ACTS: Extracting Android App Topological Signature through Graphlet Sampling

Wei Peng*, Tianchong Gao†, Devkishen Sisodia‡, Tanay Kumar Saha†, Feng Li†, Mohammad Al Hasan†
*Intel Corporation, Folsom, CA, U.S.A.
wei.peng@intel.com
†Indiana University-Purdue University Indianapolis, Indianapolis, IN, U.S.A.
{tgao, fengli}@iupui.edu, {tksaha, alhasan}@cs.iupui.edu
‡University of Oregon, Eugene, OR, U.S.A.
dsisodia@cs.uoregon.edu

***Abstract*—Android systems are widely used in mobile & wireless distributed systems. In the near future, Android is believed to dominate the mobile distributed environment. However, with the popularity of Android-based smartphones/tablets comes the rampancy of Android-based malware. In this paper, we propose a novel topological signature of Android apps based on the function call graphs (FCGs) extracted from their Android App PacKages (APKs). Specifically, by leveraging recent advances in graphlet sampling, the proposed method fully captures the invocator-invocatee relationship at local neighborhoods in an FCG without exponentially inflating the state space. Using real benign app and malware samples, we demonstrate that our method, ACTS (App topologiCal signature through graphleT Sampling), can detect malware and identify malware families robustly and efficiently. More importantly, we demonstrate that, without augmenting the FCG with any semantic features such as bytecode-based vertex typing, local topological information captured by ACTS alone can achieve a high malware detection accuracy. Since ACTS only uses structural features, which are orthogonal to semantic features, it is expected that combining them would give a greater improvement in malware detection accuracy than combining non-orthogonal semantic features.**

***Index terms*—Android; graphlet sampling; mobile applications; mobile malware; smartphone**

I. Introduction

Some rising trends in mobile systems, e.g., the wearable devices, the medical devices and the intelligent vehicle systems, are setup on Android platforms following the big success of it on smartphone market. Since Android applications are specifically designed to have as few implementation dependencies as possible, Android is believed to be adaptive to the new market and dominate the mobile distributed environment soon.

As the use of Android continues to grow, so does the threat of malware. Malicious behaviors observed in such malware include the theft of private information stored on the device, device fingerprinting, abusing premium service, and rooting the device as a backdoor for further attacks [36]. Detecting such malware is a critical task for the security research community.

It is observed that variants of malware form families through code sharing and their common lineage [36]. Therefore, instead of identifying individual malware and extracting a signature from it, we can identify the commonality within the same malware family and generate signatures that capture such commonality. Recently, various machine learning/data mining (i.e., pattern mining) techniques are applied to detect Android malware [1, 2, 10, 17, 30, 33] or closely related tasks such as identifying repackaged apps [34, 35]. Beyond the common pattern mining framework, these works differ significantly in their selection and construction of features, their quantification/metrication of such features, their choice of pattern mining algorithms, and, in totality of these fine points of design, their applicability, robustness, and efficiency in detecting malware.

A number of different app representations have been studied for malware detection. For example, Yamaguchi et al. propose a compact representation of source code, the code property graph, that combines abstract syntax trees, control flow graphs, and program dependence graphs [30]. Other approaches do not require the source, but instead focusing on features at different abstract levels: from the low-level platform opcode level [33], through the intermediate function call [10] and Android framework API [1] level, to the high semantic level that includes features such as network addresses and Android specific artifacts such as permission and Intents [2]. Yet, other works formulate malware detection as different pattern mining tasks such as frequent subgraph mining [17].

Due to the availability of off-the-shelf obfuscation solutions (such as the free ProGuard [27] and the commercial DexGuard [26]) and the growing number of Android apps, it is critical for any proposed malware detection algorithm to be robust and efficient. Our first step towards robustness is to extract from the app under investigation its function call graph (FCG) [10], in which each vertex represents a Java method and each edge represents a method invocation. We concur with Gascon et al. [10] that FCG is at a proper abstraction level for detecting malware: In addition to the non-essential transformations mentioned above, it is also immune to, for example, both lower-level opcode/instruction obfuscation or higher-level content encryption.

Based on the extracted FCG, we propose an efficient and robust Android app signature that faithfully captures the invocator-invocatee relationship between several functions, i.e., the topology of local neighborhoods on the FCG. Instead

978-1-5090-3065-1/16/$31.00 ©2016 IEEE
Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

of using vertices and edges (or extension to 1-hop neighborhoods [10]) on the FCG "as is," we leverage recent advances in graph mining to efficiently sample *graphlets* [21, 22] on the FCG. Graphlets are small (e.g., less than 6), connected, vertex-induced embedded subgraphs in an underlying graph, which is the FCG in our case. In the spectrum of purely local (e.g., individual vertices/edges and simple metrics such as degrees) and fully global (e.g., betweenness centrality [4]) scope of the FCG, our graphlet-based signature takes a unique position: It faithfully captures local topological information at a fine-grained granularity without exponentially inflating the state space.

Given these characteristics, we call our graphlet-based signature a *topological* signature and, accordingly, name our method ACTS (App topologiCal signature through graphleT Sampling). In our experiments, ACTS achieves a cross-validated accuracy as high as 87.9% . In comparison, the same method with a purely local feature (i.e., degree frequency distribution (DFD) [7]) has an average cross-validated accuracy of 75%. Since ACTS only uses structural features, which are orthogonal to semantic features such as bytecode-based vertex typing, it is expected that combining them would give a greater improvement in malware detection accuracy than combining non-orthogonal semantic features.

In summary, our contributions are:

- We propose a novel topological signature for Android apps that fully captures the invocator-invocatee relationship in an app's FCG, which is otherwise lost in a global topological metric such as betweenness centrality [4], without exponentially inflating the state space as in n-hop neighborhoods with $n \geq 3$.
- By leveraging recent advances in graphlet sampling, we make the generation of our proposed topological signature practically efficient without sacrificing its robustness.
- With experiments on real malware/benign app samples, we demonstrate that local topological information captured by our method alone can achieve a high malware detection accuracy, which can be further improved by incorporating (orthogonal) semantic features.

In the rest of the paper, after the preliminaries (Section II), we present our method (Section III) and experiment results on real malware/benign app samples (Section IV). We then reflect on our method (Section V) and conclude with a brief review of related works (Section VI).

