The 28th International Conference on Distributed Computing Systems

Analysis of Maximum Executable Length for Detecting Text-based Malware

Parbati Kumar Manna   Sanjay Ranka   Shigang Chen
Department of Computer and Information Science and Engineering, University of Florida
{pkmanna, ranka, sgchen}@cise.ufl.edu [*]

Abstract

The possibility of using purely text stream (keyboard-enterable) as carrier of malware is under-researched and often underestimated. A text attack can happen at multiple levels, from code-injection attacks at the top level to host-compromising text-based machine code at the lowest level. Since a large number of protocols are text-based, at times the servers based on those protocols use ASCII filters to allow text input only. However, simply applying ASCII filters to weed out the binary data is not enough from the security viewpoint since the assumption that malware are always binary is false. We show that although text is a subset of binary, binary malware detectors cannot always detect text malware. We analyze the MEL (Maximum Executable Length)-based detection schemes, and make two contributions by this analysis. First, although the concept of MEL has been used in various detection schemes earlier, we are the first to provide its underlying mathematical foundation. We show that the threshold value can be calculated from the input character frequencies and that it can be tuned to control the detection sensitivity. Second, we demonstrate the effectiveness of a MEL-based text malware detector by exploiting the specific properties of text streams.

1 Introduction

In the past decade, the Internet has witnessed the rapid evolution of various malware (virus, worm, Trojans). While a considerable amount of time and research has been devoted for detection of the classic (binary) malware, the possibility of using purely text stream (keyboard-enterable, Hex 0x20 through 0x7E) as carrier of malware has remained under-researched and often underestimated. Rix [9] and Eller [6] showed a few years ago that any binary code can be turned into functionally equivalent text (or even alphanumeric) code. Having a malware that is completely text-based is very appealing to the malware authors since it can open the channels that were earlier assumed to be malware-resistant simply by virtue of accepting text-only input. Many protocols (or parts thereof) are text-based, viz, the URL portion of a HTTP request, or email traffic. To ensure text-only input, these servers often employ an ASCII filter to discard or mangle the binary input [6]. However, if the filter is the only defense against any possible attack towards a vulnerability, then these servers remain subject to be compromised, as the assumption that all malware are binary is false. Worse, sometimes even malware detectors deliberately bypass text stream, e.g. SigFree usually does not process the text-only input to avoid performance degradation [12]. Thus, the notion of regarding the text data as benign and not subjecting it to malware detection is dangerous, and text should undergo the same scrutiny as binary.

Even when the text traffic is examined, today's malware detectors are not adequately suited for efficiently detecting text-based malware due to the structural properties of text. We consider two such schemes: 1) detection by disassembling the input into instructions and then checking for the validity and executability of instruction sequences (e.g. APE [10]), and 2) detection by examining the frequency distribution and other statistical properties of the payload (e.g. PAYL [11]). There are two potential problems with the former scheme. First, almost *any* text string translates into a syntactically correct sequence of instructions, which means checking for syntactic validity is of little value for detecting text malware. Second, since most branch opcodes are already text, the proportion of branch instructions for text data is significantly higher than that for binary. Since each branch instruction forks the current execution path into two directions, having a lot of them exponentially increases the total number of paths to be inspected by a detector doing instruction disassembly. In other words, to ensure quick detection of malware in text data, one must find novel ways to prune the number of the paths to be inspected. Regarding the other detection approach of examining the frequency distribution and other statistical properties of the payload, there have been instances where text malware has been shown to have successfully evaded such detectors. For example, Kolesnikov *et al* [7] showed the way to create an text malware that follows normal traffic pattern to the extent that it

[*]We would like to thank the anonymous reviewers for their suggestions that helped improve the paper. This work is supported in part by the National Science Foundation under Grant ITR 0325459 and NSF Grant number 0312038 (under a subcontract from FIU) and Cisco University Research Award (to Shigang Chen in 2007). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or Cisco.

can evade even a robust payload-based detector like PAYL [7]. Finally, we have performed experiments using a commercial malware detector to scan various binary malware and their text counterparts. Although the detector successfully caught the binary malware, no alarms were raised for the text. Therefore, we conclude that the threat of text malware is real, and we can ignore them only at our own peril.

Although binary malware detectors are relatively ineffective against text malware, they do give us directions for possible detection strategies. The Abstract Payload Execution (APE [10]) method was the first to introduce the concept of MEL (Maximum Executable Length) for detecting worms. MEL denotes the length of the longest error-free execution path in terms of number of instructions executed. Because of its inherent randomness, a benign stream of data is very likely to cause an error during runtime and thus not likely to have a long error-free execution path. In APE, the MEL of the input stream is compared to a threshold that is obtained experimentally from the test data. If the MEL is higher than the threshold, an alarm is raised. However, while this MEL concept is quite novel, it does not explain whether there is a mathematical foundation behind the theory, i.e. whether it is possible for a benign text stream to have an MEL higher than a given threshold, and if so, with what probability.

