Cypider: Building Community-Based Cyber-Defense Infrastructure for Android Malware Detection

ElMouatez Billah Karbab
Concordia University,
e_karbab@encs.concordia.ca

Mourad Debbabi
Concordia University,
debbabi@encs.concordia.ca

Abdelouahid Derhab
King Saud University
abderhab@ksu.edu.sa

Djedjiga Mouheb
Concordia University
d_mouheb@encs.concordia.ca

ABSTRACT

The popularity of Android OS has dramatically increased malware apps targeting this mobile OS. The daily amount of malware has overwhelmed the detection process. This fact has motivated the need for developing malware detection and family attribution solutions with the least manual intervention. In response, we propose Cypider framework, a set of techniques and tools aiming to perform a systematic detection of mobile malware by building an efficient and scalable similarity network infrastructure of malicious apps. Our detection method is based on a novel concept, namely *malicious community*, in which we consider, for a given family, the instances that share common features. Under this concept, we assume that multiple similar Android apps with different authors are most likely to be malicious. Cypider leverages this assumption for the detection of variants of known malware families and zero-day malware. It is important to mention that Cypider does not rely on signature-based or learning-based patterns. Alternatively, it applies *community detection algorithms* on the similarity network, which extracts sub-graphs considered as suspicious and most likely malicious communities. Furthermore, we propose a novel fingerprinting technique, namely *community fingerprint*, based on a learning model for each malicious community. Cypider shows excellent results by detecting about 50% of the malware dataset in one detection iteration. Besides, the preliminary results of the *community fingerprint* are promising as we achieved 87% of the detection.

Keywords

Android; Malware; Community Detection; Fingerprinting

1. INTRODUCTION

The ubiquitousness of mobile devices and their applications have considerably contributed to the evolvement of this technology and its expansion in both the economy and society. Indeed, mobile applications (apps) have become an essential part of our daily tasks. However, being able to access both sensitive and personal data on the network has given rise to threats targeting apps security. Android [4] is the most popular mobile OS that controls 82% of the market share [14] in the mobile world, including smart devices such as phones and tablets. Also, Android system expanded to various daily tools, such as TVs [7], watches [8], and cars [5], to connect them to the internet. Moreover, Android OS is increasingly involved in the so-called *Internet of Things* (IoT), especially with the emergence of Google's Brillo [11], an Android-based embedded OS that provides the core services for IoT developers. Brillo leverages Android's advanced functionalities to extend horizons for IoT devices. However, this comes at the expense of security since many malware instances are targeting Android devices, and thus could infect IoT applications based on Brillo.

Modern Android OS provides mechanisms and techniques, such as *sandboxing*, to empower the security of smart devices. However, due to the increasing number of attacks targeting mobile devices, this defense mechanism alone is not sufficient to mitigate such attacks. Besides, the popularity of Android OS made it a tempting target for malware. In fact, an enormous amount of malware launch attacks against users' devices on a daily basis. For example, according to G DATA [12], $1,548,129$ and $2,333,777$ new Android malware were discovered in 2014 and 2015, which represents approximately an average of $4,250$ and $6,400$ new malware per day respectively. Furthermore, about 53% of malware were SMS Trojans designed to steal money and personal information from Android-based mobile devices [25]. The malware could be classified into two broad categories: i) *malware variant*, which is a known malware with a new skin created by applying repackaging techniques, and ii) *unseen malware* or *zero-day malware*, which is a malicious app not discovered by security researchers and vendors. The vast number of existing Android apps along with the daily-created malicious ones make the manual investigation of new apps (benign and malicious) a difficult, if not an impossible task for security analysts. Traditional approaches depend on identifying specific *signature-based patterns* that belong to previously known malware families. However, a practical, yet an effective approach should not rely on signatures such as cryptographic hashes since this could easily be defeated using the simplest modification in the original malware app. Furthermore, the use of signatures limits the ability to detect new malware families, which raises concerns about the possibility

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2991079.2991124

of systematically identifying new malware apps (new families) without or with minimum human intervention. Other approaches apply *heuristic-based* or *machine-learning* methods on benign and malicious samples to generate *learning-based patterns*, which are used to identify known and new malware. Although *learning-based* approaches are more efficient than *signature-based* ones for detecting zero-day malware, their precisions mainly depend on the training set and the used features to generate the pattern.

In this research, we combat large-scale Android malware by decreasing the *analysis window* size of newly detected malware. The window starts from the first detection until the signatures generation by security vendors. The larger the window is, the more time the malicious apps are given to spread over the users' devices. Current state-of-the-art techniques have a large window due to the huge number of Android malware appearing on a daily basis. Besides, these techniques use manual analysis in some cases to investigate malware. Therefore, decreasing the need for manual detection could significantly reduce the *analysis window*. To address the previous issue, we elaborate systematic tools, methods, and approaches for the detection of both known and new malware apps (i.e., variants of existing families or unseen malware). To do so, we rely on the assumption that a couple of Android apps, with distinct authors and certificates, are most likely to be malicious if they are highly similar. This is due to the fact that the adversary usually repackages multiple app packages with the same malicious payload to hide it from the anti-malware and vetting systems. Consequently, it is difficult to detect such malicious payloads from the benign functionalities of a given Android package. Accordingly, a pair of Android apps should not be very similar in their components excluding popular libraries. This observation, as mentioned earlier, could be used to design and develop a security framework to detect Android malware apps.

In this paper, we leverage the previously-mentioned assumption to propose a cyber security framework, namely Cypider (*Cyber-Spider For Android Malware Detection*), to detect and cluster Android malware without the least prior knowledge of Android malware apps such as *signature-based* or *learning-based* patterns. Cypider consists of a novel combination of a set of techniques and methods to address the problem of Android malware clustering and fingerprinting. First, Cypider can detect repackaged malware (malware families), which constitute the vast majority of Android malware apps [57]. Second, it can detect new malware apps, and more importantly, Cypider performs the detection automatically and in an unsupervised way. The fundamental idea of Cypider relies on building a *similarity network* between the entered apps. Eventually, Cypider extracts, from this *similarity network*, sub-graphs with high connectivity, called *communities*, which are most likely to be *malicious communities*. The next step in Cypider process is the generation of a fingerprint for each detected community. For this purpose, we propose a novel technique called *community fingerprinting*. Instead of using a static signature based on cryptographic or fuzzy hashing of one app, we use *One-Class Support Vector Machine* learning model (OC-SVM) [43] to compute the *community fingerprint* of the whole Android malware family or sub-family. The OC SVM model is a machine learning technique used to learn the features of only *one class* (features of one community in our case). The resulting model, i.e., the community fingerprint, is used to detect whether a given new Android app is part of the community or not. After identifying the malware apps of the detected communities, Cypider framework continues as a periodic process, where it carries out on the remaining apps along with new ones, which form what we call the *active dataset*. We address the scalability issue in Cypider using three principal techniques: i) The extracted statistical features from the APK have a high dimensionality causing a steep decrease in the performance of the proposed solution. To address this issue, we use the *feature hashing* [45] technique to reduce this dimensionality to a fixed length while keeping the same detection rate. ii) We leverage the state-of-the-art of machine learning techniques using *Locality Sensitive Hashing* (LSH) [21] to compute the similarity between vectors of app features. The latter drastically speeds up the comparison operation compared to the brute force approaches. iii) We use a scalable *community detection* algorithm, proposed in [23], to extract the *malicious communities*. To sum up, Cypider framework is a set of algorithms, techniques, and mechanisms, which have been fashioned into one approach. The latter aims to detect Android malicious apps without pre-knowledge of the actual malware families. Furthermore, Cypider could produce unsupervised fingerprints of possible threats by leveraging the proposed community fingerprint. Therefore, our contribution is mainly the whole framework, and not only a set of components

The main contributions of this paper are:

1. We design and implement Cypider, a simple yet effective framework for malware detection based on the *community concept* and *graph analysis techniques*.

2. We propose a *community fingerprint*, a novel fingerprint based on a classification model to represent the features of a given community, which could be a malware family or a subfamily.

3. We evaluate Cypider framework on: i) Genome malware dataset [2, 57], ii) Drebin malware dataset [1] [20, 46], and iii) the previous datasets with benign apps downloaded from Google Play[13]. The evaluation shows a promising results.

The remainder of this paper is organized as follows: Section 2 presents our threat model with the usage scenarios of Cypider. Section 3 details our methodology. Afterward, we present the used *statistical features* and their processing in Section 4. The different components of Cypider are described in Sections 5, 6, and 7. The evaluation of the proposed approach is covered in Section 8. General notes about Cypider are presented in Section 9. Section 10 discusses Cypider's limitations and future work. In Section 11, we discuss related work. Finally, Section 12 concludes the paper.

