On the Infeasibility of Modeling Polymorphic Shellcode for Signature Detection

Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, Salvatore J. Stolfo
Department of Computer Science, Columbia University
{*yingbo,locasto,angel,angelos,sal*}@cs.columbia.edu

Abstract

Polymorphic malcode remains one of the most troubling threats for information security and intrusion defense systems. The ability for malcode to be automatically transformed into to a semantically equivalent variant frustrates attempts to construct a single, simple, easily verifiable representation. We present a *quantitative* analysis of the strengths and limitations of shellcode polymorphism and consider the impact of this analysis on the current practices in intrusion detection.

Our examination focuses on the nature of shellcode *decoding routines*, and the empirical evidence we gather illustrates our main result: that the challenge of modeling the class of self-modifying code is likely intractable – even when the size of the instruction sequence (*i.e.*, the decoder) is relatively small. We develop metrics to gauge the power of polymorphic engines and use them to provide insight into the strengths and weaknesses of some popular engines. We believe this analysis supplies a novel and useful way to understand the limitations of the current generation of signature-based techniques. We analyze some contemporary polymorphic techniques, explore ways to improve them in order to forecast the nature of future threats, and present our suggestions for countermeasures. Our results indicate that the class of polymorphic behavior is too greatly spread and varied to model effectively. We conclude that modeling normal content is ultimately a more promising defense mechanism than modeling malicious or abnormal content.

1 Introduction

Code injection attacks have traditionally received a great deal of attention from both security researchers and the blackhat community [2, 15], and researchers have proposed a variety of defenses, from artificial diversity of the address space [6] or instruction set [20, 5] to compiler-added integrity checking of the stack [12, 16] or heap variables [38] and "safer" versions of library functions [4]. Other systems explore the use of tainted dataflow analysis to prevent the use of untrusted network or file input [11, 32] as part of the instruction stream. Finally, a large number of schemes propose capturing a representation of the exploit to create a signature for use in detecting and filtering future versions of the attack. Signature generation methods are based on a number of content modeling strategies, including simple string-based signature matching techniques like those used in Snort [40]. Many signature generation schemes focus on relatively simple detection heuristics, such as a traffic characteristics [39, 22] (*e.g.*, frequency of various packet types) or identification of the NOP sled [42], while others derive a signature from the actual exploit code [26, 46, 28] or statistical measures of packet content [44, 43, 31], including content captured by honeypots [47].

1.1 Polymorphism

While injected malcode can follow a wide variety of internal arrangements in order to trigger a particular vulnerability, such code is conceptually structured as a set that contains a NOP sled, a sequence of positions containing the targeted return address, and the actual executable payload of the exploit *i.e.*, *shellcode*. Recent years have seen the application of polymorphism and metamorphism techniques to disguise malcode [19]. Some approaches to polymorphism were based on replacing sequences of instructions with semantically equivalent variants (metamorphism). Another approach is to use code obfuscation and masking, such as encrypting the shellcode with a randomly chosen key. A decoding engine is then inserted into the shellcode and must run before the exploit to reverse the obfuscation during runtime, resulting in a fairly standard conceptual format for shellcode: *[nop...][decoder][encrypted exploit][ret addr...]*. Only the decoding routine now need be polymorphic; this task proves less daunting than morphing arbitrary exploit code. Rapid development of polymorphic techniques has resulted in a number of off-the-shelf polymorphic engines [19, 14, 29, 7]. Countermeasures to polymorphism range from emulation methods [35, 3] to graph-theoretic paradigms aimed at detecting the underlying vulnerability [8] or signatures based on higher order information such as the control-flow graph of the exploit [25, 9].

1.2 The Challenge of Modeling Decoder Polymorphism

Our motivation is derived from the question of whether it is possible to compute and store all members of the class of decoders – and if so, how difficult such a task would be. Doing so would enable us to determine the range, type, and power of signatures required to defeat polymorphic techniques, and it would allow us to build statistical models of such malcode. Since normal network data traffic should not contain executable binaries (with program downloads being a *relatively* rare exception), being able to detect the presence of a decoder within such a data stream would provide a strong indication that the data contains malicious content. In our study of polymorphism, we assume that the payload can be disguised perfectly and do not attempt to model the payload portion of the malcode. Instead, our work focuses on examining the decoder portion of the malcode because we feel that it is the most constrained portion of the attack vector, since it represents executable code which must perform a specific decryption role. Furthermore, we do not attempt to model the NOP sled portion of the malcode (as a number of previous research efforts have); we discuss why modeling NOP sleds is hard, if not intractable, as shown by [19, 14].

The research we describe in the remainder of this paper began to address the aforementioned challenge problem by generating a mapping of decoder space. Our original goal aimed at using this pre-computed mapping to discover whether or not we could augment current signature and statistical detection techniques with a *fast* classification method to detect byte strings that were likely decoders embedded within that traffic. We shortly discovered, however, that the decoder sequences our system generates could easily be used to frustrate the capabilities of both signature-based systems and content-based anomaly detections tools – particulary those based on frequency of n-grams [43] *and* more advanced n-gram traffic profiles such as Anagram [21].

1.3 Contributions

Our research results provide a number of contributions that help improve understanding of the polymorphic shellcode problem:

- We propose *metrics* that can be used to gauge the strengths of polymorphic engines, and use these to examine some of the current state-of-the-art engines. We believe our methodology to be novel and

insightful in a space that has generally been lacking in quantitative analysis.

- We illustrate the ultimate futility of relying on straightforward string-based signature schemes by showing that the class of n-byte decoder samples spans n-space. Although our results should not be interpreted as a call for the immediate abandonment of *all* signature-based techniques, we believe that there is a strong case for investigating other protection paradigms and techniques.
- We propose methods that can be used to enhance the design of polymorphic engines in an effort to forecast potential future threats.
- We show that given any normal model, there is a significant probability that a sucessful targeted attack can be crafted against it.

The remaining sections analyze a range of polymorphic shellcode behaviors and present empirical evidence that the class of self-modifying code is indeed too large to compute and store in a compact signature-based representation. Our analysis helps explain why signature-based detection currently works, why it may work in the short term, and why it will progressively become less valuable. We also discover that shellcode behavior varies enough to present a significant challenge for statistical approaches. Finally, we study some of the state-of-the-art polymorphic engines and propose metrics to gauge their strengths. We utilize a combination of emulation, genetic algorithms, and various statistical methods to support our conclusions.

2 Design

The goal of our work is to explore the space of all byte sequences of length n and observe the magnitude of the subspace spanned by the class of self modifying code within this n-space (*i.e.,* discover how many decoders can be constructed given n bytes). This problem statement helps us to better understand the potential difficulty of modeling the decoder class; the number of unique sequences forecasts how many string-based signature sequences we would need for signature-based detection as well as provides some insight into the hardness of statistical modeling.

Since the number of possible strings grows exponentially given the length of the byte string (to be specific, $2^{8 \cdot n}$ where n is the length), we restrict our attention to byte strings of length 10 in order to make our search feasible. Most decoders in the wild have a length of 20 to 30 bytes; however, the transformation section of code usually makes up a smaller portion of the string. Our restricted, 10-byte examination reduces the search space to 2^{80} strings. This problem remains intractable if we were to explore the space one byte at a time. To overcome this difficulty, we make use of genetic algorithms [37]. The remainder of this section describes our methodology in more detail.

