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

Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an and Shakthi Bachala

Department of Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln NE 68588, USA
{zli,jsun,qyan,witty,sbachala}@cse.unl.edu

Abstract. As Android malware increasingly relies on network interfaces to perform malicious behaviors, detecting such malicious network behaviors becomes a critical challenge. Traditionally, static analysis provides soundness for Android malware detection, but it also leads to high false positives. It is also challenging to guarantee the completion of static analysis within a given time constraint, which is an important requirement for real-world security analysis. Dynamic analysis is often used to precisely detect malware within a specific time budget. However, dynamic analysis is inherently unsound as it only reports analysis results of the executed paths. In this paper, we introduce GranDroid, a graph-based hybrid 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. 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 dynamically execute each app and determine its maliciousness. 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.

1 Introduction

As Android devices become the most popular end-hosts for accessing the Internet, cybercriminals have increasingly exploited Android's network connectivity to glean sensitive information or launch devastating network-level attacks [11, 14, 20]. Significant research efforts have been spent on studying the network usage of Android devices for detecting malicious Android apps using both static and dynamic analyses approaches.

Static analysis approaches [7, 12, 19, 27] perform sound analysis in an offline manner and thus incur no runtime overhead. However, static analysis can result in excessive false positives. Dynamic analysis approaches, on the other hand, are more precise but incur additional runtime overhead [10, 15, 28]. However, recent reports indicate that dynamic analysis can be easily defeated if an app being analyzed can discover that it is being observed (e.g., running in an emulator), and as a result, it behaves as a benign app [16, 18].

Due to the aforementioned limitations, 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. Both static and dynamic information is then used 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) [12, 19, 23] or predefined programming patterns [27]. *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 and data-flow graphs). Yet, the malicious network behaviors are only induced by specific programming logic, i.e., *the network-related paths or events* 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). Consequently, it is challenging for hybrid analysis to be able to complete its analysis within a given time budget (e.g., five minutes). *Adhering to a time budget, however, is an important criterion for real-world malware analysis and vetting systems.*

In this paper, our research goal is to enhance the capability of hybrid analysis and evaluate if the analysis result is sufficiently rich to detect malicious network behaviors in malware running on real devices (to avoid evasion attacks) given a specific time budget. In this work, we introduce GranDroid, a graph-based malicious network behavior detection system. 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 contribution of our paper includes the following:

1. We develop GranDroid based on system-level dynamic graphs to detect malicious network behaviors. Unlike prior work that rely on network traffic information to detect network-related malware, GranDroid utilizes detailed network-related programming logic to automatically and precisely detect the sources of malicious network behaviors.

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

2. GranDroid enables partial static analysis to expand the analysis scope at runtime, and uncover malicious programming logic related to dynamically executed network paths. This can make our analysis approach more sound than a traditional dynamic analysis approach.
3. We perform an in-depth evaluation of GranDroid to evaluate 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.