2. THREAT MODEL AND ASSUMPTIONS

In the context of Cypider, the focus is on the detection of malware targeting Android mobile apps without the need of having any prior knowledge about these malware. In particular, instead of focusing on the detection of an individual instance of the malware, Cypider targets bulk detection of malware families and variants as malicious communities in

the similarity network of the apps dataset. Moreover, Cypider aims for a scalable yet accurate solution that can handle the overwhelming volume of the daily detected malware, which could aggressively exploit users' smart devices. More specifically, Cypider targets the detection of unobfuscated APK contents. However, Cypider could handle some types of obfuscations because it considers different static contents of the Android package in the analysis. The latter make Cypider more resilient to obfuscation as it could fingerprint malware apps with other static contents that are not obfuscated such as app permissions. Notice that some static contents could be obfuscated in one app and not in another, which depend on the authors of the apps. The evaluation of Cypider uses real malware dataset along with random Android apps from Google Play - where the apps are supposedly obfuscated through ProGuard - to prove the efficiency and the effectiveness of Cypider in real word scenario, in which the obfuscation is a part. Similarly, Cypider could not detect transform attacks malware, whose malicious payload is not in the actual APK static content and the payload is downloaded at runtime. Furthermore, Cypider aims to detect homogenous and pure malicious communities with only one malware family in each community, and hence facilitating malware analysis. In addition, Cypider aims for a high accuracy while maintaining a low false positive rate.

2.1 Usage Scenarios

Cypider proposes a generic approach to investigate apps similarity, which could have many usage scenarios. For example, software binaries comparison, where the input is a set of binaries, and the output would be the binary communities that share similar features. A possible application here is the authorship attribution where the community represents a set of software binaries owned by the same individual. Another usage scenario of Cypider approach could be malware detection and family attribution, where we try to boost the overall malware investigation in general, in which we look for communities of similar malicious binaries to infer their malware family after the detection. In this paper, we target the detection of Android malicious apps.

In the context of this research, Cypider has two main usage scenarios. In the first scenario, Cypider can be applied only on malicious Android apps. The aim is to speed up the analysis process and attribute malware to their corresponding families. Under the first scenario, the overall malware analysis process is boosted by automatically identifying malware families and minimizing the overall manual analysis. The outcome of the previous process is the communities of malicious apps. The attribution of a family to a given community can be achieved by attributing a small set (one app in most cases) among its malicious apps. On the other hand, unassigned communities are considered as suspicious apps that require manual investigation. However, the analysis can be done on only some samples of the suspicious community considering they are highly similar. We could divide the malware examination approaches into:

1. *Semi-supervised Approach:* This means that we have a reference dataset of known malicious apps with complete information about their malware families. The input to Cypider is the unknown malicious apps for (semi-) automatic investigation. Cypider will leverage the known malicious apps in addition to the new unknown ones to produce malicious communities. To this end, known malicious apps of the same malware family tend to have the same community (a strongly connected graph). Moreover, the unknown malicious apps will most likely join a given community of known malicious apps if they share the same family. Analogically, the communities of the known malicious apps play the role of a magnet for unknown ones, if they share the same features, implying having the malware family.

2. *Unsupervised Approach:* Unsupervised means we do not have any information about the suspicious apps entered to Cypider. The aim is to investigate communities (or clusters depending on the terminology; we prefer community because it expresses better the strongly connected graph notion used by Cypider to find malicious apps) instead of a single malicious app. Considering communities could drastically increase the productivity of the security investigator. In this paper, we focus on the unsupervised approach.

In the second scenario, Cypider is applied on mixed Android apps (i.e., malicious or benign). Such dataset could be the result of a preliminary suspiciousness app filtering. Therefore, a lot of *false positives* can be recorded; we assume that benign apps - meaning *false positives* - constitute $50\% - 75\%$ of the actual suspicious apps. Based on the previous assumption, we could identify malicious Android apps by detecting and extracting app communities that share a common payload. We could understand that apps with high similarity are most likely to be malicious. Moreover, Cypider could help filtering less suspicious apps (most likely to be benign apps) that do not have any similarity with the suspicious apps of the current dataset. Notice that the previous approaches, i.e., *Semi-supervised* and *Unsupervised* approaches, could be applied on the mixed dataset to target malware apps. It is important to note that the statistical features could differ from a use-case to another. However, the overall approach of Cypider is the same in all the previous applications. In our context, we focus on Android malicious app detection and family attribution. More specifically, we target to detect malicious apps without a prior knowledge using the unsupervised approach.

3. METHODOLOGY

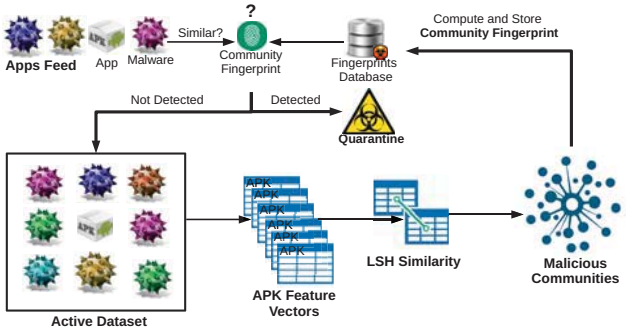

Figure 1: Cypider Overview

Cypider framework uses a dataset of unlabeled apps (malicious or mixed) in order to produce *community fingerprints* for the identified *app communities*. Cypider overall process is achieved by performing the following steps, as illustrated in Figure 1:

1. At the beginning of Cypider process, we need to filter out the apps developed by the same author; we call them *sibling apps*. We aim here to remove the noise of having app communities of sibling apps because they tend to have many similar features since authors reuse components across different apps. Cypider identifies sibling apps in the dataset based on their version, app hash, and author signature (provided in the META-INF directory in the APK file). Therefore, we only keep apps with different author identities since the adversary favors to use multiple fake author identities to prevent the removal of all apps in case of detected maliciousness in one of them. Regarding multiple apps with the same author, Cypider randomly selects one app. Afterward, if the selected app is recognized as malicious in the analysis results, Cypider will tag all its sibling apps as malicious.

2. After filtering the sibling apps, we need to derive from the actual apps package a meaningful information that could identify the app and help computing the similarity against other apps. For this purpose, Cypider extracts statistical features from the apps, which could be either benign or malicious depending on the usage scenario. The feature engineering is the most critical part of the whole framework in the context of Android malware detection usage scenario (Other usage scenarios could have different statistical features, but the overall approach is the same). It is important to mention that the selected features must be resilient to the attacker's deceiving techniques. To this end, the features need to be broad enough to cover most of the static characteristics of a given Android APK. The broader the features are, the more resilient they are. For our purposes, we leverage static analysis features of the APK in the design of Cypider. In particular, we extract them from each content category (classes.dex, resources, assembly, etc.), as described in Section 4.

3. Relying on the extracted features from each content, Cypider computes a fixed length *feature vector* for each content features. In order to reduce and normalize the size of the feature vectors, we enable Cypider with a machine learning preprocessing technique, namely *feature hashing* [45] (or *hashing trick*), as presented in Section 4.7. As a result, Cypider produces, from the extracted features of the previous stage, multiple feature vectors with a small and fixed size. The number of the generated feature vectors depends on how many APK contents are used in the feature extraction, i.e., each content type corresponds to one feature vector.

4. For efficient comparison between the apps, we empower Cypider system with a highly scalable similarity computation system based on locality-sensitive hashing (LSH) [21] technique, which computes the similarities between apps, as presented in Section 5. Given a pair of apps, we compute the similarity between each content *feature vector* from the previous stage to decide if they are connected or not with respect to that content. The result of this step is an undirected network (or similarity network), where the nodes are Android apps and the edges represent the high similarity with respect to one content between the apps. For similar apps, multiple connecting edges are expected. Besides, the more the edges are, the higher the apps are suspected to be malicious.

5. Cypider leverages the *similarity network* in order to detect malicious apps communities. This step depends, however, on the usage scenario (Section 2.1). For the malicious apps, Cypider extracts highly connected apps communities and then excludes these apps from the dataset. The remaining apps (i.e., apps that are not part of any community) are considered in another Cypider malware detection iteration. We expect to get a pure or near-pure community if the containing apps of a given community have the same or almost the same Android malware family respectively. In the case of a mixed dataset, Cypider first excludes all the app nodes with degree 1 (i.e., the app is only self-similar), which are most likely to be benign apps. Afterward, Cypider extracts the apps of malicious communities.