II. Preliminaries

A. Function call graph

Function call graph (FCG) is a graph model for functions and their invocation relationship, in which vertices represent functions and a directed edge from vertex v_1 to v_2 represents that v_1 invokes v_2. For an Android app, functions are Java methods, and their invocation relationship can be statically extracted from Java bytecode by searching for the invocation-related opcodes, i.e., invoke-*.

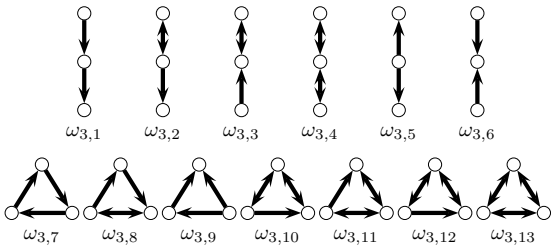

Fig. 1: The 13 unique 3-graphlet types $\omega_{3,i}$ ($i = 1, 2, \ldots, 13$).

B. Graphlets

Pržulj et al. first consider a complete set of local graph topologies with 3, 4, and 5 vertices and name them graphlets[1] in their work on characterizing biological networks [20]. Formally, given a graph G, graphlets of G are small, connected, non-isomorphic, and vertex-induced subgraphs of G. Although earlier works [20, 21, 22] on graphlets focus on undirected graphs, we consider directed graphlets to preserve the inherent directionality of FCGs.

Figure 1 enumerates all the 13 unique types of (directed) graphlets $\omega_{3,i}$[2] ($i = 1, 2, \ldots, 13$) with 3 vertices (the 3-graphlets): They are pair-wise non-isomorphic. These graphlet types do not appear equally likely in an FCG. For instance, although there are many cases in which a function invokes two others ($\omega_{3,5}$) or two different functions invoke the same one ($\omega_{3,6}$), 3 mutually recursive functions ($\omega_{3,13}$) are rare. Later, we will discuss how we use this observation to improve the performance of our method (Section III-C).

For vertices 4, 5, and 6, the number of graphlet types are 199, $9,364$, and $1,530,843$, respectively [24]. We focus on graphlets with less than 6 vertices in this work because larger graphlet types require extra computations but provide little value in capturing the structure of FCG. Figure 2 illustrates our running example: A 4-graphlet g (the grey vertices and their induced edges) embedded in a 6-vertex graph G.

C. Graphlet frequency distribution (GFD)

Graphlet frequency distribution (GFD) of a graph G is the probability distribution of the frequencies of the different graphlet types in G. For instance, since the number of 3-graphlets in a (finite) FCG G is finite, we can, in principle, enumerate all embedded graphlets in G and, for each such embedded graphlet g, identify g with one of the 13 graphlet types in Figure 1. At the end of the enumeration, suppose the count (i.e., the frequency) of graphlet type $\omega_{3,i}$ is $f_{3,i}$ ($i \in \{1, 2, \ldots, 13\}$), the frequency distribution density $d_{3,i}$ at $\omega_{3,i}$ is $f_{3,i} / \sum_{i=1}^{13} f_{3,i}$. We call the vector $(d_{3,1}, d_{3,2}, \ldots, d_{3,13})$ the *3-graphlet frequency distribution* (3-GFD) of G. We can compute n-GFD for any n with the same procedure, and concatenate several n-GFDs with different n into a single

[1]Graphlet is also used to refer wavelet decomposition of graphs [28], which is an unrelated concept to what we use in this work.

[2]The unique types of n-graphlets are enumerated as $\omega_{n,1}, \omega_{n,2}, \ldots, \omega_{n,N(n)}$, with $N(n)$ being the number of unique types for n-graphlets.

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

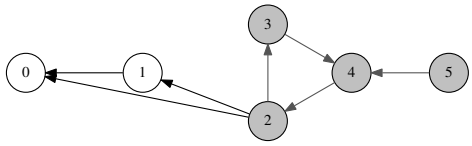

Fig. 2: Our running example: A 4-graphlet g (the grey vertices and their induced edges) embedded in a 6-vertex graph G.

vector. We can call the concatenated vector a GFD of G if there is no confusion on its constituents.

The above procedure only works in principle. In practice, the fast growing number of apps, the size of real apps' FCGs, and the combined computation complexity of graphlet enumeration and identify graphlet types make the enumeration-and-count procedure impractical to use. Nevertheless, GFD is a step forward towards our goal: It is a metrication from the (combinatorial) graphlet space into the (metric) Euclidean space, where we can apply pattern learning techniques to detect malware. In other words, GFD preserves the topological information of local neighborhoods in an FCG. Later, after giving a high-level overview of our method (Section III-A), we will focus on how to estimate GFD efficiently (Section III-B).

D. Metropolis-Hastings (M-H) algorithm

Markov chain Monte Carlo (MCMC) [11] is a class of algorithms for sampling from a probability distribution. Given an intended sampling distribution $p(x)$ over a sample space X, the idea behind general Markov chain Monte Carlo (MCMC) methods (in which the M-H algorithm is a specific method) is to construct a Markov chain over X whose stationary distribution equals to $p(x)$: After the Markov chain mixes (i.e., reaches its stationary distribution and, hence, "forgets" where it begins), the subsequently visited states of the chain can be used as samples from the intended distribution $P(x)$.

Metropolis-Hastings (M-H) algorithm [18] is a specific MCMC method that we use for estimating GFD (Section III-B). In the M-H algorithm, the transition between two consecutive states x and x' in the chain consists of two stages: proposals and acceptance/rejection. Correspondingly, there is a proposal distribution $q(x'|x)$ (the probability of *proposing* x' as the next state given the current state x) and an acceptance distribution $a(x'|x) = \min(1, A(x'|x))$ (the probability of *accepting* x' as the next state given the current state x), in which:

$$A(x'|x) = \frac{p(x')q(x|x')}{p(x)q(x'|x)}. \quad (1)$$

Intuitively, for each iteration of the sampling process, we first randomly pick x' with a probability of $q(x'|x)$, and then either accept x' (by sampling x') with a probability of $a(x'|x)$ or reject x' (by sampling x again) with a probability of $1 - a(x'|x)$.