In our previously work [8] (which was also MEL-threshold-based), we had aggressively exploited the deficiencies and constraints of the text malware to come up with a detection strategy that is fast, reliable and accurate. In this paper, we complement our prior work by providing the underlying statistical foundation of the MEL theory that has been used by both us and others for malware detection. We show how the threshold can be calculated from the input character frequencies (instead of from some experimental data [10, 2] that may be biased), and how we can control the detection sensitivity.

The rest of the paper is organized as follows. Section 2 gives a brief overview of our previous work on text malware detection, which includes the concept of MEL, especially in the context of text. Section 3 derives the underlying probabilistic model for MEL. Section 4 considers the applicability of MEL-based detection schemes for text and binary malware. Section 5 evaluates the results of our detection strategy. Section 6 provides a comparison of our work with APE. Section 7 concludes with the discussion of the limitations and resilience to evasion of our scheme.

2 Related Work

Other than DAWN [8], we are not aware of any work that *specifically* targets the text malware. SigFree does have the capability to detect both text and binary worms, but it usually bypasses the text-based data since processing it degrades its performance [12]. Below, we give a brief summary of DAWN, which provides a context for understanding our work.

2.1 Constructing a Typical Text Malware

Because text malware may contain only instructions that are completely text-based, they are limited to the following instructions under Intel architecture: register/memory/stack manipulation opcodes (*sub*, *xor*, *and*, *inc*, *imul*, *cmp*, *inc*, *dec*, *push*, *pop*, and *popa*), jump opcodes (*jo* through *jng*), I/O operation opcodes (*insb*, *insd*, *outsb* and *outsd*), miscellaneous opcodes (*aaa*, *daa*, *das*, *bound* and *arpl*), and all the Operand/Segment override prefixes (*a16*, *o16*, *cs*, *ds*, etc.). It is evident that this list does not include many opcodes (e.g., those for making system calls) that are required for a potent malware. However, using these text opcodes intelligently, the required non-text opcodes can be dynamically generated at runtime. In fact, it is possible to create *any* binary (non-text) word using text data only, *e.g.*, the binary character 0 can be generated by doing `'a'`$\oplus$`'a'`. Thus, during runtime, a binary malware can be created by "decrypting" an appropriately encoded text malware [9], and once created, the binary malware will be given the control for execution. In this paper, we refer to the process of encoding a binary malware in a text-based malware as *encryption*, and the process of decoding the text malware back into binary code as *decryption*. Thus, a text malware must contain a *decrypter*, a sequence of unencrypted text instructions that performs the decryption.

2.2 The MEL Concept

The crux of the MEL concept is that error-raising instructions occur randomly among normal data, and if an instruction stream contains a very long error-free execution path, then such a stream is not benign, since the probability of that happening purely out of chance is very slim in normal data. To formalize the concept, we use the following definitions:

- **Valid (or invalid) instruction**: an instruction that will not (or will) raise an error during runtime and cause the running process to abort.
- **Maximum Executable Length (MEL)**: length of the longest (in terms of number of instructions) sequence of consecutively-executed valid instructions in an instruction stream.

The concept of MEL was introduced in APE [10] for detecting the binary worms by finding the worm's sled, which would mostly consist of `NOP`s. A worm's sled consists of strictly valid instructions, as otherwise the process will abort and the control will not be passed to the actual worm code at the end of the sled. Therefore, a worm employing a sled would have a very long sequence of valid instructions. Exploiting this feature, it was hypothesized that if the MEL of an instruction sequence exceeded certain threshold, then the instruction stream contains a worm. While our work generally follows the direction shown by APE, there are some significant differences that we will show in Section 6.

2.3 Text-based Malware has Larger MEL

Below are the two primary reasons why a text-based malware will have a high MEL.

Opcode Unavailability: In order to be potent, a malware must perform certain actions, such as making system calls, which require opcodes that are unavailable in text data. Therefore, we see that text malware are constrained to *generate* these instructions *dynamically*. Since a malware usually has many such non-text instructions, dynamic generation of them entails long stretch of valid text instructions, which means high MEL.