2.1 Decoder Detector

Valgrind's [30] binary supervision enables us to add instrumentation to a process without modifying its source code. In particular, Valgrind provides support for examining the memory accesses a process makes. We implemented a Valgrind tool to detect self-modifying code, which we define as code that modifies bytes within a small distance of itself. This technique bears similarity to work done by Markatos *et al.* [35]. Whereas they perform online detection by filtering all network content through the detector to search for the presence of decryption engines, we use our code generator in an offline manner to precompute a set of byte strings that perform self-modification. For every 10-byte string, we constructed a new instance of a program to redirect the thread of execution into the buffer. We center our attention on instruction sequences

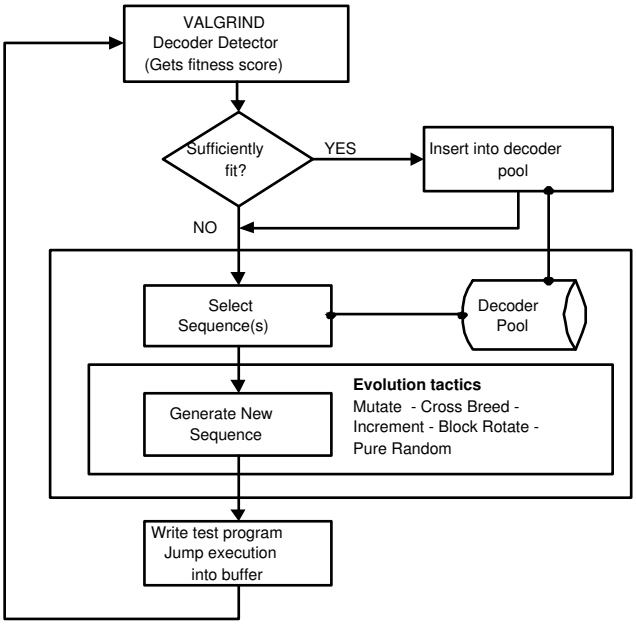

Figure 1: *Decoder search engine flow chart.* An overview of our decoder search engine. We construct our library of decoders using a feedback loop that creates candidate decoders, confirms that they exhibit sufficient decoding behavior, and uses them to generate more samples.

that modify code within 200 bytes of itself in either direction. We define *self-write* as writing to a memory location within this range. We define *self-modify* as reading from a memory location within this range and then, within the next four instructions, performing a write to the same location, as is the behavior of decoder instructions such as *xor,add,sub,...etc*.

2.2 Genetic Algorithms

Genetic algorithms represent an optimization technique from classic AI. These algorithms prove most useful in problems with a large search space domain: problems where it would otherwise be infeasible to calculate a closed form equation to directly optimize a solution. Instead, various solutions are represented in coded string form and evaluated. A function is defined to determine the "fitness" of the string. GA algorithms combine fit candidates to produce new strings over a sequence of epochs. In each epoch, the search evaluates a pool of strings, and the best strings are used to produce the next generation according to some *evolution strategy*. Genetic algorithms belong to the class of evolutionary algorithms that are modeled after biological evolutionary processes. For a more detailed discussion, we refer the reader to Norvig and Russell [37].

We defined a fitness function for our experiments that scores each *self-write* operation a 1 and each *self-modify* operation a 3. The higher score for the latter operation reflects our interest in identifying instruction sequences that represent the *xor,add,sub,...* behavior of decoders. The sum of the behavior scores of a 10-byte string defines its fitness. Any string with a non-zero score therefore exhibits polymorphic behavior. We used a dynamic threshold for minimum acceptable polymorphic behavior as 5% of the *average* polymorphic score of the previously found sequences; we bootstrapped with an overall minimum score of 6. The threshold was used in order to ignore strings which performed one or two modifications only; we wanted to capture strings that exhibited a significant amount of polymorphic behavior (*i.e.,* it encapsulated some form of a

loop construct)[1]. We stored all strings that met the polymorphic criteria in an associative array (used to preserve uniqueness) that we term the "candidate decoder" pool. We observed that the average fitness value reached into the hundreds after a few hundred epochs. We seeded our search engine with two decoder strings extracted from *ShellForge* [7] and roughly 45,000 strings from Metasploit [29] in order to obtain a good distribution of starting positions to begin the search.

Our method uses the following list of evolution strategies, as illustrated in Figure 1. These strategies are designed to maximizes the range of search in n-space.

1. `Increment`: The lowest significant byte is incremented by one modulo 255, with carry. We use this technique after finding one decoder to then undertake a local search of the surrounding space.
2. `Mutate`: A random number of bytes within the string are changed randomly. Useful for similar reasons, except we search in a less restricted neighborhood.
3. `Block swap`: A random block of bytes within one string is randomly swapped with another random block from the *same* string. This technique helps move blocks of instructions around.
4. `Cross breed`: A random block of bytes within one string is randomly swapped with another random block from *another* string. This technique helps combine different sets of instructions.
5. `Rotate`: The elements of the string are rotated to the left position-wise by some random amount with a wrap-around. This is to put the same instructions in different order.
6. `Pure random`: A new purely random string is generated. This adds variation to the pool and help prevent the search from getting stuck on local max.

For each sequence, we automatically generate a new program that writes the string into a character buffer between two nop-sleds of 200 bytes each. The program then redirects execution into that buffer, effectively simulating a buffer overflow attack. We then retrieve the fitness score of that string from the decoder detector, evaluate it, and continue with the search according to the process described above. Figure 1 shows the control flow diagram of our system.

An alternative search procedure would parameterize the actual `x86` instruction set into a genetic algorithm search package and dynamically write decoders. Unfortunately, the presence of variable length instructions used in this approach would restrict us from imposing a limit on the size of the examined strings. Instead, our analysis performs a search for self-modifying code in n-space at the byte level. We do, however, adopt this alternative approach in the second part of this paper. We connect our initial results to real-world systems by applying the same analysis techniques to decoders generated using commonly available polymorph engines.

We only study shellcode behavior for the `x86` Intel architecture. The 10-byte strings that we find mainly represent the substrings that correspond to the self-modification behavior that are present within full-length decoders that are typically 20 to 30 bytes long. These strings are likely to be the sequences that any signature-based IDS would need to extract to use in its model or signature database.

3 Evaluation

The main purpose of our evaluation is to assess the hypothesis that the class of self-modifying code *spans n-space* where n is the length of the decoder sequence.

[1] We used a four second runtime limit in our Valgrind decoder detector tool as we periodically find strings that perform infinite self modifying loops.

First sequence:	000	000	000	000	000	000	000	008	036	012
Last sequence:	255	255	156	201	159	235	220	007	012	081
Mean:	90.00	65.65	145.09	153.20	138.91	126.89	123.30	138.25	134.30	126.14
Standard deviation:	71.59	71.41	86.23	77.85	80.43	83.74	86.29	82.00	74.86	75.56

Table 1: *Candidate decode pool statistics* All measurements are shown in decimal form.

3.1 GA Search Results

Our genetic algorithm search found roughly *1.75* million unique sequences after several weeks of searching with currently no signs of slowing down. In the following sections, we show that the class of n-byte self-modifying code not only spans n-space but seemingly saturates it as well. These results give us a concrete idea of the hardness of the polymorphic modeling problem.

