A Graph Theoretic Approach to Fast and Accurate Malware Detection

Zubair Shafiq[†] and Alex Liu[‡]
[†]Department of Computer Science, The University of Iowa.
[‡]Department of Computer Science and Engineering, Michigan State University.
Email: zubair-shafiq@uiowa.edu, alexliu@cse.msu.edu

***Abstract*—Due to the unavailability of signatures for previously unknown malware, non-signature malware detection schemes typically rely on analyzing program behavior. Prior behavior based non-signature malware detection schemes are either easily evadable by obfuscation or are very inefficient in terms of storage space and detection time. In this paper, we propose GZero, a graph theoretic approach fast and accurate non-signature malware detection at end hosts. GZero it is effective while being efficient in terms of both storage space and detection time. We conducted experiments on a large set of both benign software and malware. Our results show that GZero achieves more than 99% detection rate and a false positive rate of less than 1%, with less than 1 second of average scan time per program and is relatively robust to obfuscation attacks. Due to its low overheads, GZero can complement existing malware detection solutions at end hosts.**

I. Introduction

A. Motivation

According to the threat report published by PandaLabs, a record $73,000$ new malware programs, on average, were released daily [1]. A recent study of vulnerabilities databases has shown that approximately 90% of software vulnerabilities were exploited on the disclosure day by malware [19]. Malware detection is difficult because the signature of new (or previously unknown) malware is not available at time of malware launch. Malware detection has to focus on end hosts because network based security appliances, such as firewalls and intrusion detection and prevention systems, mostly rely on malware signatures and signature based malware detection schemes can hardly detect new malware. Existing commercial anti-virus software available for end hosts primarily rely on signature based malware detection, which is ineffective. Recent Conficker and Flashback malware outbreaks have provided further evidence of the infectiveness of commercial anti-virus software against new malware attacks [18].

A non-signature malware detection scheme at end hosts needs to satisfy four requirements: *high detection rate, low false positive rate, low processing and storage complexity, and robustness to malware obfuscation*. High detection rate is critical because even a single undetected malware program can infect the host and even disable the malware detection program. Low false positive rate is desirable because false alarms are nuisance to users and often cause users to simply disable the malware detection program. Low processing and storage complexity is important because end hosts have limited processing and storage resources. Robustness to malware obfuscation is crucial because malware often obfuscate itself when it propagates from one machine to another.

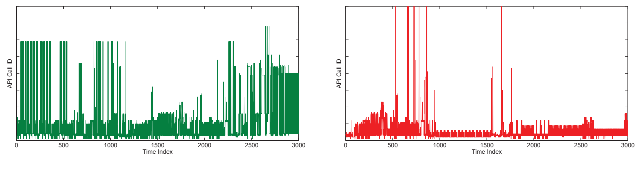
Because signatures of previously unknown malware are not available, non-signature malware detection has to rely on program behavior. Prior behavior based non-signature malware detection schemes fall into two categories: sequence based schemes and graph based schemes. Sequence based schemes identify subsequences in instruction sequences of programs whose presence or absence can be used as a binary footprint for malware detection (*e.g.*, [2], [14]). Sequence based schemes are efficient, but are vulnerable to obfuscation such as garbage insertion and reordering. Most graph based schemes store a large number of behavior graphs of known malware, and for each given program, search the behavior graph against the database to find similar graphs (*e.g.*, [9], [10], [23]). If there are known malware behavior graphs that are similar to the behavior graph of the given program, then the given program is classified as malware; otherwise, it is classified as benign software. Such graph based schemes are more robust to malware obfuscation, but they are inefficient in terms of both processing speed and storage overheads. Searching a graph database for graphs similar to a given graph is computationally expensive.

B. Proposed Approach

In this paper, we propose GZero, a graph theoretic approach to accurate, efficient, and robust non-signature malware detection at end hosts. We use a behavior graph called an *API call* (or *system call* in Linux terminology) *graph* [5]. Given the API call sequence of a program, we construct an API call graph where a vertex represents a unique API call and there is an edge from v_1 to v_2 if and only if the API call sequence contains a subsequence of two API calls v_1 followed by v_2. *Our key insight is that API call graphs of benign and malware programs have different graph theoretic properties that can be leveraged to distinguish them*. To illustrate this insight, Figure 1 and 2 show timeseries and the radial layouts of the API call graphs of a benign program and a malware program. We visually observe interesting patterns in the timeseries and behavior graph of benign and malware programs. In the timeseries plot shown in Figure 1, we observe repetitive call subsequence blocks in timeseries of both benign and malware programs.

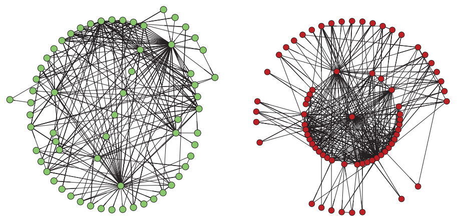
ISBN 978-3-901882-94-4 © 2017 IFIP

However, the sizes of blocks are significantly smaller for the example benign program as compared to the example malware program. In the radial layout shown in Figure 2, we randomly choose a call as a center vertex and the remaining vertices are put in concentric circles based on the distance from the center vertex. A visual comparison of the two benign and malware API call graphs highlights interesting differences. For instance, we note that the degree distribution of the API call graph of the malware program is significantly more skewed compared to the API call graph of the benign program. In addition, we observe significantly deeper branches of vertices in the API call graph of the example benign program compared to that of the example malware program.



(a) Benign (b) Malware

Fig. 1. Time series of example benign and malware programs.



(a) Benign (b) Malware

Fig. 2. API call graph of example benign and malware programs.