Difficulty in Encryption: There are two difficulties associated with encrypting binary in text (and decrypting text back to binary). First, we observe that since the text domain is a proper subset of the binary domain, we cannot have a *one-to-one* correspondence between the two. Therefore, if we are encrypting one byte of binary data into text, the size of the encrypted output will be definitely more than one byte, which means not only the size of the encrypted payload will increase but also the decryption logic will be more complex, thus resulting in a much larger decrypter. Second, in order to have a small decrypter routine, one needs to use a jump instruction with a negative displacement to “go back” to the beginning of the decrypter so that the same routine can repeatedly executed for decrypting different encrypted bytes. However, since all text bytes have 0 in their most significant bit (MSB), one cannot have a negative displacement – which means that all jumps in text instruction stream are in the forward direction. This precludes the possibility of having a small decrypter – for a n-word encrypted payload, one must have $O(n)$ decrypter blocks where each decrypter block will decrypt one individual word. Thus, we posit that text decrypters are large in size, and accordingly a text malware that employs encryption has a high MEL. While it is theoretically possible to overcome these difficulties (by generating the negative displacement dynamically or by using multilevel encryption), that would most likely make the decrypter more complicated and thus increase its size and MEL. We will discuss the multilevel encryption issue in greater detail in Section 7.

2.4 Benign Text has Smaller MEL

A high MEL implies the presence of a long valid (error-free) instruction sequence. Therefore, if invalid (error-raising) instructions are dispersed abundantly in an instruction stream, the chances of having a high MEL is minimal. It transpires that the preceding is true for normal text stream – such invalid instructions do occur frequently in the benign text data due to the following reasons:

Prevalence of Privileged Instructions: The characters ‘*l*’, ‘*m*’, ‘*n*’ and ‘*o*’, which occur frequently in text [1], correspond to privileged I/O instructions that cannot be invoked from any user-level application without generating an error. Benign text data may have these instructions, whereas malware will never have them in its execution path.

Illegal Memory Access: Text instruction streams are very prone to segmentation fault due to attempts to access out-of-bound memory. Since text characters cannot have 1 in their Most Significant Bit, register-register instructions are ruled out in text. Therefore, to manipulate a register, one operand must come from memory for text instructions of two operands. While accessing the memory, a violation can happen in the following ways: 1) uninitialized register (which may contain an arbitrary memory address), 2) wrong Segment Selector (by causing it to access wrong memory segment) and 3) explicit memory address (which could potentially create problems since nowadays it is a common practice to randomize the starting addresses of programs and static libraries [4]). In a random (benign) text stream, such memory-accessing-error events are frequent.

2.5 Detection Strategy for Text Malware

Based on the discussion above, it is evident that unlike benign text, a text malware will have a high MEL. Therefore, a threshold on MEL can be used to determine whether a text stream is malicious or benign. For example, DAWN [8] disassembles the text input and then pseudo-executes all possible execution paths. If the MEL exceeds the threshold, an alarm is raised. It should be pointed out that the contribution of DAWN was *not* limited to this rather straightforward approach of checking against an MEL threshold during pseudo-execution (which has appeared in previous works [10, 2] for classical binary worms), but rather the discovery of new text-specific techniques for identifying invalid instructions in order to prune the number of possible execution paths to be explored, and the demonstration of how adverse the detection results can be if we do not use the techniques. In this paper, we complement DAWN by establishing analytically the fundamental reasons for why the detection strategy of setting a MEL threshold will work.

3 Probabilistic Analysis of MEL

In this section, we answer this question: given a sequence of n instructions with each instruction having a probability p of generating an error, what is the distribution of the longest error-free execution path (MEL)? We elaborate below why deriving the distribution of MEL is important.

We use the detection strategy that, if an incoming instruction stream contains a contiguously valid instruction sequence longer than a certain threshold τ, then it contains a malware with a certain *false-positive* probability α (which is the chance for a benign stream to have a contiguously valid instruction stream of length more than τ purely by accident). It is intuitive to see that the larger the value of τ

is, the smaller the value of α will be. Unfortunately, if we aim at driving α close to 0 in order to avoid false alarms altogether, τ will be very large, which may lead to *false negatives* (the case that real malware is not detected). Therefore, it is important to characterize the tradeoff between false positive and false negative by deriving the mathematical relationship between α and τ. Such a formula will allow the user to select a specific combination of the two values in order to achieve certain desirable performance. While it is possible to estimate the relationship between α and τ experimentally through a training data set, such data can be biased, not representing the general case. In this section we take a probabilistic approach that correlates τ with α using the character frequency distribution of text input.

