SigMal: A Static Signal Processing Based Malware Triage

Dhilung Kirat
University of California,
Santa Barbara
dhilung@cs.ucsb.edu

Lakshmanan Nataraj
University of California,
Santa Barbara
lakshmanan_nataraj@ece.ucsb.edu

Giovanni Vigna
University of California,
Santa Barbara
vigna@cs.ucsb.edu

B.S Manjunath
University of California,
Santa Barbara
manj@ece.ucsb.edu

ABSTRACT

In this work, we propose SigMal, a fast and precise malware detection framework based on signal processing techniques. SigMal is designed to operate with systems that process large amounts of binary samples. It has been observed that many samples received by such systems are variants of previously-seen malware, and they retain some similarity at the binary level. Previous systems used this notion of malware similarity to detect new variants of previously-seen malware. SigMal improves the state-of-the-art by leveraging techniques borrowed from signal processing to extract noise-resistant similarity signatures from the samples. SigMal uses an efficient nearest-neighbor search technique, which is scalable to millions of samples. We evaluate SigMal on 1.2 million recent samples, both packed and unpacked, observed over a duration of three months. In addition, we also used a constant dataset of known benign executables. Our results show that SigMal can classify 50% of the recent incoming samples with above 99% precision. We also show that SigMal could have detected, on average, 70 malware samples per day before any antivirus vendor detected them.

Keywords

malware similarity, detection, signal processing

1. INTRODUCTION

The ever-increasing volume of new malware produced every day is a challenging problem [16]. The analysis of such a large quantity of new malware demands scalable analysis techniques as well as efficient triage systems that can quickly prioritize the incoming samples. There are three main approaches to malware analysis in practice: Static Analysis, Dynamic Analysis, and Statistical Analysis. Static analysis disassembles the code present in the executable without executing the program, and analyzes different static features of the code, such as the control flow graph, in order to identify malicious behavior. Dynamic analysis runs the executable in a controlled environment, and then analyzes its runtime behavior. In contrast, statistical analysis is agnostic to the code semantics and the dynamic behavior of the binary executable. Instead, it operates directly on the binary by computing different statistical features of the actual content without disassembly. These statistical features are then used to identify malware, usually through machine learning and classification. Compared to other malware analysis approaches, statistical analysis can be relatively fast and scalable. While static, dynamic, and statistical analyses all have trade-offs and limitations, statistical analysis has a distinct advantage in speed and complexity that makes it suitable for the goal of this paper: a triage framework that can operate on a large quantity of daily malware samples.

A large portion of the new malware samples introduced every day is composed of new variants of already-observed malware. Two malware samples are considered variants if they exhibit similar malicious dynamic behavior when executed. Malware variants usually share large portions of the code, which is reflected in the machine-level code present in the executable. This notion of binary-content-level similarity among variants of malware can be used to detect new variants of previously-seen malware. Different file similarity-based malware detection techniques have been proposed [10, 12, 19, 29, 32, 34, 35]. These techniques are based on the assumption that similar malware share similar code and file structures, and require uncompressed and unencrypted malware code for effective detection. However, easily-available polymorphic engines and generic packers are the most prevalent techniques used for generating new malware variants. These tools obfuscate the variants of malware code by adding random modifications (noise), and prevent the signature-based detection. It has been observed that malware packers commonly use weak encryption and compression algorithms, and such operations preserve certain statistical and structural properties at the binary level, which can be used to detect similarity without unpacking [9]. Most of these similarity detection techniques, including [9], are based on different versions of N-gram feature extraction. The N-gram based operations are computationally expensive, and usually not suitable in large triage systems. Some of the techniques that only extract features from the executable headers (PE structure) are fast, however, they are less precise because they ignore the content-level similarity of code. Moreover, the N-gram-based approaches are focused on the matching-substrings, regardless of their posi-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA
Copyright 2013 ACM 978-1-4503-2015-3/13/12 ...$15.00.
http://dx.doi.org/10.1145/2523649.2523682

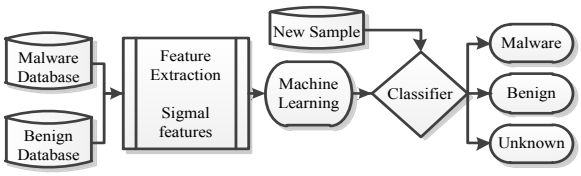

Figure 1: SigMal overview.

tion. They do not detect the similarity of the spatial structure. However, in case of the executable similarity, detecting the similarity of the spatial structure of the content is very important. This is because the executables are inherently structured, almost a replica of its memory structure, which includes the spatial structure of the machine-level code residing in the code section. The content-hashing-based techniques, such as peHash [35] and Piecewise-Hash [14], can capture some level of structural characteristics of an executable. However, they are susceptible to small noise introduced into the executables. Specially, while detecting similarity among malware executables, a robust similarity detection method needs to be noise resistant because of the random modifications introduced by polymorphic engines and packers. In Image Processing, the image similarity detection techniques deal with a very similar problem, i.e., finding similarity among spatially structured content, while remaining less susceptible to noise. We apply similar techniques to detect similarity among malware executables.

In this paper, we propose SigMal, a fast and precise signal processing-based malware similarity detection technique suitable for a large-scale malware triage. SigMal features are based on both the PE structure and the binary content of the executable. Compared to existing similarity detection features, the signal processing-based features are less susceptible to the random modifications (noise) introduced by a polymorphic or metamorphic engine.

