Extraction of Statistically Significant Malware Behaviors

Sirinda Palahan
Penn State University
sxp969@psu.edu

Domagoj Babić*
Google, Inc.
babic.domagoj@gmail.com

Swarat Chaudhuri
Rice University
swarat@rice.edu

Daniel Kifer
Penn State University
dkifer@psu.edu

ABSTRACT

Traditionally, analysis of malicious software is only a semi-automated process, often requiring a skilled human analyst. As new malware appears at an increasingly alarming rate — now over 100 thousand new variants each day — there is a need for automated techniques for identifying suspicious behavior in programs. In this paper, we propose a method for extracting statistically significant malicious behaviors from a system call dependency graph (obtained by running a binary executable in a sandbox). Our approach is based on a new method for measuring the statistical significance of subgraphs. Given a training set of graphs from two classes (e.g., goodware and malware system call dependency graphs), our method can assign p-values to subgraphs of new graph instances even if those subgraphs have not appeared before in the training data (thus possibly capturing new behaviors or disguised versions of existing behaviors).

1. INTRODUCTION

Signature-based detection has been a major technique in commercial anti-virus software. However, that approach is ineffective against code obfuscation techniques. To address this problem, most of the current work, e.g., [1, 7, 12, 10], has focused on behavior-based detection techniques because the semantics of malware are unlikely to change even after a series of syntactic code transformations.

To develop effective behavior-based detection techniques, it is important to understand how malware behaves. Previous studies (e.g., [14, 20]) typically used experts to construct malware specifications that describe malicious behaviors. This requires deep expertise and is costly in terms of time and effort. To address the problem, Christodorescu, *et al.* [7] and Fredrikson *et al.* [12] proposed methods to automatically generate specifications of malicious activity from samples of malicious and benign executables. These methods only recognize behaviors that appeared in the training data and they do not provide scores that indicate the statistical confidence that a (possibly new) behavior is malicious.

In this paper, we propose a new method for identifying malicious behavior and assigning it a *p-value* (a measure of statistical confidence that the behavior is indeed malicious). It requires a training set consisting of malware and goodware executables. Using dynamic program analysis tools, we represent each executable as a graph. We then train a linear classifier to discriminate between malware and goodware; the parameters of this classifier are crucial for our statistical test. To evaluate a new executable, we can use the linear classifier to categorize it as goodware or malware. To then identify its suspicious behaviors, we again represent it as a graph and use a subgraph mining algorithm to extract a candidate subgraph. We assign confidence scores to this subgraph with statistical procedures that use the classifier weights obtained from the training phase; the statistical procedures work for any subgraph, even if it did not appear in the training data. The framework is simple and modular – one can plug in different program analysis tools, linear classifiers, and subgraph mining algorithms to take advantage of progress in those areas. Our statistical tests work with all of these options and are easy to implement.

It is important to note that the evaluation of malware specifications is a challenging task. Manual construction of malware specifications is labor-intensive and error-prone; the resulting relatively small quantity of specifications will often have high false positive rates [14]. Other work [7, 12] compared extracted malware behavior in the form of graphs to textual descriptions provided by anti-virus companies (apparently also manually). In this paper we take a more automated approach designed to reduce the risk of experimenter bias. We use carefully designed experiments to both validate the quality of the p-values and the quality of the suspicious executable behaviors that were identified.

In summary, the contributions of this paper are:

1. A framework for identifying suspicious behavior in programs and assigning them statistical significance scores. The framework is modular, easy to implement, and does not require the malware test set to exhibit the same behaviors as the malware training set.
2. A careful empirical evaluation of the extracted behaviors. This includes an evaluation of p-values and some initial experimental evidence for the identification of malicious behaviors not seen in the training data.

We discuss related work in Section 2. We present our framework in Section 3. The framework includes a training

*This work was done while Domagoj was with UC Berkeley.

ACSAC'13 Dec. 9-13, 2013, New Orleans, Louisiana USA

phase (Section 3.1) and a deployment phase (Section 3.2). We then empirically evaluate our methods in Section 4.

2. RELATED WORK

2.1 Malware Specification

Christodorescu, *et al.* [7] use contrast subgraph mining to construct specifications by comparing syscall dependency graphs of malware and goodware samples to obtain activities appearing only in malware. They assess specification quality by manual comparison to specifications produced by an expert. This technique does not produce statistical significance scores for the specifications. Comparetti, *et al.* [8] propose a novel method to find dormant behaviors statically in binaries, based on manually-provided behavior specifications. Our work complements theirs, as our approach can be used to identify statistically significant behavior specifications. Fredrikson, *et al.* [12] use LEAP [25] to extract behaviors that are synthesized by concept analysis and simulated annealing to generate specifications. The authors evaluate their specifications by manually comparing with behavior reports from malware experts. Even though LEAP, a significant subgraph extraction algorithm, is used, no statistical significance value can be calculated for new executables.

2.2 Significant Subgraph Extraction

There are relatively few algorithms that extract significant subgraphs. LEAP [25] finds significant subgraphs that are frequent in the positive dataset and rare in the negative dataset and maximizes a user-defined significance function. For malware detection, it is used to mine the system call dependency graphs [12]. However, when such a system is deployed for analyzing new executables, searching for exact subgraphs could be a brittle approach – LEAP will not identify malicious behavior if it does not correspond to a graph it has seen before (a slight change in the graph may be enough to avoid detection). Our proposed method also mines system call dependency graphs but is more flexible and so can identify significant behaviors whose corresponding subgraphs have not appeared in the training data. Ranu and Singh [18] propose GraphSig, a frequent subgraph mining algorithm, to find significant subgraphs in large databases. GraphSig prunes out the search space by testing the significance of a subgraph; the definition of significance is based on a subgraph's frequency. GraphSig uses GraphRank [13] for the statistical significance testing. GraphRank transforms subgraphs to feature vectors and calculates p-values of vectors based on a binomial model. Note that frequent subgraphs are not necessarily indicative of malicious behavior encapsulated in system call dependency graphs.

