Auto-Sign: An Automatic Signature Generator for High-Speed Malware Filtering Devices

Gil Tahan (Gilta@bgu.ac.il)

Chanan Glezer (Chanan@bgu.ac.il[1])

Yuval Elovici (Elovici@bgu.ac.il)

Lior Rokach (liorrk@bgu.ac.il)

Deutsche Telekom Laboratory at Ben Gurion University

Beer Sheva, Israel 84105

ABSTRACT

This research proposes a novel automatic method (termed Auto-Sign) for extracting unique signatures of malware executables to be used by high-speed malware filtering devices based on deep-packet inspection and operating in real-time. Contrary to extant string and token-based signature generation methods, we implemented *Auto-Sign* an automatic signature generation method that can be used on large-size malware by disregarding signature candidates which appear in benign executables. Results from experimental evaluation of the proposed method suggest that picking a collection of executables which closely represents commonly used code, plays a key role in achieving highly specific signatures which yield low false positives.

Keywords

Malware, Automatic Signature Generation (ASG), Intrusion Detection Systems

[1] Corresponding author, Telefax + 972-3-6440414

1. INTRODUCTION

The time interval from the release of new malicious software to the wild till the time that the security software/hardware companies detects the malware, analyze it, generate a signature and release an update to its clients is highly critical. During this time interval the malware is undetectable by most of the signature-based commercial solutions and thus it can easily propagate [1]. For that reason, it is highly important to detect a new malware as soon as possible and rapidly generate a good signature so that abundant signature-based solutions can be updated.

One way to protect organizations from malware is to deploy high-speed malware filtering appliances on the communication lines that connect the organization to the outside world. Such appliances are based on deep-packet inspection in real-time and thus support very simple signatures for detecting malware. Security appliances are an appealing solution because they require no local overhead to manage them, yet they work continuously to protect the enterprise [14].

This research focuses on automating the process of generating signatures to be installed on such appliances for known malware that needs to be filtered by the appliances. Various techniques have been proposed to derive malware signatures automatically, including among others: vulnerability-based signatures [1]; payload-based signatures ([7]; [19]); content sifting [17]; semantic-aware signatures [21]; The Amd algorithm [2]; Honeypot-based signatures ([8]; [15]; [18]), and polymorphic content-based signatures [14][26]. These studies examine code by matching and analyzing the distribution of string patterns in communication packets; classifying unsuccessful connections; and modeling invariant code structures. Such signatures were tested and reported to be effective for small-sized malware (usually worms) [1, 7, 19]. Nevertheless, the employed approaches for signature generation ignore the fact that many

types of malware appear as full-fledged executable and therefore contain a significant portion of repetitive code emanating from code generators, development tools and platforms.

Considering the fact that signature-based systems do not account for such large common code segments, the quality of existing signature generation mechanisms is degraded. Such quality is measured using sensitivity (low false negative for malware) and specificity (low false positive for innocuous traffic) measures.

In order to address the problems stated above, this research proposes and evaluates a signature generation technique, termed *Auto-Sign*, for generating signatures which can be used by Network Intrusion Detection and Protection Systems (NIDS/NIPS) operating as malware filtering devices [27]. For such devices, Auto-Sign needs to generate a very simple signature that a network appliance can use for filtering malware in real-time. A very simple signature is actually a string of bytes or a simple regular expression of bytes at the most. To improve its precision, Auto-Sign employs an exhaustive and structured technique which first sanitizes malware from segments of common benign code and only then generates unique signatures which can be later used for detection of malware traffic.

The scope of this research is on tackling malicious code in the form of adware, spyware, Trojans, and viruses. Auto-Sign was tested also on large, full-fledged malicious applications and not necessarily on short stream-based malware where common code is not relevant.

Auto-Sign raises many questions with regards to various aspects of the proposed technique. In this research we were interested in finding the optimal length and selection criteria of a signature among several candidates in order to minimize false positives as well as the size and type of the training set in order to minimize false positives. This research describes the Auto-Sign technique and a set of experiments which were performed on a collection of malicious and benign executables. Auto-Sign is capable of automatic signature generation as part of the eDare framework [22] which offers "malware filtering as a service"

and is targeted for Network Service Providers (NSP), Internet service Providers (ISP), small and large enterprises.

2. RELATED WORK ON AUTOMATIC SIGNATURE GENERATION (ASG)

Automated signature generation for new attacks of this type is extremely difficult due to the following reasons [18]. In order to create a malware signature, we must identify and isolate malicious traffic from benign traffic, which is not an easy task under all circumstances due to sophistication of hacking techniques. The difficulty of signature generation for new attacks is also difficult since as soon as the signature generation methods are known to the attacker and especially the statistical ones, he may be able to defeat them by using statistical simulability as presented in [28][29]. The signature must be general enough to capture all instances malicious traffic while at the same time specific enough to avoid overlapping with the content of benign traffic in order to reduce false positives. This problem has so far been handled in an ad-hoc way based on human judgment. As a case in point, current rule-based network Intrusion Detection Systems (IDS) can do little to stop zero-day worms [19]. They depend on content, protocol-anomaly and behavioral signatures which can only be generated in a delay after the malware has been launched and already created substantial damage.

Several approaches have been employed in order to expedite the process of signature generation for effective containment of worms. *Autograph* [7] stores source and destination addresses of each inbound unsuccessful TCP connection it observes. Once an external host has made unsuccessful connection attempts to more than s internal IP addresses, the flow classifier considers it to be a scanner. All successful connections from an IP address, flagged as a scanner, are classified as suspicious, and their inbound packets written to the suspicious flow pool. Autograph next selects the most frequent byte sequences across the flows in the suspicious flow pool as signatures. At the start of a worm's propagation, the aggregate rate at which all infected hosts scan the IP address space is quite low. Because Autograph relies on

overhearing unsuccessful scans to identify suspicious source IP addresses, early in an epidemic an Autograph monitor will be slow to accumulate suspicious addresses, and in turn slow to accumulate worm payloads. To address this problem Autograph uses a tattler that, as its name suggests, shares suspicious source addresses among all monitors, toward the goal of accelerating the accumulation of worm payloads.

Honeycomb [8] tries to spot patterns in traffic previously seen on the honeypot. Honeycomb overlays parts of flows in the traffic and use a *Longest Common Substring* (LCS) [4] algorithm to spot similarities in packet payloads. [18] followed-up this work by designing a double-honeypot system, deployed in a local network for automatic detection of worm attacks from the Internet. Two algorithms based on Expectation-Maximization [10] and Gibbs sampling [9] are proposed for efficient computation of Position Aware Distribution Signature (PADS). The *PAYL sensor* [19] employ anomaly detection which is based on the principle that “zero-day” attacks are delivered in packets whose data is unusual and distinct from all prior “normal content” flowing to or from the victim's site. The *Nemean architecture* [21] is a semantic-aware Network Intrusion Detection System (NIDS) which contains two components: a data abstraction component that normalizes packets from individual sessions and renders semantic context, and a signature generation component that clusters similar sessions and uses machine-learning techniques to generate signatures for each cluster. In a related study, the *Amd* algorithm generates semantic-aware code templates and specifies the conditions for a match between the templates and the programs being checked [2]. *Polygraph* [13] provides a content-based signature generation techniques for polymorphic worms. The underlying assumption is that possible to automatically generate signatures that match many variants of polymorphic worms offering low false positives and low false negatives. Newsome et al. (2005) propose and evaluate a system that expands to notion of single substring signatures (tokens) to conjuctions, ordered sets of multiple tokens and

Bayesian (score) tokens. *EarlyBird* [17] sifts through the invariant portion of a worm's content will appear frequently on the network as it spreads or attempts to spread. In *Netspy* [19] the invariant portion of network traffic generated by a spyware program is used to derive a spyware signature. This is because a signature that has content related to specific user input will miss network activity generated by the program on other user input. Netspy uses a variant of the longest common subsequence (LCSeq) algorithm [6] to find such invariants sections.

Filiol [26] address the problem that commercially available anti-viruses are not resistant against black-box analysis. He suggested generating multiple sub-signatures that are randomly selected from a longer signature. Sub signatures are distributed such that "any sub-pattern is a fixed value which depends on the user/computer identification data".