III. Method

In this section, after a brief overview of our method (Section III-A), we zoom in on two technical points: Efficient GFD estimation (Section III-B) and FCG-specific GFD dimension reduction heuristics (III-C) that distinguish our method.

A. Overview

Given an Android app's APK (Android PacKage) binary package, we:

- extract an FCG from the APK,
- estimate the GFD of the FCG (Section III-B), and
- project the estimated GFD to a lower dimensional space to reduce the GFD's dimensions (Section III-C).

The projected GFD, which is a vector, is a signature of the app. To stress that this signature preserves detailed topological information on an app's FCG, we call it the *topological signature* (TS) of the app.

Given a pool of both malware and benign app samples, we train a classifier on their TSs to detect malware: If the TS of an app is classified as a malware, the app is flagged as malware.

B. Efficient GFD estimation

Suppose we have a *uniform sampler* of the FCG, we can approximate the whole FCG's GFD with our samples' GFD. The more samples we take, the closer the approximation is. Given the large sample space and the (relatively small) number of bins (i.e., unique graphlet types) for n-graphlets with $n < 6$, we only need to sample a tiny fraction of the sample space to get a close approximation.

This apparently solve the GFD estimation problem. However, the real problem is that we need to *uniformly* sample graphlets from the FCG *without enumerating the sample space*. Fortunately, two recent advances on graph mining, GRAFT [21] and GUISE [22], show that GFD can be estimated without enumerating all graphlets. Inspired by these works, we use MCMC to sample the directed FCG.

1) Sample space and intended distribution: Since our goal is to uniformly sample from all the embedded graphlets in the FCG:

- The sample space X consists of all the embedded graphlets in the FCG.
- The intended distribution $p(x)$ over X is the uniform distributions, i.e., $p(x) = p(x')$ for any $x, x' \in X$.

Suppose we have just sampled graphlet g in the sampling process, the M-H algorithm (Section II-D) says that, if we propose to sample graphlet g' next with a probability of $q(g'|g)$, an acceptance probability of $a(g'|g) = \min(1, A(g'|g))$ (in which $A(g'|g)$ is defined by Equation (1)) will eventually lead to a sampling process that have the desired sampling distribution $p(x)$.

2) FCG-induced graphlet graph and graphlet neighboring relationship: To define the proposal distribution $q(x'|x)$, we consider the *FCG-induced graphlet graph* $\mathcal{G}_G$ of the FCG G. The FCG-induced graphlet graph $\mathcal{G}_G$ is an undirected graph with vertices being all the embedded graphlets in the FCG, and edges defined by the *graphlet neighboring relationship*

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

between the vertices. The graphlet neighboring relationship is a symmetric relationship between two graphlet embeddings g_1 and g_2 in the FCG: g_1 and g_2 are graphlet neighbors if and only if they differ by share all but one vertex. In particular, self-neighboring is excluded by this definition because there is no vertex difference, which is required by the definition. Since graphlets on G and vertices on $\mathcal{G}_G$ have a one-to-one map, we identify a graphlet g on G with the vertex on $\mathcal{G}_G$ that it maps to, and also denote that vertex with g if there is no confusion in the context.

For example, in Figure 2, g's neighbors on $\mathcal{G}_G$ are[3] all the 3-graphlets (e.g., $\{v_2, v_3, v_4\}$, $\{v_3, v_4, v_5\}$, etc.), 4-graphlets (e.g., $\{v_1, v_2, v_3, v_4\}$, $\{v_0, v_2, v_4, v_5\}$, etc.), and 5-graphlets ($\{v_1, v_2, v_3, v_4, v_5\}$ and $\{v_0, v_2, v_3, v_4, v_5\}$) that share all but one vertex with it. Conversely, 1) $\{v_1, v_2, v_3\}$ is not a neighbor of g because it does not contain both v_4 and v_5, which are in g; 2) $\{v_0, v_1, v_2, v_3\}$ is not a neighbor of g because it does not contain g's vertices v_4 and v_5 (and g does not contain its vertices v_0 and v_1); 3) $\{v_0, v_1, \ldots, v_5\}$ is not a neighbor of g because g does not contain its vertices v_0 and v_1.

The significance of the graphlet neighboring relationship on $\mathcal{G}_G$ is that it can be efficiently generated by *local information* on the FCG G *without enumerating the whole* G. Specifically, given an embedded graphlet g of G, the neighbors of g on $\mathcal{G}_G$ can be generated by removing, changing, or adding exactly one vertex in g. Hence, we can efficiently compute the degree d_g of g in $\mathcal{G}_G$ by generating and counting g's neighbors.

3) Proposal and acceptance distributions: Let $d(g)$ and $N(g)$ be graphlet g's degree and neighbors in $\mathcal{G}_G$, respectively. Suppose the last graphlet we have sampled is g, our proposal strategy $q(g'|g)$ is to uniformly sample one of its neighbors in $\mathcal{G}_G$, i.e.,

$$q(g'|g) = \begin{cases} \frac{1}{d_g} & \text{if } g' \in N(g), \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Since d_g can be efficiently computed without enumerating the graph (see above), $q(g'|g)$ can also be efficiently computed since it only requires computing d_g.

By Equations (1) and (2), the resulting acceptance strategy $a(g'|g)$ is:

$$a(g'|g) = \begin{cases} \min(1, \frac{d_g}{d_{g'}}) & \text{if } g' \in N(g), \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

By Equations (2) and (3), the probability $s(g'|g)$ of sampling g' next given the current sample g is:

$$s(g'|g) = \begin{cases} \min(\frac{1}{d_g}, \frac{1}{d_{g'}}) & g' \in N(g), \\ 1 - \sum_{h \in N(g)} \min(\frac{1}{d_g}, \frac{1}{d_h}) & g' = g, \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

The intuition behind the sampling strategy in Equation (4) can be understood in the following two cases.

Case 1. If g is a graphlet that has the highest degree among its neighbors in $\mathcal{G}_G$, i.e., $d_g \geq d_{g'}$ for any $g' \in N(g)$, then $\min(1/d_g, 1/d_{g'}) = 1/d_g$ and, hence, by Equation (4), $s(g|g) = 1 - d_g(\frac{1}{d_g}) = 1 - 1 = 0$, i.e., the next sample will *not* be g but one of its neighbors.