The binary content of the executable can be considered as a one-dimensional signal. We first transform this one-dimensional signal into a two-dimensional digital image, so that we can computationally extract robust signatures from this image. These signatures are then used for the malware similarity detection. The intuition here is that similar malware produce similar image texture patterns. A texture is a set of metrics representing the spatial arrangement of color or intensities in an image, in our case it is the content of the executable. The similarity features are extracted from these texture-patterns. This approach of binary transformation is similar to the malware visualization technique proposed in [20], in which the one-dimensional signal of an entire binary is "reshaped" into a two-dimensional gray-scale image. However, extracting the feature from the entire binary can be problematic. If a malware reorders its internal sections, the resulting features will be significantly different. In addition, only the section or the sections of the binary that contain the malicious code are likely to be similar among similar malware. Similarity in other sections, such as the resource section, does not necessarily imply similar malware. We will see that this approach, as proposed in [20], performs the worst when compare to existing static malware detection techniques. Our method takes advantage of the information available in the PE structure of the executable to infer the "important" sections that are most likely to contain the malicious code. From these sections, we compute separate texture features for similarity detection. These features are then combined to form a larger feature-set. This approach considerably increases the dimensionality of the feature, in our case by three times, which makes it challenging to make it work in a large-scale triage system. However, we use a scalable *Balltree*-based fast nearest-neighbor technique, which can scale our malware similarity detection method to millions of malware. In particular, SigMal's feature extraction is based on the Gabor wavelets-based filters, which are commonly used in the large-scale content-based image retrieval systems. It has been shown that such signature features perform well in identifying similar images in a web-scale dataset of natural images (110 million) [7]. We will show that these features are also effective in finding similar malware samples in a large set of executable samples.

An overview of the system is presented in Fig. 1. SigMal takes sets of known malware samples and known benign samples, extracts features and builds a classifier model. The system performs per-sample analysis of new input executable samples and identifies them as *malware*, *benign*, or *unknown*. Our method is suitable in a setting where large numbers of samples are received every day, and the samples need to be quickly classified as either benign or malicious. This is indeed a common scenario in the computer security industry, research organizations, and government institutions. Fast static detection of similar malware can avoid repeated analysis, and save large amount of resources required to perform more complex analysis, such as dynamic analysis. By directly operating on the packed binary, our technique avoids the costly operation of preliminary unpacking.

We compared our approach with other static-feature-based malware detection approaches, specifically N-gram-based detection, PE-feature-based detection, and control-flow-graph-based detection. We show that our method outperforms all of these methods in terms of precision. In the 10-fold cross-validation experiment on 103,808 samples, SigMal could achieve average per-sample query response time of 47.95 milliseconds and 99% detection precision. We also performed a large-scale experiment on a dataset of 1.2 million samples observed during the period of 3 months.

The main contributions of this paper are the following:

- We propose an efficient and scalable signal processing-based malware similarity measure that can detect malware with high speed and precision.
- We evaluated its performance with respect to existing static malware similarity approaches using a dataset of 51,058 malicious and 52,750 benign samples. We show that our method outperformed all other methods in terms of precision.
- We evaluated SigMal on 1.2 million recent samples, both packed and unpacked, observed over three months. We demonstrate that SigMal can classify 50% of the incoming samples with more than 99% precision.
- We show that SigMal could have detected, on average, 70 malware samples per day before any antivirus vendor detected them.

2. SIGNAL PROCESSING-BASED FEATURES

In this section, we describe the signal processing-based feature extraction. The executable binary content is taken

as a one-dimensional *signal*, represented as a vector of bytes. This vector is "reshaped" into a two-dimensional matrix of fixed width d. In other words, the first d bytes go to the first row of the matrix, and the n_{th} group of d bytes goes to the n_{th} row of the matrix. This approach is similar to the malware visualization technique proposed in [20]. The two-dimensional matrix, after a necessary padding, is now considered as a digital gray-scale image, which is "resized" into a square image of width s for efficient computation. Image-texture-based signature features are extracted from this transformed image. The transformed image captures the short-range correlations of the signal as the texture with horizontal orientation, and the long-range correlations as the texture with vertical orientation. This is because the horizontally adjacent pixels in the image correspond to the adjacent bytes in the binary (short-range), and the vertically adjacent pixels correspond to the bytes spaced by a multiple of width d in the binary (long-range). Instead of using a variable width d, as proposed in [20], we use a fixed width transformation to maintain the consistency in the texture produced as a result of this transformation. As long as the same width is used for each executable, the choice of the width does not affect the similarity of the texture produced among similar executables. Based on this similarity of the textures, we would like to computationally obtain a signature that captures this similarity. This problem has been widely explored and texture features have been extensively used in the field of image processing for content-based image retrieval [18], scene classification [21,33], and large-scale image search [7]. Even though some sections of the image look completely random to a human eye, texture features can capture a signature of the randomness that is suitable for similarity detection. The methodology for computing the texture-based features is described in the next section.

2.1 Feature computation

We compute the features based on GIST descriptors [21]. The descriptors are computed by first filtering the image in various frequency sub-bands and then computing local block statistics on the filtered images.

Let $I(x, y)$ be the image on which the descriptor is to be computed. The GIST descriptor is computed by filtering this image through a *filter bank* of Gabor filters [6]. These filters are band-pass filters whose responses are Gaussian functions modulated with a complex sinusoid. The filter response $t(x, y)$ and its Fourier transform $T(u, v)$ are defined as:

$$t(x,y) = \frac{1}{(2\pi\sigma_x\sigma_y)} exp[-\frac{1}{2}(\frac{x^2}{{\sigma_x}^2} + \frac{y^2}{{\sigma_y}^2}) + 2\pi jWx] \quad (1)$$

$$T(u,v) = exp[-\frac{1}{2}(\frac{(u-W)^2}{(\sigma_u)^2} + \frac{v^2}{(\sigma_v)^2})] \quad (2)$$

where $\sigma_u = 1/2\pi\sigma_x$ and $\sigma_v = 1/2\pi\sigma_y$. Here, σ_x and σ_y are the standard deviations of the Gaussians function along the x direction and y direction. These parameters determine the bandwidth of the filter and W is the modulation frequency. (x, y) and (u, v) are the spatial and frequency domain coordinates.

We create a filter bank by *rotating* (orientation) and *scaling* (dilation) the basic filter response function $t(x, y)$, resulting in a set of k self-similar filters. In image processing terminology, the functions modeling the filters are often referred to as Gabor Wavelets [6]. The image is then filtered using this filter bank to produce k filtered images.