3.1 Description of the Model for MEL

In our probabilistic analysis, we use Bernoulli trials to model this problem. We start with the assumption that the instructions in a normal input stream occur randomly and independently (this assumption is verified in Section 3.3). Let I_v denote a valid instruction, I_{inv} an invalid instruction, and p the probability for an arbitrary instruction disassembled from a normal stream to be invalid. Consequently, the probability for an instruction to be valid is $(1-p)$. Let n be the number of instructions in an input stream and N be the number of invalid instructions in the stream. It is easy to see that there are $N+1$ contiguously valid instruction sequences, each containing zero or more I_vs and terminating with an I_{inv}. The instruction sequence after the last I_{inv} does not have the terminating I_{inv}. For example, with $n = 17$ and $N = 5$, the following instruction stream, $\overline{I_vI_vI_{inv}}\ \overline{I_vI_vI_vI_vI_{inv}}\ \overline{I_vI_vI_vI_{inv}}\ \overline{I_vI_vI_{inv}}\ \overline{I_{inv}}\ \overline{I_v}$, contains 6 such sequences (instructions in the same sequence are under the same bar), where the longest valid instruction sequence is $I_vI_vI_vI_v\ I_{inv}$ (MEL=5). Now, if we use the term X_i to denote the length of each of these $N+1$ valid instruction sequences, then the MEL is given by $X_{max} = \max\{X_1, X_2, ..., X_{N+1}\}$. Each X_i follows Geometric distribution with parameter p. Although $\sum_0^{N+1} X_i = n$, the X_is can be assumed to be independent (effect of this approximation will be discussed in Section 3.3). Below we derive Prob$[X_{max} \leq x]$, $\forall x \in [0..n]$, which is the cumulative density function of X_{max} (or the MEL).

First, we derive the conditional probability when the number of invalid instructions is fixed at $N = k$, for a specific number k.

$$\begin{aligned}
&\text{Prob}[X_{max} \leq x \mid N = k] \\
&\quad = \text{Prob}\ [max(X_1, X_2, X_3, ..., X_{k+1}) \leq x] \\
&\quad = \text{Prob}\ [(X_1 \leq x) \text{ and } (X_2 \leq x) \text{ ...and } (X_{k+1} \leq x)\ ] \\
&\quad = \text{Prob}\ (X_1 \leq x) \times ... \times \text{Prob}\ (X_{k+1} \leq x) \\
&\quad = [1-(1-p)^x] \times [1-(1-p)^x] ... \times [1-(1-p)^x] \\
&\quad = [1-(1-p)^x]^{k+1}
\end{aligned}$$

We stress that the probability calculated above is conditional on a specific value of N. The actual value of N may vary from 0 to n. Since N denotes the number of invalid instructions occurring among n instructions (with each of the n instructions having a probability p of being invalid), it follows the Binomial distribution with parameters (n, p). Thus, Prob$[X_{max} \leq x]$ over all possible N values is

$$\begin{aligned}
&\text{Prob}[X_{max} \leq x] \\
&= \sum_{k=0}^{n} \text{Prob}\ [N = k] \times \text{Prob}[X_{max} \leq x \mid N = k] \\
&= \sum_{k=0}^{n} \binom{n}{N} p^k (1-p)^{n-k} \times [1-(1-p)^x]^{k+1} \\
&= (1-(1-p)^x)[1-p(1-p)^x]^n
\end{aligned}$$

The Probability Mass Function (*PMF*) for MEL is Prob$[X_{max} = x]$ = Prob$[X_{max} \leq x]$ − Prob$[X_{max} \leq x-1]$ $= (1-(1-p)^x)[1-p(1-p)^x]^n - (1-(1-p)^{x-1})[1-p(1-p)^{x-1}]^n$.

3.2 Automatic Derivation of Threshold τ

We now derive the formal relation between τ and α. The resulting formula allows us to automatically derive the threshold value τ under the constraint that the false-positive probability is bounded by a given value of α.

False positive happens when X_{max} is greater than the MEL threshold τ. Thus, the false-positive probability must be $\alpha = \text{Prob}[X_{max} > \tau] = 1 - \text{Prob}[X_{max} \leq \tau] = 1-(1-(1-p)^\tau)[1-p(1-p)^\tau]^n$.

We can approximate it as $\alpha = 1-[1-p(1-p)^\tau]^n$ since $(1-(1-p)^\tau) \approx 1$. How to determine the values of n and p will be discussed in Section 5.2. The value of α is a user requirement. Given α, n and p, we can calculate the corresponding MEL threshold as $\tau = \frac{\log(1-(1-\alpha)^{\frac{1}{n}})-\log p}{\log(1-p)}$. To verify that the above approximation has insignificant impact on the value of τ, we compare the values of τ obtained using the formula with or without the approximation. For example, when $\alpha = 1\%$, $n = 1540$ and $p = 0.227$ (the parameters used in our experiments), $\tau = 40.61$ with the approximation and 40.62 without (difference of 0.02%). Other reasonable parameter settings also show that the approximation induces only small error in the computation.

Based on the above analysis, if we use the derived τ as threshold, the false-positive probability will be bounded by α. This is very important, since it gives us the flexibility to set the detection sensitivity of an MEL-based detector.