Case 2. If g is a graphlet with a relatively low degree among its neighbors in $\mathcal{G}_G$, $s(g'|g)$ in Equation (4) will be greater than 0. The greater the degree differences are, the greater $s(g'|g)$ will be. In an extreme case in which g has a single neighbor g' with a degree of 100 (i.e., $d_g = 1$ and $d_{g'} = 100$), $s(g'|g) = 0.01$ and $s(g|g) = 0.99$: If the current sample is g, 99 out of 100 times, the next sample will still be g.

In other words, the sampling process (i.e., the consecutive states of the Markov chain) is more eager to *move away from* the more popular graphlets (i.e., the ones with higher degrees in $\mathcal{G}_G$) and to *stay at* the less popular ones: The former has a better chance than the latter of being revisited later. This results in a fair (i.e., uniform) sampling of all the embedded graphlets in the FCG G.

4) GFD estimation algorithm: Finally, we estimate the GFD for the FCG G from t samples by evaluating ESTIMATE-GFD(G, t) in Algorithm 1. In our experiment, we evaluate multiple t and choose $100,000$ for having both low variance in the sampling result and acceptable efficiency. Note that, given the average size of an FCG G (thousands of vertices) and, hence, the sample space $\mathcal{G}_G$ (for a $1,000$-vertex G, $\mathcal{G}_G$ has a worst-case size of $O(1,000^3)$), $100,000$ iterations are quite small. Indeed, for the largest app in our dataset (the Facebook app, with $47,539$ vertices and $77,900$ edges), ESTIMATE-GFD(G, T) for $T = 100,000$ only takes only about 34 seconds on our desktop workstation with high convergence across multiple runs.

Algorithm 1 Estimate GFD for the FCG G from t samples.

▶ $\mathcal{C}$: all the *distinct* n-graphlet types for $n \in \{3, 4, 5\}$
▶ f_c: frequency counter for graphlet type $c \in \mathcal{C}$
▶ d_c: frequency density estimation for graphlet type $c \in \mathcal{C}$
Input: G: the FCG; t: number of iterations
function ESTIMATE-GFD(G, T)
  $g \leftarrow$ a random (initial) graphlet ▶ bootstrap the sampling process
  NEXT-SAMPLE(G, g, T) ▶ obtain the vector $(f_c|c \in \mathcal{C})$
  for $c \in \mathcal{C}$ **do** ▶ for each graphlet type $c \in C$
    $d_c \leftarrow f_c / \sum_{c \in \mathcal{C}} f_c$ ▶ estimate its graphlet density
  end for
  return $(d_c|c \in \mathcal{C})$ ▶ $(d_c|c \in \mathcal{C})$ is a vector ordered by C
end function
Input: G: the FCG; g: current graphlet sample; k: remaining iterations
procedure NEXT-SAMPLE(G, g, k)
  $N(g) \leftarrow g$'s neighbors in $\mathcal{G}_G$ ▶ Section III-B2
  choose a $g' \in N(g)$ with an equal probability of $1/d_g$ ▶ Equation (2)
  $a \leftarrow$ a number uniformly sampled from $[0, 1]$
  if $a \leq \min(1, d_g/d_{g'})$ **then** ▶ accepting g'
    $g \leftarrow g'$
  else ▶ rejecting g'
  end if
  $c \leftarrow \mathcal{C}(g)$ ▶ identify (the new) g's type
  $f_c \leftarrow f_c + 1$ ▶ increase g's count
  if $k > 0$ **then** ▶ if there are remaining iterations
    NEXT-SAMPLE$(G, g, k - 1)$ ▶ we continue the sampling process
  end if
end procedure

C. FCG-specific GFD dimension reduction heuristics

The curse of dimensionality [13] plagues many machine learning tasks. Theoretically, by confining the n-graphlets we sampled to $n \in \{3, 4, 5\}$, the GFD vectors we obtain from

[3]Given that graphlets are vertex-induced subgraphs, we use a vertex set to represent the (unique) embedded graphlet having those vertices here.

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

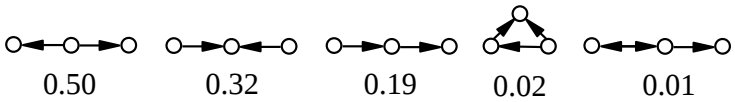

Fig. 3: The 5 3-graphlet types that have a greater-than-2% frequency density in the GFD of at least one app in our experiment, sorted by their average frequency density across all malware/benign app samples in our experiment.

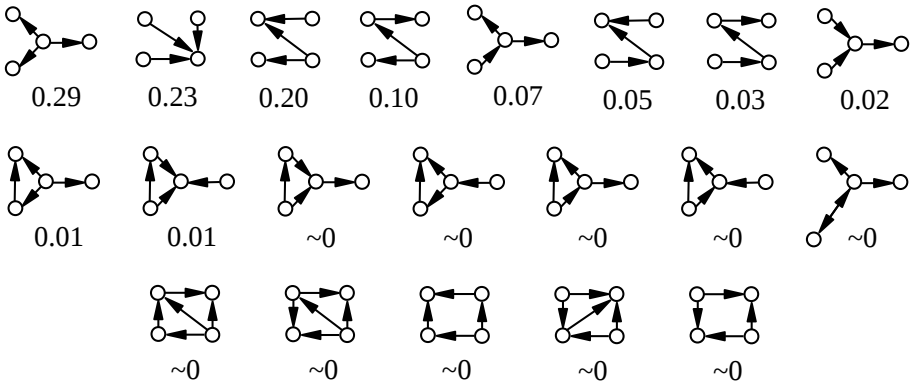

Fig. 4: The 20 4-graphlet types that have a greater-than-2% frequency density in the GFD of at least one app in our experiment, sorted by their average frequency density across all malware/benign app samples in our experiment.

Algorithm 1 are of $9,576$ ($13 + 199 + 9,364$; Section II-B) dimensions. Reducing the dimensions of these vectors is desirable.

Fortunately, as briefly discussed in Section II-B, not all graphlet types are equally likely to appear in a real FCG. Figures 3 and 4 show all 3-graphlet and 4-graphlet types (5-graphlet types are omitted for space constraints) that have more a greater-than-2% frequency density in the GFD of *at least one* of the (more than $1,400$) apps (including both malware and benign apps) in our experiment: There are 5 3-graphlet types, 20 4-graphlet types, and 71 5-graphlet types, respectively.