To demonstrate that our generated set of polymorphic sequences spans n-space, we first sort the sequences. We then show that our sample pool represents a chain of sequences of which the first string is very close to $\{$`x00,x00,...,x00`$\}$ and the last string is close to $\{$`xFF,xFF,...,xFF`$\}$. We present a metric to determine the distance between two strings and use it to show that the distance between consecutive sequences are relatively small, indicating a well spread sampling from n-space. Table 1 shows that the first and last sequences from our candidate decoder pool are relatively close to the opposite ends of n-space. We also observe a mean close to the center of the space and high standard deviations. These results support the conclusion that decoders are uniformly and densely spread throughout the space of `x86` instruction sequences.

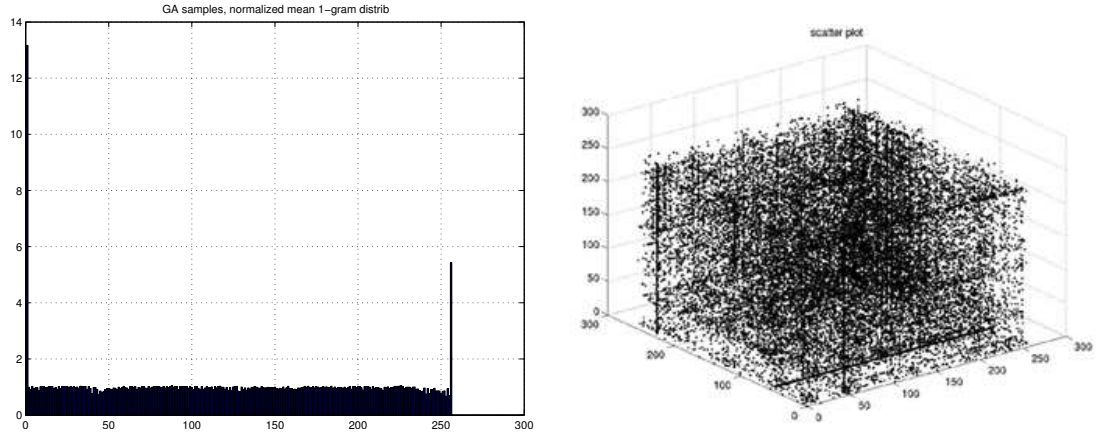

Figure 2: Results. (a) 1-gram distribution (b) 3-gram scatter plot

For each sequence in our sample pool, we compute a byte histogram, *i.e.,* a 1-gram distribution model. Statistical IDS detectors typically operate under the assumption that the class of malcode that they attempt to model exhibit a certain 1-gram distribution, or "byte spectrum" that can be modeled and used to design a classifier to separate malcode from normal traffic. This paper shows modeling malcode is very hard. In the ideal case, a particular class of malcode would exhibit a stable 1-gram distribution that can be represented by some closed form model such as a mixture of Gaussians. Figure 2(a) shows the average histogram of

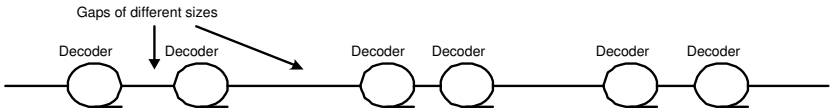

Figure 3: Gaps between decoders, defined by equation (1)

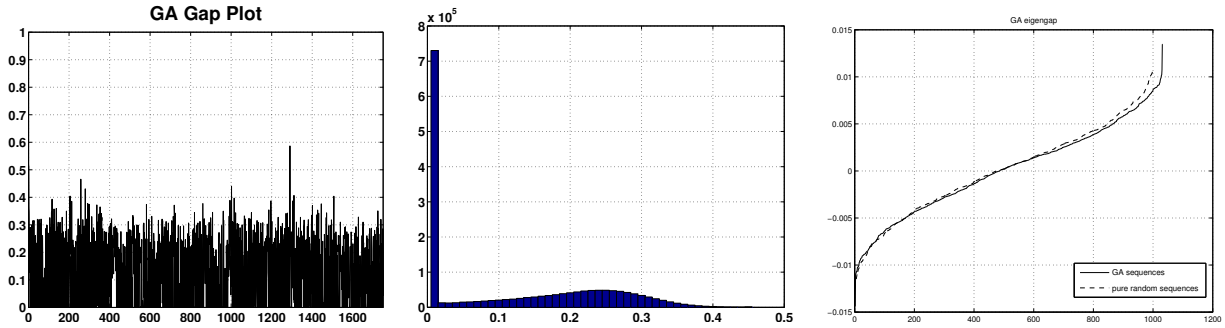

Figure 4: (a) Plot of the pair-wise gaps (b) Gap histogram (c) Eigen-gap plot. Notice that in (c) there are two almost overlapping plots: the solid corresponds to the decoder sequences and the dotted corresponds to randomly generated numbers in the same range. We are only interested in the fact that the two lines are almost identical and thus the decoder class is very close to the random distribution.

the sequences, normalized by the variance. We can see that the sample pool contains no distinguishable distribution but is rather closer to white noise — with the exception of the $\{\texttt{x00}\}$ and $\{\texttt{xFF}\}$ values, which are likely to be padding artifacts. This point in the paper simply says modeling malcode is not easy. Given this examination of 1-space, we next measured 3-space and display the results in Figure 2(b) to show the scatter plot of all 3-grams extracted from all of the candidate decoder pool. This plot shows that, for 3-grams, the space is well saturated.

Now we aim to show that our results for 3-space generalizes to the full 10-space sample pool. We do so, as previously mentioned, by examining the distance between the links of the candidate decoder chain. We define a simple metric to allow us to measure the distance between two strings.

$$\delta(x, y) = \min_{0 \le r < n} \left[\frac{\left|\left| x^{(i+r)} - y^{(i+r)} \right|\right|}{||x|| + ||y||} \right] \tag{1}$$

where $x^{(i)}$ represents the i^{th} character in string x and $||\cdot||$ represents the Euclidean norm. The distance is normalized by the norm of both strings to remove the effect of length on the metric. The smaller the value for $\delta(x, y)$ the more similar the strings are to each other.

In the ideal case we would find all possible decoder samples within 10-space. Since this is not feasible, we can only make observations of the sequences that we do have, and gauge their relative distributions. For any two consecutive sequences in the pool, we define *gap* to be the distance between them according to our metric described above. A large gap indicates that a large portion of 10-space was not explored, whereas a small gap indicates good coverage in that section of 10-space. We can then find the gaps between all sequences in the sample pool and observe if our method failed to explore any portion of 10-space, indicated by a large gap. The *gap* between the decoder is the distance from one decoder to the next after all of the decoders have been sorted. The distance metric is given by equation 1. Figure 3 shows our concept.

Figure 4(a) shows the gap plot of the sequences — uniformly sub-sampled so that plot can be recogniz-

able. We see that it is relatively well covered, the gaps are relatively stable within a certain range, and there are no sharp spikes (a big portion of 10-space missing) or large areas of low gaps (over exploration of one particular section of 10-space).

A histogram of the sequence gaps are shown in Figure 4(b). The histogram shows a bimodal Gaussian distribution with the first Gaussian centered around the low values. These gaps represent the "increment" and "mutate" evolution strategies (*i.e.,* after we find one good sequence, we modify it slightly to retest). These values correspond to roughly only a few bytes being different within the two decoders and the difference is only by a small amount. The second gaussian represent the larger gaps, which are fewer, and centered around a value of 0.25, which translates roughly to an average of 1.7-byte difference in each of the 10-bytes, which is still fairly close.