3.3 Verification of the MEL Model

In our model, we assume that valid and invalid instructions occur independently in the benign text. If we can show that the validity of an instruction is independent of the validity of the instruction prior to that, then by induction it can be

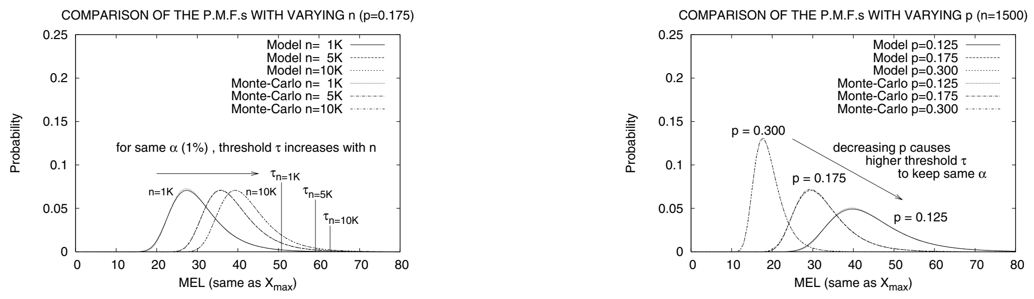

Figure 1. Juxtaposition of the *PMF*s for the MEL from the probabilistic model and from the Monte-Carlo simulation by varying n and p. A near-perfect match can be observed in almost all the cases.

shown that occurrence of any valid or invalid instruction in an instruction stream is an independent event. To prove that, we conduct the Pearson's χ^2 test with the null hypothesis H_0 : in a pair of contiguous instructions $< I_1, I_2 >$, the validity of I_2 is independent of the validity of I_1. To verify this, we construct below the 2×2 contingency table of frequencies as follows. First we disassemble the benign text data used for our testing. Then, considering all possible contiguous pairs of instructions, we count the total number of cases for each of the 4 possible validity combinations and tabulate the results under the "observed" column. The rightmost columns under "expected" indicate the expected numbers as per Pearson's χ^2 test. As we observe, the values are very close; and the corresponding p-value (0.1) is not statistically significant to reject H_0.

	Observed		Expected	
	Valid I_2	Invalid I_2	Valid I_2	Invalid I_2
Valid I_1	8960	2797	8922	2835
Invalid I_1	2797	938	2835	900

The other assumption in our model is that we do not enforce the condition that $\sum_0^{N+1} X_i = n$ and rather assume X_is occur independently. Is it evident that as n increases, the effect of this constraint becomes less pronounced. To verify this, we run Monte-Carlo simulation for the *PMF*(X_{max}) for different values of n and p. There, we toss a coin (with probability of head p) n times and calculate the MEL by taking the maximum distances between two heads that are separated by only tails and no heads in between. As heads are equivalent of invalid instructions, the maximum inter-head distance represents the MEL. The same experiment is run for thousands of rounds and averaged to obtain the distribution of MEL. Finally, we juxtapose the output *PMF* for the MEL from the Monte-Carlo simulation with the *PMF* generated by our probabilistic calculation in Figure 1. We observe a near-perfect match in all cases (especially with larger n), which vindicates our probabilistic model.

We also get one very important intuition from Figure 1. We see that if p decreases, it will require a higher threshold τ to keep the same false positive rate of α. However, higher threshold will mean that a lot of malware will also not get detected. Thus, to have a low value of false negative (in addition to a fixed low value of false positive α), we must find ways to increase p. This is why finding more ways to invalidate instructions in text streams is important, which we achieved in our prior work [8].

4 Applying Automatic Calculation of MEL Threshold in Malware Detection

In the previous section, we derived the distribution of MEL for a given input size n and invalid instruction probability p. From the shape of the distribution, it is evident that if we choose an MEL threshold of τ near the tail of the PMF curve, the probability of a random stream having an MEL higher than τ will be low, and we can calculate the error probability of that event happening accidentally as $\alpha = 1 - (1 - (1 - p)^{\tau})[1 - p(1 - p)^{\tau}]^n$. In this section, we put this theory into practice with an applicable detection scheme.

We stress that the MEL scheme is *not* for detecting every kind of malware. In fact, it applies to *only* those kinds of malware that have the property that the MEL of malicious payload is significantly higher than the MEL of the benign. We have already observed in Section 2.3 that it holds true for text traffic. We investigate the possibility of using MEL-based detection scheme on binary and text as follows.

4.1 Using MEL to Detect Binary Malware

Here, we show that the MEL method, though used for detecting binary worms previously, cannot be used any more.