   The rest of apps will be considered in another Cypider iteration. At this point, we expect to have some benign communities as *false positives*. However, the similarity network made Cypider's decision explainable because the security practitioner can track with respect to which content these apps are similar. The previous option could also help in sharpening the statistical features to prevent benign apps to be detected in malicious communities. For community detection (Section 6), we adopt a highly scalable algorithm [23] to enhance Cypider's community detection module.

6. To this end, we consider a set of malicious communities, each of which is most likely to be a malware family or a subfamily. Cypider leverages these malicious communities to generate the so-called *community fingerprint* (Section 7) that captures the app features of a given detected community. Instead of using traditional crypto or fuzzy hashing of only one malware instance, we leverage a model produced by a one-class classifier [43], which can provide a better-compressed format of a given Android malware family. This model is used to decide whether new malware apps are part of this family or not. The results are multiple *community fingerprints*, each of which corresponds to a detected community. The generated fingerprints are stored in the *signature database* for later use.

7. At this stage, Cypider is ready to start another detection iteration with a new dataset, including the rest of unassigned apps from the previous iteration. The same previous steps will be followed for the new iteration. However, we first check the *feature vectors* of the new apps against the known *malware communities fingerprint* stored in the database. The matched apps to a community fingerprint are labeled as malicious without adding them to the *active dataset*. Undetected apps are added to the *active dataset* and are considered in the next iteration of the detection process.

We consider Cypider approach as an endless process, in which we detect and extract communities from the *active dataset* that always gets new apps (malware only or mixed) on a daily basis, in addition to the rest of apps from the previous iterations.

4. STATISTICAL FEATURES

In this section, we present the statistical features of Android packaging (*APK*). We only extract, in this paper, static features from each app *APK* file, in order to generate the feature vectors than compute the similarity with other apps feature vectors. In other words, the feature vector set will be the input of the LSH similarity computation module used to build the similarity network. As previously mentioned, the features should be broad enough to cover most of the static content of the APK file. The features could be categorized, based on the main APK content types, to i) Binary features, which are related to byte-code (Dex file) of the Dalvik virtual machine, where we consider the hex dump of the Dex file along with the actual file. ii) Assembly features, which are computed from the assembly of *classes.dex*. iii) Manifest features, extracted from the Manifest [10] file, which is vital in Android apps since it provides essential information about the app to the Android OS. iv) *APK features*, which include all the remaining APK file content, such as *resources* and *assets*. In this section, we present the statistical features based on the adopted concept to extract them (e.g., N-gram). For clarity, we first start by recalling the internal structure of Android packaging.

4.1 Android APK Format

Android Application Package (*APK*) is the official Android packaging format, which is used for apps distribution and installation. By analogy, *APK* files are similar to *EXE* installation files in Windows or *RPM/DEB* files in Linux. More precisely, *APK* is a *ZIP* archive file, which contains the different components to run the app. The *APK* file content is organized into directories (namely **lib**, **res**, **assets**) and files (namely **AndroidManifest.xml** and **classes.dex**). The purpose of each item is as follows: i) **AndroidManifest.xml** contains the app meta-data, e.g., name, version, required permissions, and used libraries. ii) The **classes.dex** contains the compiled classes of the Java code. iii) The **lib** directory stores C/C++ native libraries [6]. iv) Finally, the resources directory contains the non-source code files that are packaged into the *APK* file during compilation. It mostly contains media files such as video, image, and audio files.

4.2 N-grams

The *N-gram* technique is used to compute contiguous sequences of N items from a large sequence. For our purpose, we use N-gram to extract the sequences employed by Android malware to be able to distinguish between different malware samples. The N-gram on various Android app package contents, such as *classes.dex*, reflect the APK patterns and implicitly capture the underlying Android package semantics. We compute multiple feature vectors for each APK content. Each vector $V \in D$ ($|D| = \Phi^N$ where Φ represents all the possibilities of a given APK content). Each element in the vector V contains the number of occurrences of a particular APK content N-gram.

4.2.1 The classes.dex Bytes N-grams

To increase the extracted information, we leverage two types of N-gram, namely *opcodes N-grams* and *bytes N-grams*, which are extracted from the binary *classes.dex* file and its assembly respectively. From the *hexdump* of the *classes.dex* file, we compute *Byted N-grams* by sliding a window of the hex string, where one case in that string is a byte, as depicted in Figure 2.

4.2.2 Assembly opcodes N-grams

The opcodes N-gram are the unique sequences in the disassembly of *classes.dex* file, where the instructions are stripped from their operands. An example of this N-gram is shown in Figure 2. We choose opcodes instead of the full instruction for multiple reasons: i) Using opcodes tends to be more resilient to simple obfuscations that modifying some operands such as hard-coded IPs or URLs. ii) Opcodes could be more robust to modifications, caused by repackaging, that modify or rename some operands. iii) In addition to being resilient to changes, opcodes can be efficiently extracted from Android apps.

The gained information from the opcode N-gram could be increased by considering only functions that use a sensitive API such as SMS API. Also, excluding the most common opcode sequence decreases the noise for the N-gram information. Also, the number of grams has a significant influence on the gathered semantics. The result of N-gram extraction is the list of unique N-grams with the occurrence number for each content category, i.e., *opcode instructions*, *classes.dex*. Figure 2 illustrates different N-grams on both instructions and bytes of the first portion of the **AnserverBot** malware. In addition to the opcodes, we also consider the *class names* and the *methods' names* as assembly features.

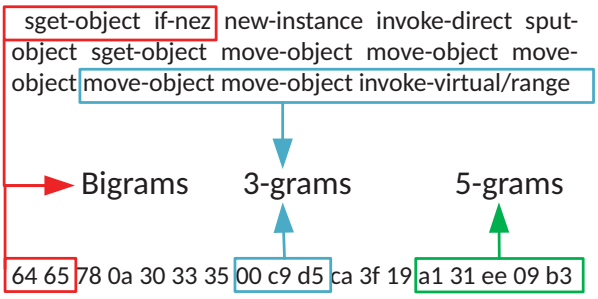

Figure 2: Opcodes & Bytes From AnserverBot

4.2.3 Native Library N-grams

The *Native Library* is part of the binary sub-fingerprint, which captures C/C++ shared libraries [6] used by malware. Using the native library for malware fingerprinting is essential in some cases to distinguish between two Android malware samples. For instance, if the malware uses a native library, it is more likely to be **DroidKungFu2** than **DroidKungFu1** because **DroidKungFu2** malware family uses C/C++ library and **DroidKungFu1** uses only *Java byte code*.

4.2.4 APK N-grams

The N-gram of the APK file can give an overview on the APK file semantics. For instance, most of the repackaged apps are built from an original app with minor modifications [30]. Consequently, applying N-gram analysis on the APK file can detect high similarity between the repackaged app and the original one. Besides, some components in the APK file, e.g., images and GUI layout structures, are preserved by the adversaries, especially if the purpose of the repackaging

process is to develop a phishing malware. Both apps, in this case, are visually similar, and hence the N-gram sequences computed from both apps will be similar in the zone related to the resource directory.

4.3 Manifest File Features

In our context, *AndroidManifest.xml* is a source of essential information that could help in identifying malicious apps. The *permissions* required by apps are the most important features. For example, apps that require *SMS send* permission are more suspicious than other apps since a big portion of Android malware apps targets sending SMS to premium phone numbers. In addition, we extract other features from *AndroidManifest.xml*, namely, *activities*, *services*, and *receivers*.

4.4 Android API Calls

The required permissions provide a global view of a possible app behavior. However, we could get a more granular view by tracking the *Android API calls*, knowing that one permission could allow access to multiple *API calls*. Therefore, we consider the *API* list used by the apps as the feature list. Furthermore, we use a filter list of *API* of the *suspicious APIs*, such as *sendTextMessage()* and *orphan APIs*, which is part of an undeclared permission. On the other hand, we extract the list of permissions, where none of their *APIs* has been used in the app.

4.5 Resources

In this category, we extract features related to the *APK* resources such as text string, file names, and their content. An important criterion when filtering the files is to not include the names of standard files, e.g., *String.xml*. Also, we include files' contents by computing **md4** hashes on each resource file. At first glance, it seems that the use of MD4 is not convenient compared to more modern cryptographic hashing algorithms such as MD5 and SHA1. However, we choose MD4 purposely because it is cheap in terms of computation. Therefore, we enhance the scalability of the system, yet, we achieve the goal of the file comparison between the malicious apps of the active dataset. Finally, we make a text string selection in the text resources, where we leverage *tf-idf* (term frequency-inverse document frequency) [15] technique for this purpose.

4.6 APK Content Types

Table 1 summarizes the proposed feature categories based on APK contents. It also depicts the features considered in the current implementation of Cypider.