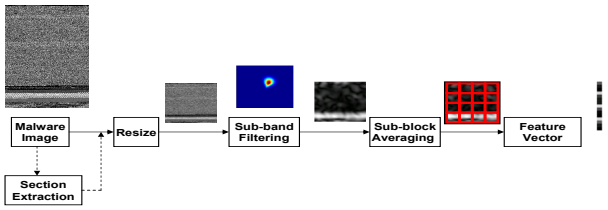

Figure 2: Texture-based feature computation. The malware image is first resized, then filtered, followed by sub-block averaging to form the feature vector.

Each filtered image is further divided into $B \times B$ sub-blocks and the average value of a sub-block is computed and stored as a vector of length $L = B^2$. This way, k vectors of length L are computed per image. These vectors are then concatenated to form a kL-dimensional feature vector called GIST. In this paper, we used 20 self-similar filters derived from different rotations (orientations) and scales. In particular, we used 3 scales, out of which the first two scales have 8 orientations and the last one has 4. We used $B = 4$ to obtain a 320-dimensional feature vector. These parameters have been previously used for image search [7]. While computing the GIST descriptor, it is a common pre-processing step to resize the image to a square image of dimensions $s \times s$ [7]. In our experiments, we used the parameters $d = 256$ and $s = 64$ and observed that choosing different values of d did not affect the results, while choosing a value of s less than $s = 64$ decreased the accuracy. Larger value of s increased the computational complexity, however, because of the sub-block averaging, this did not effectively strengthen the signature feature. In Fig. 2, we illustrate how a feature vector is calculated by filtering the image with a single filter. This process is repeated for all other filters in the filter bank, and the feature vectors thus obtained are concatenated to form the full GIST descriptor.

2.2 Section-aware feature extraction

A normal executable file structure consists of many sections, such as code or data. Apart from some special types of malware executables, such as COM files, usually all malware executable files are also structured in this way. The true malicious behavior of malware is represented by the section containing the code, which executes the actual malicious activities. The previously-proposed algorithm [20] ignores this critical information and generates malware fingerprints from the entire binary. This approach may cause the code section similarity to be out-weighted by dissimilarities of other sections, such as the resource section, and fail to identify a variant. Generic packers and installers usually share resources, such as icons and extraction routines. With relatively small packed executables, these resource similarities will produce false-positive similarity detection.

SigMal takes advantage of the internal structure of an executable. The texture properties of an executable section depends on its content type. An executable can contain different types of contents such as code, packed and unpacked data, and other resources, which produce corresponding different types of GIST filter responses. The texture feature extracted from the entire binary captures the spatial struc-

```
Data: PE Executable
Result: A list of important sections
Map sections into raw binary file;
if overlapping section exits then
    resize section to make it contiguous with adjacent sections;
end
if .text executable section exists then
    if is the largest section then
        Result.append(.text section and the second largest
        section);
    else
        if .text section is writable then
            Result.append(two largest sections);
        else
            Result.append(.text section and the largest section);
        end
    end
else
    if any non-writable executable section exists then
        Result.append(this section and the largest section);
    else
        Result.append(two largest sections);
    end
end
```

Algorithm 1: Finding important sections.

ture of these sections. To capture more localized signal information from the important regions of the binary, we extract separate GIST descriptors from each "important" section of the executable. We say a section is an important section if it is likely to contain the code (packed or unpacked) of the executable. These sections get more weight because the GIST descriptors are computed on a per-section basis and concatenated to form the final feature set. To extract separate GIST descriptors from the important sections of the binary, we first need to identify them. One way to extract this information is to use the PE structure information. However, especially in case of malware binaries, the mapping of section information from the PE structure to the binary file data is not always reliable. For example, code sections can be compressed, relocated, or obfuscated, and their size can be spuriously specified as an arbitrarily large value. We use heuristics to find boundaries of the important sections of an executable within the raw binary data, and select two important sections using the heuristics presented in Algorithm 1. These heuristics are loosely based on [9]. The section type, section flags, and the size of the section are used in the heuristics. For example, small writable executable sections are usually common loader sections instead of actual malware code. The heuristics give less priority to such sections. We also experimented with heuristics that prioritize the section selection based on entropy instead of size. However, we did not use this method because the results were slightly less precise.

3. METHODOLOGY

3.1 Feature matching

We use the Euclidean distance metrics between feature vectors to find the nearest-neighbor sample in the learning dataset. For each unknown sample q, we perform a nearest-neighbor query on the malicious and the benign datasets, which returns two distances, say d_m and d_b, respectively. If both distances are very similar, we cannot say for sure whether the sample q is malicious or benign. This confusion is even more pronounced when d_m and d_b themselves are large (i.e., when the similarity to the dataset is weak). In order to model this nature of the distances, we introduce a detection confidence parameter c as described in Eq. 3. We mark the sample q as *unknown* if the value of c is less than a certain threshold (i.e., the absolute difference of distances d_m and d_b is not large enough).

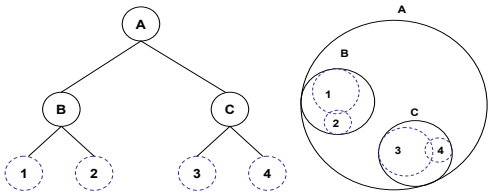

Figure 3: Illustration of a Balltree: (a) A Binary Tree (b) Corresponding Ball Tree representation.

$$c = \frac{|d_m - d_b|}{\sqrt{d_m^2 + d_b^2}} \tag{3}$$

Here, the value of detection confidence c varies from 0 to 1, such that the value of 0 implies no confidence (unknown), and the value of 1 implies the highest confidence. In case of a faulty training set, where the same sample is present in the both malware and benign datasets, the value of c for that sample will be undefined ($d_m = d_b = 0$).