In Section 2, we have showed that for text streams, the malicious payloads have much higher MEL than benign because of encryption difficulties. However, we note that the same is not true for binary, because there it is fairly easy to come up with a very short decrypter, which will result in a low MEL. Therefore, though it may appear that using MEL method is not suited for detecting binary malware, surprisingly the MEL method has already been applied in APE [10] and Stride [2] to detect a special kind of binary malware, viz. the binary worm. The reason it worked there is because those detection schemes exploited a special property of the binary worms, viz. the fact that binary worms were accompanied by a sled. Those schemes were directed not towards the actual payload (which could be encrypted and thus have a small MEL) but towards the worm's sled (a long sequence of unencrypted valid instructions, and thus having a high MEL). Unfortunately, according a recent survey [5], NOP sleds are almost never used nowadays, probably because the stack addresses today can vary by millions of bytes, and having a sled that long is improbable. Nowadays, most of the worms rather use the "register spring" method that involves no sled [5]. Thus, MEL-based methods (including APE or Stride) cannot catch binary worms any more, which means that we can no longer test our theory on binary malware.

4.2 Using MEL to Detect Text Malware

Other than DAWN [8], we are aware of only one detector (SigFree [12]) that can detect text malware. SigFree works by counting the number of *useful* instructions rather than just valid instructions, which means it is possible that instructions which do not generate any error but does not contribute actively to the flow of data will not be counted. This approach is slightly different than MEL, where all valid instructions are counted. Moreover, SigFree usually keeps the capability of detecting text malware turned off to boost its binary malware detection efficiency [12]. This makes DAWN [8] our candidate of choice to test our probabilistic MEL theory. Instead of using any user-set MEL threshold as in [10], we determine the threshold in DAWN automatically based on the the input size (n) and probability of invalid instructions (p) for any user-set false positive error rate α (using the formula $\tau = \frac{\log(1-(1-\alpha)^{\frac{1}{n}})-\log p}{\log(1-p)}$). We briefly describe the testing details of our detection scheme below.

5 Evaluation

We evaluate the effectiveness of our MEL theory in the following steps: (i) Creating the test data, (ii) Determining the appropriate threshold from the created test data using the MEL theory, (iii) Running the detection algorithm (DAWN [8]) with the threshold determined in the previous step, and (iv) Observing the false positive and false negative rates. The tests were run on an Intel(R) Pentium-IV 2.40 GHz CPU with 1 GB of RAM in a Linux machine.

5.1 Creation of the Test Data

For creating the text malware, the frameworks provided by Rix [9] and Eller [6] were used to convert multiple binary buffer overflow programs (from [3]) into their text counterparts and more than one hundred text worms were created in that way. The effectiveness of the text worms were tested by actually running the vulnerable program and then by observing the spawning of the shell. To check whether a text malware detector is at all needed, McAfee was run on both the binary and text shellcodes and it raised alarms for the binary cases only. For creating the benign dataset, nearly 0.5 MB of real web traffic from our departmental network were collected using Ethereal. After stripping off the headers, 100 cases, each containing nearly 4K text characters, were selected to serve as the benign data.

5.2 Determining MEL Threshold τ

Since we can express the threshold τ as a function of the false-positive probability α with parameters p (probability of an invalid instruction) and n (total number of instructions), we need to first calculate the values of these parameters (p and n) for our test data. For the calculations, we use only the following two entities: 1) the input size C (in number of characters), and 2) the character frequency table (indicating the probability of occurrence for each character), which can either be pre-set (from experience) or can be obtained by a linear sweep of the input character stream in case no preset data exists (like our test condition). We do not need to disassemble any data for determining p or n.

Determining *n*: We know that the total number of instructions $n = \frac{C}{\text{Avg Instr size (bytes)}}$. The average length of an instruction is given by E[length of instruction] = E[length of prefix chain] + E[length of *actual* instruction]. By *actual* instruction, we denote the rest of the instruction *after* the prefix chain, starting with the instruction opcode and including ModR/M, Immediate, Displacement, SIB etc. Now, if z denotes the probability that a character is one of the instruction-prefix characters ($z = 0.16$ in our case), then E[length of prefix chain] = $\sum_{i=0}^{\infty} i \times$ Prob[length(prefix chain) $= i$] $= \frac{z}{(1-z)} = 0.19$. Similarly, E[length of actual instruction] = $\sum_i$ length[instr(i)] $\times$ Prob[instr(i)], where instr(i) represents an *actual* text instruction. In our case, E[length of actual instruction] was found to be 2.4. Thus, E[length of instruction] turns out to be 0.19 + 2.4 $\approx$ 2.6 bytes per instruction. Since C = 4K in our case, $n = \frac{C}{E[\text{instruction size}]} = \frac{4000}{2.6} \approx 1540$.