Milo, *et al.* [16] propose an approach to mine network motifs which are significant subgraphs appearing more frequently in a complex network than in random networks. The authors generate random networks by permuting edges while maintaining network properties, such as degree of nodes and number of edges. The p-value of a subgraph is obtained by counting random networks that contain the subgraph with a support (i.e. frequency) greater than or equal to the observed support. Milo's approach may not scale to very large networks because of the complexity of subgraph isomorphism. Scott, *et al.* [19] developed a method to find significant protein interaction paths in a large scale protein network data. A color coding approach is extended to find paths between two given nodes with the minimum sum of edge weights. They adopt a randomization approach for statistical significance testing – they compare scores of paths they find to scores of paths found in random networks whose edges have been shuffled. Our approach uses randomization for statistical testing but we carefully avoid comparisons to random graphs (since they may not be plausible representations of system call dependency graphs).

3. THE STATISTICAL FRAMEWORK

We next describe the major components of our framework for identifying statistically significant malicious behaviors. An overview of the framework is shown in Figure 1. There are two phases: the training phase (where statistical information about malware and goodware is collected) and the deployment phase (for analyzing a new executable).

The training phase (Section 3.1) requires samples of goodware and malware executables. These executables are converted into system call dependency graphs (SDG) using dynamic analysis (Section 3.1.1). We build a linear classifier to distinguish malware from goodware and we use its parameters to obtain a function that assigns weights to edges (Section 3.1.2); these weights are used by our statistical tests.

The deployment phase (Section 3.2) is used to analyze a new executable for suspicious behavior. The linear classifier from the training phase can be used to classify it as malware or goodware. To extract suspicious behavior, we first build the SDG and assign edge weights based on the parameters of the linear classifier. We then use a subgraph mining algorithm to identify candidate subgraphs (Section 3.2.1); any subgraph mining algorithm for weighted or unweighted graphs can be used here as a black box. Then our statistical tests (Section 3.2.2) assign significance scores to the behaviors associated with those subgraphs. These tests use the edge weights to assign p-values and automatically correct for the multiple testing problem (explained in Section 3.2.2), which is an important concern in subgraph mining.

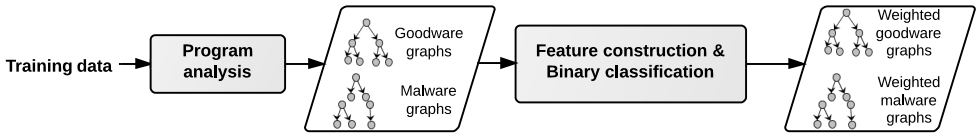
3.1 The Training Phase

We now describe the two components of the training phase: building system call dependency graphs and building a linear malware classifier whose parameters will be used to assign weights to edges in those graph.

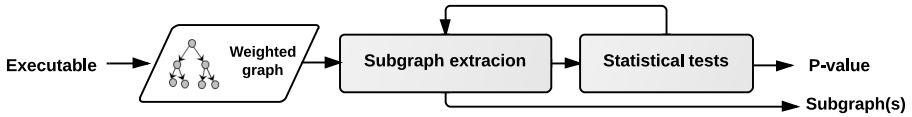
3.1.1 System Call Dependency Graphs (SDGs)

Recent research (e.g., [23, 4]) shows that a program's behavior can be inferred from its pattern of system calls. The outputs produced by some system calls can affect the inputs of other system calls. Hence, it is natural and common to represent a program's behavior using a *system call dependency graph (SDG)* whose nodes correspond to system call invocations and whose directed edges represent data flow between pairs of system calls. This abstraction converts program analysis into a graph mining problem where subgraphs correspond to program behaviors.

DEFINITION 1 (SDG). *A system call dependency graph (SDG) is a directed graph $G(E, V)$ representing data-flow dependencies among system call invocations, where V is the set of invoked system calls and $E \subset V \times V$ is a set of directed edges. The directed edge, $(x, y) \in E$, from vertex x to vertex y indicates that the output of system call invocation x is consumed by system call invocation y.*



(a) Training phase



(b) Deployment phase

Figure 1: Framework overview – identifying statistically significant malicious behavior

To create an SDG, we must execute a program in a sandbox, trace its system calls, and infer dependencies between the system call invocations. The most accurate method for doing this is *dynamic taint analysis* [17], although our attempts to reproduce existing work (such as [3]), show that faster heuristic methods can work just as well. Our framework can work with any SDG generator, however, our experiments used WUSSTrace [24] (which injects a shared library into the address space of a traced process) to generate program execution traces and Pywuss [3] to parse these traces to generate an approximate SDG. Pywuss creates a directed edge between two system call invocations x and y, if x returns a handle that is used as an input to y. An SDG is created for each executable in the training set.

Note that disassembling and static analysis or statistical analysis of a binary can provide additional information about a program. Incorporating this information into our framework is a direction for future work.

3.1.2 Adding Weights with Linear Classifiers

Given a set of SDGs belonging to goodware and malware executables, the next step is to build a malware classifier and use its parameters to assign weights to the edges of those graphs. We convert each SDG into a feature vector by generating one feature for each *ordered* pair (x, y); the value of the feature is the number of times (x, y) appeared in the SDG. Using these feature vectors, we then train a linear classifier (such as logistic regression) to discriminate between malware and goodware. In this way, the linear classifier learns a weight $w_{(x,y)}$ for each ordered pair of system calls (x, y); a positive value of $w_{(x,y)}$ indicates that (x, y) is associated with malware and a negative value indicates it is associated with goodware. The result is a weighted SDG where each edge (x, y) has weight $w_{(x,y)}$. These weights will be used for subgraph extraction and significance testing.

3.2 The Deployment Phase

The deployment phase is used to analyze a new executable. First, we obtain its SDG as in Section 3.1.1. We then assign weights to the edges of the graph using the same weights that were learned in the training phase (see Section 3.1.2). The next steps are to use a subgraph mining algorithm to identify candidate subgraphs that may correspond to suspicious behaviors and then to assign them a statistical confidence score. We discuss subgraph extraction in Section 3.2.1. The mining algorithms can use the edge weights to identify suspicious subgraphs. Since the weights are also part of the statistical test, we must automatically account for variations of the multiple testing problem. We address this issue and present our statistical tests in Section 3.2.2.

3.2.1 Subgraph Extraction

A subgraph of an SDG corresponds to a behavior exhibited by an executable. A subgraph can be deemed suspicious when it contains many edges with positive weights, especially when the concentration of positively-weighted edges in the subgraph is higher than in the rest of the SDG.