The aforementioned automatic signature generation techniques focus on analyzing similarities and anomalies in executables' substrings, code tokens as well as statistical distribution of code across variants of malware. Consequently, testing of such signatures was performed on short, stream-based malware such as Nimda, Code Red/Code Red II, MS Blaster (1.8KB long), Sober, Netsky and B[e]agle. Nevertheless, larger malware executable files, carrying full-fledged applications usually contain a significant portion of common code segments which are planted by software development platforms spawning the malware but are found also in benign executables. As a result, automatically selecting a signature that will be both sensitive and specific is a very challenging task in the case of these large files.

The goal of this research is to generalize the above work by proposing an automatic signature generation technique, termed Auto-Sign, capable of generating highly-sensitive and highly-specific signatures for malware of any size and type operating in any operating system environment (i.e., Trojan horses, spyware, adware, viruses, and worms). The technique is also capable of handling malware such as self-decrypted/self-executed files; or archive files (i.e.,

CAB, MSI, Zip). Of course a signature derived for unencrypted/uncompressed malware cannot be used for detecting the same malware in encrypted or compressed files. Short, stream-based malware (not a self-contained application) which does not include significant portions of common code is also not a typical candidate for Auto-sign.

3. THE PROPOSED AUTOMATIC SIGNATURE GENERATION METHOD

In order to create and employ signature for effective and efficient detection of malware in executables, our technique should generate a signature which complies with several requirements. First, we are interesting to find a signature σ of length s with a low probability p_0 to appear in a benign file. In classical signature-based detection, the number of appearances of a *contiguous* signature σ in a benign file of *n* bytes is distributed as[2]:

$$Z \sim N\left((n-2)\,p_0, \sqrt{(n-2)\,p_0(1-p_0)}\right) \qquad (1)$$

The false positive rate of such detector is directly determined by p_0. Thus lowering the value of p_0 will result in a lower false positive. In order to apply with a certain level of false positive rate, one should require that the p_0 will not exceed a certain value *e*.

However estimating p_0 is not an easy task. Assuming we are using a relatively small sample of benign files, then simply measuring the frequency of signature σ in the sample might be unreliable. As proposed in [26] , we can use a first-order Markov model to estimate the probability p_0 of a signature σ containing *s* bytes as:

$$\hat{p}_0(\sigma) = \Pr(b_1)\cdot\Pr(b_2|b_1)\cdot\Pr(b_3|b_2)\cdot\cdots\cdot\Pr(b_s|b_{s-1})$$

[2] This is slightly different from the probabilistic law presented in [28] because here are not refering to the general case of which the signature bytes are not necessary contiguous,

where b_i is the ith bytes in σ.

Let *f*(x)* represents a function that returns a signature of length *x* that appears in the malware code and has lowest $\hat{p}_0$. However if *f*(s)* is not sufficient, i.e. $\hat{p}_0\left(f^*(s)\right) < e$, then we are compelled to search for a larger signature *f*(s+1)*. Note that $\hat{p}_0\left(f^*(x)\right)$ is monotonically decreasing in *x* because there are only two cases:

1. If $f^*(x) \subset f^*(x+1)$ then $\hat{p}_0\left(f^*(x+1)\right) = \hat{p}_0\left(f^*(x)\right) \cdot \Pr\left(b_{x+1} | b_x\right) \leq \hat{p}_0\left(f^*(x)\right)$ (which satisfies monotony)
2. If $f^*(x) \not\subset f^*(x+1)$ then $\hat{p}_0\left(f^*(x+1)\right) \leq \hat{p}_0\left(f^*(x)\right)$ because by definition of f* $\hat{p}_0\left(f^*(x+1)\right) \leq \hat{p}_0\left(f^*(x) \& b\right)$ where & indicates the concatenation operation.

Other requirements from the signature are: The signature should be sufficiently short to decrease the problems of Internet packet splits and detection hardware storage limitations (i.e., various IDS/IPS devices such as DefensePro [5]); Third, the signature should comply to the limitations of high-speed deep packet inspection devices that can detect and remove malware in real-time in high-speed data stream. Finally, it should be well-defined to enable totally automatic generation. The major challenge in conforming to the aforementioned requirements was to develop a methodology that can locate code segment or segments highly unique to a given malware instance and can therefore serve as a powerful and unique signature meeting the stringent requirement set forth by commercial high-speed malware filtering devices.

Since many malware executables nowadays are in fact developed using 4th Generation development platforms (e.g. MSVC, J2EE, Delphi), the binary code of malware nowadays contains a large portion of code placed by these development platform. These portions of

common code are identical or almost identical except address references. Such common segments were not developed by the authoring hacker, but were linked to the malware as part of the underlying code generator's library and are termed Common Function Code (CFC). To significantly decrease the risk of selecting such abundant segments as a signature that may lead to high false positive rate, we must first identify and disregard the CFC part. The CFC of a malware file can be identified by analyzing the malware content against a repository of CFC which is termed a Common Function Library (CFL). The CFL can be derived based on a collection of benign executables and should be regularly updated in order to take into account the evolution of benign (and potentially malicious) files.

To meet the requirements stated above, we developed the Auto-Sign methodology which is schematically described in Figure 1.

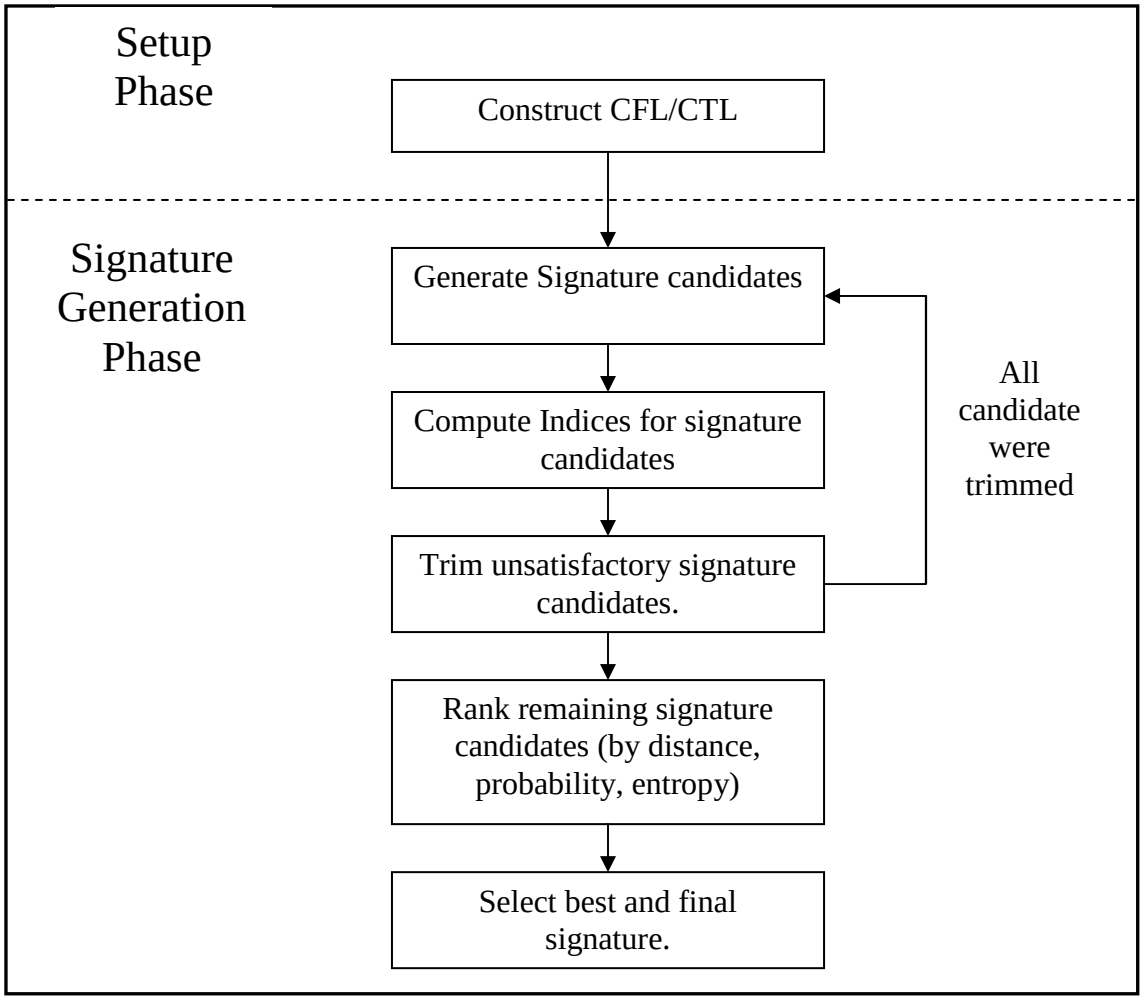


Figure 1: The Auto-Sign methodology

3.1 Setup:

During the setup phase a data structure (library) representing a collection of benign executables is constructed. The data structure is termed CFL.