3.2 Fast nearest-neighbor

Because of the high dimensionality of the feature SigMal, brute-force nearest-neighbor search is computationally expensive. For the efficient nearest-neighbor search in a high-dimensional space, we use *Balltree* data structures [22]. A Ball, in n-dimensional Euclidean space R^n, is defined as a region bounded by a hyper sphere. It is represented as $B = \{\underline{c}, r\}$, where $\underline{c}$ is an n-dimensional vector specifying the coordinates of the ball's centroid, and r is the radius of the ball. A balltree is a binary tree where each node is associated with a ball. Each ball is a minimal ball that contains all balls associated with its children nodes. The data is recursively partitioned into nodes defined by the centroid and the radius of the ball. Each point in the node lies within this region. Fig. 3 shows an illustration of a binary tree, and a balltree over four balls (1,2,3,4). Search is carried out by finding the minimal ball that completely contains all its children. This ball also overlaps the least with other balls in the tree. For a dataset of M samples and dimensionality N, the query time grows approximately as $O(N \log(M))$ (as opposed to $O(NM)$ for a brute force search).

3.3 Comparison

We compared our signal processing-based malware similarity algorithm with three popular malware detection methods.

3.3.1 N-gram based detection

The N-gram signature based approach has been extensively used in the previous works of malware similarity [3, 10, 29]. The N-gram signature of a string is a set of all substrings of the string with a length n. In the case of a binary executable, the N-gram signature is usually computed on the string of its raw bytes, or disassembled instructions.

Various similarity measures have been proposed to compare the similarity between N-gram signatures [3, 10, 29]. In

our case, we used the Jaccard similarity metric, which has been extensively used in previous work [4, 10, 25]. Jaccard similarity is the number of common occurrence of N-grams in both N-gram signatures with respect to the total number of unique N-grams. More precisely, given two N-gram signatures (sets of N-grams) s_a and s_b:

$$J(s_a, s_b) = \frac{s_a \cap s_b}{s_a \cup s_b}$$

.

3.3.2 PE structure-based detection

Several PE structure-based malware detection techniques have been proposed [30, 31, 35]. They are mainly based on the structural features of executables, which are directly extracted from the PE file structure with low computational overhead. In a recent work, only seven PE-based features were used to produce detection results comparable to methods using tens of features [26]. We used this method for our evaluation experiment. As suggested by [26], we chose the J48 decision-tree method to build models for the classes.

3.3.3 Control-flow graph-based detection

Control-flow graph (CFG) have been extensively used for detecting similarities between executables [5, 8, 15]. Kruegel et al. [15] extracted CFGs from network streams and detected polymorphic worms by identifying structural similarities. In this approach, the control flow graphs are decomposed into a set of subgraphs of fixed size k. Worm detection and classification occurs by identifying the prevalence of k-subgraph features between worm-like executable content and unknown executable content. We used the tool made available by [15] for the comparison experiment. The similarity measure between two samples is calculated as below:

$$CFG\ similarity = \frac{number\ of\ matching\ subgraphs}{total\ number\ of\ subgraphs}$$

.

As the focus of this paper is on packer-agnostic approach of malware detection, in all of our experiments, malware samples were used without unpacking. This includes the control-flow graph-based detection experiment. One can argue that this may not be a fair comparison, as the control-flow graph extracted from a packed executable is usually representative of the unpacking routines only, rather than the actual malware code. However, we are interested in the comparison of the effectiveness of these algorithms when applied directly on packed malware. The weak results of the CFG-based approach in this case show that the method is not suitable for packed executables.

4. DATASET

We prepared three datasets for the evaluation of detection methods.

Benign dataset

For the first dataset, we collected benign executables from three different sources, a fresh Windows XP SP2 install, the ZDnet Software Directory, and the National Software Reference Library (NSRL) maintained by NIST. Our benign dataset contains 377 executables from a fresh Windows installation, the top 3000 most popular downloads from ZDnet Software Directory, and 49,373 software binaries from NSRL. We consider the most downloaded software from ZDnet Software Directory as benign. We assume that a malicious software sample does not appear in the most-downloaded list of a well-reputed site. All of these samples were also cross-checked with VirusTotal [2] to make sure that none of the antivirus vendors flag them as malicious.

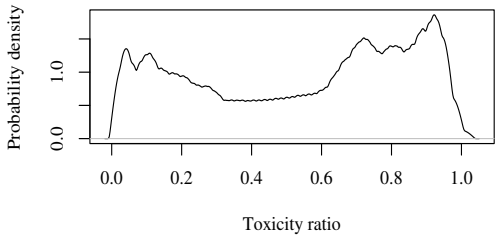

Figure 4: The *toxicity ratio* distribution of 1.2 million malware samples.

Malicious dataset

Malware dataset creation is a difficult problem [17, 27]. We built our second dataset from the executable samples submitted to *Anubis* [1]. Anubis is a dynamic malware analysis platform that receives thousands of samples for analysis everyday. We obtained the malware samples submitted to *Anubis* in 2011, along with the latest antivirus labels associated with each sample. Antivirus labels are provided by VirusTotal [2], which includes labels from different antivirus vendors for each submitted sample. To each sample, we associate a *toxicity ratio* τ, where τ is the ratio of the total number of antivirus vendors that detected the sample as malicious to the total number of antivirus vendors checked by VirusTotal. The density distribution of the *toxicity ratio* of 1.2 million malware samples is shown in Fig. 4. It clearly shows that the ratio is concentrated either towards a smaller value or towards a larger value. This means, either only a few antivirus vendors are likely to label a sample as malicious (sometimes spurious) or almost all vendors are likely to label it as malicious. From this set of samples, we built the malicious dataset by taking samples with $\tau > 0.9$, that is, the set of binary samples which were flagged as malicious by 90% of the antivirus vendors. Some samples were missing results from some of the antivirus vendors. To maintain an effective large majority, we discarded those samples that have results from less than 30 antivirus vendors (out of 49). This consensus by a majority of antivirus vendors is a result of many human experts who have analyzed the sample (or similar samples) and concluded it to be malicious. Moreover, to have a stronger confidence on this consensus, we chose older samples observed in 2011 with their latest antivirus labels obtained after a year.

The samples-per-malware-family metric of this type of datasets are usually skewed because of the abundance of some widely popular malware families, which are usually more frequently submitted to such public analysis platform, such as Anubis. Therefore, out of the large malware dataset observed in 2011, we only took at most 100 samples per malware family. The dataset after this selection contains 51,058 unique malware samples representing 15,089 malware families.