Any algorithm for finding (weighted) subgraphs can be used as a black box in our framework. The goal of subgraph mining here is to identify subgraphs with high concentrations of positive weights (rather than specifically searching for subgraphs or templates that previously appeared in the training data). This corresponds to the hypothesis that system calls which commonly participate in malicious behavior are used together to achieve their purpose. In practice, the SDG has many small connected components and so it makes sense to return a subgraph consisting of several disjoint connected components.

In our implementation, we use a variation Kruskal's spanning tree algorithm [9] to find subgraphs (not necessarily trees) in each connected component of the SDG. The key steps appear in Algorithm 2 (kMines). Initially, each node u is its own temporary subgraph (which will later grow). Graph(u) is the temporary subgraph containing u. The algorithm considers each edge (u, v) in descending order by weight. Let $Edge(u, v)$ be the union of edges in Graph(u), Graph(v), and the edges connecting them. Line 7 heuristically merges Graph(u) and Graph(v) (and the edges between them) if the sum of weights of edges in $Edge(u, v)$ is greater than or equal to k times the maximum weight in $Edge(u, v)$ (where $0 < k \leq 1$).. The condition encourages the algorithm to merge temporary subgraphs instead of returning many small subgraphs (possibly with just one edge each)

These key steps are wrapped inside Algorithm 1 which returns the final subgraph B (possibly consisting of disjoint connected components). Algorithm 1 applies Algorithm 2 (kMine) to each connected component g of the SDG. It then orders the subgraphs returned by kMine in descending order by average weight (lines 5-6) and iteratively adds them to B. The algorithm keeps iterating until the addition of a new subgraph to B increases sum of weights by less than $p\%$. Afterwards it returns B as the final extracted subgraph

(lines 8-14) . Our implementation uses $k = 0.5$ and $p = 5\%$, which were chosen subjectively from a few data samples so that the returned subgraphs were not too big (i.e. a large fraction of the original graph) nor too small (a few edges).

Algorithm 1 Kruskal-based subgraph extraction

Input: G (a weighted SDG), p, k
Output: B, a collection of subgraphs
1: $S \leftarrow \emptyset$
2: **for all** connected components g in G **do**
3: $S \leftarrow S \bigcup$kMine(g,k)
4: **end for**
5: Sort components in S by decreasing average weight
6: $B \leftarrow \max\{s|$component $s \in S\}$
7: S $\leftarrow$ S $\setminus\{$B$\}$
8: **for** each connected component s in S **do**
9: **if** (sum_weight(B,s) - sum_weight(B))$\geq p*$sum_weight(B) **then**
10: $B \leftarrow B \bigcup s$
11: **else**
12: **return** B
13: **end if**
14: **end for**

Algorithm 2 kMine(g,k)

Input: a connected component g, k
Output: S, a collection of subgraphs
1: $S \leftarrow \emptyset$
2: **for** node(u) in g **do**
3: $S \leftarrow S \bigcup$ Graph(u)
4: **end for**
5: **for** each edge, e_{uv}, in g, ordered by weights, w_{uv}, **do**
6: **if** $\sum_{e_{ij} \in Edge(u,v)} w_{ij} \geq k * \max\{w_{ij} | e_{ij} \in Edge(u,v)\}$ **then**
7: Graph(u) $\leftarrow$ merge(Graph(u), Graph(v))
8: $S \leftarrow S \setminus \{$Graph($v$)$\}$
9: **end if**
10: **end for**
11: **return** S

3.2.2 Significance Testing

We next describe our statistical tests for the candidate subgraph returned by the graph mining algorithm. We present three p-value computation techniques: *empirical* p-values that can be viewed as data-driven false positive rates, *resampling* p-values that are approximations to empirical p-values and useful when the training data is small, and *permutation* p-values which complement the other two. Permutation p-values consider how concentrated the positive edges are, while the other p-values additionally incorporate information about how many positive edges there are relative to goodware and malware.

Note that statistical significance is not a property of a subgraph – it is a property of the procedure that found the subgraph. To see why, consider the following two types of search strategies. Strategy A ignores edge weights when it returns a candidate subgraph while Strategy B returns the subgraph with largest average edge weight. If the subgraph returned by Strategy A has a high average edge weight, this is likely to be statistically significant because such an occurrence is generally unlikely to happen by chance; on the other hand, Strategy B is expected return subgraphs with large average edge weights so the bar is higher – the output returned by B is statistically significant only if its average edge weight is much higher than what we would normally expect from B. This phenomenon underlies the multiple testing problem [5] and our procedures automatically account for it.

Empirical p-values with reference populations.

Empirical p-values are a comparison between a subgraph extracted from a given SDG to subgraphs extracted from SDGs belonging to a reference population. Let $G_1, \ldots, G_N$ be weighted SDGs from a reference population (e.g., the set of goodware in the training data or the set of malware in the training data). For each G_i, let S_i be the subgraph extracted from G_i by the subgraph mining algorithm and let B_i be the average weight of edges in S_i.[1] To test a new executable G^*, let S^* be its extracted subgraph and let B^* be the average edge weight of S^*. The *empirical* p-value is the fraction of the reference population whose subgraphs had higher average edge weight: $\frac{1}{N}\sum_{i=1}^{N} \mathbf{1}_{\{B_i \geq B^*\}}$.

This p-value is affected by the concentration of positive edges (which affects the B^* that is returned), their number, and the magnitude of their weights. The null hypothesis is that the positive edge weights are not concentrated and the sampling distribution is the empirical distribution of the reference population.

Note that there are two possible reference populations: the training goodware and the training malware. A low p-value with respect to the malware population is indicative of an application that is highly suspicious. A moderate p-value with respect to the malware population and a low p-value with respect to the goodware population indicates typical malware behavior. A high p-value with respect to the malware population and a low p-value with respect to the goodware population indicates a borderline application – it performs operations that are unusual for goodware but are not that suspicious relative to previously seen malware behavior (in the training data).

Note that computation of empirical p-values requires a one-time pre-processing of the reference population.

Resampling p-values with reference populations.