The CFL is comprises of the following data structures.

1) **3-gram-frequency**: A vector of 2↑24 (~16 million) entries where each cell represents the number of occurrences of each 3-gram (three consecutive bytes) in the collection of benign files. The n-gram size was chosen to be 3 in order to comply with memory constraints. N-Gram was also used in [23] for malware detection and in [24] and [25] for

malware phylogeny. The collection of benign files used for constructing the CFL is scanned sequentially in order to record the number of occurrences of each 3-gram.

2) **3-gram-Files-association**: A 2↑24 X 64 bit-map where a '1' Binary value in a cell (i, j) indicates the appearance of a specific 3-gram i in the jth group of files. The CFL files are divided into 64 groups.

3) **3-gram-relative-position-within-file**: A 2↑24 X 64 bit-map where a '1' Binary value in a cell (i, j) indicates the appearance of 3-gram *i* in the jth internal segment of a file (assuming the file is divided into 64 equal length segments).

The common threat library termed CTL can be constructed as a separate data structure in a similar manner.

Maintaining lookup tables which capture the appearance of each possible 3-gram in a file (or group of files for large CTL/CFLs), as well as the relative position of 3-grams in each file promotes the scalability of Auto-Sign. This enables handling large repositories of CFL/CTL files representing many heterogeneous software platforms, when other techniques such as LcSeq [6] are not feasible anymore.

3.2 Signature Generation:

.

The aim of this process is to auto generate an efficient signature. The process is repreated for each malware M. First, a set of signature candidates is extracted from different positions within the malware *M*. Each signature candidate C_M looks for a fixed contiguous sequence of bytes $C_M \in \{0, 1, 2, . . . , 255\}^s$ with $s = |C_M|$. Using our 3-grams representation C_M comprises a number of 3-grams depending on the length of the signature (e.g. a signature of length 4 bytes is comprised from two 3-grams that overlap by two bytes). Employing the 3 data structures prepared during the setup stage (3.1), the following indices are calculated for each signature candidate extracted from the malware:

1) **Spread**: In subsection 3.1 we have divided each file into 64 segments. The "Spread" measure represents the spread of the signature's 3-grams along the various segments for all the files in the CFL. For example, Spread=1 indicates that the signature is located in only one segment in all the files of the CFL.

The match of a certain 3gram k in a benign subset *S* of three contiguous bytes may be described as a Bernoulli variable:

$$X^S{}_k = \begin{cases} 1 & p^{gram}{}_k \\ 0 & 1 - p^{gram}{}_k \end{cases}$$

The value $p^{gram}{}_k$ is determined by the efficiency of the k's 3-gram.

We define the following variable with respect to a segment *j:*

$$Y_j = \begin{cases} 1, iff & \forall 3gram \subset C_M, \forall file \in CFL : 3gram \subset segment_j(file) \\ 0, else & \end{cases} \quad (2)$$

where $segment_j$(file) denote the jth internal segment of file.

Let $p_{j,k,i}$ represent the probability of k's 3gram to appear at least one time in segment *j* of a benign file *i*. $p_{j,k,i}$ can be estimated the complementary probability, i.e. that the probability of the event that k is not been found in the segment:

$$p_{j,k,i} = 1 - \left(1 - p^{gram}{}_{k}\right)^{\left|segment_j(i)\right| - 2} \quad (3)$$

Thus Yj is distributed as Bernoulli with success probability of

$$p^{seg}{}_{j} = \prod_{k \subset C_M} \prod_{i \in CFL} p_{j,k,i} \quad (4)$$

Note that because the segments are equally sized then $p^{seg}{}_{j} = p^{seg}{}_{i}; \forall i \neq j$

The following formula specifies the spread measure:

$$Spread_{CFL}(C_M) = \sum_{j=1}^{64} Y_j \qquad (5)$$

Spread is distributed as the sum of 64 identically distributed Bernoulli random variables which can be approximated to Normal distribution $N\left(64 \cdot p^{seg}, \sqrt{64 \cdot p^{seg} \cdot (1 - p^{seg})}\right)$.

4) **Freq:** The average frequency of all 3-grams that comprise a candidate signature (computed using the 3-gram-frequency data structure**)**. Higher "Freq" may indicate a bad signature candidate. The following formula specifies the probability measure:

$$\mathrm{Freq}_{\mathrm{CFL}}(C_M) = \frac{\sum_{\forall k \subset C_M; |S|=3} \sum_{\forall file \in CFL} \sum_{\forall S \subset file; |S|=3} X^{S}{}_{k}}{|CFL|\left(|C_M| - 2\right)} \qquad (6)$$

Note that $|C_M| - 2$ represent the number of 3gram in the signature C_M.

Freq is a sum of distributed Bernoulli random variables. Note that the central limit theorem can not be applied here directly as the X's are not identically distributed (different success probabilities). Thus we are using Poisson approximation as proposed in [30]:

$$\Pr\left(Freq = k\right) = \frac{\lambda^k \cdot e^{-\lambda}}{k!} \qquad (7)$$

where:

$$\lambda = \frac{\sum_{\forall k \subset C_M ; |S|=3} \sum_{\forall file \in CFL} \left| file - 2 \right| \cdot p^{gram}{}_k}{\left(\left| C_M \right| - 2 \right)} \qquad (8)$$

The cumulative distribution function (CDF) is:

$$\Pr\left(Freq \le k\right) = \frac{\Gamma\left(\lfloor k+1 \rfloor, \lambda\right)}{\lfloor k \rfloor !} \qquad (9)$$

where Γ(x,y) is the Incomplete gamma function and $\lfloor k \rfloor$ is the floor function.

3) *R*=**Freq/Spread**: By dividing the aforementioned freq and spread we are able to further increase and normalize the crude probability metric for a candidate signature. For example, when all the comprising 3-grams of the candidate signature are concentrated in the same area within an executable (most likely indicating an area of common code) the normalized score will yield a higher value compared to a situation (in a different executable) where the 3-grams are found spread in numerous relative positions within the executable.

We hypothesize that those m candidates are less likely to appear within CFL files in consistent areas of a file thus less likely to belong to chunks common code. This hypothesis indicates that low ratio candidate signatures are associated with lower signature probabilities in benign files (signature *efficiency)*.

Proposition 1 mathematically examines this hypothesis under limited circumstances. In section 4.2 we empirically examine this hypothesis in general circumstances.

Proposition 1: Let R1 and R2 represent the ratios of two candidate signatures C_1 and C_2 both of length *s=3* test on single file. If $P_0(C_1) < P_0(C_2)$ then E(R1)<E(R2)

Proof:

R1 and R2 are ratio of two Poisson variables. Thus:

$$E(R_1) = E\left(\frac{Probability_1}{Spread_1}\right) = E(Probability_1) \cdot E\left(\frac{1}{Spread_1}\right)$$

Using the approximation of the mean of the inverse of a Normal distribution [32]:

$$E(R_1) \approx \lambda_1 \cdot \frac{1}{\mu_1}$$

According to the theorem condition (s=3, $|CFL| = 1$) and using Eq. (8) we obtain:

$$\lambda_1 = |file - 2| \cdot p_0(C_1)$$

$$\mu_1 = 64\left(1\text{-}\left(1\text{-}p_0(C_1)\right)^{|file|/64-2}\right)$$

i.e.:

$$E(R_1) = \frac{|file - 2| \cdot p_0(C_1)}{64\left(1\text{-}\left(1\text{-}p_0(C_1)\right)^{|file|/64-2}\right)}$$

We assume the file is large enough (i.e. larger than 64*4 which is reasonable assumptions as usually files contains thousands of bytes). In this case moving from $P_0(C_1)$ to $P_0(C_2)$, i.e. the probability is increased $\frac{P_0(C_2)}{P_0(C_1)}$ times will result in increasing the nominator by

$\frac{P_0(C_2)}{P_0(C_1)}$ times but the denominator to be increased in less than $\frac{P_0(C_2)}{P_0(C_1)}$ times. Thus the entire rate is increased as Proposition 1 argues.