The basic statistics of the benign programs and malware programs that we collected also support the above insight. For example, the average entropy of API call distributions of benign programs, which is 0.92, is significantly larger than that of malware programs, which is 0.57. Intuitively these are because benign programs tend to have a larger variety of functionalities than malware programs. Furthermore, the timeseries of benign and malware programs also support the above insight. Figures 1 (a) and (b) show the time series plot of an example benign program and that of an example malware program, respectively. The API call IDs are assigned based on the alphabetical order of their names From these two figures, we observe that the sizes of the repetitive call subsequence blocks of benign programs are significantly larger than those of malware programs.

The key idea of GZero is to use a classification model based on graph theoretic features for classifying a given program to be either benign or malware. To characterize the graph theoretic properties of API call graphs, GZero extracts features at three levels: (1) vertex level, (2) sub-graph level, and (3) graph level. Vertex level features include degree, path, and connectivity features. Examples of these three types of features respectively are degree, diameter, and clustering coefficient. At the sub-graph level, we identify and extract features based on the Markov chain model of API call sequences. We use the typicality of Markov chain states to identify API call sequences of varying lengths. Typicality is useful for efficiently identifying a small subset of sequences from a very large sample space. At the graph level, we extract the features of clique numbers, average clustering coefficient, diameter, and average path length. With these extracted features, we build a Bayesian classifier for efficient malware detection.

The architecture of our approach is shown in Figure 3. In the graph construction module, we construct a graph based on the API call sequence of an unknown executable program. In the feature extraction module, we extracts features from the constructed graph at three levels: vertex level, sub-graph level, and graph level. These features act as footprints of the behavior of an executable program and are leveraged in the detection module to differentiate between benign and malware programs. In the graph classification module, we use the Bayesian classifier trained using a set of known malware and a set of known benign software to classify the unknown program to be benign or malware.

C. *Comparison to Prior Art*

GZero fundamentally differs from prior sequence based non-signature malware detection schemes in that GZero uses *features* extracted from API call *graphs* whereas those prior sequence based schemes use *signatures* extracted from API call *sequences*. Prior sequence based malware detection schemes are vulnerable to malware obfuscation [16], [21].

GZero differs from many prior graph based malware detection schemes in that GZero builds a classifier to classify the API call graph of a program to be benign or malware whereas those graph based schemes match the API call graph of a program against a database of a large number of malware API call graphs to see whether there are similar API call graphs. We show that GZero is efficient in terms of both storage space and detection time. From the storage space perspective, GZero is more efficient than prior graph based schemes because GZero does not require to store any API call graph database. From the detection time perspective, classifying the API call graph of an unknown program using GZero's classifier is more efficient than searching a database of many graphs for ones that are similar to the API call graph.

At a high-level, GZero distills some information from a pool of known malware and then use this information to classify a given program as benign or malware. Given behavior graphs of known benign and malware programs, GZero extracts graph theoretic features from the behavior graphs of both sets of programs and use these features to build a classification model. Each program has one behavior graph where each vertex represents a distinct API call and there is a directed edge from vertex v_1 to v_2 if and only if they correspond to two consecutive API calls in the API call sequence of the program. GZero extracts graph theoretic features from its behavior graph and feeds them to its classification model for classifying the program as benign or malware. Our experimental evaluation shows that GZero uses less than one second for both behavior graph extraction and classification. Our results show that GZero achieves more than 99% detection rate and

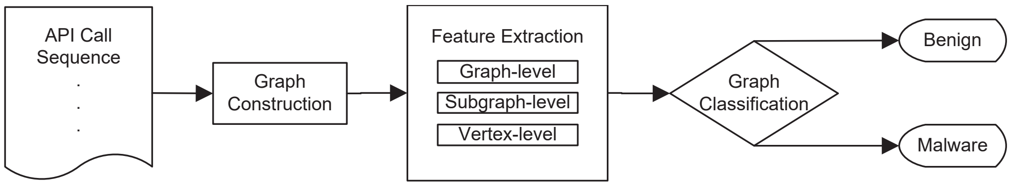


Fig. 3. Architecture of GZero

a false positive rate of less than 1%. GZero achieves high accuracy, low false alarm rate, and robustness to obfuscation because the features extracted from multiple levels of API call graphs contain highly discriminative information. GZero achieves low processing complexity because it uses a Bayesian classifier, which is highly efficient. GZero achieves low storage complexity because it only stores the trained classifier model in the order of a few kilobytes.

D. Key Contributions

We make three key contributions in this paper.

- First, we propose a API call graphs based classification model for classifying a given program into benign or malware categories.
- Second, we propose a rich and discriminative set of graph based features for classification. Using Markov chains to model API call sequences and then extracting features from the model is a highlight.
- Finally, we conducted experiments on a large set of both benign and malware programs. Our results show that GZero achieved detection rate of more than 99% and false alarm rate of less than 1%, with less than one second of average scan time per program. Furthermore, GZero's accuracy degrades slowly for increasing number of obfuscations.

II. Related Work

Previous non-signature malware detection schemes fall into two categories: sequence based and graph based.

A. Sequence Based Malware Detection

In [2], Ahmed *et al.* proposed a runtime malware analysis and detection tool that detects malware using spatio-temporal information in API call logs. The spatial information (from arguments and return values of API calls) and the temporal information (from sequence of API calls) are further used to build formal models to distinguish malware from benign programs. The most discriminative spatial and temporal features of the API calls are selected based on information gain, and are then passed on to the standard machine learning and data mining classifiers using 10-fold cross validation. The proposed scheme achieves up to an approximately 98% detection accuracy. The inherent limitation of this scheme is that it is vulnerable to even simple obfuscation techniques due to the use of API call sequences as features [16]. For a malware program, a crafty malware writer can change its temporal features by manipulating the API calls sequence of the malware as well as the spatial features by inserting garbage API calls with useless arguments.

In [14], Islam *et al.* proposed a behavior based malware detection technique, which extracts strings (system call names and function arguments) from malware and benign trace logs. The feature vectors are created on the basis of the absence or presence of a string in a particular file. The scheme achieved 97.3% detection accuracy on their test data set. Similar to the above scheme in [2], this scheme is also vulnerable to even simple obfuscation techniques.