Determining *p*: We obtain p by adding the probability of I/O instructions and wrong-Segment-override memory-accessing instructions (which are 18.5% and 4.2% according to the frequency distribution of our test data). We disregard the probability of illegal memory access due to unpopulated memory-addressing register, as it requires evaluation of prior instructions and hence it cannot be determined standalone whether it will cause an abort or not. We also take the conservative approach of not using the possible error due to explicit memory address, as the register spring technique exposes the usage of static addresses in Windows [5]. Thus, p turns out to be 0.185 + 0.042 = 0.227 in our case.

Determining the threshold τ: We set the false positive

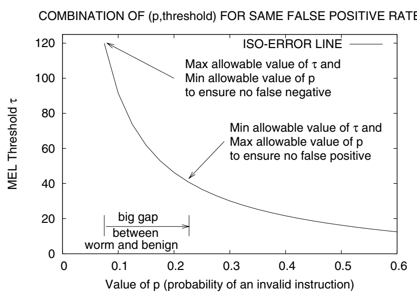

Figure 2. Correlation between τ and p for maintaining same error (false positive) rate $\alpha = 1\%$.

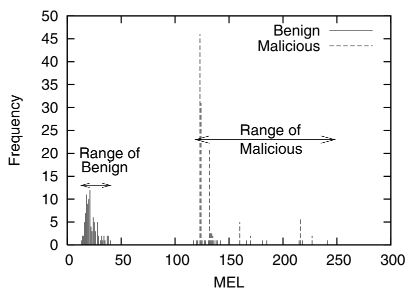

Figure 3. Comparison of MEL frequency charts for benign and malicious text traffic

rate at $\alpha = 1\%$. For this α, we calculate the corresponding threshold $\tau = \frac{\log(1-(1-0.01)^{\frac{1}{1540}})-\log 0.227}{\log(1-0.227)} = 40$ for our calculated experimental parameters $n = 1540$ and $p = 0.227$.

5.3 Experimental Results

In our experiments, the MEL threshold of 40 catches all the malicious cases and not a single benign case gets misclassified as malicious, thus yielding zero false positive and zero false negative rates. To interpret the result even further, we take the MEL from each benign as well as malicious input data, and construct the overall MEL frequency charts (equivalent of PMF of MEL). We compare the frequency distribution of the MEL for benign and malicious test data in Figure 3. For the benign data, the average MEL is near 20, and max MEL is 40 (same as τ), which matches our expectations very well. On the other hand, for the malicious data, the MEL is always above 120, thereby marking a clear differentiator. Also, if we connect the frequency points for benign, we observe that it forms into a shape somewhat similar to the PMF curves in Figure 1, which shows that our model is indeed mimicking the actual behavior. Also, we observe from Figure 2 that the gap between the false positive boundary (p value of 0.227 corresponding to MEL of 40) and false negative boundary (p value of 0.073 corresponding to MEL of 120) is quite large, which means even if the estimated p changed by a small margin, we would still have been able to distinguish the malware from the benign data. Also, the average instruction length from our actual experiment (2.65) was found to be very close to our expected value (2.6) assuming character and instruction independence.

6 Contrasting Our Work with APE

While our work generally follows the direction shown by APE [10] that introduced the concept of MEL, there are a number of significant differences:

- APE did not provide any mathematical foundation of the underlying model, which we do in this paper. As a result, any MEL thresholds in APE is obtained *experimentally*, while in our case the threshold is calculated automatically by the model – there is no “parameter tuning”.
- APE runs on random samples of data, while we examine the full content.
- Unlike our malware detection strategy, APE is not a malware detector but a worm detector. It worked for worms because previously worms used to have a sled, a feature that is almost obsoleted now [5]. As a result, APE’s effectiveness is severely dwindled today.
- Although text malware do have large MELs, we found that APE, in its current form, is not effective for detecting them either. This is because APE, which was designed for binary worms, did not exploit the text-specific properties. The definition of invalid instruction there is narrower than ours; APE considered an instruction invalid only when it is either incorrect or has a memory operand accessing an illegal address. This is a special case of our definition; we introduce new ways to invalidate more instructions in text (like I/O instructions). Moreover, the APE paper [10] did not present *specific* methods to determine which instructions are valid and which are invalid. In our previous work [8], we implemented an APE-like algorithm that did not exploit the text-specific constraints discovered by us, and compared its detection sensitivity and runtime with DAWN’s. The results of the comparison showed clearly that APE is ineffective for text.

7 Limitations and Conclusion

In this section we discuss some of the limitations of our detection strategy. We reiterate the basic principle of our detection method: 1) A text malware must self-mutate to generate potent binary opcodes, 2) this mutation requires a lot of memory-writing instructions, 3) due to difficulties in encryption (including unavailability of loops in text and lack of one-to-one correspondence between text and binary domains) the size of a decrypter is relatively big for text malware, 4) due to randomness property, benign text data does not have such a long error-free executable instruction sequence, and 5) the length of the maximal valid instruction sequence can thus be used to differentiate between benign and malicious text data.