An empirical p-value is accurate when the training data is large (since it would be an average over many data points). For smaller data sets, we propose the resampling p-values returned by Algorithm 3.

Given a new graph G^*, it first creates a set of resampled graphs $G_1^{(r)}, \ldots, G_k^{(r)}$ by replacing the edge weights of G^* with weights sampled from a distribution P_{weight} (which we will discuss shortly). To compute the p-value, it compares the average weight of the subgraph extracted from G^* to the average weight of the subgraphs extracted from the $G_i^{(r)}$.

There are several ways of obtaining a distribution P_{weight}. Resampled p-values with respect to the goodware reference population use the empirical distribution of edge weights from SDGs in the training goodware; resampled p-values with respect to the malware reference population use the empirical distribution of edge weights from SDGs in the training malware.

[1]We call B_i the test statistic. We use average edge weight but other statistics can be used too.

Algorithm 3 Resampling p-values

Input: P_{weight} (a distribution over edge weights),
Input: SubgraphMiner (a subgraph mining algorithm)
Input: G^* (a new graph to test)
Output: a p-value

1: $B^* \leftarrow$ average edge weight of SubgraphMiner(G^*)
2: **for** $i = 1, \ldots, k$ **do**
3: Create graph $G_i^{(r)}$ with the same structure as G^*.
4: For each edge in $G_i^{(r)}$ assign it a weight as a random sample from P_{weight}
5: $B_i^{(r)} \leftarrow$ average edge weight of SubgraphMiner($G_i^{(r)}$)
6: **end for**
7: **return** $\frac{1}{k}\sum_{i=1}^{k} \mathbf{1}_{\{B_i^{(r)} >= B^*\}}$

Note that this test only resampled edge weights and performs no randomization of the structure of the graph. We intentionally avoid randomizing the structure of the graph because there is no evidence that existing random graph models are plausible generative models for system call dependency graphs.

Permutation p-values.

Permutation p-values are designed only to check for concentrations of positive edges. They do not compare the magnitude of the edge weights to reference populations. Hence, their role is to help us understand empirical and resampled p-values. If empirical and resampled p-values are low it could be due to two reasons – a high concentration of positive edge weights in the extracted subgraph or large edge weight magnitudes. Generally, the concentration of positive edge weights is responsible if the permutation p-value is low; the edge weight magnitudes are responsible if the permutation p-value is high.

Permutation p-values are computed in a similar way to resampled p-values. The resampled p-value computation creates a set of graph $G_1^{(r)}, \ldots, G_N^{(r)}$ by resampling the edge weights for the graph G^*. The permutation p-value computation creates $G_i^{(r)}$ by making a copy of G^* and permuting its weights (i.e., randomly reassigning the weights to different edges). Again it extracts a subgraph S^* from G^* and a subgraph S_i from each of the $G_i^{(r)}$ and counts the fraction of S_i that have a higher average edge weight than S^*.

4. EXPERIMENTS

We collected 2393 executables from 50 malware families to produce 2393 system call dependency graphs. We collected 50 goodware programs (based on the list used by [12]) and executed the goodware binaries multiple times to generate 434 goodware system call dependency graphs. We also obtained data from the McAfee website [15] which contains a plain text description of known activities of each malware family in our collection. An example of this kind of data is shown in Figure 2, which contains the description of a sample from the LdPinch family.

To generate the SDGs, malware and goodware binaries were executed in a sandbox; invoked system calls were traced by WUSSTrace [24] (which traces system calls by injecting a shared library in the address space of the traced process). All binaries were executed for up to two minutes. The execution traces were parsed using Pywuss [3] to generate SDGs

Activities
Attempts to write to a memory location of a protected process.
Attempts to write to a memory location of a Windows system process
Attempts to write to a memory location of a previously loaded process.
Enumerates many system files and directories.
Adds or modifies Internet Explorer cookies

Figure 2: Description of LdPinch Activities [15].

(with an edge between system call invocations x and y if x returns a handle that y consumes).[2]

4.1 Malware Detection

Recall that our framework performs two distinct functions: malware detection (using a linear classifier such as logistic regression) and subsequent extraction of *statistically significant* subgraphs (from samples it labels as malware) to help prioritize an expert's analysis of the executable.

In this section we compare the malware detection rates with Holmes [12] and with two commercial anti-virus products, AVG antivirus [2] and ThreatFire [22],[3] at the 0% false positive rate on the ROC curve. Note that a comparison to Holmes is qualitative at best: we have to use reported numbers [12] because neither the code nor data was available (but our dataset was constructed to closely match the description in [12]). It is also not clear if the training goodware for Holmes was excluded from the testing set.

We randomly split the data (SDGs with malware/goodware labels) into three pieces while ensuring that malware families present in one piece do not appear in the other pieces. We ensured that one piece contained approximately 60% of the total malware and 60% of the total goodware; this piece was used for training (i.e. training logistic regression models [11] with different regularization parameters) and we used the F-score method for feature selection [6] to keep only the top 50% of the features (see Section 3.1.2). The second piece, the holdout set (used for model selection), contained approximately 20% of the total goodware and 20% of the total malware. The third piece, also containing approximately 20% of the goodware and 20% of the malware, was used to evaluate the accuracy of the selected model. We repeated this partitioning procedure five times (with different sets of families/programs in each piece) and averaged the results.

The results are shown in Table 1. The primary conclusion is that the linear classifier is good at separating malware from goodware and so its weight parameters form a reasonable basis for statistical testing of subgraphs.

4.2 The Silver Standard

Having evaluated detection rates, we must next evaluate the quality of extracted subgraphs. This involves matching

[2]In our own attempts to reproduce work such as [3], we found that producing an SDG using heuristics instead of dynamic taint analysis [17] resulted in almost the same precision and recall. Refined SDGs are useful for engineering purposes but would not be expected to produce dramatically different results.

[3]In order to test performance on *unseen* malware, we could not use the most recent versions of AVG and ThreatFire as they have been updated to include our malware samples.

AVG	ThreatFire	Holmes	Our Framework
58.37	67.08	86.56*	86.77

Table 1: Malware detection rates at 0% false positive. *Reported from [12]

them to plain text descriptions of malware behavior. One way to do this is manually [12]. However, there are drawbacks to the manual approach. First, manual judgments require considerable expertise (and can still be noisy, with high false positive rates [14]). Second, they can lead to experimenter bias. Thus we seek a more automated approach with the creation of an evaluation dataset, called the *silver standard*, which we describe next.