In addition, we employ a popular clustering approach from machine learning to analyze the randomness of the distribution of our candidate decoder sequences. Spectral clustering is a graph-theoretic data clustering algorithm and is useful because it can perform unsupervised learning without any assumptions about the underlying distributions of the data samples. Instead, it clusters data by finding the optimal cuts in a fully connected weighted graph where each node is a data sample. The optimal cuts (classification separations) are the ones that maximize the sum of the edge weights of the individual disjoint subgraphs where the edge weight is defined by some similarity metric. For our similarity metric[2] we used $\frac{1}{\delta(x,y)}$.

One can view spectral clustering as a relaxation of the classic Normalized Cuts method (which is NP-complete) into an eigenvector decomposition problem. Spectral clustering helps infer some information about the separability of the dataset based on observing the eigenvalues of the Laplacian matrix (which is a matrix representation of a graph). Unfortunately, further discussion of this topic exceeds this paper's scope; we refer the reader to Ng *et al.* [33] for more information.

Our research need only investigate these eigenvalues to show a sufficiently random and well-distributed candidate decoder pool. We find these eigenvalues using spectral clustering and then generate another equally sized set of *completely random* 10-byte strings and run the same analysis on this random set. A quick comparison of the eigenvalues of these two sets should confirm the randomness of the decoder distribution.

Figure 4(c) shows a plot of the two sets of eigenvalues. The solid line represents eigenvalues generated for the decoder sequences[3], and the dashed line represents eigenvalues from the random set. We can see that the candidate decoder pool appears very similar to the random set. *This result confirms our hypothesis that the class of* 10*-byte decoder sequences spans* 10*-space.* We have demonstrated the span of the decoder class for 1-space, 3-space and 10-space. We believe the results are likely to carry over to larger n-space since the larger sequences can contain the lower-length sequences within them.

3.2 Implication of Results

Our results have shown the span of the decoder class. The challenge of signature-based detection is to model a space somewhere on the order of $O(2^{8 \cdot n})$ sequences to catch all potential polymorphic behavior with a signature-based method. To put this task in perspective, there exist an estimated 2^{80} atoms in the universe. Thirty-byte decoders represent a space with a magnitude of $O(2^{240})$ potential signatures – we would much sooner run out of atoms in the universe before attackers run out of decoders. Thus, signature-based approaches can only play catch-up with an enemy that has a seemingly endless space to hide.

Potentially more troubling is the implication that regardless of what the normal model of traffic for a particular site may be, we have shown that there exists a certain probability that a range of decoders would

[2]The reciprocal of the distance metric has to be used in order to define a positive semi-definite Gram matrix.

[3]Spectral clustering requires eigenvalue decomposition of matrix of size $d \times d$ where d is the number of samples. Therefore, the sequences were sorted and *uniformly* sub-sampled.

fall within the span of that normal model since sequences which exhibit polymorphic behavior span most of n-space.

4 Analysis of Polymorphic Engines

Our empirical results show that the class of polymorphic malcode is too large to model. Previous research on automatic signature generation [22, 31] has reported successful detections of many existing polymorphic engines, some of which are from Metasploit.

In this section, we aim to explain in detail our approach by analyzing some of these popular state-of-the-art polymorphic engines. We show that the reason why signature based methods are successful is that individual engines may leave artifacts which can be exploited for detection purposes. However, these artifacts are not strongly correlated with polymorphic behavior itself and look very different across different engines — thus they cannot be generalized to detect polymorphic behavior outside of their training class.

In the following sections, we examine the strengths and weaknesses of some of the polymorphic engines used in the wild today and present metrics which we use to gauge their strength.

We base our work on the following conjectures about polymorphism:

1. `Variation Strength`: *A polymorphic engine is strong if it generates decoders, of length n, that spans a sufficiently large portion of n-space.*
2. `Propagation Strength`: *A polymorphic engine is strong if for the sequence of decoders that it generates ...$\vec{x}_{i-1}$,$\vec{x}_i$,$\vec{x}_{i+1}$..., $\vec{x}_i$ looks significantly different from $\vec{x}_{i+1}$.*

Note that we use the vector variable notation to refer to a decoder sample. Under these metrics, we analyze six popular engines used in the wild today: ADMmutate, CLET, and four engines from Metasploit: Shikata Gai Nai[4], Jumpcall additive, Call4dword and fnstenv mov.

For our analysis we present several metrics which can be used to gauge the strengths of polymorphic engines then present our layout for a new paradigm in engine design where the challenge is reformulated as an optimization problem; one where a metric similar to the one we proposed can be used as the optimization criteria. This work is an attempt at pre-emptively identifying the future potential for polymorphism.

4.1 Variation Strength

For any given polymorphic engine, if the decoders produced by that engine are sufficiently well spread, it is naturally more difficult to model that engine. Many signatures must be used, and for statistical methods, any model trained over the samples would increase the false positive rate as they might be forced to over-generalize.

The images in Figure 5 were generated by taking a single shellcode sample and encrypting it with each engine 10,000 times to generate 10,000 unique shellcode sequences. From these sequences, the decoder portion were extracted, sorted, sub-sampled uniformly, stacked together, and finally displayed as an image. This allows invariances in the samples to show up clearly in the form of vertical bands as shown in Figure 5. The images easily display the invariant subsequences that these engines generate; these artifacts are the exploitable signatures that several methods [22, 31] end up locking-on to in order to detect encrypted shellcode from those engines.

The important thing to note is that these invariances do not hold across different engines – as we can see from the images, even though these engines perform the same basic actions to decode a string within a small distance of itself in memory. For example for CLET, the vertical band represents clearing of registers.

[4] A common Japanese cultural phrase meaning "nothing can be done about it".

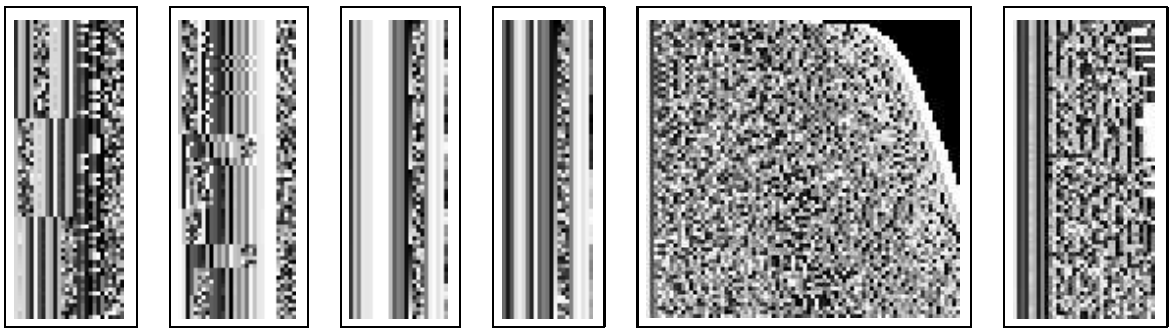

Figure 5: Spectral images to show variation strength (a) Shikata Na Gai (b) jcadd (c) call4dword (d) fnstenv mov (e) ADMmutate (f) CLET. Each pixel row represents a decoder from that engine and each individual pixel value represents the corresponding byte from that decoder. The dark region at the upper right corner of (e) is padding of zero bytes that we added so that ADMmutate's variable length decoders can be stacked together.