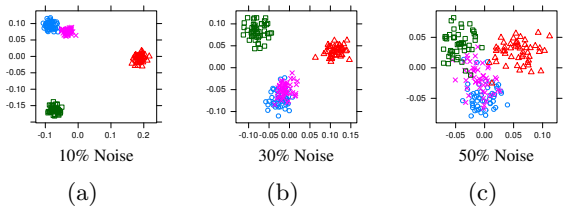

Figure 5: Feature robustness against noise. Similar symbols represent the variants from the same seed binary.

Real-world dataset

To evaluate the performance of the detection methods on real-world malware, we used the recent-feed of malware samples submitted to Anubis over three months, starting from November 2012. This dataset contains 1.2 million samples.

5. EVALUATION

5.1 Experimental setup

The signal processing-based features of both malware and benign datasets are computed and stored in memory in the *Balltree* data structure. In the N-gram-based detection, we used a bit-vector approach to encode the N-gram signature, as proposed in [10]. This transforms the Jaccard computation into more CPU-friendly logic operations, and speeds up the computation by many orders of magnitude. Like previous N-gram based works [9, 23], we set n=2 and n=3.

5.2 Experiment against noise

We performed a synthetic experiment to test the robustness of the feature against small modifications (noise) introduced into the sample. We first generated four *seed binaries* containing 200KB random bytes. We chose to use random bytes such that we do not make any specific assumption on the data pattern. From each of these seed binaries, we generated synthetic variants by introducing random noise to the original binary. Note that the differences among the variants are even more pronounced due to these random modifications. We computed signal processing features from these synthetic variants and visualized using Multidimensional scaling (MDS), as shown in Figure 5. We can see that the features can be used to cluster the variants even when 50% of the original bytes are randomly modified. In the case of malware binaries, such noise may be introduced by a polymorphic or metamorphic engine.

5.3 Detection

At first, we analyzed the classification strength of the SigMal features using various machine learning classifiers. Single nearest-neighbor distances based classifier provided the best result. In the next step, using Nearest Neighbor (NN) classifier, we performed a comparative evaluation of SigMal with existing detection methods. We used the standard 10-fold cross-validation process on the same dataset for all methods. The evaluation dataset used in this experiment is described in Section 4.

We performed the precision-recall analysis by varying the threshold value t of the detection confidence parameter c (introduced in Section 3.1). For each testing sample, SigMal returns two nearest-neighbor distances d_m and d_b corresponding to the malware and benign training sets, respectively. Fig. 6 shows the distribution of these distances computed from the 10-fold cross-validation experiment. A sample with shorter distance to the malware dataset than to the benign dataset falls in the upper left section of the graph. The area inside the dotted line represents the confusion area given by the inequality $c < t$, when the threshold value is chosen as $t = 0.1$. Samples within this confusion area are marked as *unknown* because their detection confidence value (c) is not large enough. A change in the threshold value t changes the performance of the detection. Higher values of threshold t produce more precise results by widening the confusion area, while reducing the recall rate.

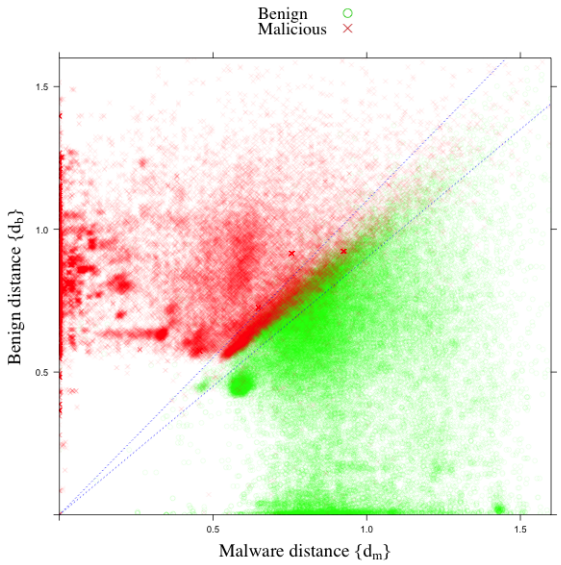

Figure 6: The nearest-neighbor distance distribution of 100K samples from the 10-fold cross-validation experiment of SigMal. For each point, X-axis represents the nearest-neighbor distance to malware dataset, and Y-axis represents the the nearest-neighbor distance to the benign dataset.

Both N-gram-based method and CFG-based method provide similarity measures instead of distance measures. To perform the precision-recall analysis, we vary a threshold s, which, in this case, is a threshold for the minimum similarity. If a resulting value of the similarity measure for a query sample is less than the threshold parameter s, we consider the value to be too weak, and the sample is marked as *unknown*. In case of the PE structure-based detection, we vary the output class probability of the decision tree classifier.

The precision-recall analysis of all detection methods is presented in Fig 7. One can see that our method has the best overall performance. We achieved very high precision (99.66%), while still maintaining a good recall rate of about 50%. When only the code-section of the executable is used to extract the features, the performance degrades. This suggests that the overall layout of different sections of an executable is also an effective feature for similarity detection. As reported in previous work [30, 31], PE structure-based methods produced overall good precision and recall rates. However, it could not improve the precision above 98%. In cases where greater recall is important, our method still has

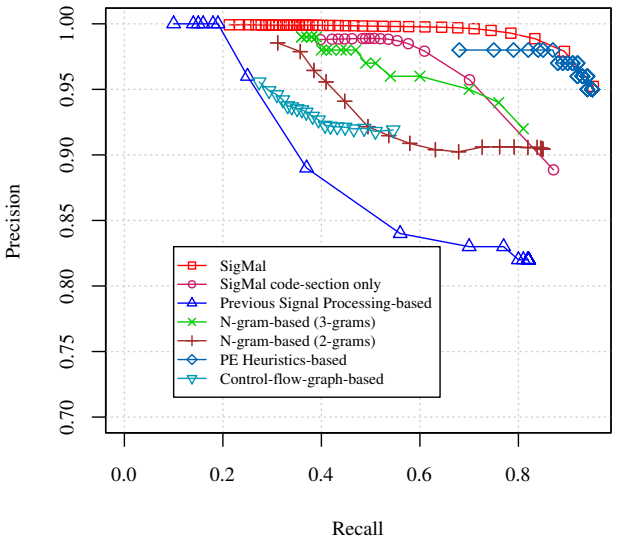

Figure 7: Comparison of malware detection algorithms.

second best precision compared to the rest of the methods. The N-gram-based method (when $n = 3$) has relatively good overall performance. However, the computational overhead is high. Because the packed samples were not unpacked, as expected, the control-flow-graph-based method did not produce good results.