B. Graph Based Malware Detection

1) Control Flow Graph Based Schemes: In [9], Christodorescu *et al.* proposed a malware detection scheme called Static Analyzer for Executables (SAFE) to detect malicious patterns in executables using static analysis. For each known malware program, SAFE generates an annotated Control Flow Graph (CFG) from the assembly code of the program where each vertex corresponds to an assembly instruction. Given a new program, SAFE first generates its annotated CFG and then searches it against a large database of the annotated CFGs of malware programs for similar ones. SAFE requires to store a large database of malware CFGs and also has high search overhead.

In [10], Christodorescu *et al.* proposed a semantics aware malware detection scheme. For each known malware program, this scheme generates templates, which are instruction sequences described using variables and symbolic constants, from the assembly code of the program. A template describes a specific semantic behavior. Given a new program, this scheme first generates templates and then searches them against a large database of templates generated from known malware programs. The main advantage of this scheme over SAFE is that templates focus on program semantics and describe program behaviors at a higher level than annotated CFGs, which leads to better detection accuracy. Compared to GZero, this scheme shares the above limitations of SAFE.

2) Dependency Graph Based Schemes: In [15], Kolbitsch *et al.* proposed a malware detection scheme that use API call graphs. For each known malware program, this scheme generates a dependency graph where each vertex is an API call and each directed edge from vertex v_1 to v_2 if and only if the API call corresponding to v_2 has a data dependency on the API call corresponding to v_1. For each new program, this scheme first generates its dependency graph and then maps this graph against a large database of dependency graphs generated from known malware programs to find similar ones. Compared to GZero, similar to the schemes in [9], [10], this scheme also has the above mentioned two limitations.

The other scheme in this category is HOLMES proposed by Fredrikson *et al.* [13]. HOLMES uses graph mining and concept analysis algorithms to analyze a set of malicious and benign programs, extracts significant malicious and benign behaviors, and creates optimally discriminative specifications.

We have given the detailed comparison between GZero and HOLMES in Section I.

C. *Other Related Work*

Observing the dependency graphs generated by [13] are too big, Chen *et al.* proposed a graph mining algorithm for generating small graphs that can be used as the summary of large graphs [8]. The focus of this work is on reducing graph sizes. In [4] and [3], Bayer *et al.* proposed an automated tool for generating human readable reports on the behavior of programs. The report is generated by tracking API calls made by a program focusing on file, registry, service, process, and network activities. The focus of this work is to facilitate malware analyst to understand the behavior of malware programs.

In [23], Yin and Song proposed a taint graph based malware detection technique called Panorama, which is based on the observation that malware (such as spyware, keyloggers, and rootkits) often accesses and processes user's private information, which is not intended for them. Panorama works by running the sample program in an emulator that contains a test engine, which runs test scripts while the sample program is running. These test scripts introduce important taint information (like password input, TCP/UDP/ICMP traffic etc). The test engine monitors the activities of the sample program and the overall system under observation. The activities or behavior of the program in a system wide context is further represented in the form of graphs, in which the vertexes represent system calls and the edges represent the data dependency between two system calls. The focus of Panorama is to facilitate malware experts and security analysts to understand malware behavior. It is designed for off-line detection and analysis of malware, whereas GZero is designed for online detection.

III. Proposed Approach

In this section, we present the details of the three key modules of GZero: graph construction, feature extraction, and graph classification.

A. *Graph Construction*

Given a sequence of API calls $\langle a_1, a_2, \cdots, a_m \rangle$ of an unknown program, we construct the API call graph as follows. For each unique API call a_i ($1 \leq i \leq m$) in the given sequence, we create a vertex denoted $V(a_i)$. For any two consecutive API calls $a_i a_{i+1}$ in the given sequence, if a_i and a_{i+1} are two unique API calls, then we create a directed edge from vertex $V(a_i)$ to vertex $V(a_{i+1})$.

Table I shows a segment of an API call sequence of a worm named as Shorm.110 and Figure 4 shows the API call graph constructed from this API call sequence. The API call trace of this malware program contains a sequence of 1145 API calls in total. This partial sequence shows 59 API calls that highlight the behavior of the malware. In this sequence, the malware program reads some file attributes and then reads and writes to the registry. Modifying the registry allows the malware to add itself to the system startup so that it is executed whenever the system is restarted. It also creates a process thread on the infected system to keep itself alive.

TABLE I
API call subsequence for Shorm.110 worm

No.	ID	API Name	No.	ID	API Name
1	162	GetFileAttributesW	31	4	HeapAlloc
2	4	HeapAlloc	32	3	HeapFree
3	3	HeapFree	33	12	LocalFree
4	48	LoadLibraryExW	34	12	LocalFree
5	4	HeapAlloc	35	12	LocalFree
6	3	HeapFree	36	4	HeapAlloc
7	10	RegOpenKeyExW	37	10	RegOpenKeyExW
8	4	HeapAlloc	38	4	HeapAlloc
9	3	HeapFree	39	3	HeapFree
10	7	RegQueryValueExW	40	10	RegOpenKeyExW
11	4	HeapAlloc	41	4	HeapAlloc
12	3	HeapFree	42	3	HeapFree
13	11	RegCloseKey	43	3	HeapFree
14	1	LocalAlloc	44	4	HeapAlloc
15	10	RegOpenKeyExW	45	10	RegOpenKeyExW
16	4	HeapAlloc	46	4	HeapAlloc
17	3	HeapFree	47	4	HeapAlloc
18	7	RegQueryValueExW	48	3	HeapFree
19	4	HeapAlloc	49	3	HeapFree
20	3	HeapFree	50	3	HeapFree
21	11	RegCloseKey	51	35	CreateProcessW
22	12	LocalFree	52	2	IsBadReadPtr
23	9	GetCurrentThread	53	2	IsBadReadPtr
24	10	RegOpenKeyExW	54	2	IsBadReadPtr
25	4	HeapAlloc	55	2	IsBadReadPtr
26	3	HeapFree	56	2	IsBadReadPtr
27	8	RegSetValueExW	57	164	GetLongPathNameW
28	4	HeapAlloc	58	4	HeapAlloc
29	3	HeapFree	59	5	IsBadWritePtr
30	11	RegCloseKey			