These spectral images demonstrate range of decoders that can be generated using existing off-the-shelf polymorphic engines. We may then ask, what is the range of sequences we can expect to see in future polymorphic engines? For this question, we refer to our earlier result that self-modifying code of length n is likely to span a significant portion of n-space. The spectral bands show that it is possible to train detectors from sequences generated by various specific engines by focusing on these areas of the decoder. However, the range of polymorphic behavior is so great that new models must be continuously trained as newer and newer engines are developed – these bands are clearly not consistent across different engines.

We now present a way to gauge such a variance so that a score may be assigned to describe the range of decoder polymorphism exhibited by a specific engine. In order to do so, we examine the magnitude and span of the covariance matrix of the dataset. The covariance matrix is a second order metric used to capture the variance between any two dimensions for a range of samples seen in a distribution of n-dimensions. The equation for the covariance matrix[5] is given as:

$$\Sigma = \frac{1}{L}\sum_{i=1}^{L}(\vec{x}_i - \vec{\mu})(\vec{x}_i - \vec{\mu})^T \tag{2}$$

Where $\vec{x}_i$ is a decoder sample, and μ is the mean sample, both of which are stored as column vectors. We want to examine the magnitude of the axis of the space defined by the covariance matrix for the decoders samples generated by a particular engine to gauge the span of the distribution of its decoders. Recall from eigenvector decomposition that the vector space can be shifted in place to find a different set of basis vectors to represent the same space.

$$\Sigma\vec{v} = \vec{v}\lambda \tag{3}$$

Where $\vec{v}$ are the eigenvectors and λ are the eigenvalues. In order to analyze the magnitude of the decoder sample distribution, we first encode a single shellcode sample 10,000 times to generate 10,000 unique sequences, then use these to generate the covariance matrix according to equation 2. Next, we perform eigenvalue decomposition of the covariance matrix. We now have a new set of eigenvectors which spans this matrix, viewable as axes in a new shifted space and the magnitude of the axes are given by the eigenvalues. We can then sum up the eigenvalues to measure the magnitude of the distribution. We therefore define the variation strength of a polymorphic engine to be the *scaled sum of the eigenvalues* of

[5]Note that in order to define a full ranked covariance matrix, you need more samples than the dimensionality of your data sample

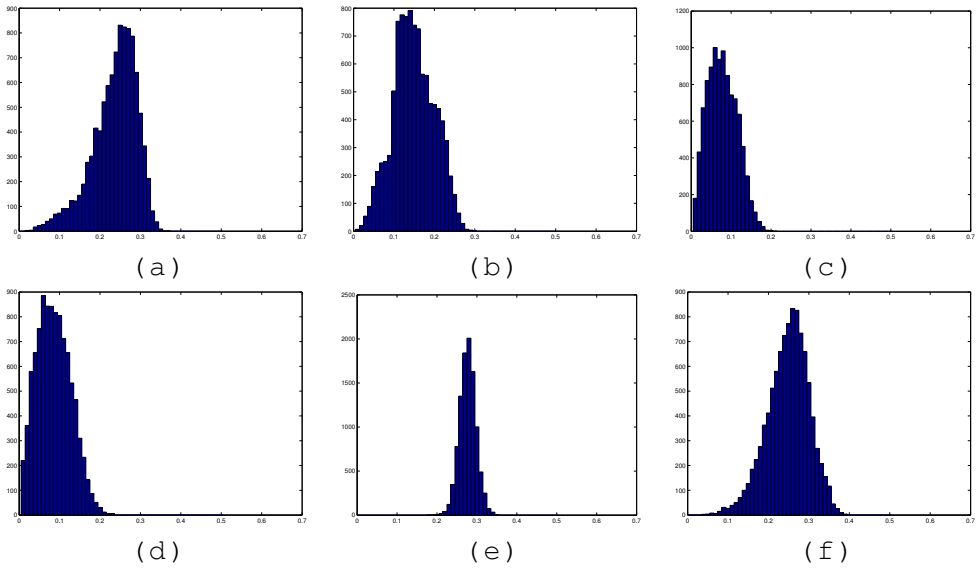

Figure 6: Gap histograms to show propagation strength (a) Shikata Na Gai (b) jcadd (c) call4dword (d) fnstenv mov (e) ADMmutate (f) CLET

the covariance matrix of the decoder samples that it generates. An important point to note is that many decoder samples must be extracted from the engine in order to get an accurate estimate, around 10,000 is an appropriate number.

$$\upsilon(engine) = \frac{1}{\xi L} \sum_{i=1}^{L} \lambda_i \tag{4}$$

Where λ_i is the i^{th} eigenvalue and L is the dimensionality of the decoders. ξ is a factor that is used scale the final score since the magnitude of the eigenvalues can get quite large.

4.2 Propagation Strength

A polymorphic engine might have a restricted span (variance), but if the sequences that it generates are sufficiently different, *i.e.* they are sparsely spread out and each decoder looks very different from the next, then it will be difficult to train any generalized statistical models or extract useful signatures until a sufficiently large number of samples from this engine are seen. The metric therefore, influences how long an engine's malcode can propagate before a detector can be properly trained. To visualize the propagation strength of an engine, we can take our previously defined similarity metric between two decoders (1) and find the *gap* between all of the subsequence decoders (sorted). Plotting a histogram of the gaps show how well spread are the decoders generated by a particular engine. For a hard score, we define the propagation strength of the engine to be the expected value of the decoder gap, *i.e.,* the centroid of the distribution, weighted by the number of static bytes present within all of the generated decoders. Figure 6 shows the gap histograms of the various engines. The histograms with their centroids at higher values exhibit stronger propagation strength. The variance of the histogram, *i.e*. the width of the distribution indicates how consistent an engine is at producing a set of non-similar decoders; the lower the variance (tighter the distribution), the better the engine performs. We define the propagation strength of an engine as follows:

Engine	Propagation Strength	Variation Strength	Overall Strength
shikata	23.60	4.79	169.88
jcadd	14.89	3.68	80.68
c4d	7.78	1.09	13.14
fnstenv	8.58	1.18	16.08
admmutate	27.74	6.10	189.26
clet	24.79	4.12	142.62

Table 2: *The decoder polymorphism strengths of various engines under our metric (the first four engines are from Metasploit)*

$$\phi(engine) = \left[100 \cdot \sum_{i=2}^{N} p(\delta(\vec{x}_{i-1}, \vec{x}_i)) \cdot \delta(\vec{x}_{i-1}, \vec{x}_i)\right] \cdot (1 - \frac{\eta}{L}) \tag{5}$$

Where $\vec{x}_i$ is the i^{th} decoder generated by the engine, N is the total number of decoders and L is the length of the decoder. Since the $\delta(x, y)$ metric returns values between [0,1] range, we multiply this value by 100 to scale the final score to a number greater than 1. In the case of variable length decoders, the average length can be used. $\delta(x, y)$ is the previously defined equation (1). η is the number of static bytes in all of the decoder samples, *i.e.,* bytes which are always present within a decoder at a fixed position. Incorporating this scaling factor into the equation allows us to enforce a heavy penalty on engines that leave consistent artifacts in their generated decoders, which can be exploited by signature based IDS implementations.