4) **Distance:**

We examine the distance of each candidate signature that does not appear in the common library to its nearest signature that does appear in the common library.

The structure of executable is built from a continuous partition of common library functions and then a unique code partition. For the sake of clarity we assume that there is only one partition of each type as illustrated in Figure 2. Each line represents a segment of 64 bytes which is also the length of the candidates that we examine. Let assume for the example that we know that the 25th segment is part of the common library (by looking into many other files). However we are not aware where the unique partition begins. However segments that are located in higher places (i.e. are located far way from the 25th segment) have larger chance to be part of the unique block (and because of that can be good candidate for a signature). Specifically in figure 2, the 70th segment is more likely to be part of the unique partition than the 60th segment. We formulate the last intuition in the following proposition.

Proposition 2: Assuming the executable consists of two contiguous partitions of common and unique code, then the expected squared distance of an unique segment to the closet known common segment is grater than the expected squared distance of a common segment to the closet known common segment.

Proof:

Without loss of generality the common code is located in the first n segments and the unique code is located in the remaining m segments. A unique segment located in position b (b>n) and a common segment is located in position a (a<=n).

The location of the closet known common segment is distributed according to *some unknown* discrete distribution:

$$\Pr\ (location = i) = \beta_i ; \sum_{i=1}^{n} \beta_i = 1 :$$

The expected squared distance from location to a and b are:

$$SD_{common} = E\left[\left(location - a\right)^2\right] = E\left[location^2\right] - 2aE\left[location\right] + a^2$$

$$SD_{unique} = E\left[\left(location - b\right)^2\right] = E\left[location^2\right] - 2bE\left[location\right] + b^2$$

Thus:

$$SD_{unique} - SD_{common} = \left(b - a\right)\left(E\left[location\right] + b + a\right) > 0$$

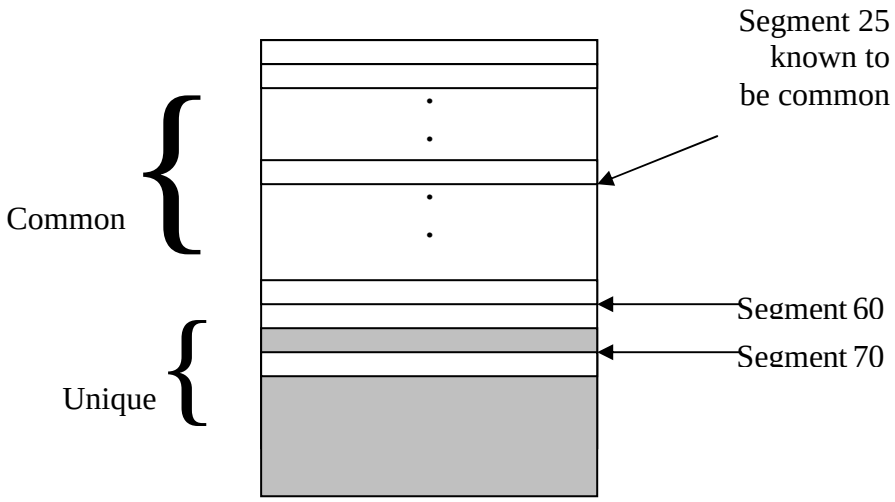


Figure 2: Illustrating the motivation of the distance estimator

5) **Entropy:** In addition the new estimators presented above, we also use entropy measure which has known to be useful for selecting effective signatures [26].

6) **Bits:** the total number of files (or file groups) that contain the signature. ‘

Measures 1- 6 were calculated for the CFL. Measures 2 and 6 were calculated for the CTL.

The signature generation process can therefore be summarized as follows:

1) **Generate signature candidates** of length L by splitting the examined malware executable to segments of equal length L.

2) **Calculate the following measures** (for each signature candidate):

   Spread (based on CFL), Freq/Spread (based on CFL), Bits (Based on CFL), Entropy (you should define it with all others), Freq (based on CTL), Bits (based on CTL). This stage is done using the data structures created in the setup stage.

3) **Mark each candidate appearing in the CFL (bits>0)**:

   Only signatures that do not have any incident where all their comprising 3-Grams appear in a file within the CFL repository are considered viable candidates

4) **Trim candidates** appearing in the CFL (CFL-bits>0) or appearing in the CTL. Note that we trim candidates that appear in the CTL because common functions of malcode are also not good candidates for identifying a certain malware.

5) **Rank remaining candidates (marked during stage 3) by distance from CFL**:

The n candidates with the highest physical distance from the CFL/CTL areas (as calculated from the 3-gram-relative-position-within-file matrix) are selected. Our assumption is that a high distance of a signature candidate from the closest chunk of common code within the executable indicates an area of code which represents the malware more uniquely.

6) **Rank remaining candidates using CFL-Freq / CFL-Spread**:

Out of the n candidates selected in phase 5, m candidates with the lowest CFL-Freq /CFL-Spread ratios are selected.

7) **Rank remaining candidates based on entropy:**

Out of the m candidates selected in phase 6, the w signature candidates with the highest degree of entropy are ranked as best and final It is customary to assume that a signature candidate with a higher degree of entropy is less likely to be associated with areas common to different executables (i.e., constants, text with repeating characters) and is therefore a more unique identifier of the malware.

8) **Select actual signature:**

Out of the w remaining candidates the one with the highest entropy is selected as the actual signature .

Table 1 provides a pseudo-code specification of the aforementioned signature selection process.

Proposition 3: The computational complexity of the algorithm in Table 1 is $O(CR \cdot (CL-2))$ where CR denote the number of requested candidates and CL is the candidate length

Proof:

The computational complexity of the algorithm in Table 1 is computed as follows: the GenerateSignatureCandidates complexity is $O(CR)$, the complexity of the first loop (lines 3-13) is $O(CR \cdot (CL-2))$, the complexity of the second loop (lines 15-19) is $O(CR)$, the complexity of the third loop (lines 21-26) is $O(CR)$, the complexity of Candidates.SortBY_DistanceFromCFL is $CR \cdot \log(CR)$, the complexity of the fourth loop on lines 29-30 is $O(CR)$ (worth case when N=0), the complexity of the fifth and sixth loop in lines 32-34 and 37-39 is also $O(CR)$, and the complexity of Candidates.SortBY_Entropy is $CR \cdot \log(CR)$. Thus the overall complexity is $O(CR \cdot (CL-2))$.

```
Candidates=GenerateSignatureCandidates(Maleware,CandidatesRequired,CandLen);

ForEach Candidate in Candidates do
        {
        Candidate.CFLspread   = CFL.CalcSpread (Candidate);
        Candidate.CFLNormProb = CFL.Average3gramProb (Candidate) / Candidate.CFLspread;
        Candidate.CFL_bits    = CFL.Count_Files_With_All_3gram (Candidate)
        Candidate.Entropy     = Entropy(Candidate.string);
        Candidate.InCFL       = (Candidate.CFL_bits > 0);

        Candidate.CTL_bits    = CTL.Count_Files_With_All_3gram (Candidate)
        Candidate.InCTL       = (Candidate.CTL_bits > T);
        }

ForEach Candidate in Candidates do
        {
        If Candidate.InCFL then continue; // disregard
        Candidate.DistanceFromCFL = DistanceFrom(Candidate.offset, Candidates);
        }

ForEach Candidate in Candidates do
        {
        If Candidate.InCFL or
           Candidate.InCTL then
          Candidates.delete(Candidate);
        }

Candidates.SortBY_DistanceFromCFL();
For i=Candidates.count downto N+1 do
        Candidates.delete(i);

Candidates.SortBY_CFLNormProb();
For i=Candidates.count downto M+1 do
        Candidates.delete(i);

Candidates.SortBY_Entropy();
For i=Candidates.count downto 2 do
        Candidates.delete(i);
Signature = Candidates[1];
```

Table 1: Pseudo-Code of signature selection process.