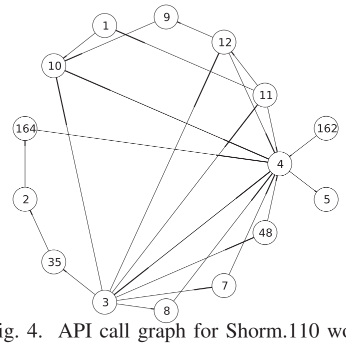


Fig. 4. API call graph for Shorm.110 worm

B. *Feature Extraction*

We characterize API call graphs using the features that capture their structural properties at different levels of granularity. More specifically, we extract graph theoretic features at three levels: vertex level, sub-graph level, and graph level.

1) Vertex Level Features: We extract three types of vertex-level features: degree, path, and connectivity. Degree features include: in-degree, out-degree, degree, and reciprocity. Path features include: betweenness centrality and closeness centrality. Connectivity features include: number of triangles, clustering coefficient, and eigenvector centrality. These features are separately computed for each vertex. Below we formally define each of the above features.

- **Degree:** The degree of a vertex is defined as the number of edges incident on it. The degree δ_i of a vertex v_i is defined as:

$$\delta_i = \left| \bigcup_{\forall j=i \vee k=i} e_{jk} \right|, \tag{1}$$

where e_{jk} denotes an inwards or outwards edge between vertices v_j and v_k.

- **In-Degree:** The degree of a vertex is defined as the number of inwards edges incident on it. The in-degree $\delta_{\downarrow i}$ of a vertex v_i is defined as follows.

$$\delta_{\downarrow i} = \left| \bigcup_{\forall k=i} e_{jk} \right| \tag{2}$$

- **Out-Degree:** The degree of a vertex is defined as the number of outwards edges incident on it. The out-degree $\delta_{\uparrow i}$ of a vertex v_i is defined as follows.

$$\delta_{\uparrow i} = \left| \bigcup_{\forall j=i} e_{jk} \right| \tag{3}$$

- **Reciprocity:** The reciprocity of a vertex is defined as the ratio of its out-degree to its in-degree. The reciprocity R_i of a vertex v_i is defined as:

$$R_i = \delta_{\uparrow i}/\delta_{\downarrow i} = \left| \bigcup_{\forall j=i} e_{jk} \right| / \left| \bigcup_{\forall k=i} e_{jk} \right|, \tag{4}$$

where e_{jk} denotes an inwards or outwards edge between vertices v_j and v_k.

- **Number of Triangles:** A triangle is defined as the subset of any three vertices in a graph that are completely connected. The triangle count $\mathit{\Delta}_i$ of a vertex v_i is defined as the number of triangles that contain the given vertex as one of their vertices. Let $\mathit{\Gamma}_i$ denote the set of vertices that a vertex v_i is connected to then its triangle count $\mathit{\Delta}_i$ is defined as:

$$\mathit{\Delta}_i = \left| \bigcup_{v_j, v_k \in \mathit{\Gamma}_i} e_{jk} \right| \tag{5}$$

- **Clustering Coefficient:** The clustering coefficient of a vertex is defined as the ratio of the number of triangles that the vertex is a part of to the total number of possible triangles. Let δ_i denote the degree of a vertex and T_i denote the number of triangles that contain the vertex. The clustering coefficient C_i is defined as:

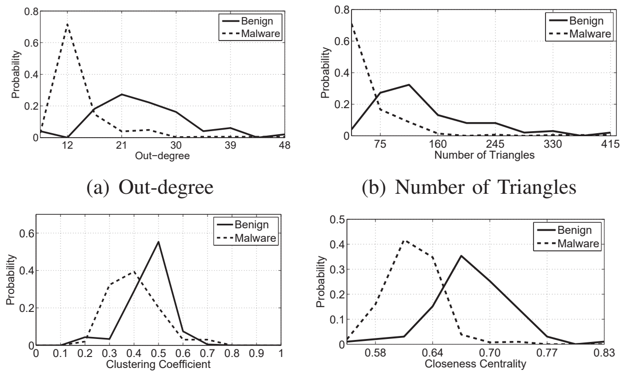
$$C_i = \frac{\mathit{\Delta}_i}{\binom{\delta_i}{2}} = \frac{2\mathit{\Delta}_i}{\delta_i(\delta_i - 1)} \tag{6}$$

- **Eigenvector Centrality:** The eigenvector centrality of a vertex is a measure of its importance in a network. Let $\mathbf{W}$ denote the adjacency matrix of the graph G where $w_{i,j}$ is 1 if an edge exists between vertices v_i and v_j, and 0 otherwise. Eigenvector centrality e_i of vertex v_i is defined as:

$$e_i = \frac{1}{\lambda} \sum_{j=1}^{N} w_{i,j} x_j \tag{7}$$

where λ is the principal eigenvalue of matrix $\mathbf{W}$.

- **Betweenness Centrality:** The betweenness centrality of a vertex is defined as the fraction of all pair shortest paths, except those originating or terminating at it, that pass through it. Let P_{jk} denote the shortest path from vertex v_j to vertex v_k, where $P_{jk} = (v_j, v_l, v_m, v_n, ..., v_k)$. The betweenness centrality b_i of a vertex v_i is defined as:



(a) Out-degree (b) Number of Triangles

(c) Clustering Coefficient (d) Closeness Centrality

Fig. 5. Distribution of some vertex level features for `LocalAlloc` API call.

$$b_i = \frac{2I(P_{jk}, i)}{|V|(|V| - 1)}, \tag{8}$$

where $I(P_{jk}, i)$ is an indicator function such that $I(P_{jk}, i) = 1$ when $v_i \in P_{jk}$ and $I(P_{jk}, i) = 0$ when $v_i \notin P_{jk}$.