There is the possibility of the classifier being biased detecting packed and unpacked executables, instead of malicious and benign. To test this, we packed our benign dataset using three popular executable packers: UPX, WinUpack, and NSPack, and used this dataset as the benign dataset for the 10-fold cross-validation experiment. The results were still comparable to the result with normal benign samples.

5.4 Performance

In this section, we compare the time and space efficiency of the algorithms. We focus on the time required by two main steps: building the features and detecting the similarity. We measured the memory requirements to compare the space efficiency. For accurate measurement of memory and time usage, all of the feature-extraction processes were modified from multi-threaded implementations to single-threaded implementations in this particular experiment. All performance experiments were done in the same computational environment (Linux 3.2.0-35 machine, Intel i7 3.33GHz/12GB).

We measured the computation time required by the feature extraction of ten thousands samples for each algorithm. The average time required to compute these different types of features are presented in Table 1. Since the PE-heuristics method inspects only the header part of the executable, it is the fastest among all. SigMal feature extraction is about five times faster than the CFG feature extraction and seven times faster than the N-gram feature extraction. We also measured the average per-sample space requirements for storing raw bytes of the features in the memory. Again, the PE-heuristics method requires the least amount of space to store features, since its feature dimensionality is the smallest. SigMal features require two times less memory compared to N-gram features (2-grams) and 78 times less memory compared to CFG features.

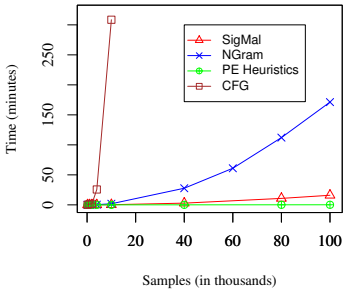

Figure 8: Query performance comparison. X-axis is the number samples used in each 10-fold experiment. Y-axis is the average of the total query time in each fold of that experiment.

Table 1: Average per sample feature extraction time in seconds and per sample memory requirements in kilobytes.

	SigMal	*N-gram*	*PE-heuristics*	*CFG*
Time	0.0265	0.1965	0.0024	0.1379
Space	3.783	8.000	0.0664	297.745

To evaluate the scalability of the algorithms, we performed the 10-fold cross-validation experiments using datasets of increasing number of samples, such that both the training samples and the testing samples are increasing in each experiment. We chose this option to resemble a real-world scenario, where both the number of new known malware samples (training set) and the number of samples to be inspected (testing set) are continuously increasing. For each 10-fold experiment, we computed the average of the total time required for the detection query in each fold. The results are presented in Fig. 8. We can see that our method is easily scalable to hundreds of thousands of samples. In a 10-fold cross-validation experiment on a dataset of 100,000 samples (80,000 in the training set, 20,000 in the testing set) average per sample query response was 47.95 milliseconds. The quadratic increase in the query response time of N-gram-based and CFG-based approaches is primarily because of the $O(NM)$ computational cost of the Jaccard-similarity comparisons. As expected, the decision-tree-based PE-heuristics method is the fastest among all.

5.5 Real-world experiments

We showed that our method works well in a limited dataset of old samples. Many of the previous works were also evaluated using a similar dataset. However, we wanted to evaluate their performance when applied to a large dataset of recent real-world samples. Results from such datasets can demonstrate the true applicability of a method.

5.5.1 *Experimental setup*

We observed that when an old malware dataset was used as the ground truth for the detection of recent malware, as expected, the detection performance was poor. When we combined both old and new malware dataset, the query response time of SigMal significantly increased. This is be-

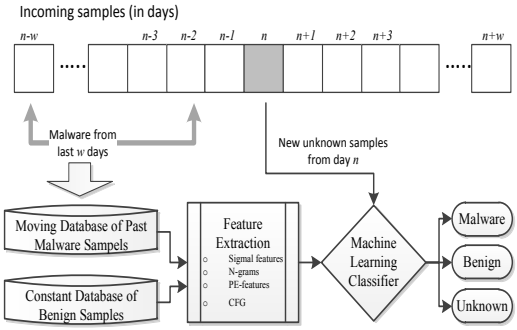

Figure 9: Overview of the sliding window experiment on the real world samples.

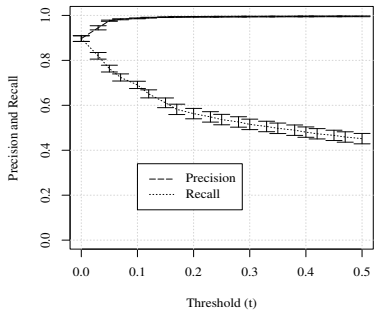

Figure 10: Precision and Recall of the SigMal detection on the real-world samples observed by Anubis in December 2012 and January 2013. The figure represents the mean values of the daily results and the standard error.

cause of the increased search-space for the nearest-neighbor query. The scalability problems with other methods were even more critical while using larger ground truth datasets. Because of this scalability problem, we prepared a fresh malware ground truth on a daily basis using the malware samples observed in the recent past. The overview of this approach is presented in Fig. 9. More precisely, for the nth day's experiment, we built a dynamic malware dataset by taking the set of malware samples observed in the past time window of w days (from day $(n-w)$ to day $(n-1)$). We can infer from the *toxicity* density distribution (Fig. 4) that the confidence starts to build up at $\tau = 0.6$. Hence, for the recent samples, we used a less conservative value of $\tau > 0.6$ for building the dynamic training set. We use the samples submitted on day n as the testing set for the nth day experiment. For the comparison of the SigMal detection results, we need to label each of these incoming samples as benign or malicious using recent antivirus labels. However, we noticed that few antivirus vendors falsely detect a benign sample as malicious, if packed with some commonly available executable packer, such as Winpack. Again from the toxicity graph, we can see that this confusion sustains up until $\tau = 0.3$. To avoid including such spurious labels, we consider $\tau <= 0.3$ as a low confidence value for the result evaluation, and exclude it in our precision-recall analysis. To find the optimal sliding time window in terms of speed and accuracy for collecting the ground truth, we performed a set of experiments using different values of w, ranging from 7 days to 60 days. The precision did not improve significantly when the time window was greater than 30 days. Therefore, we chose the time window w as 30 days for the rest of our daily experiments.