4.3 Strength of a polymorphic engine

We can now define the total strength of a polymorphic engine $\Pi(\cdot)$ to be a weighted correlation between the propagation strength and the variation strength.

$$\Pi(engine) = \underbrace{\phi(engine)^{\alpha}}_{\text{propagation st.}} \cdot \underbrace{\upsilon(engine)^{\beta}}_{\text{variation st.}} \cdot \underbrace{e^{\frac{L}{\gamma}+1}}_{\text{length-based weight}} \tag{6}$$

α and β are parameters to give different weights to the two different metrics. We also add a length based exponentially decaying factor $e^{\frac{L}{\gamma}+1}$ to the strength measurement. The adjustable γ parameter allows us to give more weight to engines that can generate decoders less than this length. This is used to put a favorable bias on shorter sequences since their variance can be trivially expanded by padding with NOP-equivalent bytes (described in the following section).

Table 2 presents our ranking of these engines based on these metrics. To get our results, we weighted the two metrics equally, $\alpha, \beta = 1$; we specified $\xi = 1000$ to reduce the range of the final score and set $\gamma = 35$ since many simple decoders are within 25-35 byte range.

4.4 Results

Note that three of the engines from Metasploit are not fully polymorphic, their scores were correspondingly lower than the other more powerful engines. The CLET polymorphic engine is in reality very good in that

it allows the user to specify an arbitrary number of decoding *i.e. xor* then *sub* then *add* etc. For our experiments, we tested on the default setting of five instruction operations. While the polymorphic performance of CLET was on par with that of ADMmutate, we found that all decoders generated by CLET contained a unique 9-byte signature string which represents a set of instructions used to clear the working registers and the appropriate jump/call instructions used to load the needed loop counter variable into memory. While overall, CLET is one of the best engines that we've seen, this particular feature makes the decoders easier to detect than the other engines, thus the lowered score. The CLET team acknowledged as one of their weaknesses this static structural layout [14].

4.5 Other Polymorphic Strengths

While we have focused our efforts on studying the range of polymorphic decoder routines, we must also take into account the other sections of malcode which will also have an important impact on detection. Recalling the conceptual structure of a polymorphic shellcode sample (*e.g., [nop...][decoder][encrypted exploit][ret addr...]*), we describe the achievements made by the shellcoder community in disguising each of these different sections.

• `nop-sled`: The most basic design of a nop-sled is a buffer of nop instructions {`x90,x90,...,x90`} which is inserted ahead of the decoder to safely catch the EIP jump. Many signature-based detection systems rely on this artifact for detection. However, many innovations have been introduced to make the nop-sled polymorphic as well, since the nop-sled does not need to consist of actual x90 nop instructions – it only has to pass the flow of execution safely into the decoder without causing system instability. Toward this end, K2 describes the discovery of at least 55 different ways to write such single byte benign instructions [19] and implemented this in the well-known ADMmutate engine. For a nop-sled of length n, this implies potentially 55^n unique nop sleds.

Another more advanced nop-sled design is deployed in the CLET polymorphic engine [14]. Their method finds benign instructions by first finding a set of 1-byte benign instructions, then finding a set of 2-byte benign instructions that contains the 1-byte instructions in the lower byte. Therefore, it does not matter if control flow lands in the 2-byte instruction or if it lands one byte to the right since that position will hold another equally benign instruction. This method can be used recursively to find benign instructions of longer length which can be combined to create the nop-sled. No analysis of the potential of this method exists as far as we know but it is likely to be a very useful polymorphic technique. We can clearly see the challenge of attempting to model this section.

• `ret addr`: Without randomization of the address space, the location of the stack and stack variables on most architectures remains consistent across program executions. Thus, the attacker has an excellent basis for guessing the value of an injected return address in order to redirect EIP into the injected malcode buffer. Using signatures based on the presence of specific address values could be possible if we were to restrict ourselves to specific forms of code-injection attack. However, return address polymorphism is trivial to implement, one needs only to modify the lower order bits [19]. This method causes control flow to jump into different positions in the stack, but as long as it lands somewhere in the nop-sled, the exploit still works. The return address section consists of the return target repeated m number of times, each repeat can be modified v times (where v is some tolerable variance in the *jmp* target) for a total of m^v possible variations in this section.

• `Spectrum shaping and byte padding`: Some recent research has demonstrated the feasibility of polymorphic blending attacks [17, 24] where the malcode attempts to appear similar to benign traffic in terms of their n-gram distributions (or at least different enough from known models). The CLET team's

polymorphic engine [14] is an example of such an technique. Their engine changes the shellcode to take on a new form: *[nop...][decoder][encrypted exploit]**[padding]**[ret addr...]*. In this new padding area, junk bytes are added to make the 1-gram distribution of the entire shellcode appear different. In addition to this, the shellcode itself is ciphered with different length keys. These keys exhibit various different byte distributions which are propagated into the shellcode through the *xor* cipher technique, re-shaping the byte spectrum of the payload. This technique increases both the variation and propagation strengths of a polymorphic engine to make it resistant to a statistical IDS such as PayL [43].

Perhaps the biggest threat to IDSes is that all of these individual techniques can be combined into one single polymorphic engine. Furthermore, *[nop...][decoder][encrypted exploit][ret addr...]* is really just a conventional design which works. There is nothing to prevent the attacker from modifying the sections between the nop-sled and the return address. By using some *jmp* instructions, it is possible to see shellcode in the future of the forms:
[nop...][encrypted exploit][decoder][ret addr...]
[nop...][decoder part 1][encrypted exploit][decoder part 2][ret addr...]
[nop...][padding][encrypted exploit][padding][decoder][ret addr...]
...
etc., a grim scenario for signature-based detection.

4.6 Future Threats

In this section we outline some of the potential implications of our results from the previous sections and provide a glimpse into possible future paradigms of shellcode polymorphism.

Polymorphism as an Optimization Problem Our proposed strength metric not only allows us to estimate the effectiveness of a polymorphic engine but also allows us to conceptually imagine a new paradigm for polymorphic engine design in the form of an optimization problem where such a metric may form the optimization criteria. Engine designers might be able to incorporate optimization methodologies from AI, machine learning, operations research *etc*, into their designs by heavily parameterizing their engines and then searching for the *optimal* parameters which maximizes some strength function such as the one we proposed.

One such parameterization might simply be to add a small recursively defined nop-equivalent sled in front of every instruction in the decoder or a subset of instructions, then break apart the decoder into sections, rearrange them, then add *jmp* instructions to reconstruct the execution flow. The presence of the nop sled in front of individual instructions also allows polymorphism in the *jmp* targets.

These simple methods can be parameterized so that one set of values determine how to rearrange the instruction blocks and another set of values determine how to generate the nop sleds as well as the positions to place them. It would then be possible to use optimization functions such as the genetics algorithms method we used in this paper to find the optimal set of values that maximizes the output of a strength function, thus maximizing the power of the polymorphic engine.