- **Closeness Centrality:** The closeness centrality of a vertex is defined as the average length of shortest paths to all vertices reachable from it. Let $|P_{ij}|$ denote the shortest path length from vertex v_i to vertex v_j. The closeness centrality c_i of a vertex v_i is defined as:

$$c_i = \frac{\sum_{j=1}^{N} |P_{ij}|}{|V|} \tag{9}$$

Figure 5 shows the distribution of four example vertex level features for API call `LocalAlloc`: out-degree, number of triangles, clustering coefficient, and closeness centrality. This particular API call allocates the specified number of bytes from the heap and is important for memory management. We observe significant distinction between distributions of benign and malware programs for all of these vertex level features.

2) Sub-graph Level Features: Let $\langle a_1, a_2, \cdots, a_m \rangle$ be an API call sequence and $G = (V, E)$ be the corresponding graph that we construct from it. For any subsequence $\langle x_1, x_2, ..., x_n \rangle$, where $n \leq m$, let $P(x_1, x_2, ..., x_n)$ denote the probability of finding $\langle x_1, x_2, ..., x_n \rangle$ in $\langle a_1, a_2, \cdots, a_m \rangle$. Let k be the total number of unique API calls in $\langle a_1, a_2, \cdots, a_m \rangle$, which is equal to $|V|$. Here k also represents the sample space size of each x_i $(1 \leq i \leq n)$. The size of the sample space for subsequence $\langle x_1, x_2, ..., x_n \rangle$ is k^n. Here $P(x_1, x_2, ..., x_n)$ can also be represented as the joint probability $P(x_1 \cap x_2 ... \cap x_n)$. Using Bayes theorem, this joint probability can be defined by a combination of the conditional and marginal probabilities. Specifically,

$$P(x_1 \cap x_2 ... \cap x_n) = P(x_1 | x_2 ... \cap x_n) P(x_2 ... \cap x_n).$$

In fact, the conditional probability contains more precise information due to its reduced sample space compared to the joint probability. These conditional n-gram probabilities can be conveniently represented using a discrete time Markov chain of order $n-1$ containing k^{n-1} states. Note that each conditional n-gram sequence corresponds to a unique state in the multi-order Markov chain. Now let a binary indicator random variable $I_i, i = 1, 2, ..., k^{n-1}$ represent the presence of a state in Markov chain, where k^{n-1} is the total number of states of the Markov chain of order $n-1$. Hence, $P(I_i = 1)$ represents the probability for the presence of state X_i.

It is important to select an appropriate order for a Markov chain model. It is also important to note that we are typically interested in employing a single Markov chain to model a set of multiple sequences $\mathbb{S}$. Here let $|\mathbb{S}|$ denote the size of the set of sequences that we want to model. For each sequence, autocorrelation is a well-known heuristic for selecting appropriate order for its Markov chain model [7]. For a given lag t, the autocorrelation function of a sequence, S_m (where m is the index), is defined as:

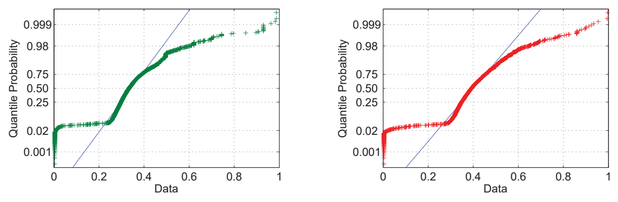
$$\rho[t] = \frac{E\{S_0 S_t\} - E\{S_0\}E\{S_t\}}{\sigma_{S_0}\sigma_{S_t}}, \tag{10}$$

where $E(S_i)$ and σ_{S_i} respectively represent the expectation and standard deviation of S at lag i. The value of the autocorrelation function lies in the range $[-1, 1]$, where $|\rho[t]| = 1$ indicates the perfect correlation at lag t and $\rho[t] = 0$ means no correlation at lag t. The minimum value of lag t_{min} for which $\rho[t_{min}]$ falls inside the 95% confidence interval band is selected to be the appropriate order for a Markov chain. For a set of multiple sequences, let $\mathbb{T}$ denote the set of selected orders as per the aforementioned criterion. We select the maximum value in $\mathbb{T}$, denoted by $\mathbb{T}_{max}$, as the order of a single Markov chain model that we want to employ.

The number of states in a Markov chain increases exponentially for higher orders and so does the complexity of the underlying model. Furthermore, higher order Markov chains require a large amount of training data to identify a subset of states that actually appear in the training data. In other words, a Markov chain model trained with limited data is typically sparse. To overcome these challenges, we combine multiple states in a higher order Markov chain to reduce its total number of states. We are essentially using states from lower order Markov chains as we combine different states in a multi-order Markov chain.

Now, we also need to establish a criterion to combine states in a multi-order Markov chain. Towards this end, we use the typicality of Markov chain states as the criterion. Typicality allows us to identify a "typical" subset of Markov chain states by generating its realizations [7]. Before delving into further details, we first state the well-known typicality theorem as follows: For any stationary and irreducible Markov process X and a constant c, the sequence $x_1, x_2, ..., x_m$ is almost surely (n, ϵ)-typical for every $n \leq c \log m$ as $m \to \infty$. A sequence $x_1, x_2, ..., x_m$ is called (n, ϵ)-typical for a Markov process X if $\hat{P}(x_1, x_2, ..., x_n) = 0$, whenever $P(x_1, x_2, ..., x_n) = 0$, and

$$\left| \frac{\hat{P}(x_1, x_2, ..., x_n)}{P(x_1, x_2, ..., x_n)} - 1 \right| < \epsilon, \text{ when } P(x_1, x_2, ..., x_n) > 0.$$



(a) Benign transition matrix (b) Malware transition matrix

Fig. 6. Q-Q probability plot of sub-graph level features.

