0	76.0	79.7	73.1	73.1	73.1
II. Precision	53.8	75.3	77.0	65.8	65.8	65.8
III. Recall	99.3	77.3	84.7	96.0	96.0	96.0
IV. F-Measure	69.8	76.3	80.6	78.0	78.0	78.0

Table 3.3: The performance comparison of two different approaches (Approach 1 and Approach 2) and three different Machine Learning algorithms: Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF).

In summary, GranDroid outperforms two other popular approaches in terms of Android malicious network behavior detection. We observe that the overall performance of Random Forest is better than other classifiers. Table 3.4 summarizes the overall performances of all approaches consisting of DroidMiner (F1), Approach-1, Approach-2 and GranDroid. For DroidMiner's results, we use the Decision

Method	DroidMiner (F1) (%)	HTTP (Approach 1) (%)	HTTP (Approach 2) (%)	GranDroid (%)
Accuracy	84.3	79.7	73.1	93.0
F-Measure	80.9	80.6	78	93.2

Table 3.4: Detection Result Comparison

Tree. For GranDroid's results, we use Random Forest. We see that GranDroid achieves higher detection accuracy and F-measure than other approaches. Particularly, GranDroid achieves a 93.0% detection accuracy, much higher than that of DroidMiner (84.3%), that of Approach-1 (79.7%) and that of Approach-2 (73.1%). Furthermore, GranDroid also achieves a higher F-Measure than those of other approaches.

3.3.5 Average Malware Detection Time

The goal of GranDroid is to provide time-sensitive malware detection for security analysts. As such, the steps to detect malware are as follows, assuming that we already build the detection model. In the first step, GranDroid relies on dynamic analysis to generate runtime information. We typically run each app for a fixed time (five to eight minutes) to generate the three graphs (classloader, class, and method call graphs). As previously mentioned, these graphs have been appended with results from the partial static analysis so that the method call graph also includes other unexecuted paths.

Note that our execution time is determined by the amount of time UIAutomator needs to exercise reachable buttons. However, we can also set the time limit by using Monkey[2] to generate event sequences. In the second step, we extract the four features (F1 – F4) to be used for classification. In the third step, we submit these features to our previously generated model to determine whether the submitted app is malicious or benign. As such, the malware detection time consists of the time to complete these three steps. On average, the time to execute an application using UIAutomator was 489 seconds, our feature extraction time was 1.76 seconds, and the model training time using Random Forest, the best performing algorithm, was 1.14 seconds.

Consider a situation when a security analyst needs to vet an app for malicious

[2]Available from: https://developer.android.com/studio/test/monkey.html.

components. With BOUNCER, each app is executed for 5 minutes to observe if there are any malicious behaviors. Our approach also executes an app for about 8 minutes. Within that time, it can achieve the average accuracy and F-measure that are comparable to those achieved by approaches that rely on sound static analysis. Based on this preliminary result, GRANDROID has the potential to significantly increase the effectiveness of dynamic vetting processes commonly used by various organizations without incurring additional vetting time.

In addition, the time requires to train a detection model is also very short (i.e., 1.14 seconds). This means that we can quickly update the model with newly generated features, which indicates that GRANDROID can be practically used by security analysts to perform time-sensitive malware detection.

3.4 Discussion

We have shown that GRANDROID can be quite effective in detecting network-related malware. However, similar to other hybrid analysis or classifier based detectors, GRANDROID also has several limitations.

First, as an approach that relies on executing apps, the quality of event sequences used to exercise the apps can have a significant impact on code coverage. Currently, automatically generating event sequences for Android apps that can reach any specific code location or provide good coverage is still an open research problem [76]. As such, our system can perform better if we have a better way to generate input that can provide higher code coverage. In this regard, employing static analysis would be able to explore more code, but it might not be able to adhere to a strict vetting time budget.

Second, our analysis engine, JITANA, only works on dex code and cannot analyze

native code. As such, implementations of network-related APIs that utilize JNI to execute native code directly would not be fully analyzed. However, analyzing dex code and native code is typically done in two separate steps, so our approach can still be incorporated into any cross-domain analysis approaches.

Third, as a learning-based detector, evasion is a common problem as cybercriminals may try to develop attacks that are so much different than those used in the training dataset [77]. However, as mentioned by the authors of DroidSIFT, semantic- or action-based approaches are more robust and resilient to attack variations than syntax- or signature-based approaches [63]. This is because semantic- or action-based approaches focus their efforts on actual events. It is difficult to instigate a particular network related event (e.g., downloading a malicious component) without utilizing network-related APIs. While it is possible for cybercriminals to evade our detector, it would require significantly more effort than trying to evade signature-based detectors.

Fourth, our current implementation only supports network-related APIs, which are widely used to carry out malicious attacks. However, our approach can be extended to cover other classes of APIs. The key to doing so is to identify relevant APIs that can be exploited to conduct a specific type of attack. For example, a malicious app that destroys the file system would need to use file-related APIs. Fortunately, there are already existing approaches that can help to identify these relevant APIs [78].

3.5 Related Work

Network traffic has been used to detect mobile malware. Notably, prior research efforts aim at detecting malicious behaviors through network flows by capturing actual network traffic [13, 74, 75]. However, these studies have also shown that such systems can be evaded by simply delay malicious behaviors so that only benign traffic is

generated within the observation window. Another important observation is that malicious attacks often occur through invocations of various network APIs, which are provided by the Android framework. Therefore, merely looking at the usage of such APIs is not sufficient to distinguish between benign and malicious apps as both types of apps with network functionalities would need to use those APIs. Our approach tries to overcome this ambiguity by considering execution paths that include framework, system, and the third-party library's code that often invokes network-related APIs [8].

Past research efforts to address this problem statically analyze various program contexts to help distinguish between benign and malicious apps [6, 7, 8, 9, 10, 11]. AppContext creates contexts by combining events that can trigger the security-sensitive behaviors (referred to as *activation events*) with control flow information starting from each entry point to the method call that triggers an activation event (referred to as *context factors*). Machine learning (i.e., SVM) method is then applied to these contexts to detect malware, achieving 92.5% precision and 77.3% recall. The average program analysis time is about 5 minutes per app [10]. However, they analyze much older apps, and newer apps are more complex and can take a longer time to analyze [61].

Another approach is DroidMiner, which applies static program analysis to generate a two-tiered behavior graph to extract modalities (i.e., known logic segments in the graph that correspond to malicious behaviors). It then aggregates these modalities into vectors that can be used to perform classification. Their evaluation result indicates that DroidMiner achieves a detection rate of 95.3% and a false positive rate of 0.4%. The average time spent to extract modalities is 19.8 seconds [8]. It is worth noting that their approach suffers from scalability issues. As the number of methods in an app increases from 5,000 to 19,000, the analysis time also increases from a few seconds to over 250 seconds [79].

The work that is most closely related to our work is DroidSIFT [63], which uses API dependency graphs to classify Android malware. The basic idea is to develop program semantics by establishing an API dependency graph that is then used to construct a feature set. Because API usage ultimately determines the actions that an app can take, focusing on API dependency makes their system more tolerant to techniques that perform dex code transformations or polymorphic variants. This fundamental observation is also the underlying principle of our approach. That is, we also focus on the actions that an app can take instead of focusing on programming syntax. However, their main feature is weighted graph similarity, while our approach considers network path-related features that aim at detecting malicious network behaviors.

While GranDroid takes a hybrid program analysis approach, DroidSIFT, on the other hand, takes a static analysis approach. It uses Soot as the program analysis platform. GranDroid presents several advantages. First, DroidSIFT only focuses on application code and does not include the underlying framework or third-party library code, while our analysis can capture the third-party and framework code. Second, as a static analysis approach, DroidSIFT cannot deal with components that are loaded at runtime through Java reflection or Android Dynamic Code Loading (DCL). Our approach, in contrast, can easily deal with these dynamically loaded components. Third, their analysis time can also vary due to different application size and complexity. They report an average detection time of 3 minutes, but the detection time for some apps can exceed 10 minutes. Thus, the approach cannot guarantee to complete under tight vetting time budget. We have reached out to the authors of DroidSIFT to access their implementation to be used as another baseline system. Unfortunately, we have not received the response.