As evident in the performance experiments in Section 5.4, N-gram-based and CFG-based methods have a high computational cost. For example, with the 30 days sliding-window dataset, the CFG-based method required several days to complete a single sliding window experiment even with a parallelized implementation on a 24-core 96GB machine. Hence, this part of the comparison experiment on the real-world dataset is limited to a few days.

5.5.2 Results

The precision and recall performance of the sliding window experiments are presented in Fig. 10. We can see that at $t = 0.33$, more than 50% of the recent daily samples can be accurately detected as malicious or benign with a precision of 99.5% (standard error 0.000835). This can essentially reduce about 50% of the resource requirements of a triage system by avoiding further, more expensive, analysis.

Fig. 11 presents the detection results of all methods, when applied to the sliding window dataset (i.e., samples from the month of November 2012 as the training set, and the samples from December 1st 2012 as the testing set). One can see that the performance of other methods is less impressive in terms of precision. The PE-heuristics-based method and CFG-based method have a relatively larger recall. However, they do not produce high precision results. In case of the CFG-based method, its computation cost is another critical factor that makes it unsuitable for large-scale malware triage.

5.5.3 Evaluation with the current AV labels

Notice that the detection results were checked with the VirusTotal results obtained at the time of the submission to Anubis. However, there is a possibility that some malware may not have been detected by the majority of antivirus vendors at that time. Moreover, antivirus vendors may not eventually detect all malware. Hence, an accurate analysis of such false positives is difficult. Here, we are only interested in how many malware samples SigMal could have detected that AV vendors missed at the time of submission, but later identified them as malware. We used an old dataset observed in 2011, for which we have old antivirus labels retrieved from VirusTotal during the time of the submission to Anubis. We retrieved the latest antivirus labels for these samples. We performed a simulated daily sliding-window experiment on this dataset and re-evaluated our results with the updated antivirus labels. We found out that SigMal could have detected, on average, 70 malware samples per day before any antivirus vendor detected them as malicious.

5.6 Limitations

In this section, we discuss the limitations of our approach. Because our approach relies on instance-based learning, its main limitation is that it can only detect malware similar to what has already been observed. It cannot detect a zero-day malware that is structurally dissimilar from the previously-seen malware. This is a generic problem with any similarity-based malware detection system.

Malware can infect a benign executable by patching and embedding malicious code into it. If the embedded content is very small relative to the actual benign file-size, then the

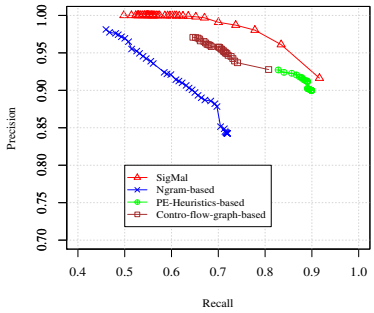

Figure 11: Comparison of malware detection methods with a live malware feed (2012-12-01).

infected file is likely to be considered similar to the benign file. When we manually analyzed the false negatives, we found that the majority of them were a patched system binary. For example, we found instances of TDSS rootkit that embeds its code into a small existing *.rsrc* section of Windows driver files, such as `netbt.sys`. We also found false positive cases of benign input files, which were not present in the benign dataset, but its infected version was present in the malware dataset. These problems are also generic to file-similarity based detection techniques. One countermeasure can be flagging an input file as suspicious, if it is very similar to a system file, but not exactly the same file.

If some strong cryptographic methods, such as AES, are used by packers, it will be hard to find statistical similarity among such encrypted samples. If the block chaining is enabled, this problem becomes almost impossible. Our approach is unable to identify similarity in such cases. However, the use of strong encryption itself can be considered suspicious, which malware writers would want to avoid. In fact, our experiment showed that the majority of antivirus vendors mark packed executables as suspicious or malicious, even if the original executables were benign.

As a targeted attack to our system, an adversary could insert large unused sections with random data into an executable. This may cause our heuristics to select wrong sections as important sections. Since it is likely that no previous sample matches with the random data, such samples will be considered *unknown*. In a more crafted attack, an adversary could embed code section of a benign executable as its largest section. This will generate the exact same feature vectors corresponding to those sections. However, the malicious code still needs to be embedded in the file to make the crafted executable malicious. Because of this, the part of the feature vector generated from the entire file will still be dissimilar from the feature vector of the actual benign executable. Hence, the crafted attack will not match completely with the benign executable.

6. RELATED WORK

6.1 Signal processing

The SigMal feature extraction method is similar to the malware visualization method proposed in [20]. However, our method differs in the way we extract signals from the sample using heuristics based on its PE structure. There are several signal similarity features depending on the type of signal (speech, image, video, seismic, and others). We restrict ourselves to image similarity features within the scope of this paper. Some of the common image similarity features are the Homogeneous Texture Descriptor (HTD) [28] and the Color Layout Descriptor (CLD) [28]. The HTD is a 96-dimensional texture-based image similarity descriptor, where an image is filtered over 48 sub-bands after which the mean and standard deviation on each filtered image are grouped to form the feature vector. The CLD, on the other hand, is a layout-based descriptor. The image is divided into an 8x8 grid and the mean value of every grid block is computed to obtain an 8x8 matrix. The Discrete Cosine Transform is then computed and the top energy coefficients form the feature vector.