4.7 Feature Preprocessing

Feature extraction and similarity computation are the most atomic operations in the proposed framework. Therefore, we need to optimize both their design and implementation to get the intended scalability. The expected output from the *feature processing* operation is a vector, which can straightforwardly be used to compute the similarity between apps. Apps feature vectors are the input to the Cypider community detection system.

N-gram technique, presented in Section 4.2, suffers from its very high dimensionality D. The dimension number D dramatically influences the computation and the memory needed by Cypider for Android malware detection. The complexity of computing the extracted N-grams features increases exponentially with N. For example, for the *opcodes N-grams*, described in Section 4.2, the dimension D equals to R^2 for bi-grams, where $R = 200$, the number of possible opcodes in Dalvik VM. Similarly, for *3-grams*, the dimension $D = R^3$; for *4-grams*, $D = R^4$. Furthermore, N has to be at least 3 or 5 to capture the semantics of some Android APK content.

#	Content Type Features	Implemented Feature
0	APK Byte N-grams	X
1	Classes.dex Byte N-grams	X
2	Native Library Bytes N-grams	
3	Assembly Opcodes Ngrams	X
4	Assembly Class Names	X
5	Assembly Method Names	X
6	Android API	X
7	Orphan Android API	
8	Manifest Permissions	X
9	Manifest Activities	
10	Manifest Services	
11	Manifest Receivers	
12	IPs & URLs	X
13	APK Files names	X
14	APK File light hashes (md4)	
15	Text Strings	X

Table 1: Content feature categories

Algorithm 1: Feature Vector Computation

```
input  : Content Features: Set,
         L: Feature Vector Length
output: Feature Vector
vector = new vector[L];
for Item in Content Features do
    H = hash(Item) ;
    feature_index = H mod L ;
    vector[feature_index] = vector[feature_index] + 1 ;
end
```

To address this issue, we leverage the *hashing trick* technique [45] to reduce the high dimensionality of an arbitrary vector to a fixed-size feature vector. More formally, *hashing trick* reduces a vector V with $D = R^N$ to a compressed version with $D = R^M$, where $M << N$. The compacted vector boosts Cypider, both computation-wise and memory-wise, by allowing the clustering system to handle a large volume of Android apps. A previous research [51, 45] has shown that the hash kernel approximately preserves the vector distance. Moreover, the computational cost incurred by using the hashing technique for reducing dimensionality grows **logarithmically** with the number of samples and groups. Algorithm 1 illustrates the overall process of computing the compacted feature vector from a N-grams set. Furthermore, it helps to control the length of the compressed vector in an associated feature space.

5. LSH SIMILARITY COMPUTATION

Building the *similarity network* is the backbone of Cypider framework. We generated the *similarity network* by computing the pair-wise similarity between each feature vector of the apps APKs. As a result, we obtain multiple similarities according to the number of these content vectors. Using various similarities gives flexibility and modularity to Cypider. In other words, we could add any new feature vector to

the similarity network without disturbing the Cypider process. Also, we could remove features smoothly, which makes the experimentation of selecting the best features more convenient. *More importantly, having multiple similarities between apps static contents in the similarity network leads to explainable decisions, where the investigator can track based on which contents a pair of apps are similar in the final similarity network.* Similarity computation needs to be conducted in an efficient way that is much faster than the brute-force computation. For this purpose, we leverage *Locality Sensitive Hashing* (LSH) techniques, and more precisely *LSH Forest* [21], a tunable high performance algorithm for the similarity computation of Cypider framework. The key idea behind *LSH Forest* is that similar items hashed using LSH are most likely to be in the same bucket (collide) and dissimilar items in different ones. Many similarity measures correspond to *LSH* function with this property. In our case, we use the well-known *Euclidean* distance for this purpose.

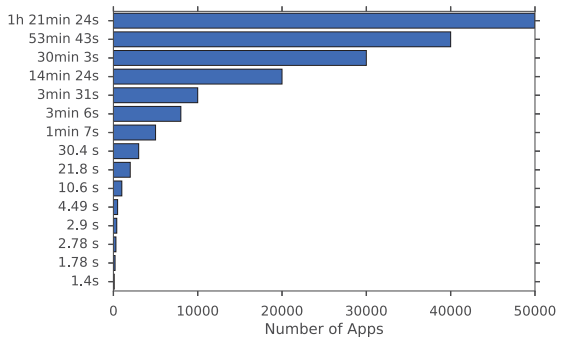

Figure 3: LSH Similarity Computational Time

$$d(m,n) = \|V_m - V_n\| = \sqrt{\sum_{|i=1|}^{|S|} (V_m(i) - V_n(i))^2} \quad (1)$$

Given a pair of Android apps, after extracting one content feature vector, we use the *Euclidean* distance to compute the distance between two feature vectors m and n of one APK content, as depicted in Formula 1. Figure 3 shows the LSH computational time with respect to the number of apps using *one CPU core* and *one thread* for the permission feature vector. Even though the current performance using *LSH Forest* is acceptable for a large number of daily malware samples (reaching $40,000$ apps per hour), we believe that we could drastically improve these results by just leveraging an implementation that exploits all the CPU cores in addition to multi-threading. The final result of the similarity computation is a *heterogeneous network*, where the nodes are the apps and the edges represent the similarity between apps if it exceeds a certain threshold. The latter is manually fixed based on our evaluation, and the same threshold is used in all experiments. Note that the network is heterogeneous because there are multiple types of edges, where the edge type is a static content type.

6. COMMUNITY DETECTION

A scalable community detection algorithm is essential to extract *suspicious communities*. For this reason, we empower Cypider with the *Fast unfolding community detection* algorithm [23], which can scale to billions of network links. The algorithm achieves excellent results by measuring the *modularity* of communities. The latter is a scalar value $M \in [-1, 1]$ that measures the density of edges inside a given community compared to the edges between communities. The algorithm uses an approximation of the *modularity* since finding the exact value is computationally hard [23]. The previous algorithm requires a homogeneous network as input to work properly. For this reason, we propose using a *majority-voting* mechanism to homogenize the heterogeneous network generated by the similarity computation. Given the number of content similarity links s, the *majority-voting* method decides whether a pair of apps are similar or not by computing the ratio s/S, where S is the number of all contents used in the current Cypider configuration. If the ratio is above the average, the apps will only have one link in the *similarity network*. Otherwise, all the links will be removed. Notice that content similarity links could be kept for later use, for example, to conduct a thorough investigation about given apps to figure out how similar they are, and in which content they are similar. The prior use case could be of great importance for security analysts.

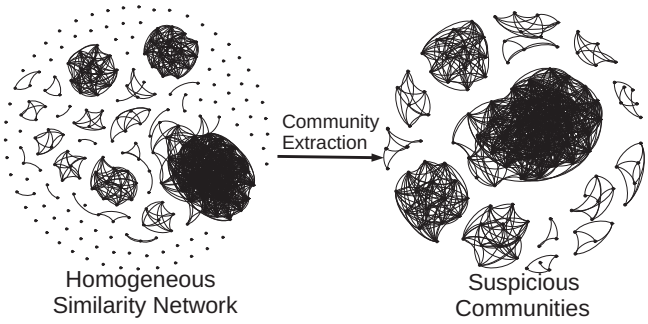

Figure 4: Applying Cypider On a Small Dataset

We propose the *majority-voting* mechanism to filter the links between the nodes (apps) to prevent having inaccurate *suspicious communities*. Furthermore, we use a *degree filtering* parameter to filter all node links with a degree that is less than this parameter. The previous hyperparameter keeps only edges of a given node when their number is above the threshold. Consequently, only nodes with a high connectivity will maintain their edges, which are supposedly like similar malicious apps. Notice that all the parameters have been fixed in our evaluations. In the case of a *mixed dataset* scenario, we use the degree 1 to filter all apps having a similarity link to themselves since they are not similar to any other app in the *active dataset*. Cypider filters these apps and consider them as benign apps. At this point, Cypider applies the community detection algorithm [23] to extract a set of communities with different sizes. Afterward, we filter all communities with a community cardinal that is less than the *minimum community size* parameter (fixed for all the evaluations). The purpose of filtration is to prevent the extraction of bad quality communities. Figure 1 depicts an example of using Cypider on a small Android dataset (250 malware apps), where the process of the community detec-

tion starts with the *homogeneous network* and ends up with *suspicious communities*.