Here $\hat{P}(x_1, x_2, ..., x_n)$ and $P(x_1, x_2, ..., x_n)$ are the empirical relative frequency and the actual probability of the sequence $x_1, x_2, ..., x_n$, respectively. In other words,

$$\hat{P}(x_1, x_2, ..., x_n) \approx P(x_1, x_2, ..., x_n).$$

This theorem shows us a way of empirically identifying "typical" sample paths of arbitrary length for a given Markov process. Based on this theorem, we generate realizations (or sample paths) of arbitrary lengths from the transition matrix of the Markov process. By generating a sufficiently large number of sample paths of a given length, we can accurately identify a relatively small subset of sample paths that are typical.

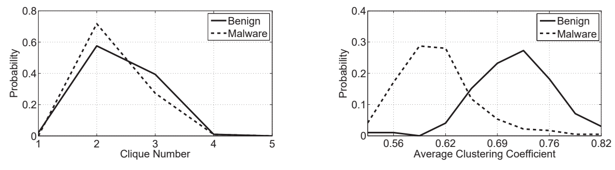
Using this methodology, we select a subset of top-10000 typical states $\mathbb{X}_{10000}$ as potential features, whose lengths vary in the range $[0, \mathbb{T}_{max}]$. To further cut down the number of sub-graph level features (up to 100), we use an information-theoretic measure called information gain to rank features [11]. Information gain is used to quantify the differentiation power of features, which are Markov chain states in our case. In this context, information gain is the mutual information between a given feature X_i and the class variable Y. For a given feature X_i and the class variable Y, the information gain of X_i with respect to Y is defined as:

$$IG(X_i; Y) = H(Y) - H(Y|X_i), \tag{11}$$

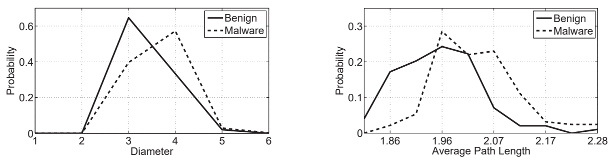
where $H(Y)$ denotes the marginal entropy of the class variable Y and $H(Y|X_i)$ represents the conditional entropy of Y given feature X_i. In other words, information gain quantifies the reduction in the uncertainty of the class variable Y given that we have the complete knowledge of feature X_i. For our application, the class variable Y is $\{\text{Benign}, \text{Malware}\}$. Using information gain, we finally select a subset of top-100 typical states $\mathbb{X}_{100}$ as features.

Recall we can identify a typical sample paths of the Markov chain model of API call graphs. We use the presence or absence of these typical sample paths as sub-graph level binary features. Here we are interested in studying the distribution of their empirically estimated probabilities. Ideally, we want the sample paths in the typical set to have high probabilities. Figure 6 show the Q-Q probability plot of sub-graph level features. We observe that most sample paths have moderate to high occurrence probabilities, which is desirable. This shows that we can successfully capture variable length Markov chain sample paths.

3) Graph Level Features: Graph level features capture the properties of the complete graph, whereas the vertex



(a) Clique Number (b) Average Clustering Coefficient



(c) Diameter (d) Average Path Length

Fig. 7. Distribution of some graph level features.

level features capture the properties of individual vertices. We capture four graph level features: clique number, average clustering coefficient, diameter, and average path length.

- **Clique Number:** A clique is a sub-graph such that all vertices in it are directly connected to each other by an edge. The clique number ω of a graph is defined as the total number of vertices in its largest clique.
- **Average Clustering Coefficient:** The average clustering coefficient $\bar{C}$ of a graph $G(V, E)$ is defined as:

$$\bar{C} = \frac{1}{|V|} \sum_{i=1}^{|V|} C_i \tag{12}$$

- **Diameter:** The diameter D of a graph $G(V, E)$ is defined as:

$$D(G) = \max_{\forall j,k}(|P_{jk}|), \tag{13}$$

where P_{jk} is the shortest path length between vertices v_j and v_k.
- **Average Path Length:** The average path length l of a graph $G(V, E)$ is defined as:

$$l(G) = \frac{\sum_{\forall j,k}(|P_{jk}|)}{|V|(|V|-1)}, \tag{14}$$

where P_{jk} is the shortest path length between vertices v_j and v_k.

Figure 7 shows the distributions of graph level features for both benign and malware programs. We observe that API call graphs of benign programs tend to have larger values of average clustering coefficient compared to malware programs. Recall from Section III that the average clustering coefficient measures the connectivity of a graph, where a larger value indicates more connectivity. This observation means that the API call graphs of benign programs are more tightly connected than those of malware programs. As tightly connected graphs have smaller average path lengths and diameters, we observe that the API call graphs of benign programs have smaller average path lengths and diameters than those of malware programs. The above observations are due to the fact that the malware programs tend to call some API sequences over and over again. For example, in the API call sequence in Table I, we observe that the subsequence `HeapAlloc` $\rightarrow$ `HeapFree` $\rightarrow$ `RegQueryValueExW` show up in the sequence many times. Actually this subsequence appears more often in the complete API call sequence for worm Shorm.110. Because such repetition does not contribute much to adding new edges, the API call graphs of malware programs are less tightly connected than those of benign programs.