3.6 Conclusion

In this chapter, we present GranDroid, a graph-based malware detection system that utilizes dynamic analysis and partial static analysis to deliver high detection performance that is comparable to approaches that rely mainly on static analysis. When we use Random Forest with two of our feature sets, we can achieve over 93.2% F-measure, which is about 10% higher than the F-Measure that can be achieved by DroidMiner when applied to our dataset. We also demonstrate that we can achieve this level of performance by spending, on average, 8 minutes per apps on analysis and detection. While we only focus on detecting network-related malware in this work, our approach, by considering sensitive APIs, can be extended to detect other types of malicious apps designed to, for example, drain power or destroy resources. Such an extension is possible because GranDroid focuses its analysis efforts on paths that can lead to specific API invocations. It is thus possible to detect different forms of malware by knowing specific APIs that they use to perform attacks.

Chapter 4

Obfusifier: Obfuscation-resistant Android Malware Detection System

Portions of this material have previously appeared in the following publication:

Z. Li, J. Sun, Q. Yan, W. Srisa-an, and Y. Tsutano, "Obfusifier: Obfuscation-Resistant Android Malware Detection System," in International Conference on Security and Privacy in Communication Systems. Springer, 2019, pp. 214–234.

In this chapter, we propose OBFUSIFIER, a machine-learning-based malware detector that is constructed using features from unobfuscated samples but can provide accurate and robust detection results when security analysts submit obfuscated samples for detection. Code obfuscation is a common approach used by developers to help protect the intellectual properties of their software. The goal of obfuscation is to make code and data unreadable or hard to understand [23]. This, in effect, makes reverse-engineering of their applications more difficult. Typically, there are three major types of obfuscation methods:

1. trivial obfuscations, which most tools can easily handle,

2. data-flow and control-flow obfuscations, which can be Detectable by Static Analysis (DSA), and

3. encryption-based obfuscations, which often involve some forms of encryption to hide the actual code and data.

Recently, various obfuscation techniques have been applied to malicious apps to evade security analysis. These techniques are especially useful in defeating existing malware and virus scanners, which often rely on signature matching or program analysis. As will be shown in Section 4.2, we apply DSA based obfuscation techniques to known malware samples and evaluate them by VirusTotal [24]. The analysis results indicate that many existing techniques deployed by VirusTotal cannot detect obfuscated malware samples and would indicate them as benign.

These DSA based obfuscation techniques change the flow of the program by adding (e.g., junk code insertion), reordering (e.g., code reordering, function inlining, function outlining), or redirecting code (e.g., method indirection), making them effective in defeating malware detectors. These code manipulations can change the signatures of a program and complicate program analysis. Besides, these techniques also change method, variable, and class names so that static analysis techniques that look for previously known values would fail to locate them. Also, note that encryption-based obfuscation techniques are effective in defeating malware detectors because they "hide" the entire code-base and data through encryption. Before running, however, these encrypted applications must be decrypted to reveal the real codes (that may or may not have been obfuscated using DSA techniques) and data for execution. Encryption-based obfuscation is beyond the scope of this work.

Recently, machine learning has become widely used for Android malware detection in the state-of-the-art systems [9, 11, 13, 25, 26, 27]. These existing systems extract

features from benign and malware Android samples to build classifiers to detect malware. Currently, we do not obfuscate the samples used in building classifiers. However, one recent work [28], as well as Section 4.2 have shown that when analysts submit obfuscated Android malware samples to classifiers, these classifiers can miscategorize them since the features used by these classifiers are now more ambiguous due to obfuscation [29].

Our critical insight is that there are portions of codes that malware authors cannot obfuscate because the obfuscation of them would break the functionality. One of these portions is *the API invocations into the Android framework.* As a result, our feature selection focuses mainly on the usage of Android APIs. Our approach then extracts features that are related to such usage. In total, we extract 28 features to build our classifier. We use 4,300 unobfuscated benign apps and 4,300 unobfuscated malware samples obtained from VirusShare. We then test our system using 568 obfuscated malware. The result indicates that our system can achieve 95% precision, recall, and F-measure, corroborating the obfuscation resilience of Obfusifier.

The rest of this chapter is organized as follows. Section 4.1 describes different obfuscation techniques that can change a program's structure and can be used by malware to evade detection. Section 4.2 reports our preliminary results to investigate the effects of obfuscation on malware detection effectiveness. Section 4.3 describes the design and implementation of Obfusifier. Section 4.4 reports the evaluation results. Section 4.5 discusses the limitation of our work. We describe the related work in Section 4.6. The last section concludes this chapter.

4.1 Background on Code Obfuscation

In this work, we use ALAN, a Java-based code obfuscation tool for Android. Next, we describe the obfuscation features supported by ALAN. As will be discussed in Section 4.4, we employed all techniques in a composite fashion to obfuscate our malware samples to make them as challenging as possible to be detected by OBFUSIFIER.

Disassembling & Reassembling. The Dalvik bytecode in the DEX file of the Android app can be disassembled and reassembled. The arrangement of classes, strings, methods in the DEX files can be changed in different ways. In other words, the architecture or the arrangement of the DEX files can be modified, and this transformation creates changes that significantly alter the structures of the program, rendering signature-based detector ineffective.

Repackaging. Developers must sign their Android app before it is released to the market. Cybercriminals can unzip the released Android app and repack it via tools in the Android SDK. After repacking, hackers must sign the repackaged app with their own keys, because they do not have the developers original keys; this newly signed app does not have the same checksum with the original app. This process neutralizes the effectiveness of malware detectors that compare checksums primarily for detection.

Data Encoding. The strings and arrays in the DEX files can be used as signatures to identify malicious behaviors. Encryption of strings and arrays can make signature-based detection ineffective [80].

Code Reordering. This feature aims to change the order of the instructions randomly, and the original execution order is preserved by inserting `goto` instructions. Because this reordering is random, the signature generated by this malware would be significantly different from the signature of the original malware. This approach is by far the strongest obfuscation technique for evading the signature-based detectors [81].

Junk Code Insertion. This technique does not change the programming logic of the code. As such, compared with other transformations, its impact towards the detector is less significant, and malware only obfuscated with Junk Code Insertion are very likely to be detected [82]. There are three common types of junk codes: `nop` instructions, unconditional jumps, and additional registers for garbage operations.

Identifier Renaming. This transformation modifies package and class names with random strings to evade signature-based detection.

Call Indirection. Some malware scanners take advantage of the structure of the method graphs to generate signatures. The original method call can be modified by inserting a newly and randomly generated method before calling the original method. This transformation can insert many irrelevant nodes into the method call graph of an obfuscated app. If a detector is relying on a signature based on a method call graph, this obfuscation technique can be effective in evading the detection. Furthermore, a machine learning detector based on method call graph features would also likely fail to detect malware samples employing this obfuscation technique.

4.2 Effects of Obfuscation on Malware Detection

Obfuscation techniques that can transform the structure of an application have the potential to allow malware to evade detection of many antivirus scanners. To elaborate and quantify the magnitude of this phenomenon, we investigated the effects of obfuscations on the effectiveness of existing virus scanners. The data collection process to conduct our experiments (described next) and the subsequent evaluation of our proposed system is described in Section 4.4.

In the first experiment, we assessed the effect of obfuscation on the accuracy of detection by about 60 scanners deployed by VirusTotal [24]. The experiment involved

randomly selecting 30 malware samples from VirusShare (we downloaded them in June, 2018). We then applied obfuscation using ALAN, a Java-based code obfuscation tool that is capable of applying several types of structure-altering transformations, including code reordering, junk code insertion and call indirection directly on DEX code of an Android app [28,83]. Once these apps have been obfuscated, we resubmitted them for scanning again on VirusTotal. We report the scanning result in Figure 4.1.