7. COMMUNITY FINGERPRINT

Detecting malicious communities is not the only goal of Cypider framework. Since Cypider tends to be completely unsupervised, we aim to generate fingerprints from the extracted communities automatically. Therefore, in the following iteration, Cypider filters the known apps without adding them to the *active dataset*. Traditional *cryptography fingerprints* or *fuzzy hashing* techniques are not suitable for our case since we aim to generate a fuzzy fingerprint, not only for one app but also for the whole malicious community whether it is a malware family or subfamily. In this context, we present a novel fingerprinting technique using *One-Class Support Vector Machine* learning model (OC-SVM) [43]. The latter could be fuzzy enough to cover the community malicious apps. The *One-Class SVM* model is employed for a novelty detection of a set of malicious apps of a given community. In particular, it detects the *soft boundary model* of that set. Adopting the *one-class model*, Cypider classifies new apps as belonging to this community or not. The proposed fingerprinting technique produces a much more compressed signature database compared to the traditional methods, where the signature is only for one malicious app. Moreover, it highly reduces the computation since we check only with the community fingerprints instead of checking with each single malware hash signature. In order to generate a *community fingerprint* from a set of malicious apps, Cypider first extracts the features, as presented in Section 4. Afterward, we train the *one-class model* using the statistical features of the malicious community apps. However, we need to apply *feature selection* techniques to improve the accuracy of the community fingerprint (model), as described in the following.

7.1 Feature Selection

To improve efficiency given a large number of features, we only keep the best N features with the highest information gains and remove irrelevant ones that potentially cause less accuracy along with the overhead of the unnecessary computational complexity. In order to filter the features, we apply two metrics in the selection of Android APK valuable features, namely *variance threshold* and *inverse document frequency (IDF)*. First, we compute the *variance* of the Android apps features. Afterward, we use a fixed threshold T to filter APK with a *variance* that is lower than T. For example, filtering with $T = 0$ results in removing all the features with the same value for all the entered Android apps. It makes sense because the information gain from a fixed number for all the community apps is zero.

$$idf(t) = log\frac{|F|}{1 + |\{f_{apk} : t \in f_{apk}\}|} \quad (2)$$

Second, we compute the *inverse document frequency (IDF)* on the list $F = \{f_{apk1}, ..., f_{apkN}\}$, where f_{apk} is the set of Android APK features and $|F|$ is the number of inputted APKs. The *idf(t)* is computed using Formula 2, $|\{f_{apk} : t \in f_{apk}\}|$ is the number of Android APKs where the feature value t appears. To avoid zero-division, we add 1 to the formula. We use $idf(t)$ as an alternative to compute frequencies, which shows less effectiveness in similar solutions.

8. EXPERIMENTAL RESULTS

In this section, we start with an overview of Cypider implementation and the testing setup, including the dataset and the performance measurement techniques. Afterward, we present the achieved results regarding the defined metrics for both usage scenarios that are adopted in Cypider framework, namely *malware only and mixed* datasets.

8.1 Implementation

We have implemented Cypider using *Python* programming language and *bash* command line tools. To generate binary N-grams, we used *xxd* tool to convert the package content into a sequence of bytes, in addition to a set of command tools such as *awk* and *grep* to filter the results. To perform reverse engineering of the *Dex* byte-code, we used *dexdump*, a tool provided with Android SDK. The generated assembly is filtered using the standard Unix tools. To extract the permissions from *AndroidManifest.xml*, we first converted the binary XML to a readable format using *aapt*, an Android SDK tool. Then, we parse the produced XML using the standard Python XML parsing library.

8.2 Dataset and Test Setup

In order to evaluate Cypider, we leverage well-known Android datasets, namely, i) Genome malware dataset [2] [57], ii) Drebin malware dataset [1] [20, 46]. As presented in Table 2, we build two additional datasets based on the previous ones by adding Android apps downloaded from Google Play in late 2014 and beginning of 2015. These apps have been randomly downloaded without considering their popularity or any other factor. In order to build *Drebin Mixed* dataset, we added 4,403 benign apps to the original *Drebin* dataset. The result is a mixed dataset (malware & benign) with 50% of apps in each category. Similarly, we build *Genome Mixed* dataset with 75% of benign apps (the *mixed dataset* will be publicly available for the community).

	Drebin	*DrebinMixed*	*Genome*	*GenomeMixed*
Size	4330	8733	1168	4239
Malware	4330	4330	1168	1168
Benign	0	4403	0	3071
Families	46	46	14	14

Table 2: Evaluation Datasets

The aim of using these datasets is to evaluate Cypider in the unsupervised usage scenarios, with and without benign apps, as presented in Section 2.1. First, we assess Cypider on malware only using *Drebin* and *Genome* datasets. This use case is the most attractive one in bulk malware analysis since it decreases the number of malware to be analyzed by considering only a sample from each detected community. Second, *Cypider* is evaluated against mixed datasets. The second scenario is more challenging because we expect not only the *suspicious communities* as output but also benign communities (false positives) along with filtered benign apps. To this end, various metrics are needed to measure Cypider performance in each dataset. We adopted the flowing metrics:

Apps Detection Metrics.

A1: *True Malware*: This metric computes the number of

malware apps that are detected by Cypider. It is applied in both usage scenarios.

A2: *False Malware*: This metric computes the number of benign apps that have been detected as a malware app. It is applied only on the *mixed dataset* since there are no benign apps in the other datasets.

A3: *True Benign*: This metric computes the number of filtered benign apps by Cypider. It is only applied on *mixed dataset* evaluation.

A4: *False Benign*: This metric counts the number of malware apps that are considered as benign in the *mixed dataset* evaluation.

Community Detection Metrics.

C1: *Detected Communities*: It indicates the number of suspicious communities that have been extracted by Cypider.

C2: *Pure Detected Communities*: This metric computes the number of communities with a unique Android malware family. In other words, a community is pure if it contains instances of the same family. In this task, we rely on the labels of the used datasets to check the purity of a given community. This metric is applied in both usage scenarios.

C3: *K-Mixed Communities*: This metric counts the communities with K-mixed malware families, where K is the number of families in a detected community. This metric is applied to both usage scenarios.

C4: *Benign Communities*: This metric computes the number of benign communities that have been detected as suspicious. This metric is used in the *mixed dataset* evaluation.

8.3 Mixed Dataset Results

Table 3 presents the evaluation results of Cypider using *Drebin Mixed* and *Genome Mixed* datasets. The most noticeable result is the fact that Cypider could detect about *half* of the actual malware in a single iteration in both datasets even though the noise of benign apps is about $50\% - 75\%$ of the actual dataset. On the other hand, Cypider was able to filter a considerable number of benign apps from the dataset. However, in both dataset evaluations, we obtained some *false malware* $(190 - 103$ apps) and *false benign* $(38 - 10)$ respectively to datasets. According to our results, these *false positives,* and *false negatives* appear, in most cases, in communities with the same label (malware or benign). Therefore, the investigation would be straightforward by analyzing some samples from a given suspicious community. The similarity network and the resultant communities are illustrated in Figure 7 and 8 respectively in the Appendix Section.

Table 4 presents the results of Cypider's evaluation using the *community metrics*. A very interesting result here is the number of *pure detected communities*, which is 179 out of 188 detected communities in *Mixed Drebin* and 61 out of 61 detected communities in *Mixed Genome*. Consequently, almost all the detected communities have instances in the same malware family or benign ones. Even the *mixed communities* are composed of only two labels (2-mixed). It is important to mention that all the *detected benign communities* are pure without any malware instance, which makes the investigation much easier. Furthermore, according to our analysis, most malware labels in the *2-mixed* malicious communities are just a naming variation of the same malware, which is caused by the name convention differences between vendors. For example, in one *2-mixed* community, we found *FakeInstaller* and *Opfake* malware instances. Actually, these names point to the same malware [9], which is *FakeInstaller*. Similarly, we found *FakeInstaller* and *TrojanSMS.Boxer.AQ*, which point to the same malware [3] with different vendor namings.

Community Metrics	**Drebin Mixed**	**Genome Mixed**
True Malware A1	2413	449
False Malware A2	190	103
True Benign A3	257	171
False Benign A4	38	10

Table 3: Evaluation Using Apps Metrics (Mixed)

Apps Metrics	**Drebin Mixed**	**Genome Mixed**
Detected C1	188	61
Pure Detected C2	179	61
2-Mixed C3	9	0
Benign C4	18	16

Table 4: Evaluation Using Community Metrics

Community Metrics	**Drebin**	**Genome**
True Malware A1	2223	449

Table 5: Evaluation Using Apps Metrics

Apps Metrics	**Drebin**	**Genome**
Detected C1	170	45
Pure Detected C2	161	45
2-Mixed C3	9	0

Table 6: Evaluation Using Community Metrics

8.4 Results of Malware-only Datasets

Tables 5 and 6 present the performance results of Cypider using the *app metrics* and *community metrics* utilizing malware only datasets. Since we use the same malware dataset as the *mixed dataset* by only excluding the benign apps, we obtained almost the same results. Cypider was able to detect about 50% of all malware in one iteration. Moreover, nearly all the recognized communities are pure. The precious result is a significant advantage of Cypider in malware investigation since the security analyst could automatically attribute the family to a given suspicious community by only matching one or two samples. Furthermore, the analysis complexity dramatically decreased from 2413 detected malware to only 188 discovered communities. We believe that this could reduce the analysis window and help overcome the overwhelming number of the daily detected Android malware. Notice that there are nine *2-mixed* communities in the *Drebin dataset*, which contain different malware labeled for the same actual malware, as mentioned before. Figure 5 depicts the *similarity network* of the Drebin malware dataset.