2 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 is running on an emulator instead of a real device. Figure 1 illustrates a code snippet from Android.Feiwo adware [5], 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:   ...}
```

Fig. 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. As an 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 [29], 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 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 2. Next, we describe each phase in turn.

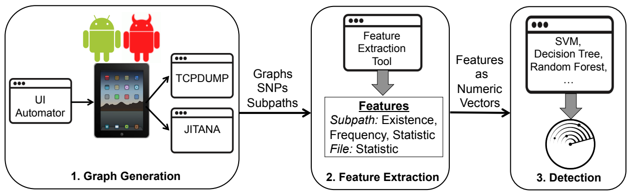

Fig. 2. System Architecture

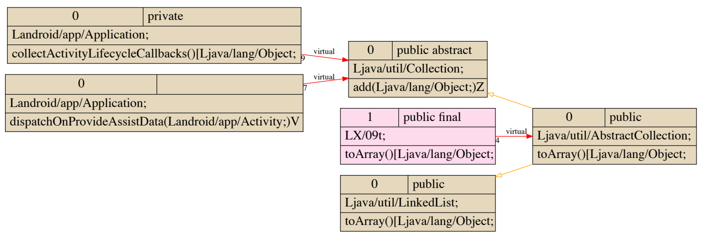
0
private
Landroid/app/Application;
collectActivityLifecycleCallbacks()[Ljava/lang/Object;
9
virtual
0
Landroid/app/Application;
dispatchOnProvideAssistData(Landroid/app/Activity;)V
7
virtual
0
public abstract
Ljava/util/Collection;
add(Ljava/lang/Object;)Z
1
public final
LX/09t;
toArray()[Ljava/lang/Object;
4
virtual
0
public
Ljava/util/AbstractCollection;
toArray()[Ljava/lang/Object;
0
public
Ljava/util/LinkedList;
toArray()[Ljava/lang/Object;

Fig. 3. Method Graph

3.1 Graph Generation

GranDroid detects malicious network behaviors by analyzing program contexts based on system-level graphs. As illustrated in Figure 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* via UIAutomator and can *produce network traffic* (we monitored traffic via TCPDump). Incomplete malware samples and the ones that produce no network traffic are discarded, as GranDroid currently focuses on detecting malicious network behaviors. For future work, we plan to extend GranDroid to cover other types of malware (e.g., those that leak information via intents).

Next, we use Jitana [21], a high-performance hybrid program analysis tool to perform on-the-fly program analysis. While UIAutomator exercises these apps installed on a tablet, Jitana *concurrently* analyzes loaded classes to generate three types of graphs: classloader, class, and method call graphs that our technique utilizes. 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. 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.

Method Graph. Our system detects malicious network behaviors by exploring the invoking relationship of methods in the Method Graph. As shown in Figure 3, blocks represent methods, and edges indicate invoking relationship 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.

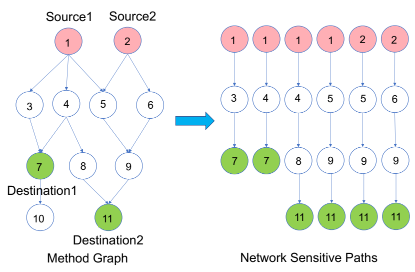

Fig. 4. Path Generation

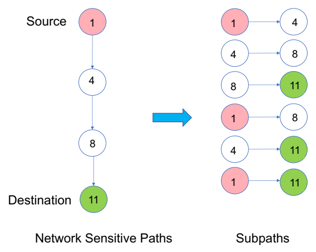

Fig. 5. Subpath Generation

Note that these dynamically generated graphs are determined by the event sequences that exercise each app. As such, they actually reflect the runtime behavior of an app. Another useful information contained in these graphs include the specific Android 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 the 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 is able to 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 [4] and Okhttp [2]. In the Method Graph, we consider all nodes whose in-degree are 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 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. 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. In order 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 5 shows the process of generating subpaths. These subpaths are then converted into numeric vectors in the Feature Extraction phase.

3.2 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 is used to store each feature for 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 enriched 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.

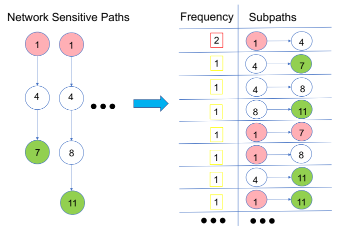

Fig. 6. Subpath Frequency Feature

To do so, we use both SNP and SNS information. As shown in Figure 6, instead of marking 1 or 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.
File Statistic Feature (F4). For each app, we save all of the 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, because the file size accurately reflects the amount of generated information that can provide some insight about 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).

3.3 Detection

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

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

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), Decision Tree and Random Forest.

4 Empirical Evaluation

We present the results of our empirical evaluation of GranDroid. We first explain the process to collect 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.

4.1 Data Collection

Initially, our dataset consists of 20,795 apps from APKPure [1] collected from January 2017 to March 2017. We also downloaded 24,317 malware samples from VirusShare [3]. Note that these samples are newer than those from the Android Genome Project [31], a popular malware repository that was also used by DroidMiner.

To ensure that our experimental environment has not been contaminated after executing a malware sample, we turn off common features that generate network traffic such as auto updates for apps and systems. We also manually checked that there is no background traffic. This is done to ensure that the network traffic seen is generated by our malware sample and not from residual effects from previously exercised malware samples. After running a few samples, we also reflash our devices to ensure that they are free from contaminations.

As previously mentioned, we also run TCPDUMP packet analyzer in each tablet 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. In order to have a balanced dataset, we randomly select 1,500 benign and 1,500 malicious apps to form our dataset.