We have already mentioned that generating loops dynamically makes the decrypter complicated and thus longer. We discuss in detail a possible argument against the other encryption difficulty (lack of one-to-one correspondence) below.

We have argued that the absence of one-to-one correspondence between text and binary makes the task of decryption more complex and thus causes the decrypter to be large with high MEL. However, one may overcome this obstacle by using multilevel encryption (Russian doll architecture) in the following manner. First, convert the binary malware into text, and then encrypt this text malware in such a way that the output is yet again text. We observe that in the second step, we are doing encryption within the *same* text domain, which signals the possibility of having a one-to-one correspondence. On the surface, this approach appears to have merit since 1) the final encrypted text data will show very little trend of a text malware, and 2) because of the one-to-one correspondence, one may be able to use *simple* decryption schemes, which means a short decrypter. While it is impossible to consider all possible encryption methods, we put forth our rebuttal to this argument by demonstrating the case of using `xor`, which is usually a favorite choice for encryption. First of all, we observe that there is no *single* decryption key (a text byte) with the property that `xor`-ing it with any other text byte will still yield text data. This is because the text data (0x20–0x7E) occupies a somewhat odd slot in the original ASCII table, and `xor`-ing two characters from text data often yields a result that is not text. As shown in Figure 4, if we divide the 95-char text domain into three nearly equal-sized parts (viz. 0x20–0x3F, 0x40–0x5F, and 0x60–0x7E), and if we `xor` *any* two bytes from the *same* part, then the output will belong to the non-text domain 0x00-0x1F. This means that, in order to use `xor` directly, we cannot use a *constant* key for all of the text. Consequently, the decryption logic will have to be more complex too, leading to a not-so-small decrypter.

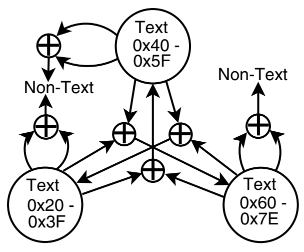

Figure 4. Result of XOR-ing 2 text characters

We emphasize that while we offer a novel way to differentiate between benign and malicious text traffic, this means that we have merely made the task of an attacker significantly harder. As per our limited experiment, the difference between the maximum length of the valid instruction sequence between benign and malicious traffic is currently significantly large. To the best of our knowledge, no text malware employing encryption to this date has been able to come up with a decrypter smaller than our current threshold. However, as security is a cat-and-mouse game, in future we will invariably see such malware, and we must strive to find more exploits to counter that.

We would like to reiterate that while our approach is similar to some other existing MEL-based schemes [10, 12, 2], a fundamental difference exists. In our detection scheme the MEL threshold is obtained purely from the statistical properties of text traffic, while for the rest it is obtained experimentally from the MEL of the benign data.

To conclude, we have analyzed the Maximum Executable Length method and laid mathematical foundation to this theory. We have shown that such theory can be used to detect malware in the text domain but no longer in the binary domain. We have incorporated our MEL model into a text malware detector that is easily deployable, signature-free, requires no parameter tuning, has user-configurable detection sensitivity, and is extremely robust.

References

[1] Compact Oxford Dictionary.

[2] P. Akritidis, E. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic Sled Detection through Instruction Sequence Analysis. *In Proc. of the* 20th *IFIP International Information Security Conference*, May 2005.

[3] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[4] S. Bhatkar, R. Sekar, and D. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *In Proc. of the* 14th *USENIX Security Symposium*, July 2005.

[5] J. Crandall, S. Wu, and F. Chong. Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. *In Proc. of DIMVA*, July 2005.

[6] R. Eller. Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel platforms. *http://community.core-sdi.com/~juliano/bypass-msb.txt*, 2003.

[7] O. Kolesnikov and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. *Technical report, Georgia Tech*, 2004.

[8] P. K. Manna, S. Ranka, and S. Chen. DAWN: A Novel Strategy for Detecting ASCII Worms in Networks. *In Proc. of 27th IEEE INFOCOM (short paper)*, April 2008.

[9] rix. Writing IA32 Alphanumeric Shellcodes. *Phrack*, 11(57), 2001.

[10] T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. *In Proc. of* 5th *International Symposium on RAID*, October 2002.

[11] K. Wang and S. Stolfo. Anomalous Payload-based Network Intrusion Detection. *RAID 2004: In Proc. of* 7th *Internation Symposium on Recent Advances in Intrusion Detection*, September 2004.

[12] X. Wang, C. Pan, P. Liu, and S. Zhu. A Signature-free Buffer Overflow Attack Blocker. *In Proc. of* 15th *USENIX Security Symposium*, July 2006.