Note that, as we discuss in Section II-B and is verified here, graphlet types $\omega_{3,5}$ (outgoing invocations) and $\omega_{3,6}$ (incoming invocations) rank among the most frequent 3-graphlet types, while the mutually recursive type ($\omega_{3,13}$) is not. Moreover, except for a few cases of mutual recursion, loops among a few functions of are rare. This suggests that: 1) either inter-function loops have a long chain of invocations, 2) or most functions have a clear invocator-invocatee relationship that is not reciprocal.

These observations suggest that we can significantly cut down the dimensions of GFDs by projecting the GFD vectors onto the *most frequent dimensions*. Indeed, this is what we do in our method after obtaining the full-spectrum (i.e., $9,576$-dimensional) GFD estimation.

IV. Experiment Results

A. Datasets

In our experiment, we use the benign app samples from PlayDrone [29] and use the malware samples from the Android Malware Genome Project (AMGP) [36].

For the benign app portion of our datasets, we download the dataset of PlayDrone. There are total 49000 benign samples in 9 different archives. To test the scalability and robust of our algorithm, we randomly and repeatedly choose sets from the PlayDrone and each set has thousands of benign samples. We also check the package name, the version code and the $MD5$ message of each sample to prevent the duplicate in it.

For the malware portion of our datasets, the AMGP lists $1,249$ malware samples of 49 families. The top 9 malware families that have over 40 samples are: DroidKungFu3 (303 samples), AnserverBot (185 samples), BaseBridge (118 samples), DroidKungFu4 (96 samples), Geinimi (69 samples), Pjapps (56 samples), KMin (52 samples), GoldDream (47 samples), and DroidDreamLight (46 samples).

B. Procedure

We first use Androguard [15] Android app reverse engineering toolkit to extract FCGs from the APK samples. Specifically, we use the androgexf.py script to extract a GEXF[4]-format file that encodes the Java methods and their invocation relations in the APK.

We implement our GFD estimation algorithm (Algorithm 1) to generate a GFD vector for all n-graphlet types for $n \in \{3, 4, 5\}$. The majority of dimensions have a frequency of 0; hence, we use the FCG-specific GFD dimension reduction heuristics (Section III-C) to reduce these $9,576$-dimensional vectors to 96-dimensional ones, which only contain the dimensions that have a frequency density over 2% in at least one of the apps in our datasets. These 96-dimensional vectors are the topological signatures of their corresponding apps.

We then use the LIBSVM [5] support vector machine (SVM) library for classification; the details are mentioned below along with corresponding results.

C. Results

[4]GEXF (Graph Exchange XML Format); http://gexf.net/format/.

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

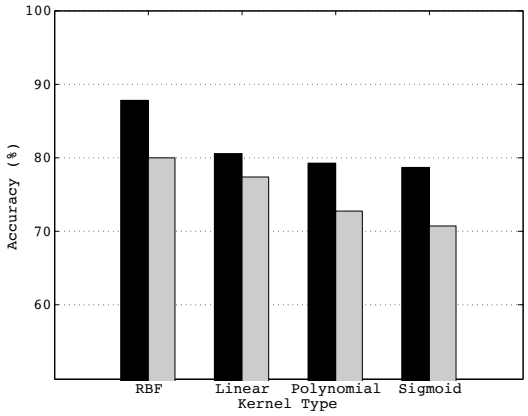

Fig. 5: Malware detection accuracy of SVM-GFD (SVMs with GFD-based signature; dark) and SVM-DFD (SVMs with DFD-based signature; grey) using C-SVC (C-support vector classification) SVMs (support vector machines) with different kernels: RBF (radial basis function), linear, polynomial, and sigmoid.

TABLE I: Malware detection false positives (FPs) and false negatives (FNs): SVM-GFD vs. SVM-DFD with different kernels.

	RBF		linear	
	FP	FN	FP	FN
GFD	11.53%	12.78%	19.30%	19.55%
DFD	13.03%	27.07%	17.54%	27.82%
	polynomial		sigmoid	
	FP	FN	FP	FN
GFD	20.80%	20.55%	22.01%	20.55%
DFD	21.30%	33.08%	26.57%	32.08%

1) Malware detection performance: To understand how the local-topology-preservation property of GFD helps in enhancing malware detection performance, we compare our method with another method in which both the (preceding) FCG extraction phase and (subsequent) learning phase are the same. The only difference is the feature we extract from FCG. Specifically, we use the degree frequency distribution (DFD) for comparison. In DFD, vertices with the same degree frequencies are binned together and counted. DFD is the probability distributions of element counts over these bins. In other words, the only difference between the two methods is whether local topology information of FCG is used in the subsequent learning phase: Our GFD-based method uses this information, while the DFD- based method does not.

For reasons that will be explained shortly, in this experiment, we randomly and repeatedly pick 1200 samples from the benign dataset to compare with the 1200 malware samples. In each comparison, we use the 10-fold cross-verification method, which means that each time 120 benign samples and 120 malware samples are randomly chosen as test set, other samples will be feed as training set and the result shows the overall average accuracy. Then we compare malware detection performance of SVMs with GFD-based signature (SVM-GFD) and SVMs with DFD-based signature (SVM-DFD) using all 4 built-in SVM kernel functions in LIBSVM: RBF (radial basis function: $e^{\gamma|u-v|^2}$), linear ($u' \cdot v$), polynomial ($(\gamma u' \cdot v)^3$), and sigmoid ($\tanh(\gamma u' \cdot v)$), in which u and v are feature vectors, $\gamma = 1/N$, and N is the feature vector dimension. Figure 5 shows the accuracy (the samples that are correctly labeled by the SVMs) comparison and Table I shows the detailed false positives/negatives (the samples that are incorrectly labeled by the SVMs). We do observe similar results on the repeated experiments but we just choose to report one due to the space constraint.

The reason we use a 1:1 ratio between malware and benign app dataset is that a skewed dataset may give misleading performance results. Later in this part we will also present the influence of sample bias. In both Figure 5 and Table I, the performance of SVM-GFD and SVM-DFD appear to be consistent across learning kernels. The high accuracy of the two algorithms implies that both of them could successfully capture topological features and the information is helpful to Android malware detection.