6.2 Static malware similarity

Different approaches to static-feature-based malware analysis and triage systems have been proposed in the past. Most of them use N-gram-based feature extraction [3, 9, 10, 13, 23, 29]. The recent work from Jacob et al. [9] studied the preserved statistical similarity over packed binaries and proposed a packer-agnostic bigram-based similarity measure. N-gram-based approaches are less scalable because of the computationally expensive feature matching operation over relatively large dimensionality of the N-gram feature space. Jang et al. [10] proposed feature hashing to reduce the high-dimensional feature space in malware analysis and implemented feature hashing on N-gram based features. However, its evaluation was performed on the clustering of an unpacked malware dataset, and no benign samples were used to test the accuracy of the system. A malware phylogeny generation technique was proposed using N-perms to match every possible permuted code [11]. Other packer-agnostic approaches include the detection method based on features extracted from the PE file structure [24, 26, 30, 31, 35]. Although this approach is time efficient, results show that achieving high accuracy is difficult.

Other work on static-features-based detection requires unpacked code [5, 8, 10, 15]. With unpacked code, static features can be extracted from the disassembled code. Hu et al. [8] proposed function call graphs to implement an efficient nearest-neighbor search on a large graph database of malware. Kruegel et al. [15] proposed extracting control-flow-graph from network streams to detect worms. We have used CFG-based methods from [15] for the comparison of approaches in our evaluation.

7. CONCLUSIONS

In this paper, we presented SigMal, a fast signal processing-based malware similarity detection framework. It can operate on both packed and unpacked samples, avoiding the resource intensive unpacking process. We used heuristics based on PE structure information to improve the signal processing-based features. Our results showed that SigMal outperforms all existing static malware detection methods in terms of precision. Large-scale experiments on 1.2 million recent samples, both packed and unpacked, observed over three months demonstrated that our method can classify 50% of the incoming samples with above 99% precision.

8. ACKNOWLEDGMENTS

This work is supported by the Office of Naval Research (ONR) under grant N00014-11-10111, the Army Research

Office (ARO) under grant W911NF0910553, and the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537.

9. REFERENCES

[1] Anubis. http://anubis.iseclab.org.

[2] VirusTotal. http://www.virustotal.com.

[3] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Proc. of COMPSAC'04*. IEEE, 2004.

[4] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of NDSS'09*. Citeseer, 2009.

[5] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus Bulletin Conference*, 2004.

[6] J. G. Daugman. Complete discrete 2-d gabor transforms by neural networks for image analysis and compression. *IEEE Transactions on ASSP*, 1988.

[7] M. Douze, H. Jégou, H. Sandhawalia, L. Amsaleg, and C. Schmid. Evaluation of gist descriptors for web-scale image search. In *ACM International Conference on Image and Video Retrieval*. ACM, 2009.

[8] X. Hu, T. Chiueh, and K. Shin. Large-scale malware indexing using function-call graphs. In *Proc. of CCS'09*. ACM, 2009.

[9] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna. A static, packer-agnostic filter to detect similar malware samples. In *Proc. of DIMVA'13*. Springer, 2013.

[10] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proc. of CCS'11*. ACM, 2011.

[11] M. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1), 2005.

[12] A. Karnik, S. Goswami, and R. Guha. Detecting obfuscated viruses using cosine similarity analysis. In *Proc. of AMS'07*, pages 165–170. IEEE, 2007.

[13] J. Kolter and M. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7:2721–2744, 2006.

[14] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[15] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proc. of RAID'06*. Springer, 2006.

[16] M. Labs. Mcafee threats report: Second quarter 2013. Technical report, McAfee, 2013.

[17] P. Li, L. Liu, D. Gao, and M. K. Reiter. On challenges in evaluating malware clustering. In *Proc. of RAID'10*. Springer, 2010.

[18] B. S. Manjunath and W. Ma. Texture features for browsing and retrieval of image data. *IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI)*, 18(8), Aug 1996.

[19] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, N. Japkowicz, and Y. Elovici. Unknown malcode detection and the imbalance problem. *J. Computer Virology*, 5(4):295–308, 2009.

[20] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: visualization and automatic classification. In *Proc. of VizSec'11*, VizSec '11. ACM, 2011.

[21] A. Olivia and A. Torralba. Modeling the shape of a scene: a holistic representation of the spatial envelope. *Intl. Journal of Computer Vision*, 42(3):145–175, 2001.

[22] S. M. Omohundro. Five balltree construction algorithms. Technical report, International Computer Science Institute Berkeley, 1989.

[23] R. Perdisci and A. Lanzi. McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. *Computer Security Applications*, pages 301–310, Dec. 2008.

[24] R. Perdisci, A. Lanzi, and W. Lee. Classification of packed executables for accurate computer virus detection. *Pattern Recognition Letters*, 29(14), 2008.

[25] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proc. of NSDI*, 2010.

[26] K. Raman. Selecting features to classify malware. In *InfoSec Southwest*, 2012.

[27] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Proc of SP'12*, pages 65–79. IEEE, 2012.

[28] P. Salembier and T. Sikora. *Introduction to MPEG-7: Multimedia Content Description Interface*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[29] I. Santos, Y. Penya, J. Devesa, and P. Bringas. N-grams-based file signatures for malware detection. In *Proc. of ICEIS'09*, 2009.

[30] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo. Data mining methods for detection of new malicious executables. In *Proc of SP'01*. IEEE, 2001.

[31] M. Shafiq, S. Tabish, F. Mirza, and M. Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In *Proc. of RAID'09*. Springer, 2009.

[32] S. M. Tabish, M. Z. Shafiq, and M. Farooq. Malware detection using statistical analysis of byte-level file content. In *ACM SIGKDD Workshop CyberSecurity and Intelligence Informatics*, 2009.

[33] A. Torralba, K. Murphy, W. Freeman, and M. Rubin. Context-based vision systems for place and object recognition. In *Proceedings of ICCV*, 2003.

[34] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proc. BlackHat DC Conf.*, 2007.

[35] G. Wicherski. pehash: A novel approach to fast malware clustering. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.