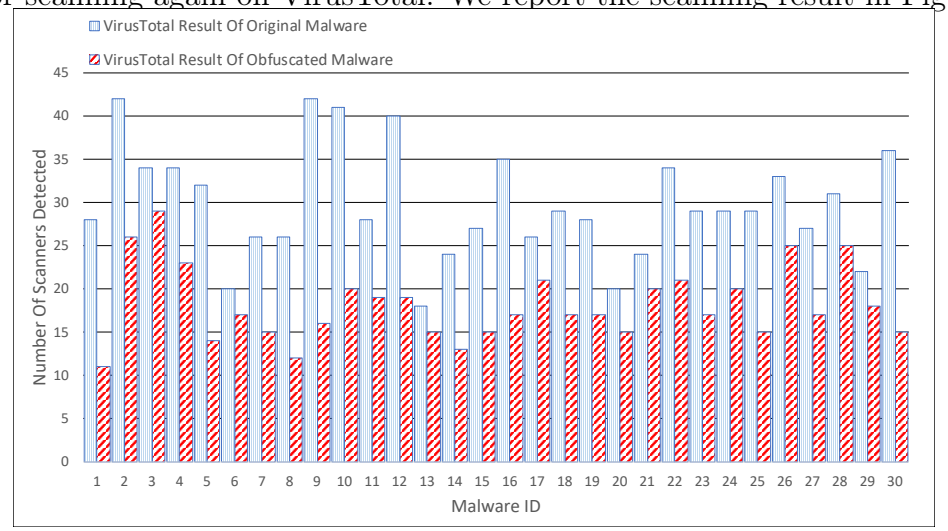

Figure 4.1: The Difference in Detection Rate of Original and Obfuscated Malware

In the figure, the horizontal axis lists malware ID (from 1 to 30). The vertical axis presents how many antivirus scanners identify an app as malware. The light blue bar is the detection number for the original app, and the red twilled bar is the result of the obfuscated app—the number of scanners that can accurately identify each obfuscated app as malicious decreases dramatically. The most significant drop occurs in App 9 as 42 scanners detect its unobfuscated version, but only 16 scanners detect its obfuscated version–a reduction of 62%.

In the second experiment, we focused on the accuracy of 14 popular scanners in detecting obfuscated malware. We randomly obfuscated 1,540 apps using ALAN. Table 4.1 shows the detection difference between these 1,540 unobfuscated malicious apps and their obfuscated versions. The scanner Antiy-AVL can identify 1,427 as malware

before obfuscation, but can only identify 260 obfuscated versions. The difference between McAfee and Symantec is 641 before and after obfuscation, which is surprisingly high. Ad-Aware and Baidu cannot detect obfuscated malware at all. We checked 60 scanners, and the number of scanners which could still identify the obfuscated apps as malicious decreased by 34.4% on average. Prior work called DroidChameleon [28] has shown that 10 popular antivirus products such as Kaspersky, AVG, and Symantec, lose their detection effectiveness when used with obfuscated malware samples.

Scanner	Number of detected (original)	Number of detected (obfuscated)	Differece
Antiy-AVL	1427	260	1167
MAX	1429	463	966
Comodo	999	122	877
F-Prot	830	54	776
Alibaba	975	291	684
K7GW	1348	679	669
McAfee	1446	805	641
Symantec	763	122	641
McAfee-GW-Edition	1265	669	596
DrWeb	1119	607	512
BitDefender	464	20	444
eScan	434	2	432
Ad-Aware	430	0	430
Baidu	308	0	308

Table 4.1: Detection Difference By Scanners

We conducted the third experiment to understand the effects of obfuscation on malware detection effectiveness of existing scanners. To do so, we focused our analysis on a malware sample that belongs to Adware:android/dowgin [84] family, which is an advertising module that can leak or harvest information such as its IMEI number, location, and contact information from the device.

We then obfuscated this malware sample using Alan [85]. Before it was obfuscated, 20 scanners from the VirusTotal [24] were able to identify it as malware. However,

after obfuscation, only 8 scanners could detect it. We statically analyzed this app and its obfuscated counterpart. We checked its method call graph and found that there were 4,948 methods, 7,244 function calls before obfuscation. After obfuscation, the number of methods increased to 6,387, and the number of function calls increased to 8,683; the obfuscation process inserts some additional methods

Figure 4.2: Obfuscation Process

Figure 4.2 illustrates this obfuscation process. we have A→B as the original function call, but in the obfuscated graph, we have A→C, C→B instead. This is called Call Indirection. The structure of the original method graph is modified so that signature-based virus detectors would not be able to detect such changes.

We also compared their DEX codes. There were 93,077 lines in the original DEX file, but there were 148,819 lines after obfuscation. Scanners that rely on the order of the instruction as signatures would be ineffective by such changes. Prior work called RevealDroid [29] has shown that even for machine learning-based detectors, obfuscation is still problematic.

There is a need to create a malware detector that maintains its effectiveness despite obfuscation. Our approach, Obfusifier applies static analysis to identify code that cannot be obfuscated and then efficiently extracts useful features to build a machine learning-based detection system. In the next section, we introduce our proposed system.

4.3 Introducing Obfusifier

The main goal of the Obfusifier is to identify malicious apps, transformed via different obfuscation techniques, and challenging to detect via common antivirus

scanners. Thus, the features selected must satisfy the following four policies. First, these features must give a good representation of the difference between malware and benign apps. Second, they must produce a very high detection accuracy when handling unobfuscated malware. Third, the detection time must be sufficiently short for real-world application scenarios. Fourth, the system must be resilient when used to detect o

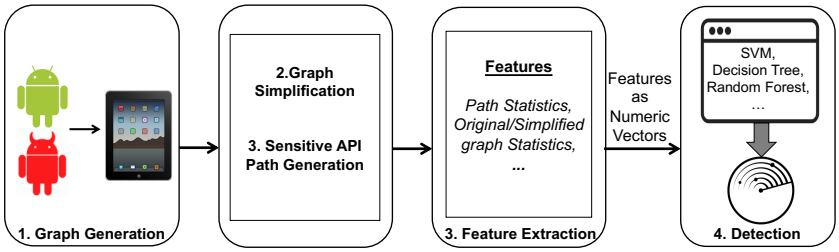

Figure 4.3: System Architecture

In this section, we describe the architectural overview of our proposed system, which operates in five phases: Graph Generation, Graph Simplification, Sensitive API Path Generation, Feature Extraction, and Malware Detection, as shown in Figure 4.3. Next, we will describe each phase in turn.

4.3.1 Graph Generation

The method graph is a good representation of the malware structure based on the calling relationship between different methods and subroutines. Each node in the graph represents a method, and a directed edge from one node to the other shows their calling relationship. We implement OBFUSIFIER based on JITANA [61], a high-performance hybrid program analysis tool to perform static and dynamic program analysis. JITANA can analyze DEX file, which includes the user-defined code, third party library code,

framework code (including implementations of various Android APIs), and underlying system code. JITANA analyzes the classes to uncover all methods and generates the method graph for the app. OBFUSIFIER takes advantage of the calling relationship of methods to detect malware. As shown in Figure 4.4, blocks represent methods, and directed edges indicate calling relationships among methods. Each block contains the name of the method, its modifiers, and the class name to which this method belongs. OBFUSIFIER captures the interactions of these methods, understands the semantic information that can help detect malware.

There are three types of methods in the graph: Android API method, system-level method, and user-defined method. All of these methods can be exploited by malware writers to conduct malicious behaviors. The user-defined methods and the classes to which the methods belong can be renamed, so that the malware can evade the antivirus scanners. As such, only relying on the original method graph may not enough to build a obfuscation-resistant detector because of the negative impact of code obfuscation. Lightweight features can be extracted from these method graph to build the malware detection system.