A Multi-decoder chained polymorphic engine The attacker's ultimate goal is to be able to generate stealthy polymorphic engines that have the ability to transform the payload to a specific n-gram distribution and be themselves part of a site's normal traffic. It is very unlikely that a single decoder can achieve both goals simultaneousl, but it is not impossible. We know, for example, that CLET is capable of mapping the attack payload to a wide range of n-grams, that may cover the n-grams that appear in a site's normal traffic. But the CLET engine itself may be detectable. Hence, we can use another polymorph engine to encode CLET and create a chain of decoders that once decoded completely can map an attack exploit to the set of a target site's normal n-grams. To decode the full byte sequence we will have to decode CLET and then CLET

will decode the payload. In general, we combine two decoders A and B. B can map the payload to a normal distribution of our choice. A encodes B rendering B undetectable (A is selected appropriately to be part of the site's normality model also). The byte sequence A, B and payload and nop sled will not be detectable by both signature- and content-based IDS. Our work, as discussed in section 3.2 shows that constructing A is feasible and thus it is likely that standard non-randomized content-based anomaly detection systems will fail to detect such polymorphic code.

Distributed Calculated Attacks We present a potential scenario for a k-vector worm propagation strategy. Using these concepts the attacker can pre-calculate a large set of highly random encrypted worms which, as a whole, is difficult to model as confirmed by the analysis tools. He then divides these worms among multiple k different online repositories (nodes from a botnet perhaps). For each of these repositories there exists a pre-calculated hash map which links together worms that look significantly different. The worm behaves normally as it propagates, typically changing its encryption key with each new attack, but it periodically checks back with the repository and, using the hash map, retrieves and sends out a variant that is significantly different from itself. This method would make it very difficult for automated signature generation methods to keep up with the worms — in all probability, by the time a useful set of signatures have been generated, a new wave of variants has been released (a wave pre-computed to be statistically different from the old). If somehow the defense succeeds in dissecting the worm and uncovering the location of its designated repository, it will only be one in k vectors compromised, since the worms belonging to each vector of attack are known to be statistically different from the others. Signature based IDS are likely to be more vulnerable to this attack.

Training Attacks While one way to attack an IDS is to develop an engine that can generate attacks which can slip by its sensors. Another attack, as described by Chung *et al.* [10], is to send a wide range of attacks against an IDS system, forcing the system to expand their models to catch all of these new threats. If the attack can force the IDS to sufficiently over-generalize, then the large amount of false positives generated would make the engine unusable. The optimization paradigm we discussed can be used to launch exactly this form of attack. By optimizing the variation strength of the engine, the attacker can create a very wide range of decoders, then artificially lowering the propagation strength, lowers the overall polymorphic strength so that his decoders are caught in the wild. (This would correspond to generating large amount of compact clusters of very similar decoders.) However since the variance is high, the cumulative affect of so many alerts forces the IDS to cover too much ground, reaching into the domain of normal traffic. A statistical IDS would naturally be more affected by this attack. This idea of exploiting the false positive rate is well known among the shellcoder community – it is discussed in the documentation and *Phrack* articles of both the ADMmutate and CLET engines.

Distributed Denial-of-Sensor Attacks Along the same line of argument, an attacker can find a model of normal traffic content, then using a distance metric similar to the one proposed in this paper, calculate a large chain of malcode which are statistically similar to this normal model to some degree d, calculated using a metric similar to the one we proposed (1). d can be constrained to be just below the threshold for a front-line malicious content detection system such as Snort but just above the threshold for a second-line anomaly detection system so that an alert is triggered. On systems where emulation and instrumentation based defenses are used such as [35], this flood of *odd* looking content therefore would cause a *Denial-of-Senor* attack as the defense cannot process anymore alerts while it is busy in the emulation environment and possibly a *Denial-of-Service*, depending on the processing policy. The number of distributed attackers needed is the inverse of the slow-down factor of the emulation environment. An instrumentation based IDS would be most affected by this attack.

The results from the first part of the paper show that, since n-byte self-modifying code spans n-space,

the amount of ammunition for these last two attacks would seem endless and would be simple to generate – just by mimicking our experiments.

5 Related Work

Although our work illustrates the extreme difficulty IDSs face in reliably detecting polymorphic shellcode, we would not report such results without being confident in the ability of the IDS community to answer this challenge. The next section proposes outlines of some countermeasures that we believe may be helpful, and we review the current state-of-the-art in this section to help motivate that discussion. Since we cover various detection and protection techniques in Section 1 as part of our introduction to the problem of signature generation for polymorphic shellcode, we provide only a brief treatment of these topics here. For related work in complexity analysis, Spinellis showed that identification of bounded length metamorphic virsuses is NP-complete [41]. In addition, Fogla et.al[17] showed that finding a polymorphic blending attack is also an NP-complete problem.

Attack Techniques Aleph0ne illustrated the basics of smashing the stack [2]. The virus writer Dark Avenger's Mutation Engine influenced the shellcoder K2 to develop shellcode polymorphism [19]. rix [36] proceeds to show how to perform alphanumeric encoding, Sinan Eran [15] showed how to smash the kernel stack, obscue [34] described how to encode shellcode to make it survive ASCII to unicode transformations, the CLET team [14] developed the technique of spectrum spoofing and how to construct a recursive NOP sled and most recently the Metasploit [29] project combined vulnerability probing, code injection, and polymorphism, among other features, into one complete system.

Intrusion Defense Snort [40] is a widely deployed open-source signature-based detector. Exploring how to automatically generate exploit signatures has been the focus of a great deal of research [22, 39, 31, 26, 47, 46, 28, 3]. To generate a signature, most of these systems either examine the content or characteristics of network traffic or instrument the host to identify malicious input. Host-based approaches filter traffic through an instrumented version of the application to detect malcode. If confirmed, the malcode is dissected to dynamically generate a signature to stop similar future attacks.

Abstract Payload Execution (APE) [42] examines network traffic and treats packet content as machine instructions. Instruction decoding of packets can identify the *sled*, or sequence of instructions in an exploit whose purpose is to guide the program counter to the exploit code. Krugel *et al.* [25] detect polymorphic worms by learning a control flow graph for the worm binary with similar techniques. *Convergent static analysis* [9] also aims at revealing the control flow of a random sequence of bytes. The SigFree [45] system adopts similar processing techniques, and Markatos *et al.* [35] propose running every packet through an emulation environment to detect the presence of polymorphic malcode.

Statistical content anomaly detection is another avenue of research, and PayL [43] models the 1-gram distributions of normal traffic using the Mahalanobis distance as a metric to gauge the normality of incoming packets. Anagram [21] caches known benign n-grams extracted from normal content in a fast hash map and compare ratios of seen and unseen grams to determine normality.

Research on *vulnerability-specific* protection techniques [13, 8, 18] (and dynamic taint analysis [11, 32] in particular) explores methods for defeating exploits despite differences between instances of their encoded form. The underlying idea relies on capturing the characteristics of the vulnerability (such as a conjunction of equivalence relations on the set of jump addresses that lead to the vulnerability being exercised: in other words, the control flow path).

Proactive Techniques Preventing intrusions by removing the weaknesses of current execution environments is another area of active research. Data Execution Prevention (DEP) is used in the latest Windows

OS to flag certain memory areas as non-executable in order to prevent code injection, similar to the WˆX feature in BSD systems. StackGuard and similar techniques [12, 16] instrument the stack to detect unauthorized modifications such as those stemming from a buffer overflow. *Program shepherding* [23] validates branch instructions in IA-32 binaries to prevent transfer of control to injected code and to ensure that calls into native libraries originate from valid sources. Abadi *et al.* [1] propose formalizing the concept of Control Flow Integrity, observing that high-level programming often assumes properties of control flow that are not enforced at the machine language level. CFI statically verifies that execution remains within a control-flow graph (the CFG effectively serves as a policy).