After applying the community detection algorithm, we end up with the *malicious communities*, as depicted in Figure 6.

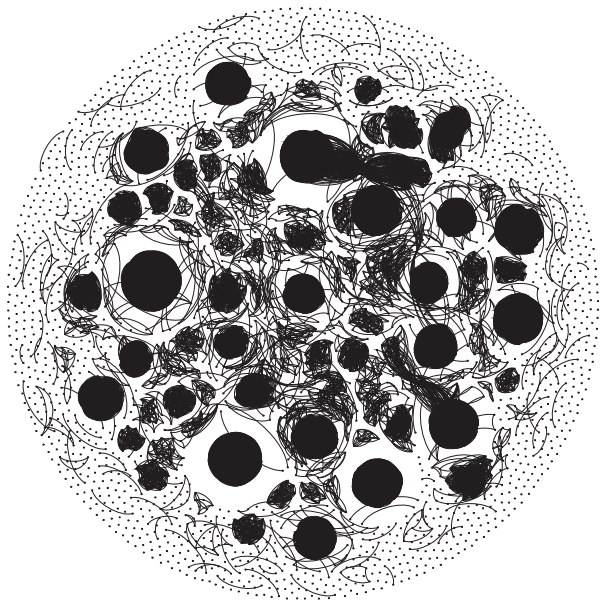

Figure 5: Cypider Network of *Drebin* Dataset

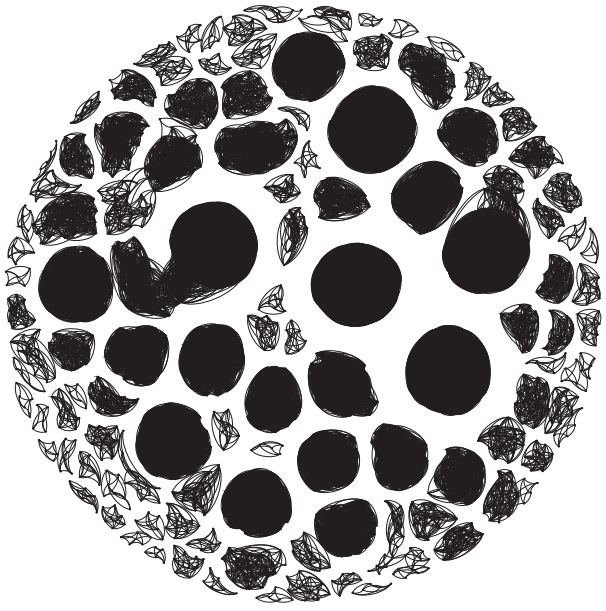

Figure 6: Detected Communities in *Drebin* Dataset

8.5 Community Fingerprint Results

The preliminary results of the *community fingerprint* are very promising, where we achieved, in best cases, a detection rate of 87% on new malware samples sharing the same family with a given *community fingerprint* in the *signature database*. Moreover, its compact format (learning model in binary format) could fingerprint a whole Android family, which implies a far more compressed *signature database*. The performance of this fingerprint significantly varies with the number of malware in the detected community. The more malware instances are in the community, the higher detection performance is. For this reason, we plan to set up a threshold for the needed community cardinal to compute the fingerprint and store it in the *signature database*. However, determining such threshold requires more evaluations of the proposed fingerprint in different communities, which we leave for future work.

9. DISCUSSION

Cypider framework achieves a good result regarding the percentage of the detected malware and purity of the communities. However, we believe that the detection performance of Cypider could be improved by considering more content vectors. The more static coverage of the APK file is, the more broad and accurate the suspicious communities are. Fortunately, Cypider could easily add new vectors due to the *majority-voting* mechanism to decide about the similarity. Using only static features leads to the right results. However, including dynamic features in Cypider's detection process could boost the detection since this covers Android malware that download and execute the payload at runtime. Cypider with only static features could be a complementary solution to other detection approaches based on *dynamic features*. As we have noticed in the evaluation results, there are more detected communities than the number of the actual malware families. According to our analyses, multiple communities could be part of the same malware family. However, these communities could represent malware family variants. For example, **DroidKungFu** family is considered as one family in *Drebin dataset*. However, there are many variants for this family in the *Genome dataset* such as **DroidKungFu1**, **DroidKungFu2** and **DroidKungFu4**. Moreover, we notice that some malware instances could have various labels from the vendors. For example, certain instances of **FakeInst** malware have their community with no connection shared with the rest of the family. After a small investigation using the hashes of the instances, we identified a different label (**Adwo**), which represents a different malware family. Moreover, the evaluation shows the effectiveness of *Cypider* in identifying Zero-day malicious apps since the detection was without any prior knowledge of the actual dataset. The unsupervised feature of *Cypider* framework could make it handy for security practitioner.

The concept of the malicious community, which is proposed by *Cypider*, could be similar to malware family concept. However, there are some differences: i) One malware family could be detected in multiple communities of malicious apps. So, the community gives a more granular view of the malicious app similarity based on a given statical features. ii) Security practitioner could decide about a given malicious app family using manual analyses. However, the community concept comes from a purely unsupervised automatic analysis. An important feature of Cypider' detected suspicious communities is the explainability, which facilitates defining what exactly is shared among the apps of a given community. The explainable results are due to the multiple content similarity links between the apps. For example, malware apps sharing the same IP addresses that are hard-coded in the binary are most likely to be part of a Command and Control (C&C) infrastructure and are the sign of a botnet.

10. LIMITATION AND FUTURE WORK

Obfuscation is considered a big issue for malware detection systems including *Cypider*, where the adversary uses an obfuscated content or transform attacks. The latter attack, which could download the malicious code and execute it at runtime is undetectable by Cypider unless the malware instances share other covered contents. However, *Cypider* attempts to deal with obfuscated apps with a range of techniques. i) *Cypider* leverages multiple Android package contents (such as permissions) which allow the system to be more robust against obfuscation since it also uses other unobfuscated contents. ii) In the case of Dex disassembly, *Cypider* uses instruction opcode instead of the whole instruction to compute the N-grams. Therefore, the latter would be more resilient to obfuscation in the instruction operands. The previous techniques show their effectiveness in the evaluation since we used a real malware dataset in addition to apps from Google play (where the apps are supposedly obfuscated through ProGuard). Besides, Cypider framework could be complementary to dynamic analysis solutions to achieve better results.

Cypider's detection process relies on a loop. In each iteration, Cypider detects a portion of the malware sample by defining its communities. The size of the *active dataset* is an important parameter that should be set according to the computation resources, the hard line of the detection window, and the usage scenario. Also, we consider other parameters related to similarity network building, as presented in Section 6. However, once the parameters are defined, there is no need to adjust them as is the case of our evaluation. Furthermore, in the current implementation of the proposed feature vectors, we did not include contents such as the *native library N-grams* and *manifest activities*, a part of the similarity computation using *majority-voting* technique. We plan to consider these contents in future work, where we focus on other Android malware that uses, for example, C/C++ native code.

The current *majority-voting* mechanism is agnostic to the content type. Hence, it considers all the content similarity when deciding about a similarity link. However, a more accurate representation could be achieved using *weighted majority voting*. For instance, the similarity in the opcode could be more valuable than the similarity of strings in the resources files. Similarly, malware instances with identical IP addresses are more suspicious than URLs since IP addresses are less used in benign apps and could indicate the existence of a botnet. For the aforementioned reasons, we plan to implement and evaluate the *weighted majority voting* in future work.