C. Graph Classification

We now present our graph classification method that uses the three levels of features that we extract from API call graphs. We select Naïve Bayes algorithm as the machine learning classifier for graph classification because it is a light-weight probabilistic classifier that has been widely used for problems like text and malware classification and is also known to outperform more complex techniques in terms of accuracy [22]. We train the algorithm using two sets of probabilities: the prior, which represents the marginal probability $P(Y)$ of the class variable Y, and the a-priori conditional probabilities $P(X_i|Y)$ of the features X_i given the class variable Y. For continuous features, Naïve Bayes uses Gaussian distributions and estimates its parameters from the data to compute these probabilities [6]. These probabilities can be computed from the training set. Given an unknown API call graph with observed features X_i, $i = 1, 2, ..., n$, the *a-posteriori* probability $P(Y|X^{(n)})$ can be computed for both classes $Y \in \{\text{Benign}, \text{Malware}\}$, where $X^{(n)} = (X_1, X_2, ..., X_n)$ is the vector of observed features in the test cascade under consideration:

$$P(Y|X^{(n)}) = \frac{P(X^{(n)}, Y)}{P(X^{(n)})} = \frac{P(X^{(n)}|Y)P(Y)}{P(X^{(n)})}. \tag{15}$$

The Naïve Bayes classifier then combines the a-posteriori probabilities by assuming conditional independence among the features as follows:

$$P(X^{(n)}|Y) = \prod_{i=1}^{n} P(X_i|Y). \tag{16}$$

Although the independence assumption among features makes it feasible to evaluate the a-posteriori probabilities with much lower complexity, it is unlikely that this assumption truly holds all the time in reality. We mitigate the effect of the independence assumption by pre-processing the features using the Karhunen-Loeve Transform (KLT) [12], which is closely related to the Principal Component Analysis (PCA). KLT is a non-parametric method that transforms the features into a domain where they are perfectly uncorrelated. The feature transformation helps in reducing dependency among distinct features, which in turn improves the performance of classification algorithms.

IV. Experimental Results

In this section, we evaluate the effectiveness, efficiency, and robustness of GZero. To evaluate GZero, we collected a data set containing a total of 564 benign programs of various types (such as games, word processors, archivers, web browsers, system utilities, and file sharing softwares) and 14,215 diverse malware programs (from `http://openmalware.org/`). The number of API calls that we logged per program is on the order of tens of thousands, but the number of unique API calls is only 237.

A. *Effectiveness*

We evaluate the effectiveness of GZero using the 5-fold cross-validation process in our experiments. Specifically, we divide the data set of benign programs into 5 subsets and so does for the data set of malware programs. We reported results based on the average of 5 runs, where in each run 4 subsets are used for training and the left over subset is used for testing. To systematically evaluate the effectiveness of these feature sets in classifying benign and malware programs, we first conduct experiments using stand alone feature sets and then evaluate their all possible combinations. We characterize the classification performance in terms of false positive rate (fraction of benign programs falsely detected as malware programs) and true positive rate (fraction of malware programs correctly detected as malware programs). Both true positive and false positive rates are jointly incorporated in a single metric called Area Under ROC Curve (AUC). The AUC for an ideal classifier is 1.0. We select ROC operating point during the training phase to keep the false positive rate below 1%. An alternate measure of classification performance is precision, which is defined as the number of true positives divided by the sum of the number of true positives and false positives.

Table II shows the effectiveness of GZero using different feature sets in terms of AUC, true positive rate, false positive rate, and precision. Among stand alone feature sets, sub-graph features provide the best classification performance. It is followed by the vertex feature set and then the graph feature set. We think the sub-graph feature set outperformed the graph feature sets because it captures more detailed information than graph-level features. Moreover, the vertex feature set is relatively sparse than the sub-graph feature set. We also observe that using combinations of feature sets does improve the classification performance. The best classification performance is achieved when all feature sets are combined. Note that using all feature sets consistently provides the best classification performance in terms of all metrics compared to otherwise. Using all feature sets, GZero achieves a true positive rate, a false positive rate, and a precision of 99.6%, 0.9%, and 99.4%, respectively.

TABLE II
EFFECTIVENESS OF GZERO USING COMBINATIONS OF FEATURE SETS.

Feature Set	AUC	True Positive Rate	False Positive Rate	Precision
Graph	0.865	0.855	0.145	0.881
Vertex	0.989	0.979	0.021	0.987
Sub-graph	0.990	0.985	0.017	0.990
Vertex, Graph	0.990	0.981	0.019	0.990
Sub-graph, Graph	0.992	0.987	0.015	0.991
Vertex, Sub-graph	0.996	0.988	0.016	0.992
All	**0.997**	**0.996**	**0.009**	**0.994**

B. *Efficiency*

We conducted experiments on a desktop PC with AMD Phentom II 3.01 GHz processor and 12.0 GB DRAM. Overall, our proposed approach takes less than a second on average to scan a given program. Thus, GZero can be efficiently deployed at end hosts. The efficiency of GZero can be separately analyzed for three modules: graph construction, feature extraction, and detection. Unlike existing dependency graph based malware detection schemes that need to track data dependency, the graph construction module in GZero simply maps an API call sequence into a graph, which can be efficiently done in the order of milliseconds. The feature extraction module of GZero only extracts a handful of features from the constructed API call graph. The complexity of feature extraction is primarily dependent on the size of the API call graph in terms of number of vertices. In our evaluations, we found that all of the three feature sets takes less than half a second on average for a given program. For detection, GZero uses the Bayesian classifier instead of other more complex machine learning algorithms, such as support vector machines, because of its efficiency. We observed that it takes less than a hundredth of a second on average to classify the feature set of a given program.

C. *Robustness*

We evaluated the robustness of GZero to obfuscation attempts by an attacker. We use the following obfuscation techniques to generate the obfuscated variants of all benign and malware programs in our data set.

• **Garbage insertion:** This technique adds garbage API calls at random locations in the program trace. In effect, edges between randomly selected vertices are added to the API call graph.

• **Reordering:** This technique swaps random API calls in the program trace. In effect, random edges in the behavior graph are re-wired.

For systematic robustness analysis, we have created obfuscated versions of our data sets by performing varying number of these obfuscation operations. These obfuscation operations make API call graphs random; thus, we expect the effectiveness of GZero to degrade for increasing obfuscation operations [17], [20]. Figure 8 shows the effectiveness of GZero on obfuscated data sets. The x-axis represents the percentage of total API calls for each program that are obfuscated. For example, 50% of garbage insertion for a program trace consisting of m API calls means that $m/2$ garbage API calls were randomly inserted in the original trace. Likewise, 50% of reordering for a program trace consisting of m API calls means that $m/2$ API calls were swapped with other randomly selected calls in the original trace. Comparing garbage insertion, reordering, and their combination, we note that GZero is less robust to reordering than garbage insertion. We observe that combining garbage insertion and reordering has a minor impact as compared to standalone obfuscation techniques. This observation indicates that reordering impacts the underlying API graph structure more than garbage insertion.