Randomization based defenses work on the principle of making the host a moving target. Address space randomization prevents return-into-libc type attacks. Instruction set randomization [20, 5] has been proposed to make it impossible for the attacker to write executable code for a target system. Randomized stack header padding is used in BSD to add a small offset to the location of the return address buffer on the stack in order to cause the `[return addr..]` portion of the attacker's code to misalign.

6 Countermeasures

Our analysis of polymorphism and its future potential suggests that it may be unwise to continue relying on modeling malcode as a form of intrusion detection. Our analysis (which confirms the work of other researchers) also questions the ultimate utility of string-matching signatures. Such reactive measures fundamentally surrender the initiative to the adversary and force the defender to continuously re-adjust signatures and defenses based on the attacker's innovation. The growing potential of polymorphism raises the complexity of the malcode models that the defenders have to identify and threatens to make each attack a zero-day.

A change of strategy can help intrusion detection systems recapture the initiative. We believe that, in many cases, switching to a combined-arms strategy founded on host-based randomized defense offers a way forward. In addition, we believe that randomizing the modeling of normal content or behavior can help system defenders recapture the initiative and force attackers to continuously re-adjust their methods against a moving target.

Our research also suggests that straightforward normal content modeling also requires improvment. Because network traffic may look similar enough across sites, an attacker can pre-train his attacks. If malicious code spans n-space, then an attacker has a large space to match against these normal traffic models. However, if we model selected sub-portions of the traffic, we would weaken an attacker's ability to train his attack. Instead of focusing on a malcode modeling problem of complexity $O(2^{8 \cdot n})$, we force the attacker to guess what part of n-space to aim his malcode into. As a result, the attacker has a $O(\frac{1}{2^{8 \cdot n}})$ chance of success; a move that effectively turns the vast scale of n-space to our favor, and defeats most of the previously mentioned future threats.

Many of the defensive measures we mention in Section 5 have a good deal of potential, although they are not always trivial to implement or deploy, especially for legacy systems. For example, DEP can be costly, and it is only activated for system services by default. StackGuard, WˆX, and randomized instructions cannot be easily applied to legacy systems, and emulation environments (in the case of ISR) incur a heavy performance penalty. Therefore, IDSs that act as a first line of defense will always have a valuable place. For example, anomaly detectors can rapidly drop confirmed attacks against a randomized model, and sending unverified ones (ideally a very small percentage of traffic) into a more heavily instrumented, host-based environment for testing can help reactively update the IDS models. When available, the public-facing services of the host would utilize randomized defense such as DEP, ISR, *etc*. Finally, sites can employ a collaborative security approach [27]: multiple sites with similar services can automatically share data on

the malicious content they encounter so that all participating members may have a chance to update their models before they face the same threat.

7 Conclusions

Our empirical results demonstrate the difficulty of modeling polymorphic behavior. We presented a set of analysis techniques that can help gauge the strengths of polymorphic engines, and we examined some of the state-of-the-art polymorphic engines with these techniques. We explained why signature-based modeling works in some cases, and confirm that the long-term viability of such approaches matches the intuitive belief that polymorphism will eventually defeat these methodologies. We illustrated scenarios that the polymorphic threat can explore in the future and discussed the implications for existing IDS systems.

We argue that while signature-based methods may work in the short term, empirical evidence shows that they cannot generalize enough to protect against future attacks. Therefore, we believe that the strategy of modeling malicious behavior leads to an endless game of keeping up with the attacker. To help counter this threat, we presented our recommendations for countermeasures and conclude that future intrusion detection systems must prevent attacks by modeling the class of good content or application behavior. In short, we believe whitelisting normal content or behavior is ultimately safer than blacklisting arbitrary and highly varied malicious behavior or content.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[2] Aleph0ne. Smashing the Stack for Fun and Profit. *Phrack*, 7(49-14), 2001.

[3] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium.*, August 2005.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[5] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[6] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[7] P. Biondi. Shellforge Project, 2006. http://www.secdev.org/projects/shellforge/.

[8] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[9] R. Chinchani and E. V. D. Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 284–304, September 2005.

[10] S. P. Chung and A. K. Mok. Allergy Attack Against Automatic Signature Generation. In *Proceedings of the* 9^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[11] M. Costa, J. Crowcroft, M. Castro, and A. Rowstron. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, October 2005.

[12] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium*, 1998.

[13] J. R. Crandall, Z. Su, S. F. Wu, and F. T. Chong. On Deriving Unknown Vulnerabilities from Zero-Day Polymorphic and Metamorphic Worm Exploits. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[14] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack*, 11(61-9), 2003.

[15] S. Eren. Smashing the Kernel Stack for Fun and Profit. *Phrack*, 11(60-6), 2003.

[16] J. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. In *http://www.trl.ibm.com/projects/security/ssp*, June 2000.

[17] P. Fogla and W. Lee. Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the* 13^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 59–68, 2006.

[18] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through Vulnerability-Specific Predicates. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)*, October 2005.

[19] K2. ADMmutate documentation, 2003. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

[20] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 272–280, October 2003.

[21] S. J. S. Ke Wang, Janak J. Parekh. Anagram: A Content Anomaly Detector Resistant To Mimicry Attack. In *Proceedings of the* 9^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[22] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference*, 2004.

[23] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[24] A. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. In *Proceedings of the USENIX Security Conference*, 2006.

[25] C. Krugel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 207–226, September 2005.

[26] Z. Liang and R. Sekar. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[27] M. Locasto, J. Parehk, A. Keromytis, and S. Stolfo. Towards Collaborative Security and P2P Intrusion Detection. In *IEEE Information Assurance Workshop. West Point N.Y.*, 2005.

[28] M. E. Locasto, K. Wang, A. D. Keromytis, and S. J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 82–101, September 2005.

[29] Metasploit Developement Team. Metasploit Project, 2006. http://www.metasploit.com.

[30] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[31] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[32] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12^{th} *Symposium on Network and Distributed System Security (NDSS)*, February 2005.

[33] A. Ng, M. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems*, 2001.

[34] Obscou. Building IA32 'Unicode-Proof' Shellcodes. *Phrack*, 11(61-11), 2003.

[35] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-level polymorhpic shellcode detection using emulation. In *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2006.

[36] Rix. Writing IA-32 Alphanumeric Shellcodes. *Phrack*, 11(57-15), 2001.

[37] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[38] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the* 8^{th} *Information Security Conference (ISC)*, pages 1–15, September 2005.

[39] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[40] Snort Development Team. Snort Project. http://www.snort.org/.

[41] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. In *IEEE Transactions on Information Theory*, volume 49, pages 280–284, January 2003.

[42] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 274–291, October 2002.

[43] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 227–246, September 2005.

[44] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 203–222, September 2004.

[45] X. Wang, C.-C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the* 15^{th} *USENIX Security Symposium*, pages 225–240, 2006.

[46] J. Xu, P. Ning, C. Kil, Y. Zhai, and C. Bookholt. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12^{th} *ACM Conference on Computer and Communications Security (CCS)*, November 2005.

[47] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, 2005.