11. RELATED WORK

Previous works on Android malware detection mainly use two basic approaches: static and dynamic analyses. Static analysis-based approaches [20, 39, 32, 52] rely on static features extracted from the app, such as requested permissions and APIs, to detect malicious apps. These methods are generally not resistant to obfuscation. On the other hand, dynamic analysis-based approaches [26, 46, 16, 54, 19, 50, 50, 37] aim to identify a behavioral signature or anomaly of the running app. These methods are more resistant to obfuscation than the static ones as there are many ways to hide the malicious code, while it is difficult to hide the malicious behavior. However, the dynamic methods are limited in scalability as they incur additional costs in terms of processing and memory in order to run the app. In addition, it is not known when the malicious behavior will occur, and hence each app requires a long-running time. The hybrid analysis methods [53, 34, 22, 49] combine between static and dynamic analyses. Some approaches [44, 40, 18] do not offer scalable analysis as they use multi-stage machine learning or employ program control graph. Cai and Yap [24] perform a large-scale experiment on Android Anti-Viruses (AVs). Their results show that a majority of AVs detect malware using simple static features. Such features can be easily obfuscated by renaming or encrypting strings and data, making it easy to evade some AVs. Hoffmann et al. [36] present a new obfuscation framework that aims to break the assumptions used by static and dynamic analyses tools. Our work adopts the static analysis approach, but being different from previous work, it uses a set of statistical features that cover most of the static characteristics of the APK. In this way, it is difficult for an adversary to evade detection as it has to obfuscate all the features. In some cases, malicious apps, which belong to the same family, establish connection with the same $C\&C$ server. The adversary cannot change the server's IP. Otherwise, the malicious activity cannot be performed.

A significant number of research work has been recently proposed to detect repackaged apps by performing similarity analysis. The latter either identifies the apps that use the same malicious code (i.e., detection of malware families) [41, 17, 30, 58, 33, 47, 38, 42, 31], or those that repackage the same original app (i.e., code reuse detection) [27, 48, 56, 35, 28, 55, 29]. However, most of them use non-scalable techniques such as control flow graphs to detect similar codes. Differently, our work is novel in the sense it represents an app as a vector of contents. If one content is shared between pair of apps, both of them are similar and are suspected to be malicious.

12. CONCLUSION

In this paper, we have presented Cypider, an efficient and scalable framework for Android malware detection. The detection mechanism relies on the community concept. Cypider consists of a systematic framework that can generate a fingerprint for each community, and identify known and unknown malicious communities. Cypider has been implemented and evaluated on different malicious and mixed datasets. Our findings show that Cypider is a valuable and promising approach in detecting application similarity and malicious communities in Android applications. Cypider only needs less than 82 minutes to build a network similarity of $50,000$ apps. The preliminary results of the community fingerprint are very promising as 87% of the detection is achieved. As future work, we plan to investigate more features selection, attempting to increase the detection rate.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments that helped improving this paper.

13. REFERENCES

[1] Drebin Dataset - http://tinyurl.com/pdsrtez, 2015.

[2] Genome Dataset - http://tinyurl.com/combopx, 2015.
[3] Andr/Boxer-C - https://tinyurl.com/zpk3xkd, 2016.
[4] Android - http://tinyurl.com/prhj8zn, 2016.
[5] Android Auto - http://tinyurl.com/hdsunht, 2016.
[6] Android NDK - http://tinyurl.com/ppn559l, 2016.
[7] Android TV - http://tinyurl.com/zlb6hk6, 2016.
[8] Android Wear - http://tinyurl.com/qfa55o4, 2016.
[9] Andr/OpFake-U - https://tinyurl.com/h9lxk7p, 2016.
[10] App Manifest - http://tinyurl.com/zdjps4o, 2016.
[11] Brillo - http://tinyurl.com/q5ko3zu, 2016.
[12] G DATA Mobile Malware Report - http://tinyurl.com/jecm8gg, 2016.
[13] Google Play - https://play.google.com/, 2016.
[14] IDC - http://tinyurl.com/p7znq9m, 2016.
[15] tf-idf - https://tinyurl.com/mcdf46g, 2016.
[16] A. Ali-Gombe, I. Ahmed, G. G. Richard III, and V. Roussev. AspectDroid: Android App Analysis System. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 145–147, New York, NY, USA, 2016. ACM, ACM.
[17] A. Ali-Gombe, I. Ahmed, G. G. Richard III, V. Roussev, A. A. Gombe, I. Ahmed, G. G. R. III, and V. Roussev. OpSeq: Android Malware Fingerprinting. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 7:1—-7:12. ACM, 2015.
[18] K. Allix, T. F. Bissyandé, Q. Jerome, J. Klein, Y. Le Traon, and Others. Large-scale machine learning-based malware detection: confronting the 10-fold cross validation scheme with reality. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 163–166, 2014.
[19] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 1666–1671. IEEE, 2013.
[20] D. Arp, M. Spreitzenbarth, H. Malte, H. Gascon, K. Rieck, M. Hubner, H. Gascon, and K. Rieck. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Symposium on Network and Distributed System Security (NDSS)*, pages 23–26, 2014.
[21] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. *Proceedings of the 14th international conference on World Wide Web - WWW '05*, page 651, 2005.
[22] S. Bhandari, R. Gupta, V. Laxmi, M. S. Gaur, A. Zemmari, and M. Anikeev. DRACO: DRoid analyst combo an android malware analysis framework. In *Proceedings of the 8th International Conference on Security of Information and Networks*, pages 283–289. ACM, 2015.
[23] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 10008(10):6, 2008.
[24] Z. Cai and R. H. C. Yap. Inferring the Detection Logic and Evaluating the Effectiveness of Android Anti-Virus Apps. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, number January 2009 in CODASPY '16, pages 172–182, 2016.
[25] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Detecting Android malware using sequences of system calls. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015, Bergamo, Italy, August 31 - September 4, 2015*, DeMobile 2015, pages 13–20, 2015.
[26] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio. Acquiring and Analyzing App Metrics for Effective Mobile Malware Detection. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, number March in IWSPA '16, pages 50–57. ACM, 2016.
[27] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674. USENIX Association, 2015.
[28] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Computer Security - {ESORICS} 2012 - 17th European Symposium on Research in Computer Security Proceedings*, pages 37–54. Springer, 2012.
[29] J. Crussell, C. Gibler, and H. Chen. AnDarwin: Scalable Detection of Semantically Similar Android Applications. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, volume PP, pages 182–199, 2013.
[30] L. Deshotels, V. Notani, and A. Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014*, PPREW'14, page 3, 2014.
[31] P. Faruki, V. Laxmi, A. Bharmal, M. S. Gaur, and V. Ganmoor. AndroSimilar: Robust signature for detecting variants of Android malware. *Journal of Information Security and Applications*, 22:66–80, 2015.
[32] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587. ACM, 2014.
[33] 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*, AISec '13, pages 45–54, 2013.
[34] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294. ACM, ACM, 2012.
[35] S. Hanna, L. Huang, E. X. Wu, S. Li, C. Chen, and D. Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection*

of Intrusions and Malware, and Vulnerability Assessment, volume 7591 LNCS, pages 62–81. Springer, 2012.

[36] J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, and T. Holz. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 139–141, 2016.

[37] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang. AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046. ACM, 2014.

[38] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim. Detecting and classifying android malware using static analysis along with creator information. *International Journal of Distributed Sensor Networks*, 2015:7, 2015.

[39] E. B. Karbab, M. Debbabi, and D. Mouheb. Fingerprinting Android packaging: Generating DNAs for malware detection. *Digital Investigation*, 18:S33–S45, 2016.

[40] P. M. Kate and S. V. Dhavale. Two Phase Static Analysis Technique for Android Malware Detection. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*, pages 650–655. ACM, 2015.

[41] J. Kim, T. G. Kim, and E. G. Im. Structural information based malicious app similarity calculation and clustering. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, pages 314–318. ACM, 2015.

[42] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

[43] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, a. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural computation*, 13(7):1443–1471, 2001.

[44] S. Sheen and A. Ramalingam. Malware Detection in Android files based on Multiple levels of Learning and Diverse Data Sources. In *Proceedings of the Third International Symposium on Women in Computing and Informatics, {WCI} 2015, co-located with {ICACCI} 2015, Kochi, India, August 10-13, 2015*, pages 553–559. ACM, 2015.

[45] Q. Shi, J. Petterson, G. Dror, J. Langford, A. J. Smola, A. L. Strehl, and V. Vishwanathan. Hash Kernels. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics, AISTATS 2009, Clearwater*, pages 496–503, 2009.

[46] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815. ACM, 2013.

[47] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[48] M. Sun, M. Li, and J. C. S. Lui. DroidEagle: seamless detection of visually similar Android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, page 9, 2015.

[49] T. Vidas, J. Tan, J. Nahata, C. L. Tan, N. Christin, and P. Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 39–50, 2014.

[50] T.-E. E. Wei, C.-H. H. Mao, A. B. Jeng, H.-M. M. Lee, H.-T. T. Wang, and D.-J. J. Wu. Android malware detection via a latent network behavior analysis. In *Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1251–1258. IEEE, 2012.