4. EVALUATION

4.1 Examining the effectiveness of Auto-Sign

The evaluation described in this section comprises of a set of experiments we conducted to test the effectiveness of Auto-Sign. The experimental design was aimed at assessing the impact of various independent variables on the quality of the signature which is characterized by the number of false positive appearances of a signature in a set of clean (benign) files. The independent variables used by this study are the proportion of CFL versus test files and the size of a signature (32, 64, 128 bytes).

The first task in the evaluation (Figure 4) is to assemble two repositories: one comprising benign and the other comprising malicious (threat) executables. The benign repository is randomly split 10 times into training- and testing-sets. A distinct CFL is constructed 10 times for each of these sampled training sets where the size of the training set is increased in a linear fashion during each of these 10 iterations. The threat repository, on the other hand, is held fixed during the evaluation and therefore the CTL is constructed once before being used by the signature generation algorithm.

In each such configuration we generated three signatures with lengths of 32, 64, 128 bytes for all 849 malware files. The false positive count of the best signatures generated by Auto-sign was calculated by performing a cross-validation of the selected signature against the test set.

The repository of benign files included 5494 files with lengths ranging from 3Kb to 8MB. The repository of malware instances comprised of 849 files with lengths ranging from 6Kb to 4.25MB (11 executables where above 1 MB and 200 above 300KB). The distribution of malware file types is depicted in Figure 5. Generating the signatures for the malware repository took 5.7 seconds using one 3Ghz Pentium processor. Figure 6 depict a sample of

Auto-Sign's output with a list of files, their signatures and the indices calculated for these signatures.

Table 2 depicts the average false positive counts for monotonically increasing proportions of the CFL in the following two configurations: 33% training set (CFL with CTL)/67% testing set; and 50% training (CFL with CTL)/50% testing set. Table 3 compares the average false positive counts for 25% training-75% testing set once with the CTL and once without. For each such configuration we used a 10 cross-fold validation over the 849 files for the 3 signature lengths.

The results indicate the fact that even a large CFL of up to 50% of the clean files repository cannot compensate for short signatures of 32 Bytes. The size of the signature is optimal at 64 Bytes as the improvement from 64 to 128 Bytes is not substantial. Moreover, the important factor with regards to the CFL selection is the "proximity" of the common code to the code generated by coding platforms. This is evident from the fact that in the case of the 25-75%, removing the files of the CTL hampers the precision of the signatures and increases the amount of false positives. Moreover, the results also indicate that there is no point increasing the CFL beyond some optimal threshold. Once the CFL represents a critical mass of the common code (in the case of choosing a 25% CFL randomly, approximately 7.5% of the clean repository), adding files to the CFL does not yield any marginal decrease in the false positive rates of candidates.

Finally, we also generated signatures for the malware using random selection and entropy-maximization for the various signature lengths. Under the entropy-maximization approach, the signature was picked from an area within the malware which has the highest entropy score. This was done in order to minimize the likelihood of the signature's appearance in benign executables which would yield frequent false-positive detections.

Table 4, depicts the number of malware files whose signature was found in the benign repository of 5494 files. The results indicate that the entropy maximization technique is

superior to random selection in all signature lengths however both techniques are far from being feasible for meeting the quality of the CFL and especially CFL+CTL performance.