An ideal dataset would contain annotations of system call dependency graphs that indicate which subgraphs correspond to malicious activity. Such a "gold standard" does not exist so we constructed an approximation to it, called the *silver standard*, using the plain-text descriptions obtained from the McAfee website [15] (see Figure 2).

For each piece of malware, we first convert its plain-text activity into a list of system calls. For example, the activity 'Creates registry keys and data values to persist on OS reboot' is converted to the list {NtOpenKey, NtSetValueKey}. Note that there is no unique translation between textual description and system calls and so some noise is necessarily introduced in this process.

Now, the system call dependency graphs consist of a disjoint union of many (usually small) connected components. For each connected component, we keep it if it contains an edge between any two system calls on this list. Then we remove edges whose vertices are not in the system call list. Next, we apply the Kruskal-based algorithm (Algorithm 1) to each component in order to prune irrelevant edges. We remove vertices for wait-related system calls and NtClose (whose presence/absence has no causal effect on malicious behavior). The wait-related system calls, such as NtWaitForMultipleObjects, are used to wait until a specified criteria is met or a time-out interval has elapsed. Finally, we also remove repetitive/redundant NtFreeVirtualMemory and NtFlushVirtualMemory invocations $y_1, y_2, \ldots, y_k$ that have an incoming edge from the same node x and have no outgoing edges; we only keep y_1, the first of these duplicate calls.

In this way, for each malware SDG, we obtain a subgraph (consisting of possibly many connected components) that is marked as malicious activity. Such a subgraph is called a *silver standard graph*. As an example, Figure 3 shows the connected components of a silver standard graph from a sample in the Banbra family. According to Symantec [21], Banbra is a Trojan horse that attempts to steal financial information from the compromised computer and send the information to a remote location. Component SS1 was induced by an attempt to launch an instance of Internet Explorer. Components SS2 and SS3 resulted from writing stolen information to a file and sending it over the internet respectively. SS4 resulted from creating a mutex to avoid infecting the same computer twice. SS5 and SS6 resulted from adding/modifying registry keys. SS7 was induced by checking a process's privilege.

It is important to note that the creation of the silver standard graphs is a noisy process due to the conversion of a textual description to a system call list and its automated

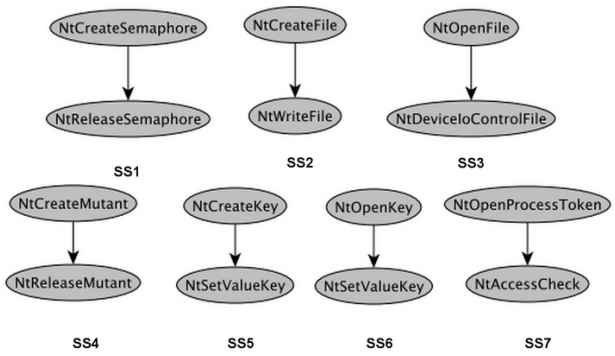


Figure 3: Components of a silver standard graph of a sample in the Banbra family

matching to subgraphs of malware SDGs. Another complication is that malware does not always perform malicious activity every time it is run. Note, however, that the alternative is a manual inspection, which is also noisy and which risks introducing experimenter bias.

4.3 Evaluation of Extracted Subgraphs

We evaluate the subgraphs extracted by our framework using the Kruskal-based algorithm (Algorithm 1). We can measure two quantities for these subgraphs: their p-values and their similarities to the silver standard graphs.

To measure the similarity between a subgraph S extracted from a malware sample and the corresponding silver standard graph S_{ag} for that sample, we use F1, precision, and recall scores defined in the following way. *Precision* is the fraction of distinct edges in S that are also in S_{ag}, *recall* is the fraction of distinct edges in S_{ag} that also appear in S, and *F1* is the harmonic mean of precision and recall.

4.3.1 Correlation between p-values and F1 scores

The silver standard graphs were constructed with the help of textual descriptions of malware behavior. The computation of p-values does not have access to this information. Thus the next set of experiments are designed to measure the level of agreement between the p-value of an extracted subgraph and the F1 similarity score (between that subgraph and the silver standard graph).

We used a train/holdout/evaluation data partition as described in Section 4.1. We used the evaluation set for extracting subgraphs and computing p-values. For reference, the evaluation set contained executables from the malware families Rbot, Downloader, Mydoom, LdPinch, Gaobot, OnLineGames, Hupigon, Stration, and Banbra; it also contained the goodware openOfficeWriter, 7zip, Bitcomet, SpeedFan installation, openOfficeDraw, Chrome, mysql, winrar, and AVG Antivirus.

For each malware sample, we ran the subgraph mining algorithm. We computed the p-value of the returned subgraph along with its F1 similarity to that sample's silver standard graph. Ideally, the similarity score would be high and the p-value would be low (to indicate statistical significance). Thus, the p-value of the extracted subgraph should be negatively correlated with the F1 score.

For a quantitative assessment, we calculated Pearson correlation between p-values (using the permutation method)

Malware	Correlation
Stration	-0.4123
OnLineGames	-0.1643
Rbot	-0.0289
Hupigon	0.5666
Banbra	-0.6952
Gaobot	-0.3932
Downloader	-0.1136
LdPinch	-0.2201
Mydoom	-0.2365

Table 2: Correlations between p-values and F1 scores of malware samples.

and F1 scores in all malware families in the evaluation set. The results are shown in Table 2. With the exception of the Hupigon family, all correlations are indeed negative. The reason that Hupigon samples had a positive correlation was the following. This family is classified by McAfee[15] as a backdoor. In general, a backdoor will simply provide a hacker with a convenient access point to a machine to enable future malicious activities. Without a command from a hacker, we suspected that our backdoor samples will not exhibit much malicious behavior. To check this, we divided the Hupigon samples into two groups: the significant samples, from which our framework extracted subgraphs with low p-values, and the non-significant samples (i.e., the rest of the samples). There were only 2 significant samples and their exhibited behaviors consisted of: 1) checking access tokens of other processes (possibly with an attempt to use memory of another process), 2) sending data over the network, and 3) checking a mutant (windows terminology for a mutex) and creating one if it didn't exist. There were 28 non-significant samples and 24 of them only exhibited the third behavior, which was part of their extracted subgraphs and which is not necessarily malicious.