[51] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola. Feature Hashing for Large Scale Multitask Learning. *Proceedings of the 26th Annual International Conference on Machine Learning*, pages (pp. 1113–1120)., 2009.

[52] W. Yang, J. Li, Y. Zhang, Y. Li, J. Shu, and D. Gu. APKLancet: tumor payload diagnosis and purification for android applications. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, ASIA CCS '14, pages 483–494, 2014.

[53] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue. Droid-Sec: deep learning in android malware detection. In F. E. Bustamante, Y. C. Hu, A. Krishnamurthy, and S. Ratnasamy, editors, *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372, 2014.

[54] 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 *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 611–622. ACM, 2013.

[55] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY '13, pages 185–196, 2013.

[56] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, 2012.

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

[58] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Symposium on Network and Distributed System Security (NDSS)*, volume 25, pages 50–52, 2012.

APPENDIX

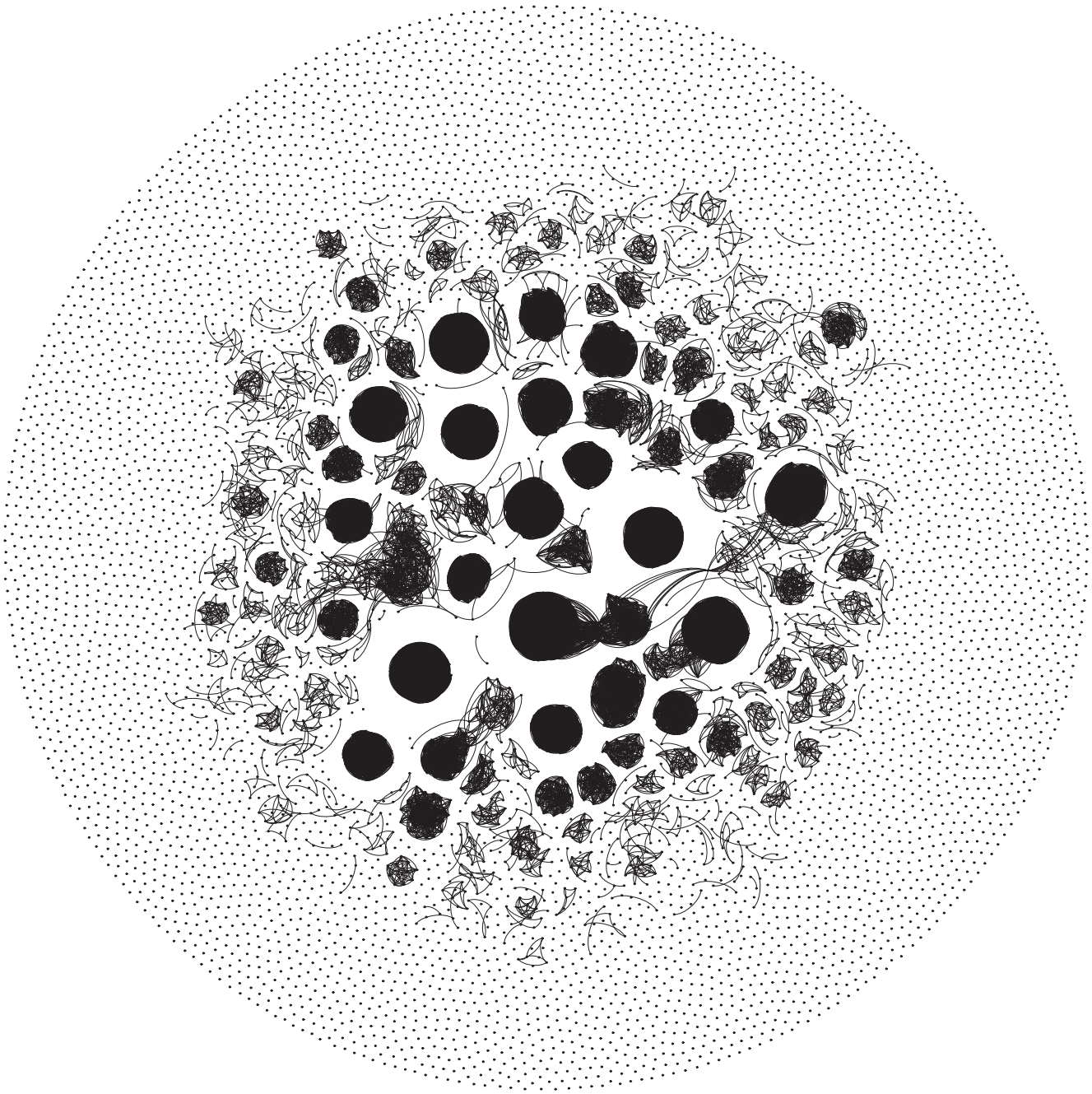

Figure 7: Cypider Network of *Mixed Drebin* Dataset

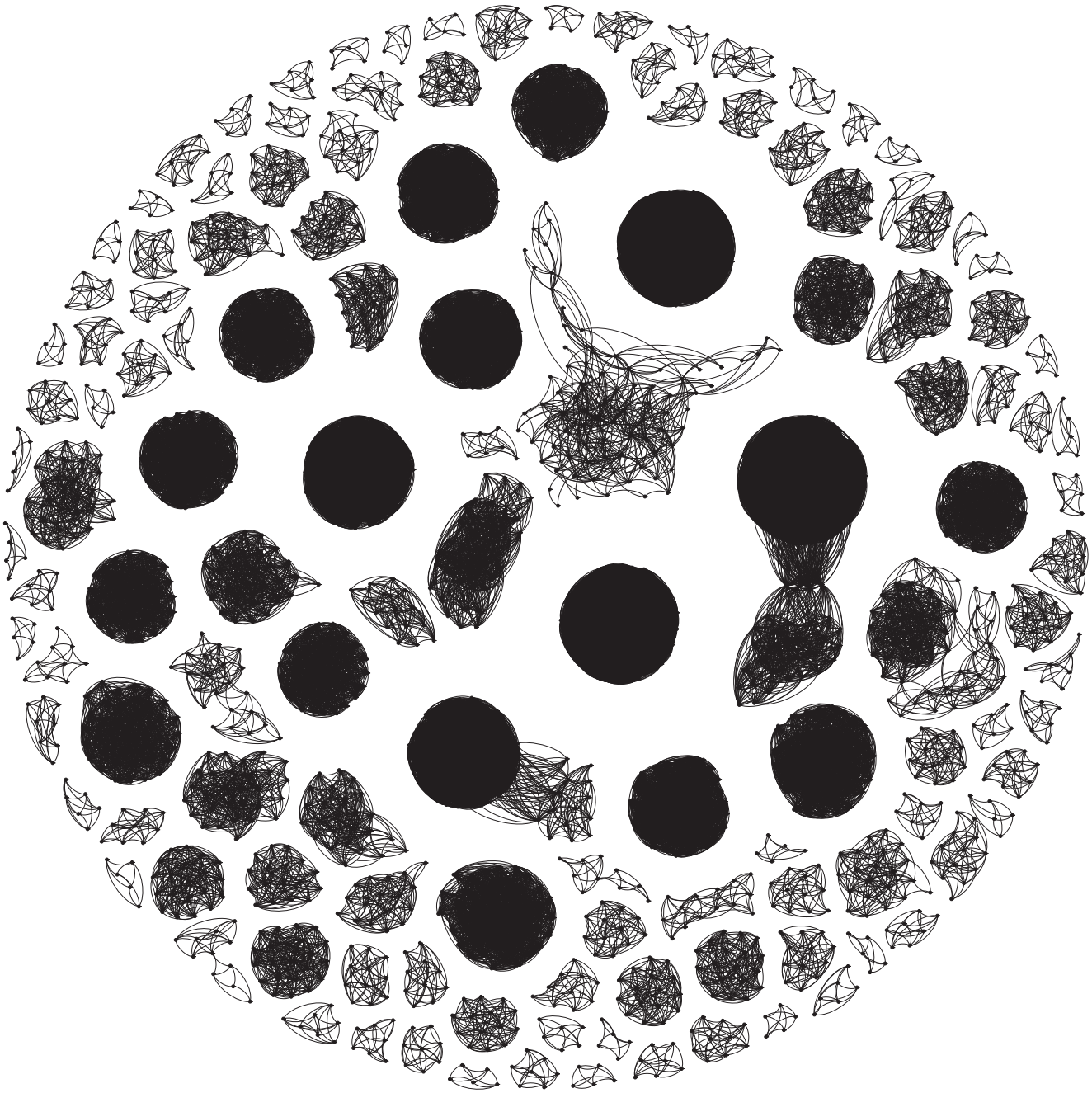

Figure 8: Detected Communities in *Mixed Drebin* Dataset