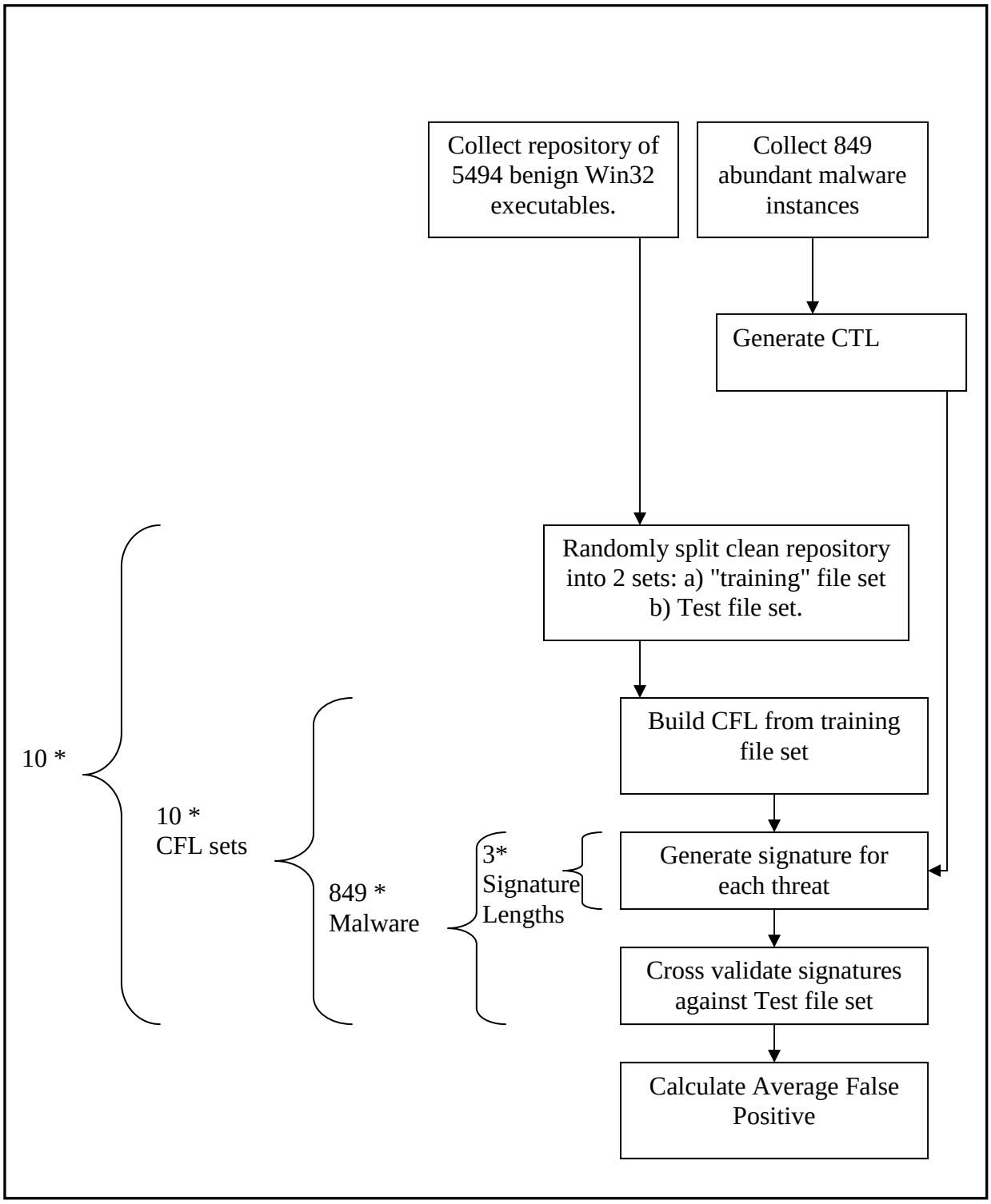


Figure 4: The Auto-Sign Evaluation Plan

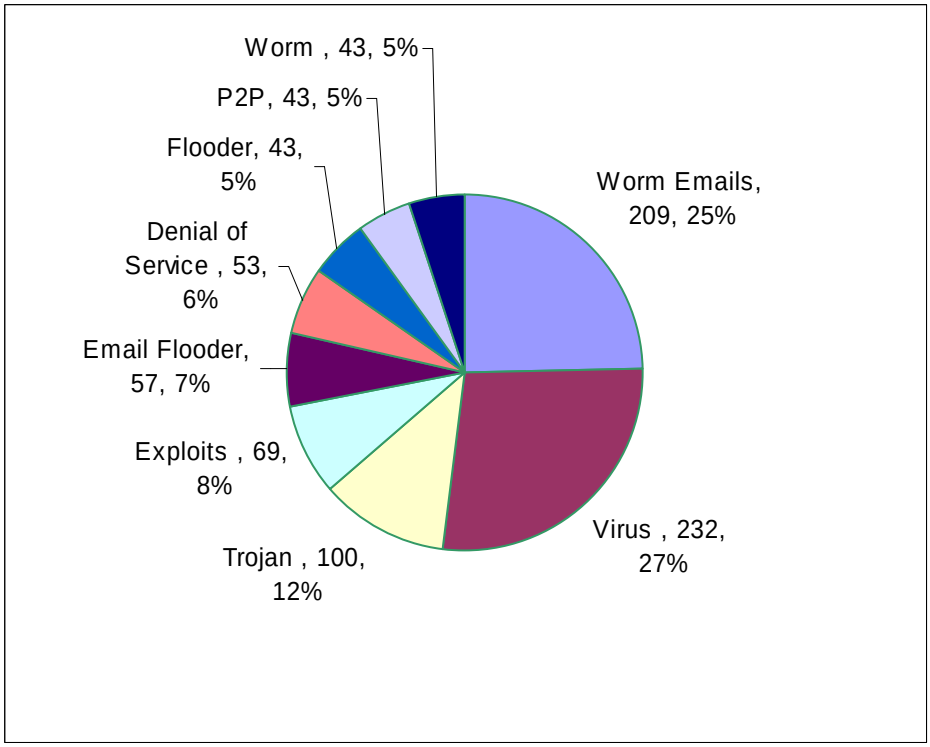
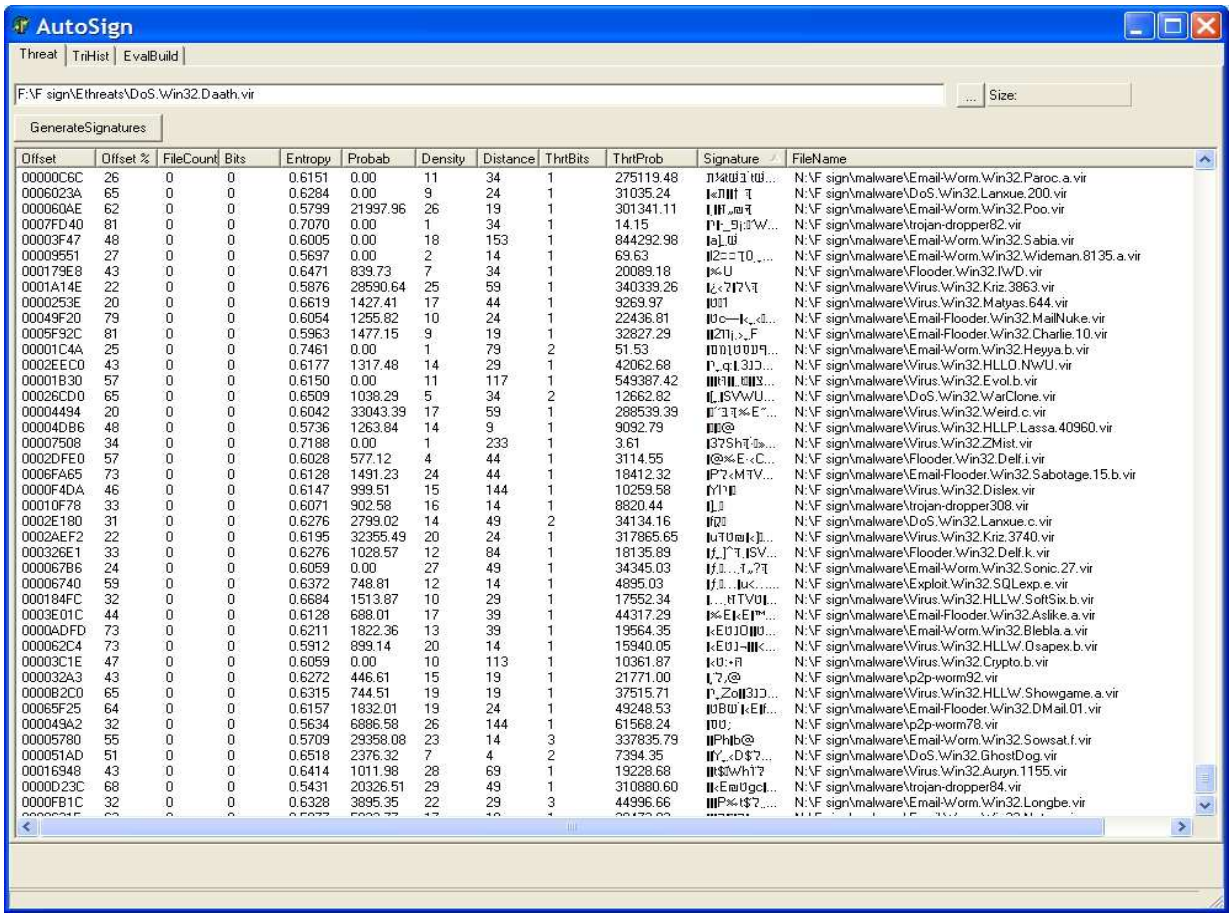