Comparing these two algorithms, SVM-GFD always give better results (by average 6% margin over the SVM-DFD algorithm, to over 80% accuracy). A recent study [2] on commercial anti-virus scanners' (AntiVir, AVG, BitDefender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda, Sophos) performance on the AMGP dataset shows that, except for two outliers (23.68% and 1.12%), the commercial AV scanners have accuracy ranging from 84.23% to 98.90%. SVM-GFD attains a comparable accuracy of 87.85% on the full AMGP dataset using only the structural features without any semantic augmentation.

Figure 5 suggests that RBF kernel could give a better result than other three kernels both for SVM-GFD and SVM-DFD. SVM-GFD could perform a 78% or higher results on different kernels, while SVM-DFD show 70% accuracy when choosing polynomial or sigmoid kernel. So the SVM-GFD seems more robust than SVM-DFD. Table I shows that they have different performance among false positives (FP) and false negatives (FN). Because the dataset is 1:1 ratio, FP and FN achieving a nearly 1:1 ratio means the SVM could successfully divide the hyperplane. From Table I we can see that these two SVM methods tend to give high accuracy under the specified circumstances. And SVM-GFD often have a same FP or FN percentage as SVM-DFD while the other is much better. Also there is a trade-off between FP and FN. Taking the result of SVM-GFD with linear kernel as an example, it has the slightly higher FP than the SVM-DFD while the FP is relative low. In other words, comparing with SVM-DFD, SVM-GFD with linear kernel is an aggressive malware detector that misses less malware at the cost of flagging more benign apps as malicious. The mechanism behind this calls for further research.

2) Malware family labeling accuracy: To further understand the significance of capturing local topology in FCG for malware detection, we compare our SVM-GFD together with the SVM-DFD in their *malware family labeling accuracy* on the 8 malware families that have over 40 samples in the AMGP dataset (Section IV-A). Specifically, we take the family labels on the malware samples in the AMGP dataset as the ground truth, and compare the two methods' accuracy in assigning the

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Pair-wise malware family label accuracy (in percentage) of SVM-GFD (GFD) vs. SVM-DFD (DFD) with the linear kernel of the 8 malware families that have over 40 samples in the AMGP dataset: DroidKungFu3 (DKF3; 303 samples) AnserverBot (AB; 185 samples), BaseBridge (BB; 118 samples), DroidKungFu4 (DKF4; 96 samples), Pjapps (P; 56 samples), KMin (KM; 52 samples), GoldDream (GD; 47 samples), and DroidDreamLight (DDL; 46 samples). Since this matrix is symmetric, we only show the upper half of it.

	DKF3		AB		BB		DKF4		P		KM		GD		DDL		Benign	
	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD	GFD	DFD
DKF3	-	-	92.6	60.09	71.63	67.77	75.94	71.08	84.40	77.38	85.35	78.08	86.82	78.96	86.82	79.14	76.73	71.78
AB	-	-	-	-	76.14	58.29	84.04	62.03	80.58	69.46	94.54	70.30	80.17	71.38	80.17	71.60	84.86	81.62
BB	-	-	-	-	-	-	77.78	53.90	68.18	61.94	83.72	62.88	72.29	64.09	72.29	64.34	81.25	56.25
DFK4	-	-	-	-	-	-	-	-	63.15	58.20	87.16	59.17	67.61	60.43	67.61	60.63	71.88	53.13
P	-	-	-	-	-	-	-	-	-	-	76.85	51.25	54.90	52.38	54.90	52.58	81.58	76.32
KM	-	-	-	-	-	-	-	-	-	-	-	-	89.80	51.31	86.73	51.58	72.12	60.58
GD	-	-	-	-	-	-	-	-	-	-	-	-	-	-	57.61	52.97	73.40	63.83
DDL	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	75.00	67.39
Benign	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

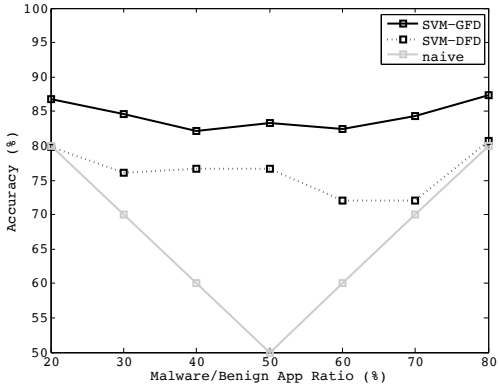

Fig. 6: Accuracy response to different malware/benign-app ratios: SVM-GFD (full line) vs SVM-DFD (dotted line) vs the naive strategy. Percentage on the x axis is the ratio of malware over benign apps in the dataset; y axis is the malware detection accuracy.

correct family labels for the test data sets. We also compare each family with a dynamic benign dataset that has the same number of samples as the malware family to show the accuracy of malware detection in one certain family. Table II shows the pair-wise malware family labeling accuracy of SVM-GFD vs. SVM-DFD.

SVM-GFD outperforms SVM-DFD in all pairs of malware families by a margin from 2.32% (P/Pjapps vs. DDL/DroidDreamLight) to 38.49% (KM/KMin vs. GD/GoldDream). The malware and benign software classification result in each family also shows SVM-GFD could achieve 3.24% (AB/AnserverBot vs. Benign) to 25% (BB/BaseBridge vs. Benign) higher performance. Note again, the additional local topological information on FCG captured by GFD, alone, takes the credit for this improvement in accuracy.

Given that we accept the manual labels as the ground truth, malware family labeling accuracy can be interpreted as a measure of how close two malware families are due to, for example, code sharing. For instance, in Table II, on the row of DKF3/DroidKungFu3, DKF4/DroiKungFu4 has a low accuracy (75.94%). This lower labeling accuracy may derive from the higher similarity between DKF4 to DKF3 due to their common lineage in the DroidKungFu mega-family.

3) Performance against sample bias: In Section IV-C1, we mention the peril of sample bias: If the ratio between positive and negative samples (i.e., benign app and malware samples) is skewed, even a naive strategy can give a misleadingly high accuracy without actually identifying malware from benign apps. In real-world malware detection, positive/negative samples rarely comes in evenly: It is highly likely we have to work with a skewed dataset.