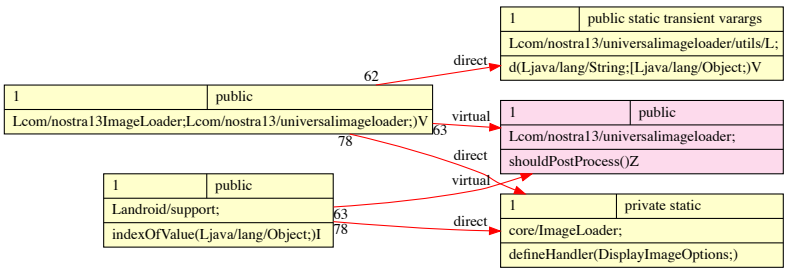

Figure 4.4: Method Graph

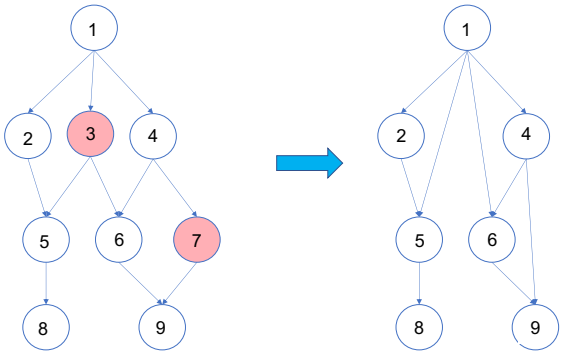

Figure 4.5: Graph Simplification Process

4.3.2 Graph Simplification

Our critical insight is that the Android APIs and system-level methods cannot be transformed by code obfuscation, and we can exploit these characteristics to extract obfuscation-resistant features. Google publishes Android APIs, so we can easily create a list of these APIs. System-level methods include the Android OS source code and the Linux kernel source code, so it is not as convenient to gather all these methods, and therefore, we do not collect them. We simply rely on the list of Android APIs that we collected.

To generate obfuscation-resistant graphs, we only keep the Android APIs in the original method graph and ignore the system-level methods, user-defined methods, and those from third-party libraries. For example, as shown in the Figure 4.5, nodes 1, 2, 4, 5, 6, 8 and 9 are Android APIs, and node 3 and node 7 are system-level or user-defined method. In this situation, our system simply ignores nodes 3 and 7, and generates a new call edge from node 1 to node 5 and another edge from node 4 to node 7. By doing this, we remove two nodes and combine four method calls into two.

By performing graph simplification, we can reconstruct a graph that is obfuscation-resistant while keeping the structural and semantic information concerning Android

API usage of the original graph. In addition, the API-only graph contain as much as an order of magnitude less information than the original graph, allowing feature extraction to be much faster, especially during the path traversal phase.

4.3.3 Sensitive API Path (SAP) Generation

SAP is the program execution path from one node to the other in the API-only graph. An SAP can be used to differentiate between malicious and benign behaviors. To generate SAPs, we need to select the critical APIs which are used for path generation, since these APIs reflect the semantic information about the behaviors of apps. We analyze the call frequency of APIs and keep APIs which are used only by malware because they can directly reflect the malicious behaviors. We also extract some frequently used APIs by both malware and benign apps. Even though both use them, the additional program context (e.g., method call characteristics) can still represent the difference between malware and benign apps. In the API-only graph, we consider all nodes whose in-degree are zero as sources and nodes whose out-degree are zero as destinations. OBFUSIFIER generates SAPs from sources to destinations via depth-first search (DFS).

Figure 4.6 illustrates the process of generating SAPs. In the figure, there are two sources (Node 1 and Node 2, marked as green) and two destinations (Node 4 and Node 10, marked as red) in the graph. Node 1 and Node 2 are sources (in-degree is zero) as no edges are flowing into them. Node 4 and Node 10 are destinations as they are selected and frequently used APIs. Starting from Node 1, Node 2, and ending with Node 4, Node 10, there are four SAPs:

1→4, 1→5→9→10, 2→5→9→10 and 2→6→9→10.

SAP reflects the running behaviors of apps, and these paths form patterns, which can be useful to distinguish between malicious apps and benign ones.

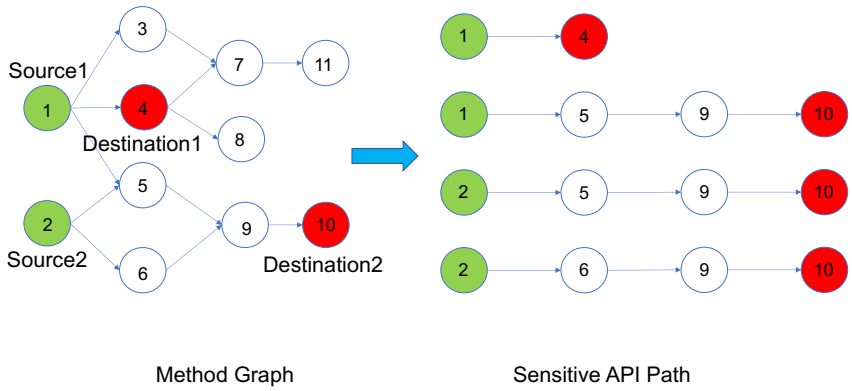

Figure 4.6: Sensitive API Path(SAP) Generation

4.3.4 Feature Extraction

We now describe the features, which our system extracts the original graphs, API-only graphs, and SAPs.

Path Statistic Feature (F1). We collect seven statistic features from Sensitive API Path. These features include the lengths of the longest and short paths, the number of paths, the sum of lengths of all paths, the average length per path, the number of methods in all paths, and the average number of methods per path. These statistical features can indicate path characteristics and represent malicious behaviors. For example, malware that conducts malicious behaviors tends to generate shorter and fewer paths than benign apps. Since the paths in API-only graphs only consist of APIs, this feature set is not affected by code obfuscation. These features are concatenated and construct a numeric vector to reflect the unique characteristics of app behaviors that can further represent these paths in detail.

Simplified Graph Statistic Feature (F2). We select eight features from the simplified graph. They are the number of methods, the number of classes and the number of edges in the graph, graph density, the average in-degree and out-degree of the graph, the number of sources (nodes of which in-degree are zero) and destinations

(nodes of which out-degree are zero). Compared with the original graph, the simplified graph does not include the renamed user-defined classes and methods.

Original Graph Statistic Feature (F3). We also collect eight features from the original graph. These features are the same as F2. Even though some of the methods in the original graph are obfuscated, we still think these graphs can reflect malicious behaviors. Keeping features from the original graph might still be useful to identify malware, whether it is obfuscated or not.

Other Statistic Feature (F4). We save the original graph, simplified graph and SAP in separate files, and use the size of these files to form three new numeric features. We assume that the size of the file can reflect the amount of generated information, which indicates the complexity of these graphs and paths. We also calculate the ratio of the number of methods in the original graph to that in the simplified graph. We also calculate the ratio of the number of classes in the original graph to that in the simplified graph. The ratio can reflect the level of obfuscation accurately, and we hypothesize this will contribute to the malware detection too. Finally, we form F4 as a vector of five features.

4.3.5 Detection

In the Detection phase, we apply three well-recognized machine learning algorithms to determine if an Android app is malicious or benign. Our proposed system utilizes four different features (F1 - F4), as previously mentioned. Intuitively, we consider that each of the four feature sets can reflect malicious behaviors in some specific patterns. For API-only Graph Statistic Feature, because we remove all the user-defined classes and methods, which are usually transformed by obfuscation techniques, to generate a simplified graph, these features are less likely affected by obfuscation. Besides, these features also reflect the structural difference between malware and benign apps. For

example, we find that benign apps usually have more sources, destinations, classes, and methods than malware. Thus, we need the Original Graph Statistic Feature because the graph simplification process also eliminates some of the original structural characteristics of graphs. The size of files where we store graphs and paths can also help build our detection system, for example, we observe that the size of the file storing graph and paths from malware is usually smaller than benign apps. We also notice that the graph density from malware is greater than benign apps, so we gather these file size and graph density information to form Other Statistic Feature.

We evaluated the performance of our system by using different feature sets individually. Also, we also concatenated different feature sets to construct the combined new feature set and assess its impact on the detection result. In terms of the classification policy, we apply three popular algorithms: Decision Tree, Random Forest, and Support Vector Machine(SVM) [68, 69, 70]. Prior work shows that these machine learning algorithms can achieve superior performance in addressing classification problems experienced by OBFUSIFIER.

4.4 Empirical Evaluation

To evaluate OBFUSIFIER, we show its detection performance in terms of accuracy, precision, recall, and f-measure. We also illustrate its resistance against obfuscation, and ultimately its runtime performance. We first present the process of collecting our experimental apps, both benign and malicious, and explain how to transform malware using obfuscation techniques. Next, we show our detection results based on different sets of features. We also compare our system with several related approaches. Finally, we present the runtime performance of OBFUSIFIER.