Figure 5: Distribution of Malware Types



Offset	Offset %	FileCount	Bits	Entropy	Probab	Density	Distance	ThrtBits	ThrtProb	Signature	FileName
00000C6C	26	0	0	0.6151	0.00	11	34	1	275119.48	[illegible]	N:\F sign\malware\Email-Worm.Win32.Paroc.a.vir
0006023A	65	0	0	0.6284	0.00	9	24	1	31035.24	[illegible]	N:\F sign\malware\DoS.Win32.Lanxue.200.vir
000060AE	62	0	0	0.5799	21997.96	26	19	1	301341.11	[illegible]	N:\F sign\malware\Email-Worm.Win32.Poo.vir
0007FD40	81	0	0	0.7070	0.00	1	34	1	14.15	[illegible]	N:\F sign\malware\trojan-dropper82.vir
00003F47	48	0	0	0.6005	0.00	18	153	1	844292.98	[illegible]	N:\F sign\malware\Email-Worm.Win32.Sabia.vir
00009551	27	0	0	0.5697	0.00	2	14	1	69.63	[illegible]	N:\F sign\malware\Email-Worm.Win32.Wideman.8135.a.vir
000179E8	43	0	0	0.6471	839.73	7	34	1	20089.18	[illegible]	N:\F sign\malware\Flooder.Win32.IWD.vir
0001A14E	22	0	0	0.5876	28590.64	25	59	1	340339.26	[illegible]	N:\F sign\malware\Virus.Win32.Kriz.3863.vir
0000253E	20	0	0	0.6619	1427.41	17	44	1	9269.97	[illegible]	N:\F sign\malware\Virus.Win32.Matyas.644.vir
00049F20	79	0	0	0.6054	1255.82	10	24	1	22436.81	[illegible]	N:\F sign\malware\Email-Flooder.Win32.MailNuke.vir
0005F92C	81	0	0	0.5963	1477.15	9	19	1	32827.29	[illegible]	N:\F sign\malware\Email-Flooder.Win32.Charlie.10.vir
00001C4A	25	0	0	0.7461	0.00	1	79	2	51.53	[illegible]	N:\F sign\malware\Email-Worm.Win32.Heyya.b.vir
0002EEC0	43	0	0	0.6177	1317.48	14	29	1	42062.68	[illegible]	N:\F sign\malware\Virus.Win32.HLLO.NWU.vir
00001B30	57	0	0	0.6150	0.00	11	117	1	549387.42	[illegible]	N:\F sign\malware\Virus.Win32.Evol.b.vir
00026CD0	65	0	0	0.6509	1038.29	5	34	2	12662.82	[illegible]	N:\F sign\malware\DoS.Win32.WarClone.vir
00004494	20	0	0	0.6042	33043.39	17	59	1	288539.39	[illegible]	N:\F sign\malware\Virus.Win32.Weird.c.vir
00004DB6	48	0	0	0.5736	1263.84	14	9	1	9092.79	[illegible]	N:\F sign\malware\Virus.Win32.HLLP.Lassa.40960.vir
00007508	34	0	0	0.7188	0.00	1	233	1	3.61	[illegible]	N:\F sign\malware\Virus.Win32.ZMist.vir
0002DFE0	57	0	0	0.6028	577.12	4	44	1	3114.55	[illegible]	N:\F sign\malware\Flooder.Win32.Delf.i.vir
0006FA65	73	0	0	0.6128	1491.23	24	44	1	18412.32	[illegible]	N:\F sign\malware\Email-Flooder.Win32.Sabotage.15.b.vir
0000F4DA	46	0	0	0.6147	999.51	15	144	1	10259.58	[illegible]	N:\F sign\malware\Virus.Win32.Dislex.vir
00010F78	33	0	0	0.6071	902.58	16	14	1	8820.44	[illegible]	N:\F sign\malware\trojan-dropper308.vir
0002E180	31	0	0	0.6276	2799.02	14	49	2	34134.16	[illegible]	N:\F sign\malware\DoS.Win32.Lanxue.c.vir
0002AEF2	22	0	0	0.6195	32355.49	20	24	1	317865.65	[illegible]	N:\F sign\malware\Virus.Win32.Kriz.3740.vir
000326E1	33	0	0	0.6276	1028.57	12	84	1	18135.89	[illegible]	N:\F sign\malware\Flooder.Win32.Delf.k.vir
000067B6	24	0	0	0.6059	0.00	27	49	1	34345.03	[illegible]	N:\F sign\malware\Email-Worm.Win32.Sonic.27.vir
00006740	59	0	0	0.6372	748.81	12	14	1	4895.03	[illegible]	N:\F sign\malware\Exploit.Win32.SQLexp.e.vir
000184FC	32	0	0	0.6684	1513.87	10	29	1	17552.34	[illegible]	N:\F sign\malware\Virus.Win32.HLLW.SoftSix.b.vir
0003E01C	44	0	0	0.6128	688.01	17	39	1	44317.29	[illegible]	N:\F sign\malware\Email-Flooder.Win32.Aslike.a.vir
0000ADFD	73	0	0	0.6211	1822.36	13	39	1	19564.35	[illegible]	N:\F sign\malware\Email-Worm.Win32.Blebla.a.vir
000062C4	73	0	0	0.5912	899.14	20	14	1	15940.05	[illegible]	N:\F sign\malware\Virus.Win32.HLLW.Osapex.b.vir
00003C1E	47	0	0	0.6059	0.00	10	113	1	10361.87	[illegible]	N:\F sign\malware\Virus.Win32.Crypto.b.vir
000032A3	43	0	0	0.6272	446.61	15	19	1	21771.00	[illegible]	N:\F sign\malware\p2p-worm92.vir
0000B2C0	65	0	0	0.6315	744.51	19	19	1	37515.71	[illegible]	N:\F sign\malware\Virus.Win32.HLLW.Showgame.a.vir
00065F25	64	0	0	0.6157	1832.01	19	24	1	49248.53	[illegible]	N:\F sign\malware\Email-Flooder.Win32.DMail.01.vir
000049A2	32	0	0	0.5634	6886.58	26	144	1	61568.24	[illegible]	N:\F sign\malware\p2p-worm78.vir
00005780	55	0	0	0.5709	29358.08	23	14	3	337835.79	[illegible]	N:\F sign\malware\Email-Worm.Win32.Sowsat.f.vir
000051AD	51	0	0	0.6518	2376.32	7	4	2	7394.35	[illegible]	N:\F sign\malware\DoS.Win32.GhostDog.vir
00016948	43	0	0	0.6414	1011.98	28	69	1	19228.68	[illegible]	N:\F sign\malware\Virus.Win32.Auryn.1155.vir
0000D23C	68	0	0	0.5431	20326.51	29	49	1	310880.60	[illegible]	N:\F sign\malware\trojan-dropper84.vir
0000FB1C	32	0	0	0.6328	3895.35	22	29	3	44996.66	[illegible]	N:\F sign\malware\Email-Worm.Win32.Longbe.vir

Figure 6: Sample of Signature Data with Indices

		50%-50% with CTL				33%-67% with CTL	
	32Byte	64Byte	128Byte		32Byte	64Byte	128Byte
CFL=5%	32	0	0	CFL=3.3%	54	9	10
CFL=10%	28	1	0	CFL=6.6%	31	1	1
CFL=15%	19	0	0	CFL=9.9%	25	1	0
CFL=20%	17	0	0	CFL=13.2%	30	0	0
CFL=25%	21	0	0	CFL=16.5%	25	0	0
CFL=30%	20	0	0	CFL=19.8%	22	0	0
CFL=35%	22	0	0	CFL=23.1%	22	0	0
CFL=40%	26	0	0	CFL=26.4%	19	0	0
CFL=45%	23	0	0	CFL=29.7%	17	0	0
CFL=50%	25	0	0	CFL=33%	21	0	0

Table 2: False positive counts as a function of the CFL size

		25%-75% with CTL				25%-75% without CTL	
	32Byte	64Byte	128Byte		32Byte	64Byte	128Byte
CFL=2.5%	43	10	7	CFL=2.5%	58	20	10
CFL=5%	43	6	5	CFL=5%	55	14	10
CFL=7.5%	46	10	9	CFL=7.5%	52	11	10
CFL=10%	34	0	0	CFL=10%	39	2	2
CFL=12.5%	25	0	0	CFL=12.5%	36	1	1
CFL=15%	24	0	0	CFL=15%	32	2	1
CFL=17.5%	35	0	0	CFL=17.5%	40	2	1
CFL=20%	35	0	0	CFL=20%	38	1	1
CFL=22.5%	25	0	0	CFL=22.5%	36	3	2
CFL=25%	27	0	0	CFL=25%	32	2	1

Table 3: False positive counts as function of the CFL size - with/without CTL

	Random			Entropy	
32Bytes	64Bytes	128Bytes	32Bytes	64Bytes	128Bytes
297	249	227	197	124	91

Table 4: False Positive Counts: Random vs. Entropy-Maximization

4.2 Are the estimators' good indicators for the signature efficiency?

Our method filters the signature candidate list by employing different estimators in a cascade manner. In order to obtain a good signature (i.e. which does not appear in the benign file) , the estimators need to be informative discriminator to if the signature might appear or not in a benign file. In this section we examine how indicative the estimators are.

For this purpose we performed the following experiment. We randomly sampled 2,000 64 bytes signatures from various malcode and calculated their estimators. Then we search the signature on a benign corpus to obtain the fact if the signature appears in benign files. Table 5 summarizes the results of t-test of two-sample assuming unequal variances which compares the estimators values in case that the signature appeared in the benign corpus or not. The results are very encouraging. For all estimators the null hypothesis are rejected with $\alpha = 5\%$. Thus the proposed estimators are good indicators for predicting if a signature would appear in benign files.

Estimator	Mean value of the estimator for signatures that appeared in benign corpus	Mean value of the estimator for signatures that did not appear in benign corpus	p-value on two-tail	Conclusion
Entropy	0.55±0.008833	0.32±0.075756	4.05E-37	Significant
Ratio	1493.06±2441	3728.99±3306	1.7E-25	Significant
Distance	13.28±24.22	0.586+ 3.963629	3.62E-82	Significant
Number of observations	1695	305		

Table 5: Results of t-test of two-sample assuming unequal variances

4.3 Are the estimators' statistically independence?

In addition to the fact that estimators should be indicative for choosing the best signature. We would like that the estimators will be diverse in the way they rank signatures. Otherwise there is no point in using multiple estimators. This requirement is similar to the diversity requirement in mixture-of-experts techniques in AI [33] and specifically in malware detection [34]. These methods are very effective, mainly due to the phenomenon that various types of models have different ``inductive biases". It has shown that such diversity can be used to increase the predictive performance of the system. In order to examine the diverseness of the estimators we used the data described in the previous section and check whether the estimators are statistically independent using chi-square test.

Unfortunately, the results indicate that the estimators are statistically dependent with $\alpha = 5\%$. However, Auto-Sign does not simply weight these estimators. Instead, it uses it in a cascade manner: first it uses distance estimator then the ratio estimator and finally the entropy estimator. Thus, it is more reasonable to examine the statistics dependence which reflects this cascading procedure. Assuming that each estimator filters half of the candidates to the next cascade, we examine the statistic dependence between estimator's values for the chosen (unfiltered) candidates and the corresponding values of the subsequent estimator. The results that are presented in Table 7, indicate that the estimators are statistically independent with $\alpha = 5\%$ when they are used in a cascade manner.