Therefore, we study how SVM-GFD responds to sample bias. In order to avoid the influence of the dataset's size, we first fix the total number of benign and malicious softwares to 1000. Then we perturb the ratio between malware and benign app samples, and study the accuracy response of SVM-GFD/SVM-DFD with the linear kernel. The 10-fold cross-validation method is also employed in this experiment. Figure 6 shows the results and indicates that SVM-GFD get higher accuracy among all kinds of malware and benign software combination. SVM-GFD has a variance of 4.1 while SVM-DFD has a variance of 11.4. We conclude that SVM-GFD is more robust than SVM-DFD against sample bias, especially when malware or benign software accounts a small proportion. When the ration between malware and benign software is 2:8, as mentioned above it is a common real-world situation, SVM-GFD outperforms 7% accuracy but SVM-DFD is just the same as the naive strategy.

4) Most frequent graphlets: To understand why malware detection accuracy improves only by replacing DFD with GFD, we study the most frequent graphlets that appear in benign apps and in malware. Figures 7 and 8 show the top 5 most frequent graphlet types for all benign app and malware samples in our datasets, respectively. "Most frequent" in this case means that these graphlet types have the highest average GFD densities in that category (benign app or malware).

It is interesting to note that, in addition to different average density values, the types of the most frequent graphlets are different. For example, while $\omega_{3,5}$ (outgoing invocations; Figure 1) ranks the first and $w_{3,6}$ (incoming invocations) ranks the third for malware, $\omega_{3,5}$ ranks the third and $\omega_{3,6}$ ranks the first for benign apps. In both cases, these two graphlet types have a graphlet frequency density gap of 0.1 or more between them. And it also happens when a function invokes/is invoked by 3 or more other functions. This suggests that incoming invocations to a same function is more frequent than outgoing

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

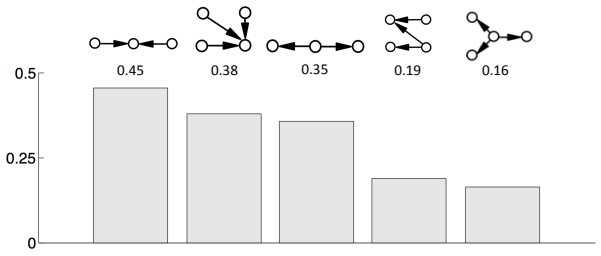

Fig. 7: The top 5 most frequent graphlet types for benign apps, i.e., the ones that have the highest average graphlet frequency densities across all benign apps.

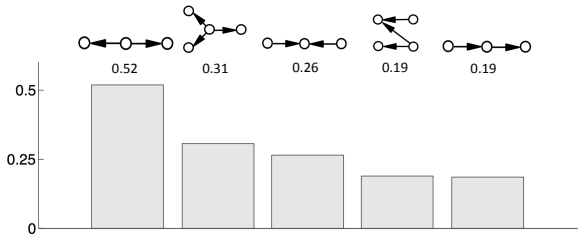

Fig. 8: The top 5 most frequent graphlet types for malware, i.e., the ones that have the highest average graphlet frequency densities across all malware.

invocations from a single function in benign apps, while the reverse is true for malware. The mechanism behind this calls for further research.

5) GFD estimation efficiency: In our experiment on a desktop workstation (8-core Intel Core i7-3820 CPU at 3.60GHz with 12GB RAM) with $100,000$ sampling iterations (at which point, the GFD estimation has already converged), our GFD estimation algorithm (Algorithm 1) takes less than 3 seconds to complete for many apps whose FCGs have less than $1,000$ vertices. For apps whose FCGs have less than $20,000$ vertices, GFD estimation takes an average of less than 10 seconds. For the most complex app in our data set, Facebook, which has $47,539$ vertices and $77,900$ edges, GFD estimation takes about 34 seconds on average with about 2 seconds variance. While the GFD estimation just takes seconds of work to analyze each single app, the total calculation time mainly depends on the size of the dataset. Because each apps and their FCGs are independent, the topological features extraction work is absolutely convenient for distributed computing system. Analyzing single extraction work, we note that GFD estimation is dominated by the generation of 1-hop neighborhood on $\mathcal{G}_G$ and the graphlet-type identification, which are independent to the size of the graph unless the graph is dense.

By contrast, the DFD calculation needs to traverse every edge and employ a sorting algorithm to the vertices. So it takes more time to do the DFD calculation especially on the complex networks. For instance, DFD calculation takes about 41 seconds for the Facebook application, 7 seconds longer than the GFD estimation. Therefore, GFD estimation, and hence ACTS, is practically efficient and accurate (Section I).

V. Further Discussion

In order to verify the effectiveness of the graphlet-based analysis and to better understand why the topological features used in ACTS could result in good performance of benign/malicious software classification, we conducted a few case studies using dynamic analysis that based on semantic features [25].

In detail, we obtain the critical API calls with the help of online analysis tools, such as Andrubis and SanDroid. These critical calls are represented as edges in the FCG. And if a function invokes one or more times of the critical API calls, we label the mapping vertex as a critical vertex. Instead of taking the full FCG graph into account, now we can just focus on the graphlets that contain the critical vertices.

Our experiment were taken on four APK files randomly chosen from four different malware families, TapSnake [19], SndApps [14], NickySpy [12] and LoveTrap [16]. The result shows that for each particular malware, its top-2 graphlets with critical vertices are always the same as the top-2 graphlets in GFD generated by ACTS. And obviously, they are different from the top-2 graphlets generated from the benign softwares. It implies that the most frequent graphlets of malware generated by ACTS in Section IV-C4 always contain the critical API calls. ACTS catches the critical API calls by counting the graphlet distribution, which uses a different route from dynamic analysis but achieves the similar result in malware detection.

We also in-depth analyzed one application *com.typ3studios.airhorn* in the malware family SndApps [14]. There are just four critical graphlets that were obtained through dynamic analysis tools. After embedding the 3-node graphlets in 4&5-node graphlets, we find that there are only 2 kinds of 3-node graphlets that contain the critical API calls, $\omega_{3,5}$ and $\omega_{3,1}$ in Figure 1, while the possible 3-node graphlets has 13 types. Also, $\omega_{3,5}$ (outgoing invocations) is included but $w_{3,6}$ (incoming invocations) is not. It supports the result in Figure 8 of Section IV-C4 that outgoing invocations to a same function is more frequent than incoming invocations from a single function in malware.