4.4.1 Experimental Objects

To evaluate the performance of our proposed system, we collected a dataset containing both malware and benign apps. We downloaded 24,317 malware samples from VirusShare [72]. Compared to Android Genome Project [36], which is often used by many researchers [13, 86], we included more malware samples, and they are also newer. However, they also include many of the samples in the Android Genome Project. For benign apps, we collect 20,795 apps from APKPure [71], a third party website providing Android apps. Note that we also used these collected apps to conduct the experiment in Section 4.2.

To avoid polluting our benign dataset with malware samples, we cross-checked all apps downloaded from APKPure with VirusTotal, and remove those apps identified as malware by VirusTotal from the benign dataset. After we completed the cross-checking process, there are only 11,238 apps left for the benign dataset. This checking process took 29 days. Note that VirusTotal identifies all the samples in the malware dataset as malicious.

4.4.2 Experimental Methodology

To evaluate the performance of our system, and guarantee the balance of the data, we randomly chose 4,300 malicious samples, 4,300 benign apps as training/testing samples from our dataset. We also applied 10-fold cross-validation.

To verify Obfusifier's ability to resist code obfuscation, we randomly choose another 568 benign apps and 568 malware as an additional testing set. We transform the additional 568 malicious samples using Alan by applying all its transformations mentioned in Section 4.1.

As previously mentioned, our system utilizes four sets of features (F1, F2, F3, F4)

to construct our classifier and perform detection. We also use four metrics to evaluate the performance of our system: Accuracy, Precision, Recall, and F-measure. We also assess the performance of our system by combinations of different sets of features. For example, by concatenating F1 and F2(F1 U F2), we form a new feature vector. Besides, we also compare our system with several popular approaches based on very similar dataset.

We used Macbook Pro with a dual-core 2.8 GHz Intel Core i7 running OS-X High Sierra and 16 GB of 1.33 GHz main memory to perform our evaluations.

4.4.3 Detection Result

We discuss two usage scenarios in this section. The first scenario is when we evaluate our classification system by 10-fold cross-validation. All samples in the dataset are unobfuscated apps. We use unobfuscated apps to illustrate that the classifier is effective and can detect unobfuscated malware with high accuracy. In a typical application, we imagine that security analysts would use obtainable, unobfuscated malware, and benign samples for training and testing. Table 4.2 reports our result.

In the second scenario, we continued to use the original unobfuscated samples as in the first scenario for training; i.e., we used the same classifier built in the first scenario. However, we expand the testing dataset to include 568 more benign apps and then 568 more obfuscated malware samples (using ALAN) so that we can evaluate the ability of our system to maintain accurate detection despite obfuscation. Note that we applied all obfuscation methods supported by ALAN to make detection more challenging, and our testing dataset also includes the same number of unobfuscated benign apps to maintain balance.

Table 4.3 shows the result of the second scenario, in which all the testing malware samples are obfuscated. But above all, in both cases, there are not obfuscated apps

in the training dataset, which means we do not need obfuscated apps in the training phase, and this characteristic guarantees that our system is robust and able to resist to obfuscated malware. One most significant contribution of our system is that, in the training phase, even though no obfuscated apps are needed, the system can still successfully identify malware. Next, we discuss the results based on each feature.

Result Based on F1. Based on the Path Statistic Feature (F1), we implement and evaluate our learning-based system. Table 4.2–F1 shows the detection result without obfuscation in terms of three approaches: SVM, Decision Tree, and Random Forest. F1 is constructed by seven statistic features from Sensitive API Path (SAP). We generate all the SAPs from the API-only graphs. Because obfuscation has little to no effects on this graph, this feature set is essential to build the proposed obfuscation-resistant malware detection system.

In the first scenario (without obfuscation), we calculate the four metrics, as shown in Table 4.2–F1 for each classification technique based on F1. The Random Forest and Decision Tree achieve the f-measure of 87.9% and 85.3%, respectively. On the other hand, SVM only yields the f-measure of 60.2%. The Random Forest also has an accuracy of 87.6%, which outperforms the SVM and Decision Tree. This result indicates that our system can incorrectly detect malware if we only rely on F1.

In the second scenario, we assess our system with obfuscated apps. As shown in Table 4.3–F1. Similar to the case without obfuscation, Random Forest performs better than SVM and Decision Tree. It has an accuracy of 89.7% and the f-measure of 89.8%. This result shows that our system is somewhat effective at identifying obfuscated Android malware. Interestingly, by checking accuracy and f-measure for F1, the result with obfuscation in Table 4.3–F1 is slightly better than the one without obfuscation in Table 4.2–F1. We achieve this result because the impact of these transformations on the SAP feature is minimal, so the system trained using SAP can resist obfuscation

naturally. However, the SAP is from the simplified graph, which removes a lot of user-defined methods from the original graph. Because of this, some contexts of the program, which are helpful to recognize the non-obfuscated malware, are missing. As such, F1 performs better when handling obfuscated malware.

Result Based on F2. API-only Graph Statistic Feature (F2) is the feature vector directly from the simplified graph. This feature is significant because it reflects the structural difference between malware and benign apps, and the influence of obfuscation on F2 is minimal due to the elimination of all the newly added methods or renamed methods(Junk Code and Call Indirection) in the obfuscated and original graph.

Table 4.2–F2 shows the evaluation result without obfuscation. For F2, Random Forest achieves the best accuracy of 92.9%. It also attains the highest f-measure of 93.1% with 91.0% precision and 95.2% recall. The result of F2 is better than that of F1. This observation indicates that features directly from the graph are more effective than features from paths.

Table 4.3–F2 shows that our system is very effective even dealing with obfuscated malware. In terms of Random Forest, we can achieve very high accuracy of 94.3% and 94.6% f-measure. This result validates our assumption that features from these simplified API-only graphs, in which obfuscated methods are removed, are very useful in identifying malware and resisting the negative impact of code obfuscation. As such, a system trained based on F2 is more obfuscation-resistant.

Result Based on F3. F1 and F2 are created based on API-only graphs to reduce the impact of code obfuscation on malware detection. Based on our reported results, these two feature sets not only help to identify non-obfuscated apps, but also show a remarkable efficacy when dealing with obfuscated malware. However, when graphs are simplified, some structural information that is beneficial to distinguish non-obfuscated malware might be lost. In the case that original malware samples are also available,

	F1			F2			F3			F4			F1∪F2∪F3∪F4		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	71.3	85.3	87.6	70.2	91.0	92.9	69.6	92.3	94.0	63.9	90.8	92.6	63.9	94.0	95.5
II. Precision	97.8	82.8	85.7	99.6	89.9	91.0	99.9	90.2	92.2	99.9	88.9	91.3	99.9	92.4	93.9
III. Recall	43.5	89.1	90.3	40.5	92.3	95.2	39.2	95.0	96.2	27.7	93.2	94.3	27.9	95.9	97.3
IV. F-Measure	60.2	85.3	87.9	57.6	91.1	93.1	56.3	92.5	94.1	43.4	91.0	92.7	43.6	94.1	95.5

Table 4.2: The performance of Obfusifier on non-obfuscated apps using five different features (F1 – F4, F1UF2UF3UF4) and three different Machine Learning algorithms: Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF).

	F1			F2			F3			F4			F1∪F2∪F3∪F4		
	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)	SVM (%)	DT (%)	RF (%)
I. Accuracy	73.6	89.2	89.7	71.3	92.9	94.3	51.5	90.9	80.6	50.1	89.3	91.5	50.0	93.3	90.2
II. Precision	99.3	87.9	89.1	100.0	90.5	91.2	100.0	91.7	90.4	0	88.9	92.3	0	92.5	92.7
III. Recall	47.4	91.0	90.5	42.7	95.8	98.2	1.9	89.9	68.4	0	89.9	90.7	0	94.2	87.3
IV. F-Measure	64.2	89.4	89.8	59.8	93.1	94.6	3.8	90.8	77.9	0	89.4	91.5	0	93.4	89.9

Table 4.3: The performance of Obfusifier with obfuscated apps as testing set

there is a potential to improve effectiveness by extracting features from the original method graph to form a feature set called Original Graph Statistic Feature (F3). The meaning of each feature in F3 is the same as F2.