4.2 Detection Result

For each experiment, we run the 10-fold cross validation on the dataset. We generate different sets of features for these dataset by ways explained in previous sections, and apply three different machine learning methods to build our detection system. In order to compare the performance with other methods, we also implement one popular approach based on our dataset.

Result Based on F1. We first implement our system based on Subpath Existence Feature (F1). Table 1:F1 shows the result of applying SVM, Decision Tree and Random Forest on F1. We compare four metrics for each classification method in Table 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.

Result Based on F2. As explained in Section 3, Subpath Frequency Feature (F2) is based on F1. It builds feature vector based on the frequency of each subpath. Table 1:F2 shows the detection result. For F2, Decision Tree achieves the highest F-measure of 85.1%. It achieves the accuracy of 82.7% with 74.7%

	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 1. The performance of GrandDroid 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).

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 existence of a subpath and its frequency to malware characteristic.

To overcome this shortcoming, we extract statistical information from SNP to construct Path Statistic Feature (F3). As illustrated in Table 1:F3 obviously achieves higher performance than F1 and F2 in terms of all four metrics. This indicates that statistical information related to paths is an important factor that can improve detection performance.

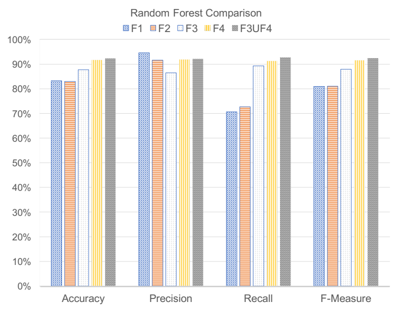

Fig. 7. Performance of Random Forest

Result Based on F4. Besides the statistical feature from paths, we also convert the size all the graph and feature files into numeric vectors. We refer to this feature as File Statistic Feature (F4). Table 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 F-measure of 91.6%.

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 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 7 graphically illustrates the comparison of different feature sets via Random Forest, which also shows that F3 ∪ F4 yields the best F-Measure.

4.3 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 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 1).

4.4 Comparison with Related Approaches

Existing dynamic analysis techniques use network traffic behaviors to detect malware and botnets [15, 30]. 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. If a malicious traffic behavior is detected by executing an app, the app is then classified as malware. Next, we show how GranDroid performs against one of these purely dynamic analysis approaches.

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 2. Utilized HTTP Statistic Features

HTTP Statistic Feature. Prior research efforts have used network traffic information to conduct the malware or botnet detection [30]. 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.

To facilitate a comparison with GranDroid, we re-implement their system. Table 2 lists all the extracted features. Table 3 reports 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%.

	HTTP Statistic Approach		
	SVM (%)	DT (%)	RF (%)
I. Accuracy	57.0	76.0	79.7
II. Precision	53.8	75.3	77.0
III. Recall	99.3	77.3	84.7
IV. F-Measure	69.8	76.3	80.6

Table 3. The performance of HTTP Statistic Approach based on using Support Vector Machine (SVM), Decision Tree (DT) and Random Forest (RF).

In summary, GranDroid outperforms this popular approach in terms of Android malicious network behavior detection. We observe that the overall performance of Random Forest is better than other classifiers. Table 4 summarizes the overall performances of all approaches consisting of DroidMiner (F1), HTTP Statistic Approach and GranDroid. For DroidMiner's results, we use Decision 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%) and that of HTTP Statistic Approach(79.7%). Even though DroidMiner has about 5% FP rate, and GranDroid has about 8% FP rate, GranDroid achieves a much higher F-Measure than those of other approaches.

Method	DroidMiner (F1) (%)	HTTP Statistic Approach (%)	GranDroid (%)
Accuracy	84.3	79.7	93.0
F-Measure	80.9	80.6	93.2

Table 4. Detection Result Comparison

4.5 Average Malware Detection Time

On average, the time to execute an application using UIAutomator was about 5 minutes, 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 components. Prior work by With Bouncer, each app is also executed for 5 minutes to observe if there are any malicious behaviors. The time of 5 minutes is also confirmed by Chen et al. [8] when they reported that most malware would generate malicious traffic in the first 5 minutes. As such, our approach also executes an app for about 5 minutes and 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.

5 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 major 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 [9]. As such, it is possible that our system can perform better if we have a better way to generate input that can provide higher code coverage. In this regard, static analysis would be able to explore more code but it might not be able to adhere to 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 directly execute native code would not be fully analyzed by our approach. However, there are existing work that attempt to perform native code analysis. For example, Afonso et al. [6] perform an analysis of the native code usage in Android apps, and they report that sandboxing native code can be quite useful in practice. Approaches such as this can be incorporated into our work to extend the analysis capability.