Test	p-value	Conclusion
Distance vs. Entropy	0.037	Statistically dependent
Ratio vs. Entropy	1.53636E-52	Statistically dependent
Ratio vs. Distance	0.005	Statistically dependent

Table 6: Results of chi-square test for independent

Test	p-value	Conclusion
Distance vs. Ratio	0.363	Statistically independent
Ratio vs. Entropy	0.070	Statistically independent

Table 7: Results of chi-square test for independence in cascade manner

5. CONCLUSION

This paper proposes a new approach for automatic generation of signatures for malware executable of all sizes with an intention to be used by high-speed malware filtering devices. We consider the fact that large executables are comprised of substantial amounts of code that originates from the underlying standard development platforms and is thus replicated across various instances of both benign and malware developed by these platforms. In order to minimize the risk of false positive classification of benign executables as malware, we propose and evaluate a method to discard signature candidates that contain such replicated chunks of code.

The empirical findings presented in section 4 support the viability of the general approach proposed by this research and suggest that eliminating signature candidates belonging to

common code segments has a more profound effect on restricting the level of false positives than increasing the length of a signature. N-grams, however, are not the only measure which can be used to realize the general approach posed by Auto-Sign and alternative ones can be used as a substitute for representing signatures (i.e., segment prefix/suffix, hash values etc.)

The main benefit of the proposed method is that it enables analysis at the binary level and does not require a semantic interpretation of code into function blocks using techniques such as code markers, disassembly, state-machines etc. This benefit means that the methodology is generic and is not affected by changes in CPU or introduction of new development platforms.

Nevertheless. enterprises which would like realize Auto-Sign in generating signatures for high-throughput network security appliances need to follow a more exhaustive and systematic methodology for building their CFL repository. Considering the global variety of development platforms and the mobility of threats facilitated by the Internet, ensuring the external validity of this study relies substantially on reaching a critical mass of CFL files which represents abundant development platforms. Furthermore, it often does not suffice for a signature to be available—deployed signatures must be managed, distributed and kept up-to-date by security administrators [16].

We plan to repeat the evaluation Auto-Sign on a larger scale with much more malware files and CFLs generated for different development environments. We also plan to evaluate additional methods for trimming, ranking and choosing the best signature out of the collection of candidates. In addition, in order to further strengthen the signatures and minimize the risks of false positives we propose to use "composite signatures" which are generated by using two or more distinct signatures for each malware. This activity addresses the biggest challenge of Auto-Sign which is the need to reduce to zero the amount of false positives before being deployed for generating signatures in high-speed malware filtering

devices. In the future we plan to use Auto-Sign to generate multiple signatures in order to increase the resistance against black box analysis as described in [26].

Acknowledgments

The authors gratefully thank the action editors and the anonymous reviewers whose constructive comments significantly helped in improving the quality and accuracy of this paper.

6. REFERENCES

[1] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proc. of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[2] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection, In *IEEE Symposium on Security and Privacy*.Oakland, California, May 2005.

[3] S. P. Chung, A.K. Mok. Allergy attack against automatic signature generation, *Recent Advances in Intrusion Detection*, 61-80, 2006

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

[5] DefensePro, Radware, http://www.radware.com/.

[6] D.S. Hirschberg, 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (Oct. 1977), 664-675.

[7] HA. Kim. and B. Karp, Autograph: Toward automated, distributed worm Signature detection, in *Proceedings of the 13th Usenix Security Symposium (Security 2004)*, San Diego, CA, August, 2004.

[8] C. Kreibich, and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *SIGCOMM Comput. Commun. Rev.* 34, 1 (Jan. 2004), 51-56.

[9] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: A Gibbs sampling strategy for multiple alignment," *Science*, vol. 262, 208–214, Oct. 1993.

[10] C. E. Lawrence and A. A. Reilly. An Expectation Maximization (EM) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences, *PROTEINS: Structure, Function and Genetics*, 7, 41–51, 1990.

[11] R. Lemos, Counting the Cost of Slammer. *CNET news.com*. http://news.com.com/2100-1001-982955.html, Jan. 2003.

[12] D. Moore, C, Shannon, G., Voelker., S. and Savage. Internet Quarantine: Requirements for containing self-Propagating code. In Proceedings of *IEEE INFOCOM* 2003 (Mar. 2003).

[13] J. Newsome, B. Karp, D. Song, Polygraph: Automatically generating signatures for polymorphic worms, *2005 IEEE Symposium on Security and Privacy (S&P'05)*, pp. 226-241

[14] P. Szor, *The Art of Computer Virus Research and Defense*, Addison-Wesley, 2005.

[15] N. Provos. A virtual honeypot framework, *Tech. Rep. CITI Technical Report 03-1*, Center for Information Technology Integration, University of Michigan, Ann Arbor, Michigan, USA, , Oct. 2003.

[16] K. Rieck, P.Laskov, Language models for detection of unknown attacks in network traffic, *Journal in Computer Virology*, 2(4), 243- 256, 2007.

[17] S. Singh, C. Eitan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[18] Y. Tang, S. Chen, Defending against Internet worms: A signature-based approach, in *Proc. of IEEE INFOCOM'05*, Miami, Florida, USA, May 2005.

[19] K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advance in Intrusion Detection (RAID)*, Sep 2004.

[20] H. Wang, S. Jha and V. Ganapathy. NetSpy: Automatic Generation of Spyware Signatures for NIDS *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06),* 2006.

[21] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics-aware signatures, In *14th USENIX Security Symposium*. Baltimore, Maryland, August 2005.

[22] Y. Elovici, A., Shabtai, R., Moskovitch, G., Tahan, C., Glezer Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic", *The 30th Annual German Conference on Artificial Intelligence (KI-2007),* Springer, LNCS Vol. 4667, 44-50, Osnabrück, Germany, September 10-13, 2007.

[23] T. Abou-Assaleh, N. Cercone, V. Kešelj, R. Sweidan, "NGram Based Detection of New Malicious Code," *28th Annual International Computer Software and Applications Conference Workshops and Fast Abstracts (COMPSAC'04),* pp. 41-42, 2004,

[24] Goldberg, L. A., Goldberg, P.W., Phillips, C. A. & Sorkin, G., "Constructing Computer virus phylogenies". *Journal of Algorithms*, 26(1), pp.188-208.

[25] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, "Malware Phylogeny Using Maximal πPatterns", *EICAR 2005 Conference: Best Paper Proceedings*, 2005, 167-174.

[26] Filiol, E., "Malware Pattern Scanning Schemes Secure Against Black-box Analysis", *Journal in Computer Virology*, 2(1), pp. 35-50, 2006.

[27] Benjamin Morin, Ludovic Mé, "Intrusion detection and virology: an analysis of differences, similarities and complementariness", *Journal in Computer Virology*, 3(1), pp. 39-49, 2007.

[28] Filiol E. and Josse S. "A Statistical Model for Viral Detection Undecidability", *Journal in Computer Virology*, 3 (2), 65-74, 2007.

[29] Filiol E. and Raynal F. Malicioux, "Malicious Cryptography … reloaded and also malicious statistics", CanSecWest 2008, Vancouver, 26--28 Mars 2008.

.

[30] Le Cam, L. (1960). An approximation theorem for Poisson binomial distribution, Pacific J. Math., 10, 1181-1197.

[31] Ederer, F., and N. Mantel. 1974. Confidence limits on the ratio of two Poisson variables. Am. J. Epidemiol. 100:165-167.

[32] C. D. Lai, G. R. Wood, C. G. Qiao, The Mean of the Inverse of a Punctured Normal Distribution and Its Application, Biometrical Journal, Volume 46 Issue 4, Pages 420 - 429, 2004

[33] L. Rokach, Collective Agreement-based Pruning of Ensembles, Computational Statistics and Data Analysis (to appear).

[34] E. Menahem, A. Shabtai, L. Rokach, Y. Elovici, Improving malware detection by applying multi-inducer ensemble, Computational Statistics and Data Analysis (to appear).