As illustrated in Table 4.2–F3, F3 achieves higher performance than F1 and F2 in all three classification techniques. Random Forest performs better than Decision Tree and SVM for F3 as it attains f-measure of 94.1% while the other two approaches (SVM and Decision Tree) achieve 56.3% and 92.5%, respectively.

For obfuscated malware, the performance of F3 is not as good as F1 and F2. As illustrated in Table 4.3–F3, most of the metrics show F3 cannot handle the obfuscated apps as good as F1 and F2. For example, in terms of Random Forest, F3 only has a f-measure of 77.9%, which is lower than those of F1 and F2, which achieve 89.8%

and 94.6%, respectively. As such, F3 alone is not a sufficient feature set to achieve obfuscation-resistant capability.

Result Based on F4. We transform the sizes of several files, the ratio of the number of methods in original graph to the number in the simplified graph, and the ratio of the number of classes in original graph to the number in the simplified graph into a new feature referred to Other Statistic Feature (F4). We assume that these file sizes and the ratios are also efficient features for distinguishing between malware and benign apps.

Table 4.2–F4 shows the detection result on non-obfuscated malware. Random Forest achieves the highest accuracy of 92.6% and f-measure 92.7%, compared with SVM and Decision Tree. Table 4.3–F4 illustrates the result with obfuscation. Random Forest also performs best yielding f-measure of 91.5%. Results based on F4 verify our assumption, and these sizes of files and ratios can provide another efficient way to build the malware detection system.

Result Based on F1 U F2 U F3 U F4. By aggregating all our feature sets, as shown in Table 4.2, in terms of Random Forest, we achieve the accuracy of 95.5% and f-measure of 95.5%. Table 4.3 shows that the combination of all feature sets also works well for obfuscated malware.

4.4.4 Comparison with Related Approaches

Next, we compare the performance of Obfusifier with other research efforts including RevealDroid [29], MUDFLOW [26], Adagio [25] and Drebin [9]. More information about these systems are available in Section 4.6.

In this work, we relied on the data provided in the RevealDroid paper as a base for comparison. They conducted an investigation that compared the detection

performance of RevealDroid with the other three systems. Thus, we simply compared our system's performance against the reported performances.

Another noticeable difference is that RevealDroid obfuscated their malware using DroidChameleon [28]. RevealDroid applies four sets of transformations on there dataset, including call indirection, rename classes, encrypt arrays, and encrypt strings. We, on the other hand, obfuscated our dataset with Alan, and enabled all transformations described in Section 4.1. In our approach, "Data Encoding" technique includes the "Encrypt Arrays and Encrypt Strings" by DroidChameleon, and our "Identifier Renaming" includes "Rename Classes" by DroidChameleon. The level of obfuscation in our dataset is higher than RevealDroid, so our transformed malware should be more difficult to detect.

The malicious apps used to investigate RevealDroid are from Android Malware Genome [36], the Drebin dataset [9] and VirusShare [72]. Our malicious dataset is only from VirusShare. However, the samples on VirusShare contain similar apps in Android Malware Genome and Drebin dataset. The similarity of the dataset ensures the fairness of comparisons.

When comparing with the other four systems, we consider two scenarios. The first scenario is testing the non-obfuscated malware (without obfuscation). The second scenario is testing the obfuscated malware (with obfuscation). In the first scenario, RevealDroid splits a dataset, including 1,742 benign apps and 7,989 malicious ones into two parts evenly. One part is the training dataset, and the other part is for testing. The training dataset has half of the benign apps and half of the malicious apps. For this case, we also split our dataset consisting of 4,300 benign apps and 4,300 malicious apps randomly into two parts evenly, one part for training, and the other part for testing. In the second scenario, RevealDroid has 7,995 malicious apps and 878 benign apps in the training set, and 1,188 obfuscated malicious apps and 869

benign apps for testing. Similar to their dataset, there are 4,300 benign apps and 4,300 malicious ones in our training set, and we form a testing set with 568 benign apps and 568 obfuscated malicious ones.

Note that all of our samples are chosen and split randomly. Compared with the imbalanced dataset from RevealDroid, our dataset is very balanced. When training imbalanced data, which the number of malware is greater than benign apps, the classifier often favors the majority class and form a biased prediction model. The imbalance in the testing set can cause significant inaccuracy.

Table 4.4 shows the comparison result without obfuscation. Table 4.5 presents the comparison result with obfuscated malware. Without obfuscation, as illustrated in Table 4.4, Drebin shows the best performance with the average precision, recall and f-measure 99%, we think this is because Drebin gathers all types of features, such as permission, API call, intents and the diversity of the feature set plays a significant role to detect malware. Obfusifier has the average f-measure of 96%, which is the same as RevealDroid. Even though the performance is not as good as Drebin, both Obfusifier and RevealDroid outperform Adagio, of which average f-measure is 90%. MUDFLOW has the worst result, with only average 71% f-measure and 66% recall.

With obfuscation, as illustrated in Table 4.5, Obfusifier outperforms all other four systems. This result is from feature combinations of F1 U F2 U F4. Note that F3 is a feature set extracted from the original method graph, so the F3 feature set is not obfuscation resistant. It achieves surprisingly high metrics, with an average of 95% precision, recall, and f-measure. Note that the f-measure of MUDFLOW with obfuscation is only 74%. This result is very close to the result without obfuscation (f-measure of 71%).

We suspect that this is because the obfuscation techniques do not influence its

feature sets. DREBIN shows poor performance with obfuscation; the average precision, recall and f-measure are 0%. This is because all of DREBIN's feature sets are negatively influenced by obfuscation, and this result indicates that DREBIN is not resilient against obfuscation. ADAGIO achieves the average f-measure 62% with obfuscation, but this is not as good as its result (f-measure 90%) without obfuscation. Still, it shows the ability to detect obfuscated malware. The average f-measure and recall of REVEALDROID is 85%, which is not as high as OBFUSIFIER.

	MUDFLOW (%)			RevealDroid (%)			Adagio (%)			Drebin (%)			Obfusifier (%)		
	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm
Ben	85	34	49	90	88	89	90	76	83	97	100	98	97	94	95
Mal	87	99	93	97	98	98	95	98	96	100	99	100	94	97	96
AVG	86	66	71	96	96	96	92	87	90	99	99	99	96	96	96

Table 4.4: Comparison Without Obfuscation (Pr = Precision, Re = Recall, and Fm = F-measure)

	MUDFLOW (%)			RevealDroid (%)			Adagio (%)			Drebin (%)			Obfusifier (%)		
	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm	Pr	Re	Fm
Ben	98	47	64	91	72	80	54	73	62	42	100	59	97	92	95
Mal	72	99	84	82	95	88	73	54	62	0	0	0	93	98	95
AVG	88	73	74	86	85	85	63	63	62	18	42	25	95	95	95

Table 4.5: Comparison With Other Methods (Pr = Precision, Re = Recall, and Fm = F-measure)

4.4.5 Runtime Performance

For real-world applications, a malware detector must be both effective and efficient. To evaluate the efficiency of OBFUSIFIER, we measured the time taken to analyze and detect a malware sample. As part of the analysis, one critical factor that can affect efficiency is the time to train the classification model and the time needed to test each app. The training time is the time to build the prediction model. The testing time is the average number to test each app. Another key factor is the time we spend to

statically analyze apps and extract its features. We also list the average time needed to analyze each app in different phases: Graph Generation, Graph Simplification, SAP Generation, and Feature Extraction. We measure the time of 100 apps (50 benign and 50 malicious apps, respectively) and calculate the average execution time for each app in each phase. We found the system took the average total of 35.06 seconds to analyze each app, generate graphs, simplify paths, and extract features. This runtime result should be acceptable for detecting obfuscated and sophisticated malware in real-world settings.

4.5 Discussion

Our evaluations have shown OBFUSIFIER's robustness, and its ability to handle obfuscated Android with high efficiency and accuracy. However, there are still some limitations of our system.