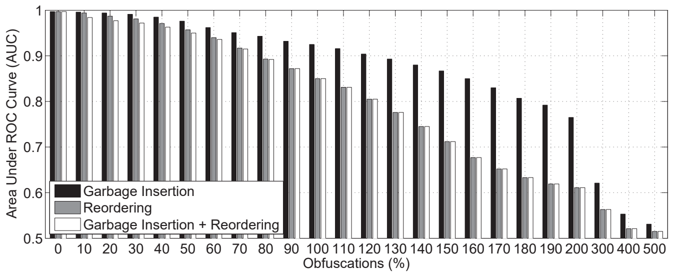


Fig. 8. Effectiveness of GZero in response to obfuscation.

We note that the degradation in GZero's effectiveness is gradual for increasing levels of obfuscation. For instance, the effectiveness of GZero at 100% obfuscation reduces AUC values of 92.6% and 85.1% for garbage insertion and reordering, respectively. At 200% obfuscation rate, the effectiveness of GZero further reduces AUC values of 76.5% and 61.1% for garbage insertion and reordering, respectively. For much more obfuscation operations (e.g., up to 500%), the AUC values near the 50% mark, which represents a random classifier. We note that at high levels of obfuscation, a program's API call graph is very much randomized and its underlying behavior is impacted. Thus, a program will likely not run and may lose its core functionality at these high levels of obfuscation.

V. Conclusion

In this paper, we propose GZero, a non-signature malware detection approach that uses a classification model based on graph theoretic features extracted from API call graphs. We gain two key insights from this study. First, benign programs and malware programs do exhibit different graph theoretic characteristics on their API call graphs. Second, the three levels of graph theoretic features that we extract from API call graphs indeed can be used to effectively and efficiently distinguish malware from benign software. Our experimental results show that GZero achieves more than 99% detection rate and a false positive rate of less than 1%. Furthermore, GZero is efficient in terms of time and storage; thus, it can complement existing malware detection solutions at end hosts due to its low overheads.

In this work, we focused on optimizing the performance of GZero in terms of its effectiveness and efficiency. We ended up using a bucket list of standard graph features, without worrying about any behavior insights for specific features. Although providing domain-specific insights into the proposed features may be useful to understanding why certain features perform as well as they do, we believe that the domain-agnostic nature of the features is actually an advantage because it confirms our earlier hypothesis that API call graphs of benign and malware programs have *different* graph theoretic properties. Thus, the research community can use GZero as a blueprint and propose additional features to improve its performance for specific design goals (e.g., efficiency vs. robustness).

Acknowledgment

Zubair Shafiq's work is partially supported by the National Science Foundation under Grant Numbers CNS-1464110, CNS-1524329, and CNS-1617288. Alex Liu's work is partially supported by the National Science Foundation under Grant Numbers CNS-1318563, CNS-1524698, and CNS-1421407, the National Natural Science Foundation of China under Grant Numbers 61472184 and 61321491, and the Jiangsu Innovation and Entrepreneurship (Shuangchuang) Program.

References

[1] Pandalabs annual report. http://press.pandasecurity.com/press-room/reports/.

[2] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq:. Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In *ACM Workshop on Security and Artificial Intelligence (AISec)*, pages 55–62, 2009.

[3] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *European Institute for Computer Antivirus Research Annual Conference (EICAR)*, 2006.

[4] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[5] D. Bilar. On callgraphs and generative mechanisms. *Journal in Computer Virology*, 3(4):285–297, 2007.

[6] R. R. Bouckaert. Naive Bayes Classifiers That Perform Well with Continuous Variables. In *17th Australian Joint Conference on Artificial Intelligence*, 2004.

[7] P. Bremaud. *Markov Chains*. Springer, 2008.

[8] C. Chen, C. Lin, M. Fredrikson, M. Christodorescu, X. Yan, and J. Han. Mining graph patterns efficiently via randomized summaries. In *ACM Conference on Very Large Data Bases*, 2009.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *12th USENIX Security Symposium*, pages 169–186, 2003.

[10] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[11] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscience, 1991.

[12] R. Dony. *The Transform and Data Compression Handbook, Chapter 1*. CRC Press, 2001.

[13] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[14] R. Islam, R. Tian, L. M. Batten, and S. Versteeg. Differentiating malware from cleanware using behavioural analysis. In *IEEE International Conference on Malicious and Unwanted Software*, pages 23–30, 2010.

[15] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *USENIX Security Symposium*, pages 351–366, 2009.

[16] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu. Shadow attacks: Automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1-2):1–13, May 2012.

[17] M. Musale, T. H. Austin, and M. Stamp. Hunting for metamorphic javascript malware. *Journal of Computer Virology and Hacking Techniques*, 2014.

[18] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, February 2009.

[19] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *34th International Conference on Software Engineering (ICSE)*, pages 771–781, 2012.

[20] S. M. Sridhara and M. Stamp. Metamorphic worm that carries its own morphing engine. *Journal of Computer Virology and Hacking Techniques*, 9(2):49–58, 2013.

[21] A. Srivastava, A. Lanzi, and J. Giffin. System call API obfuscation. In *11th International Symposium On Recent Advances In Intrusion Detection (RAID)*, volume 5230 of *Lecture Notes in Computer Science*, pages 421–422, 2008.

[22] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2011.

[23] H. Yin and D. Song. Panorama: Capturing system-wide information flow for malware detection and analysis. In *ACM conference on Computer and Communications Security (CCS)*, pages 116–127, 2007.