We can perform a similar experiment with goodware. From each SDG we can extract a subgraph and compute its p-value. We can also compute its F1 score with respect to the best matching silver standard graph out of all malware samples. In this case, the correlation should not be negative. The average correlation was 0.3348.

4.3.2 Comparison of p-value computations

Recall that we presented three methods for computing p-values: empirical, resampling, and permutation p-values. Empirical p-values with respect to the goodware reference population have similar interpretations to false-positive rates (i.e., how many goodware training samples exhibit more suspicious behavior), while empirical p-values with respect to the malware reference population compare a sample's behavior to typical malware behavior. Resampling p-values are an approximation to empirical p-values, can be computed without storing the training data, and can be preferable when the size of the training data is small. Both empirical and resampling p-values are affected by how many suspicious edges (i.e., edges more often associated with malware) there are relative to the reference population and by how clustered those edges are. On the other hand, permutation p-values only consider the how clustered those edges are and are essentially designed to measure whether such edges are grouped together in a manner that is not random (e.g., they are chained together for a common purpose).

Table 3 shows *average* p-values of subgraphs from malware and goodware families using the empirical, resampling, and permutation techniques. For the case of empirical and resampling, we show p-values with respect to malware and goodware reference populations.

Since malware does not always exhibit malicious behavior in every execution, the purpose of this table is not to highlight malicious activity (this will be done in Section 4.3.3, where we focus specifically on samples that have low p-values). Instead, the primary purpose of this table is to highlight agreement/disagreement between these three approaches. For example, we notice that average p-values with respect to goodware reference populations are lower than with respect to malware populations (e.g., because behavior that is atypical for goodware is not necessarily atypical for malware). We also note that the average p-value of malware samples is generally lower than the average p-values for goodware samples, even though malware does not always exhibit malicious behavior. There are two malware families, Hupigon and Gaobot, that have higher p-values than the other malware families. They are both backdoors [15] with similar behavior. The explanation for their high p-values is similar to the discussion of Hupigon in Section 4.3.1.

Note that several goodware have low permutation p-values: SpeedFan installation, Chrome, 7zip, and winrar. For SpeedFan installation and Chrome, the p-values are significant. As discussed in Section 3.2.2, the cause is due to a concentration of positive edges. For example, SpeedFan installation contained a very large connected component that consisted of positive edges. The system calls involved various virtual memory and process management functions that are only slightly more indicative of malware (in our training data) and hence many edges had small but positive edge weights. The empirical and resampling p-values were not significant because of these edge magnitudes. Note that both malware and goodware SDGs contain edges with positive weights and edges with negative weights. As Figure 4 shows, the positive edge magnitudes in SpeedFan installation are generally smaller than is typical even for goodware.

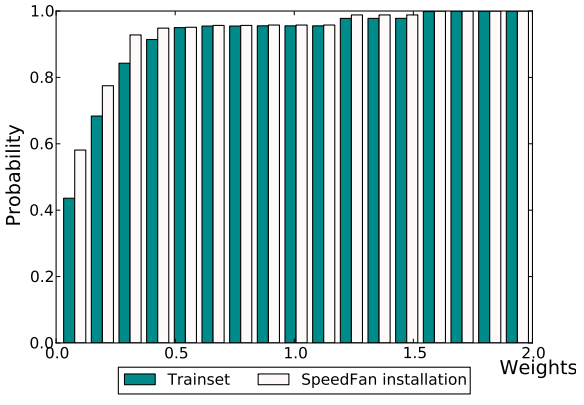


Figure 4: Cumulative distribution function of positive edge weights in SpeedFan installation and the average cumulative distribution function of positive edge weights in training goodware. The cumulative distribution function that increases fastest is the one that has more of the smaller values (i.e. more edges with smaller positive weights).

Kruskal-based subgraph extraction						
	Family	Permutation	Empirical		Resampling	
			Goodware-reference	Malware-reference	Goodware-reference	Malware-reference
Malware	Stration	0.0736	0.4719	0.9286	0.4977	0.9405
	OnLineGames	0.0900	0.4719	0.9286	0.5170	0.9525
	Rbot	0.0131	0.0612	0.1939	0.0707	0.1280
	Hupigon	0.7035	0.4722	0.9550	0.5103	0.9483
	Banbra	0.2088	0.1244	0.5134	0.2323	0.5192
	Gaobot	0.7255	0.5778	0.9087	0.4644	0.9250
	Downloader	0.0419	0.3066	0.7355	0.3640	0.7646
	LdPinch	0.1109	0.2536	0.6897	0.2759	0.8263
	Mydoom	0.0013	0.1707	0.6386	0.2967	0.8733
Goodware	AVG Antivirus	0.6590	0.8502	0.9992	0.1800	0.7300
	Bitcomet	1.0000	0.8014	0.9907	0.7067	1.0000
	SpeedFan installation	0.0000	0.5071	0.9684	0.7973	1.0000
	mysql	0.9710	0.4634	0.9831	0.6900	1.0000
	Chrome	0.0000	0.2648	0.9022	0.5700	0.9022
	7zip	0.1447	0.2190	0.7572	0.4907	0.9560
	winrar	0.1400	0.2997	0.9511	0.7667	1.0000
	openOfficeDraw	1.0000	0.9992	1.0000	0.9870	1.0000
	openOfficeWriter	1.0000	0.9022	1.0000	0.9862	1.0000

Table 3: Average p-values of subgraphs extracted by the Kruskal-based algorithm. Permutation, empirical, and resampling p-values are described in Section 3.2.2. Note that malware samples do not always perform malicious activity in every execution.