In the future, we plan to firmly combine ACTS with the dynamic analysis methods. Both the graphlet frequency and the semantic features will be analyzed to reveal the hidden mechanisms of malware.

VI. Related Works

The present work follows a line of recent works [1, 2, 10, 17, 30, 33] that apply advances in machine learning and data mining for Android malware detection. One main focus is on extracting learning features at the different app representation levels: Droid Analytics [33] focuses on the low-level platform Dalvik opcode level; Gascon et al. study function call graphs [10]; DroidAPIMiner [1] extracts features from Android API calls; Drebin [2] extracts string features from multiple Android-specific sources, e.g., intent/permission requests, API calls, network addresses. Martinelli et al. for-

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.

mulates the malware detection problem as a subgraph mining problem[17].

Pržulj et al. first propose and coin the term graphlet [20]. Two recent advances on graph mining, GRAFT [21] and GUISE [22], inspire our use of GFD as a robust and efficient topological signature for apps.

A related problem to malware detection is app repackaging, in which an app is transformed for a similar but different app through repackaging [34]. Repackaged apps are often seen on alternative Android app market, and is a major vector for carrying and propagating malware. Zhou et al. propose a system called AppInk that applies watermarking to prevent app repackaging [35].

Tainting analysis (e.g., TaintDroid [8] and FlowDroid [3, 9]) and Android app analysis frameworks (e.g., DroidScope [31] and CopperDroid [23]) can be used to further analyze malware families identified by ACTS.

VII. Conclusion

In this paper, we propose GFD as a feature for Android malware detection and adapt recent advances in graph mining to make GFD estimation robust and efficient. We demonstrate that local topological information (captured by graphlets) is attributed to improvement in malware detection accuracy and efficiency. This provides a new angle to Android malware detection research, and suggests that finding structural features (e.g., graphlets) on a graphical representation of Android apps (e.g., the FCG) that situates between local and global scope as a fertile ground for future research.

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. DroidAPIMiner: Mining API-level features for robust malware detection in android. In *Security and Privacy in Communication Networks*, pages 86–103. Springer, 2013.

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Proc. of ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[3] Steven Arzt, Siegfried Rasthofer, E Bodden, A Bartel, J Klein, Y Le Traon, D Octeau, and P McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[4] Stephen P Borgatti and Martin G Everett. A graph-theoretic perspective on centrality. *Social networks*, 28(4):466–484, 2006.

[5] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[6] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. of ACM Symposium on Theory of Computing (STOC)*, 1971.

[7] SN Dorogovtsev, JFF Mendes, and AN Samukhin. Size-dependent degree distribution of a scale-free growing network. *Physical Review E*, 63(6):062101, 2001.

[8] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[9] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android applications. Technical report, TU Darmstadt, 2013.

[10] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *Proc. of ACM Workshop on Artificial Intelligence and Security (AISec)*, 2013.

[11] Walter R Gilks, Sylvia Richardson, and David J Spiegelhalter. Introducing Markov chain Monte Carlo. In *Markov chain Monte Carlo in practice*, pages 1–19. Springer, 1996.

[12] Josh Grunzweig. Nickyspy. https://www.trustwave.com/Resources/SpiderLabs-Blog/NickiSpy-C---Android-Malware-Analysis--Demo/, Oct. 2011.

[13] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proc of ACM Symposium on Theory of Computing (STOC)*, 1998.

[14] Xuxian Jiang. Sndapps. http://www.csc.ncsu.edu/faculty/jiang/SndApps/, July 2011.

[15] Antiy Labs. androguard. https://code.google.com/p/androguard/, 2014.

[16] Yi Li. Lovetrap. https://www.symantec.com/security_response/writeup.jsp?docid=2011-072806-2905-99&tabid=2, July 2011.

[17] Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Classifying Android malware through subgraph mining. In *Data Privacy Management and Autonomous Spontaneous Security*. Springer, 2014.

[18] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6): 1087–1092, 1953.

[19] Satnam Narang. Tapsnake. http://www.symantec.com/connect/blogs/android-tapsnake-mobile-scareware-ads-push-antivirus, Dec 2013.

[20] N Pržulj, Derek G Corneil, and Igor Jurisica. Modeling interactome: scale-free or geometric? *Bioinformatics*, 20(18):3508–3515, 2004.

[21] Mahmudur Rahman, Mansurul Bhuiyan, and Mohammad Al Hasan. GRAFT: An approximate graphlet counting algorithm for large graph analysis. In *Proc. of ACM International Conference on Information and Knowledge Management (CIKM)*, 2012.

[22] Mahmudur Rahman, Mansurul Alam Bhuiyan, Mahmuda Rahman, and Mohammad Al Hasan. GUISE: a uniform sampler for constructing frequency histogram of graphlets. *Knowledge and information systems*, 38(3):511–536, 2014.

[23] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *Proc. of European Workshop on System Security (EuroSec)*, 2013.

[24] P Ribeiro. *Efficient and Scalable Algorithms for Network Motifs Discovery*. PhD thesis, University of Porto, 2011.

[25] Juan Rodriguez. Linking static and dynamic android malware analysis through graph mining.

[26] Saikoa. Dexguard. https://www.saikoa.com/dexguard, 2014.

[27] Saikoa. Proguard. http://proguard.sourceforge.net/, 2014.

[28] Hossein Azari Soufiani and Edoardo M Airoldi. Graphlet decomposition of a weighted network. *arXiv preprint arXiv:1203.2821*, 2012.

[29] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233. ACM, 2014.

[30] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2014.

[31] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In *Proc. of USENIX Security*, 2012.

[32] Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *Proc. of IEEE International Conference on Data Mining (ICDM)*, 2002.

[33] Min Zheng, Mingshen Sun, and John Lui. Droid Analytics: A signature based analytic system to collect, extract, analyze and associate Android malware. In *Proc. of IEEE Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2013.

[34] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proc. of ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[35] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk: watermarking Android apps for repackaging deterrence. In *Proc. of ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS)*, 2013.

[36] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*, 2012.

Authorized licensed use limited to: UNIVERSITY OF OREGON. Downloaded on October 07,2020 at 18:21:36 UTC from IEEE Xplore. Restrictions apply.