First, we only obfuscate malicious apps using ALAN. According to the results from VirusTotal, ALAN provides several very effective obfuscation techniques that help malware evade many existing antivirus scanners. However, to verify OBFUSIFIER's ability to deal with different obfuscation techniques, we plan to experiment with more Android obfuscation tools, such as DashO [87], DexGuard [88] to transform malware.

Second, our system cannot handle the malware transformed by the obfuscation on the native code. Malware authors can take advantage of this loop-hole to encrypt the strings and arrays in the native code, and then decrypt them during runtime to hide the malicious behaviors. One important tool that can close this loop-hole is OBFUSCATOR-LLVM [89], which targets the native code obfuscation. We plan to experiment with this tool and attempt to integrate it into our workflow.

Third, our system is based on static analysis of the DEX code, but if the DEX code

is encrypted and then decrypted at runtime, we cannot capture its method graph and malicious behaviors. A special obfuscation technique called packing [90], which is used to protect Android apps from reverse engineering. It creates a wrapper application and hides the original DEX code. This wrapper app loads necessary libraries to unpack the original code at runtime.

4.6 Related Work

In this section, we describe closely related work, including the four baseline systems used in Section 4.4 and other prior works about malware detection.

Garcia et al. [29] introduced RevealDroid as a lightweight machine learning-based system to detect Android malware and identify Android malware families. It constructs features from the Android API usage, reflection characteristic, and native binaries of the app. The evaluation shows that RevealDroid can detect malware (both non-obfuscated and obfuscated malware) and identify malware families with high accuracy.

MUDFLOW [26] is built on the static analysis tool FLOWDROID [91]. It extracts the normal data flow from benign apps as patterns, mines these benign patterns, and use these pattern to identify malicious behaviors automatically. The novelty of their work is that they only use information from benign apps to train their system and identify abnormal flows in malicious apps. Our evaluations indicate that MUDFLOW has some ability to detect obfuscated malware, with a precision of 88% and f-measure of 74%. Adagio [25] extracts the function call from Android apps and map these function calls to features, and build a machine learning system based on these features. As shown, the proposed system loses its accuracy when used with obfuscated malware samples.

DREBIN [9] is a machine learning-based malware detector. It performs static analysis on Android apps to collect many features such as permission, API calls, intents from app's code and the Manifest file. It embeds them in a vector space that can be used to discover patterns of malware. These patterns are then used to build a machine learning detection system. The system is accurate, but it requires running on a rooted device. As shown, DREBIN is not able to detect obfuscated malware.

APPCONTEXT [92] is a machine-learning-based malware detector that focusing on the context difference between malware and benign apps. It leverages SOOT [93] as the static analysis engine and uses the permission mappings offered by PScout [94] to extracts the contexts in Android components, Android permissions, and Intent. They achieve 92.5% precision and 77.3%, which is lower than our system. The average analysis time of APPCONTEXT for each app is about 5 minutes [92], but we only need 35 seconds. This behavior-based approach might be able to resist obfuscation, and we hope we can its source code and assess its performance over obfuscated malware in the future.

DROIDMINER [79] is a system that mines the program logic from Android malware, extract this logic to modalities ordered sequence of APIs, and construct malicious patterns for malware detection. It builds a method call graph for each app and control flow graph. It also generates modalities (API paths and subpaths) from sensitive methods. A feature vector based on the existence of modalities is formed for classification. They replace user-defined methods with framework API functions. We, on the other hand, remove the user-defined methods for efficiency.

DROIDSIFT [63] is also a machine learning-based detector based on static analysis. They generate weighted contextual API dependency graphs, build graph databases, and construct a graph-based feature vector by performing graph similarity queries.

Their features represent program behaviors at the semantic level. Note that the average detection time is about 176.8 seconds [63], but we only need 35 seconds.

4.7 Conclusion

We introduce OBFUSIFIER, a machine learning based malware detection system that is highly resistant to code obfuscation. The critical insight is that obfuscation cannot be applied to portions of code that include calls to Android APIs, kernel functions, and third party library APIs. As such, our system extracts mainly features based on these portions of code unaffected by obfuscation. In total, we use four feature sets consisting of 28 features. Our results showed that the effectiveness of the system is not affected by obfuscation. When used to detect non-obfuscated malware, the system can achieve an average f-measure of 96%. However, when these samples are obfuscated, the system can achieve an average f-measure of 95%, suffering only a 1% drop in performance.

Chapter 5

Conclusions and Future Work

In this dissertation, we have designed and implemented three malware detection systems to detect sophisticated malware. These three frameworks aim to address three different malware issues. First, we have introduced DroidClassifier, which utilizes multiple dimensions of mobile traffic information from different families of Android malware to extract features and build a malware classification system. Second, we have developed GranDroid, which relies on partial static and dynamic graph information, analyzes the programming logic of malicious apps, and detect malicious network behaviors. Third, we have implemented Obfusifier, a machine-learning-based malware detector that extracts features from unobfuscated samples but can be used to detect obfuscated malware. Our three frameworks solve the malware detection problems from different perspectives. The evaluation results show that they are more effective when compared to other systems.

We 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 a feasible computational cost. Thus, DroidClassifier can facilitate network management in a vast 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.

Our empirical evaluation of GranDroid has shown that it can be very effective in detecting network-related malware. Our evaluation using 1,500 malware samples and 1,500 benign apps shows that our approach achieves 93% accuracy while spending only eight minutes to execute each app and determine its maliciousness dynamically. GranDroid can be used to provide rich and precise detection results while incurring similar analysis time as a typical malware detector based on pure dynamic analysis.

Our experimental evaluation has shown that Obfusifier can achieve the precision, recall, and F-measure that exceed 95% for detecting obfuscated Android malware, well surpassing any of the previous approaches. The training of our system is based on obfuscation-resistant features extracted from unobfuscated apps, while the classifier retains high effectiveness for detecting obfuscated malware.

These three implemented frameworks have already laid a solid foundation for the detection of complex Android malware. In future work, we need to consider several aspects to further verify our research work and improve the performance of our existing frameworks.

For DroidClassifier, events triggered by Monkey tool are random, and therefore, may not replicate real-world events especially in situations that complex event sequences are needed to trigger malicious behaviors. In such scenarios, malicious network traffic may not be generated. As such, we intend to use more sophisticated event sequence generation approaches to generate more real-world network traffic.

Similarly, GranDroid may perform better if we have a better way to generate input that can provide higher code coverage. Also, our current implementation only supports detection based on network-related APIs to carry out malicious attacks.

However, we intend to extend our approach to cover other classes of relevant and exploitable APIs vulnerable to other types of attacks.

For OBFUSIFIER, we only obfuscate malicious apps using Alan. We intend to experiment with other Android obfuscation tools. In addition, we only apply classical machine learning algorithms to build the system. One disadvantage of the classical approach is the need to extract features manually, and it requires sophisticated feature engineering. To eliminate the need for complex feature engineering, we plan to utilize deep neural networks in our system. Usually, deep neural networks can achieve higher accuracy than classical machine learning methods.

Bibliography

[1] F-Secure, "F-secure mobile threat report," https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q1_2014.pdf, March, 2014.

[2] 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.

[3] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, 2011, p. 2.

[4] Symantec, "Latest intelligence for march 2016," in *Symantec Official Blog*, 2016.

[5] G. Kelly, "Report: 97% of mobile malware is on android. this is the easy way you stay safe," in *Forbes Tech*, 2014.

[6] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[7] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *International conference on Mobile systems, applications, and services*. ACM, 2012, pp. 281–294.

[8] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, *DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications*. Cham: Springer International Publishing, 2014, pp. 163–182.

[9] 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.

[10] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. of ICSE*, Florence, Italy, 2015, pp. 303–313.

[11] L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan, "Sigpid: significant permission identification for android malware detection," in *Proc. of MALWARE*. IEEE, 2016, pp. 1–8.

[12] 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)*, vol. 32, no. 2, p. 5, 2014.

[13] Z. Li, L. Sun, Q. Yan, W. Srisa-an, and Z. Chen, "Droidclassifier: Efficient adaptive mining of application-layer header for classifying android malware," in *Proc. of Securecomm*. Springer, 2016, pp. 597–616.