Family	F1	Precision	Recall
Stration	0.4505	0.3836	0.7279
OnLineGames	0.3041	0.2351	0.4977
Rbot	0.4075	0.5347	0.4144
Hupigon	0.2759	0.2667	0.2857
Banbra	0.4534	0.4490	0.6212
Gaobot	0.3884	0.4163	0.3675
Downloader	0.3813	0.3869	0.4324
LdPinch	0.3862	0.4984	0.3198
Mydoom	0.4362	0.5625	0.3563

Table 4: Average F1, precision and recall scores between the silver standard and extracted subgraphs with permutation p-values ≤ 0.05

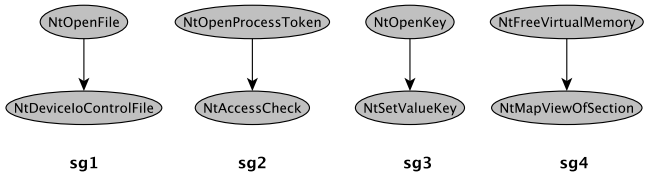


Figure 5: A significant subgraph of an LdPinch sample

Family	F1	Precision	Recall
Stration	0.1304	0.1071	0.1667
Banbra	0.3000	0.2143	0.5000
Gaobot	0.3000	0.2143	0.5000

Table 5: Average similarity scores of 95% significant malware subgraphs extracted by gSpan.

4.3.3 *Similarity scores of significant subgraphs*

Next, we computed similarity scores of significant subgraphs with respect to the silver standard graphs. The results are shown in Table 4. We extracted a subgraph for each executable in each malware family using the Kruskal-based algorithm. We kept the subgraphs that were statistically significant (i.e. permutation p-values ≤ 0.05) and computed the similarity of these subgraphs to the silver standard.

As an example of the types of subgraphs returned by the Kruskal-based algorithm, see Figure 5, which contains a subgraph from a sample in the LdPinch family that has a permutation p-value that is below 0.05. Note that the suspicious behavior consists of all of the connected components *collectively* (not individually). Connected components $sg1$, $sg2$, $sg3$ and $sg4$ in Figure 5 can be induced by the actions of sending stolen data over the network, checking its privilege, adding an entry into the registry and making sure that the memory space is free before sharing the memory between their processes, respectively.

4.4 Comparison of graph mining algorithms

In this section, we evaluate the subgraph extraction component (Section 3.2.1) of our framework. We consider the Kruskal-based algorithm (Algorithm 1) and gSpan [26] (a frequent subgraph mining algorithm) and LEAP [25] (a subgraph mining algorithm designed to discriminate between two classes and which was used in [12] as part of a malware detection framework). To use gSpan within our framework, we modified line 3 in Algorithm 1 to call gSpan to obtain frequent subgraphs at 5% frequency. To use LEAP, we simply replaced Algorithm 1 with a call LEAP. The call to LEAP requires positive samples and negative samples. For the positive samples, we used graphs from the malware executable being tested; for the negative samples, we used the goodware graphs so that it can learn to distinguish between that malware sample and the goodware. There was an imbalance in size between the positive and negative samples provided to LEAP and as a result it produced no significant subgraphs. Hence, all of our subsequent comparisons are restricted to the Kruskal-based algorithm and gSpan.

4.4.1 *P-values*

The average p-values of subgraphs, extracted by gSpan, using permutation, empirical and resampling methods are shown in Table 6. From the Table, average p-values of malware and goodware subgraphs are not that different. Results

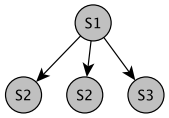


Figure 6: A component with multiple edges of type $(S1, S2)$

from Tables 3 and 6 show that the Kruskal-based method can extract subgraphs with more reasonable p-values than gSpan. The reason for the difference is that gSpan does not use edge weights (it uses subgraph frequency instead). Thus a comparison of these two tables show that frequent behaviors and malicious behaviors are entirely different concepts.

4.4.2 Similarity scores

The gSpan mining algorithm produced significant subgraphs (p-value below 0.05) only from samples from the Stration, Banbra and Gaobot families. Table 5 shows the average similarity of those subgraphs to the corresponding silver standard graphs. By comparison, Table 4 shows the corresponding results for the Kruskal-based algorithm, which produced higher similarity scores.

4.5 Unseen Behaviors

Template-based malware detection frameworks look for fixed patterns, such as the presence of pre-specified subgraphs in SDGs of new programs. One of their disadvantages, therefore, is their limited ability to identify malicious behavior that has not previously appeared in their training sets. On the other hand, our statistical testing framework is more flexible because it considers the presence of clusters of certain types of edges without pre-defined connection patterns between the edges. As a result, subgraphs that did not appear in the training data can still be flagged as malicious (furthermore, it is more difficult for malware authors to counter this approach relative to fixed templates).

In the final set of experiments, we search for an empirical demonstration by checking our testing set for behavior that did not appear in the SDGs used for training and model selection. For each malware sample in our testing set, we extract a subgraph using the Kruskal-based algorithm. We kept only the extracted subgraphs that were significant (in this case, those that had permutation p-values below 0.05). Now, each extracted subgraph S_i may consist of several disjoint connected components $S_i^{(1)}, \ldots, S_i^{(k)}$. For each component, we check whether it is isomorphic to a subgraph of any training malware SDG and only keep those components that are not subgraph isomorphic to training malware SDGs.

There were many connected components $S_i^{(j)}$ belonging to extracted subgraphs that survived this pruning. However, we were not satisfied with most of them because of the following reason. Many components $S_i^{(j)}$ had multiple edges of the same type (i.e. they connected nodes with the same labels, as in Figure 6). In many cases, we found that we could remove the extra copies of those edges so that the resulting component was still connected and was also isomorphic to a subgraph of a training malware SDG.

However, there was one component of an extracted subgraph from the Stration family that survived even this pruning step. Figure 7 shows the part of the SDG containing the extracted component. Nodes and edges belonging to the extracted component are marked in bold (the rest of the nodes and edges are shown to give context to this example). This malware instance was trying to execute its code in another process's context. To do so, it first created a process in a suspended process with the CREATE_SUSPENDED parameter to suspend the target's main thread. Next, it queried the base address value of the suspended process. It then read the code from its process and wrote into the memory space of the suspended process, starting at the base address. When the copy was done, it resumed the thread with the instruction pointer of the suspended thread to the location of the copied code. We note that some malware in our training data also has this high-level behavior. However, their edges are connected together in a way that is different from Figure 7. This reflects multiple ways of achieving the same goal, but with a different graph structure (something that template-based schemes may have difficulty with).

5. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a framework for classifying malware and identifying suspicious program behavior using statistical techniques. Our framework uses information contained in the system call dependency graph of an executable. Other approaches, such as static analysis and statistical analysis of a binary are also useful. In the future, we plan to incorporate these sources of information as well as refine the subgraph extraction algorithms.

6. ACKNOWLEDGMENTS

This work is partially supported by University of the Thai Chamber of Commerce and NSF grant #1054389.

7. REFERENCES

[1] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *J Comput Virol*, 7(4):247–258, 2011.

[2] AVG Antivirus 7.5.519a `http://www.oldversion.com/download-AVG-Anti-Virus-7.5.519a.html`.

[3] D. Babić, D. Reynaud, and D. Song. Malware analysis with tree automata inference. In *CAV*. Springer, 2011.

[4] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[5] G. Casella and R. Berger. *Statistical inference*. Duxbury Press, 2001.

[6] Y. Chen and C. Lin. Combining svms with various feature selection strategies. *Feature Extraction*, 2006.

[7] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, 2008.

[8] P. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *S&P*, 2010.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.

[10] J. Devesa, I. Santos, X. Cantero, Y. Penya, and P. Bringas. Automatic behaviour-based analysis and classification system for malware detection. In *ICEIS*, 2010.

gSpan						
	Family	Permutation	Empirical		Resampling	
			Goodware-reference	Malware-reference	Goodware-reference	Malware-reference
Malware	Stration	0.2633	0.4455	0.4927	0.6029	0.9976
	OnLineGames	0.2716	0.3560	0.3894	0.7363	0.9975
	Rbot	0.3367	0.7236	0.8372	0.5727	0.9928
	Hupigon	0.4040	0.7414	0.8599	0.7027	0.9993
	Banbra	0.3738	0.6417	0.7308	0.4778	0.9913
	Gaobot	0.2769	0.3994	0.4143	0.4013	0.9519
	Downloader	0.4322	0.6272	0.6884	0.6611	0.9976
	LdPinch	0.4243	0.6483	0.7149	0.5120	0.9717
	Mydoom	0.3913	0.3532	0.3903	0.4793	0.9960
Goodware	AVG Antivirus	0.3140	0.4153	0.4299	0.1720	0.8480
	Bitcomet	0.0857	0.1855	0.1861	0.0600	0.4622
	SpeedFan installation	0.2354	0.8392	0.9579	0.5887	0.9127
	mysql	0.2462	0.8377	0.9569	0.1163	0.6188
	Chrome	0.2500	0.8952	0.9875	0.1100	1.0000
	7zip	0.2580	0.8145	0.9516	0.0907	0.6387
	winrar	0.2600	0.3669	0.4032	0.0800	0.5100
	openOfficeDraw	0.0020	0.3629	0.2830	0.3490	0.7300
	openOfficeWriter	0.0000	0.3629	0.2830	0.5023	0.8369

Table 6: Average p-values of subgraphs extracted by gSpan. The malware and goodware p-values are generally similar, showing that those subgraphs are unrelated to malicious activity.

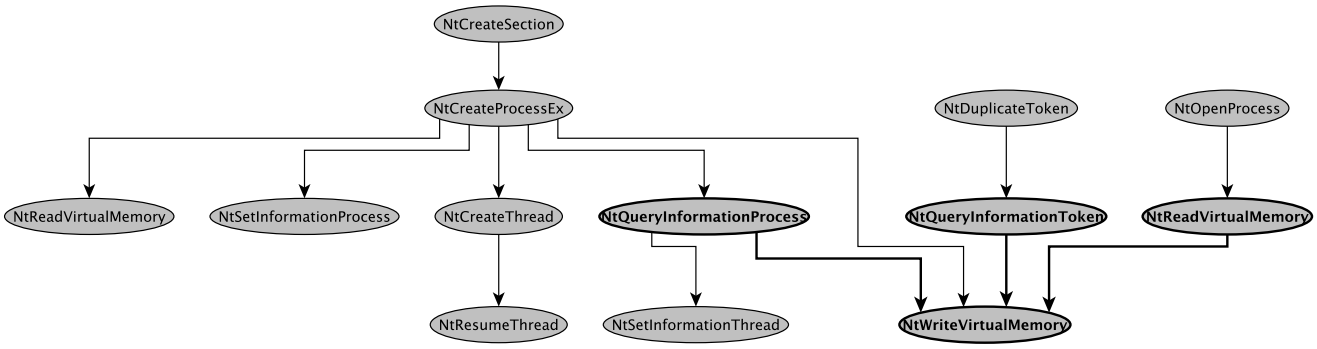


Figure 7: Unseen subgraph of a sample from Stration family

[11] R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. Liblinear: A library for large linear classification. *JMLR*, 9:1871–1874, 2008.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *S&P*, 2010.

[13] H. He and A. Singh. Graphrank: Statistical modeling and mining of significant subgraphs in the feature space. In *ICDM*, 2006.

[14] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A layered architecture for detecting malicious behaviors. In *RAID*. Springer, 2008.

[15] Mcafee labs threat center. `www.mcafee.com/us/mcafee-labs.aspx/`, May 2012.

[16] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824, 2002.

[17] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *NDSS*, 2005.

[18] S. Ranu and A. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*, 2009.

[19] J. Scott, T. Ideker, R. Karp, and R. Sharan. Efficient algorithms for detecting signaling pathways in protein interaction networks. *J. Comp. Bio.*, 13(2):133–144, 2006.

[20] E. Stinson and J. Mitchell. Characterizing bots' remote control behavior. *DIMVA*, 2007.

[21] Symantec security research centers. `www.symantec.com/security_response/`, Nov 2012.

[22] ThreatFire v3.0.0.15 Beta 1 `http://www.afterdawn.com/software/general/download_splash.cfm/threatfire?software_id=1369&version_id=6190`.

[23] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *S&P*, 5(2), 2007.

[24] Wusstrace. `http://code.google.com/p/wusstrace/`, June 2012.

[25] X. Yan, H. Cheng, J. Han, and P. Yu. Mining significant graph patterns by leap search. In *SIGMOD*, 2008.

[26] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, 2002.