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 [24]. 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 [29]. This is because semantic- or action-based approaches focuses 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 cybercriminal to evade our detector, it would require significant more efforts 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 in doing so is to identify relevant APIs that can be exploited to conduct a specific type of attacks. For example, a malicious app that destroys file system would need to use file related APIs. Fortunately, there are some existing approaches that can help identify these relevant APIs [17].

6 Related Work

Network traffic has been used to detect mobile malware [15, 22, 30]. However, these studies have also shown that such systems can be evaded by simply delay the malicious behaviors so that only benign traffic is generated within observation window. Another important observation is that by simply looking at 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 [25].

Past research efforts to address this problem statically analyze various program contexts to help distinguish between benign and malicious apps [7,12,13, 19,25,27]. 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 on these contexts to detect malware, achieving 92.5% precision and 77.3% recall. DroidMiner applies static program analysis to generate two-tiered behavior graph to extract modalities (i.e., known logic segments in the graph that correspond to malicious behaviors) and then aggregates these modalities into vectors that can be used to perform classification. 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 [26].

The work that is most closely related to our work is DroidSIFT [29], which uses API dependency graphs to classify Android malware. The basic idea is to develop program semantics by establishing API dependency graph that is then used to construct a feature set. 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 underlying framework or third party library code, while our analysis can capture these third party and framework codes. 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), but our approach 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.

7 Conclusion

In this work, 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 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.

Acknowledgement

This work was supported in part by US National Science Foundation under grant CNS-1566388.

References

1. Apkpure.com. https://apkpure.com/, December 2017.
2. An http client for android and java applications. http://square.github.io/okhttp/, December 2017.
3. Virusshare.com. https://virusshare.com/, December 2017.
4. Volley overview. https://developer.android.com/training/volley, December 2017.
5. Android feiwo. https://goo.gl/AAY8xp, Accessed at Feb. 2018.
6. V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.
7. D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.
8. 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 *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 206–213. IEEE, 2015.
9. S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? In *Proc. of ASE*, pages 429–440, Lincoln, NE, 2015.
10. 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 TOCS*, 32(2):5, 2014.
11. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX security symposium*, volume 2, page 2, 2011.
12. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of CCS*, pages 627–638. ACM, 2011.
13. M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of MobiSys*, pages 281–294, 2012.

14. G. Kelly. Report: 97% of mobile malware is on android. this is the easy way you stay safe. In *Forbes Tech*, 2014.
15. 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*, pages 597–616. Springer, 2016.
16. E. Messmer. Black Hat demo: Google Bouncer Can Be Beaten. http://www.networkworld.com/news/2012/072312-black-hat-google-bouncer-261048.html.
17. S. Rasthofer, S. Arzt, and E. Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Proc. of NDSS*, 2014.
18. 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.
19. L. Sun, Z. Li, Q. Yan, W. Srisa-an, and Y. Pan. Sigpid: significant permission identification for android malware detection. In *Proc. of MALWARE*, pages 1–8. IEEE, 2016.
20. Symantec. Latest intelligence for march 2016. In *Symantec Official Blog*, 2016.
21. 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.
22. 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*, pages 1–6. IEEE, 2016.
23. 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*, 9(11):1869–1882, 2014.
24. W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proc. of NDSS*, 2016.
25. C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. *DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications*, pages 163–182. Springer International Publishing, Cham, 2014.
26. C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. Technical report, Texas A&M, 2014.
27. 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*, pages 303–313, Florence, Italy, 2015.
28. 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.
29. M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proc. of CCS*, pages 1105–1116, 2014.
30. 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*, 39:2–16, 2013.
31. Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of IEEE S&P*, pages 95–109, 2012.