[14] Y. Yang, Z. Wei, Y. Xu, H. He, and W. Wang, "Droidward: An effective dynamic analysis method for vetting android applications," *Cluster Computing*, Dec 2016. [Online]. Available: https://doi.org/10.1007/s10586-016-0703-5

[15] E. Messmer, "Black Hat demo: Google Bouncer Can Be Beaten," http://www.networkworld.com/news/2012/072312-black-hat-google-bouncer-261048.html.

[16] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: hindering dynamic analysis of android malware," in *Proceedings of the Seventh European Workshop on System Security*. ACM, 2014, p. 5.

[17] O. Storey, "More malware found on google play store," https://www.eset.com/uk/about/newsroom/blog/more-malware-found-on-google-play-store/, June 2017.

[18] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *Proc. of CODASPY*, 2015, pp. 37–48.

[19] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. V igna, "Execute this! analyzing unsafe and malicious dynamic code loading in android applications." in *Proc. of NDSS*, vol. 14, 2014, pp. 23–26.

[20] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *Proc. of NDSS*, 2016.

[21] 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*. ACM, 2013, pp. 611–622.

[22] W. Wang, X. Wang, D. Feng, J. Liu, Z. Han, and X. Zhang, "Exploring permission-induced risk in android applications for malicious application detection," *Information Forensics and Security, IEEE Transactions on*, vol. 9, no. 11, pp. 1869–1882, 2014.

[23] J. Cannell, "Obfuscation: Malware's best friend," https://blog.malwarebytes.com/\threat-analysis/2013/03/obfuscation-malwares-best-friend/, March 2016.

[24] "Virustotal.com," https://www.virustotal.com/#/home/upload, April 2019.

[25] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 2013, pp. 45–54.

[26] V. Avdiienko, K. Kuznetsov, A. Gorla, A. Zeller, S. Arzt, S. Rasthofer, and E. Bodden, "Mining apps for abnormal usage of sensitive data," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, 2015, pp. 426–436.

[27] Z. Li, J. Sun, Q. Yan, W. Srisa-an, and S. Bachala, "Grandroid: Graph-based detection of malicious network behaviors in android applications," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2018, pp. 264–280.

[28] V. Rastogi, Y. Chen, and X. Jiang, "Catch me if you can: Evaluating android anti-malware against transformation attacks," *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 1, pp. 99–108, 2014.

[29] J. Garcia, M. Hammad, and S. Malek, "Obfuscation-resilient, efficient, and accurate detection and family identification of android malware," *ACM Transactions of Software Engineering and Methodology*, vol. 9, no. 4, June 2017.

[30] 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.

[31] G. DATA, "G DATA MOBILE MALWARE REPORT," https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf, July, 2015.

[32] 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.

[33] 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.

[34] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces." in *NSDI*, 2010, pp. 391–404.

[35] 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.

[36] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[37] Tutorialspoint, "HTTP header," http://www.tutorialspoint.com/http/http_header_fields.htm, Accessed on June 21, 2016.

[38] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals." *Forschungsbericht.–707–710 S*, 1966.

[39] L. Rokach and O. Maimon, "Clustering methods," in *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 321–352.

[40] D. Pelleg, A. W. Moore *et al.*, "X-means: Extending k-means with efficient estimation of the number of clusters." in *ICML*, vol. 1, 2000.

[41] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[42] P. Jaccard, *Etude comparative de la distribution florale dans une portion des Alpes et du Jura*. Impr. Corbaz, 1901.

[43] 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*, vol. 11, no. 1, pp. 9–17, 2015.

[44] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," http://ibotpeaches.github.io/Apktool/, Accessed on June 21, 2016.

[45] 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.

[46] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," in *International Symposium on Software Testing and Analysis*, ser. ISSTA 2013, Lugano, Switzerland, 2013, pp. 67–77.

[47] 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.

[48] 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*. ACM, 2013, pp. 141–148.

[49] 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*. ACM, 2012, pp. 343–356.

[50] 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*. ACM, 2012, pp. 65–66.

[51] 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*. ACM, 2015, pp. 15–20.

[52] 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*. ACM, 2011, pp. 639–652.

[53] 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*. IEEE, 2014, pp. 66–71.

[54] N. Kheir, "Analyzing http user agent anomalies for malware detection," in *Data Privacy Management and Autonomous Spontaneous Security*. Springer, 2013, pp. 187–200.

[55] 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*. Springer, 2009, pp. 232–249.

[56] 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*. IEEE, 2015, pp. 630–641.

[57] S. Nari and A. A. Ghorbani, "Automated malware classification based on network behavior," in *Computing, Networking and Communications (ICNC), 2013 International Conference on*. IEEE, 2013, pp. 642–647.

[58] 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*, vol. 38, no. 1, pp. 161–190, 2012.

[59] F. A. Narudin, A. Feizollah, N. B. Anuar, and A. Gani, "Evaluation of machine learning classifiers for mobile malware detection," *Soft Computing*, vol. 20, no. 1, pp. 343–357, 2016.

[60] M. Z. Rafique and J. Caballero, "Firma: Malware clustering and network signature generation with mixed network behaviors," in *Research in Attacks, Intrusions, and Defenses*. Springer, 2013, pp. 144–163.

[61] Y. Tsutano, S. Bachala, W. Srisa-an, G. Rothermel, and J. Dinh, "An efficient, robust, and scalable approach for analyzing interacting android apps," in *Proc. of ICSE*, Buenos Aires, Argentina, May 2017.

[62] "Android feiwo," https://goo.gl/AAY8xp, Accessed at Feb. 2018.

[63] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. of CCS*, 2014, pp. 1105–1116.

[64] "Volley overview," https://developer.android.com/training/volley, December 2017.

[65] "An http client for android and java applications," http://square.github.io/okhttp/, December 2017.

[66] "A powerful image downloading and caching library for android," http://square.github.io/picasso/, December 2017.

[67] "Android asynchronous http client," http://loopj.com/android-async-http/, Accessed at Dec. 2017.

[68] I. Steinwart and A. Christmann, *Support Vector Machines*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[69] N. Landwehr, M. Hall, and E. Frank, "Logistic model trees," vol. 95, no. 1-2, pp. 161–205, 2005.

[70] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[71] "Apkpure.com," https://apkpure.com/, December 2017.

[72] "Virusshare.com," https://virusshare.com/, December 2017.

[73] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[74] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2–16, 2013.

[75] S. Wang, Z. Chen, L. Zhang, Q. Yan, B. Yang, L. Peng, and Z. Jia, "Trafficav: An effective and explainable detection of mobile malware behavior using network traffic," in *Proc. of IWQoS*. IEEE, 2016, pp. 1–6.

[76] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?" in *Proc. of ASE*, Lincoln, NE, 2015, pp. 429–440.

[77] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proc. of NDSS*, 2016.

[78] S. Rasthofer, S. Arzt, and E. Bodden, "A machine-learning approach for classifying and categorizing android sources and sinks." in *Proc. of NDSS*, 2014.

[79] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors

in android applications," Texas A&M, Tech. Rep., 2014. [Online]. Available: http://faculty.cse.tamu.edu/guofei/paper/DroidMiner_TechReport_2014.pdf

[80] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, no. 3, pp. 211–220, 2008.

[81] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, pp. 297–300.

[82] C. S. Collberg, C. D. Thomborson, and D. W. K. Low, "Obfuscation techniques for enhancing software security," Dec. 23 2003, uS Patent 6,668,325.

[83] A. Bacci, A. Bartoli, F. Martinelli, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Impact of code obfuscation on android malware detection based on static and dynamic analysis." in *ICISSP*, 2018, pp. 379–385.

[84] "f-secure.com," https://www.f-secure.com/sw-desc/adware_android_dowgin.shtml.

[85] "Alan-android-malware.com," http://seclist.us/alan-android-malware-evaluating-tools-released.html, April 2019.

[86] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[87] "preemptive.com," https://www.preemptive.com/products/dasho/overview.

[88] "guardsquare.com," https://www.guardsquare.com/en/products/dexguard.

[89] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International*

Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[90] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: a systematic study based on whole-system emulation," in *25th Annual Network and Distributed System Security Symposium, NDSS*, 2018, pp. 18–21.

[91] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[92] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 303–313.

[93] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[94] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.