On the Infeasibility of Modeling Polymorphic Shellcode[*]

Yingbo Song
Dept. of Computer Science
Columbia University
yingbo@cs.columbia.edu

Michael E. Locasto
Dept. of Computer Science
Columbia University
locasto@cs.columbia.edu

Angelos Stavrou
Dept. of Computer Science
Columbia University
angel@cs.columbia.edu

Angelos D. Keromytis
Dept. of Computer Science
Columbia University
angelos@cs.columbia.edu

Salvatore J. Stolfo
Dept. of Computer Science
Columbia University
sal@cs.columbia.edu

ABSTRACT

Polymorphic malcode remains a troubling threat. The ability for malcode to automatically transform into semantically equivalent variants frustrates attempts to rapidly construct a single, simple, easily verifiable representation. We present a *quantitative* analysis of the strengths and limitations of shellcode polymorphism and consider its impact on current intrusion detection practice.

We focus on the nature of shellcode *decoding routines*. The empirical evidence we gather helps show that modeling the *class* of self–modifying code is likely intractable by known methods, including both statistical constructs and string signatures. In addition, we develop and present measures that provide insight into the capabilities, strengths, and weaknesses of polymorphic engines. In order to explore countermeasures to future polymorphic threats, we show how to improve polymorphic techniques and create a proof-of-concept engine expressing these improvements.

Our results indicate that the class of polymorphic behavior is too greatly spread and varied to model effectively. Our analysis also supplies a novel way to understand the limitations of current signature–based techniques. We conclude that modeling normal content is ultimately a more promising defense mechanism than modeling malicious or abnormal content.

Categories and Subject Descriptors

H.1.1 [**Models and Principles**]: Systems and Information Theory—*Value of Information*

[*]This material is based on research sponsored by the Air Force Research Laboratory under agreement number FA8750-06-2-0221, Army Research Office contract number W911NF0610151, and by NSF Grant 06-27473, with additional support from Google. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

General Terms

Experimentation, Measurement, Security

Keywords

polymorphism, shellcode, signature generation, statistical models

1. INTRODUCTION

Code injection attacks have traditionally received a great deal of attention from both security researchers and the blackhat community [1, 14], and researchers have proposed a variety of defenses, from artificial diversity of the address space [5] or instruction set [20, 4] to compiler-added integrity checking of the stack [10, 15] or heap variables [34] and "safer" versions of library functions [3]. Other systems explore the use of tainted dataflow analysis to prevent the use of untrusted network or file input [9, 29] 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 [36]. Many signature generation schemes focus on relatively simple detection heuristics, such as traffic characteristics [35, 22] (*e.g.*, frequency of various packet types) or identification of the NOP sled [38], while others derive a signature from the actual exploit code [24, 43, 25] or statistical measures of packet content [41, 40, 28], including content captured by honeypots [44].

This paper presents a study of the efficacy of contemporary polymorphism techniques, as well as methods to combine and improve them. Our analysis focuses on what we consider the most constrained section of malcode, the *decoder* portion. Since this section of a malcode sample or exploit instance must contain executable code, it cannot easily be disguised (unlike most other parts of a malcode sample, except, perhaps, the higher order bits of the return address section).

We derive our motivation from the challenge of modeling this particular type of malcode data. We wondered whether, given unlimited samples of polymorphic code, it is possible to compute and store a set of signatures or a statistical model that could represent this class of code. If so, how costly would such a task be in terms of memory and processing time? In the span of the n-byte space that these samples of code populate, how much overlap is there with the class of benign network traffic? Unlike current research on polymorphic engines [17], our work focuses on the general class of code that performs decryption independent of the payload. Although other research focuses on determining if an arbitrary sequence of bytes represents executable malcode (either by employing content anomaly detection or detecting streams of executable code in network traffic), our objective is quite different: we aim to determine if malcode itself has any distinguishing features that might support the construction and use of exploit signatures or statistical models.

1.1 Shellcode Background

Aleph0ne first illustrated the basics of smashing the stack [1]. The virus writer Dark Avenger's Mutation Engine influenced the shellcoder K2 to develop shellcode polymorphism [19]. rix then proceeds to show how to perform alphanumeric encoding [32], Obscue described how to encode shellcode to make it survive ASCII to unicode transformations [30], the CLET team developed the technique of spectrum spoofing and implemented a recursive NOP sled [13] and most recently the Metasploit project combined vulnerability probing, code injection, and polymorphism (among other features) into one complete system [26].

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 executable payload of the exploit *i.e., shellcode*. Recently, polymorphism has been successfully employed in disguising shellcode. One 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][ENCPAYLOAD][RETADDR]
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, 13, 26, 6]. Countermeasures to polymorphism range from emulation methods [31] to graph–theoretic paradigms aimed at detecting the underlying vulnerability [7] or signatures based on higher order information such as the control-flow graph of the exploit [23, 8] or correlating protocol format information with memory corruption vulnerabilities [12]. We elaborate on defense techniques in Section 5.

There are two main ways to disguise shellcode. The first rewrites the code each time so that it differs syntactically but retains the same operational semantics. This process, akin to metamorphism, is decomposable to graph isomorphism [37], and unlike virus metamorphism (see Zmist [16]), it is, in general, a non–trivial solution to implement. The other more common approach is self-ciphering: the exploit is wrapped as payload within a larger malcode component and is disguised using a reversible cipher (usually a sequence of operational loops *e.g.,* xor, add, subtract, ror, rol, *etc.*, although a looping construct is not always required). An attacker typically uses several rounds of ciphering. Polymorphism is obtained by randomizing the order of these ciphers and using different keys. In order to reverse the cipher, a clear–text program must exist immediately before (in terms of execution flow) the payload. This program decodes the exploit payload at runtime. Such "decoders" typically have a length of 30 to 50 bytes and can decode arbitrary payloads. Decoders provide an effective technique for rapid and simple dissemination of malcode variants. Attackers reuse exploits in arbitrarily different forms. In fact, many polymorphic engine in the wild carry a copy of the shellcode listed in Aleph0ne's seminal paper [1].

```
address   byte values     x86 code
--------  ------------    ------------------
00000000  EB2D            jmp short 0x2f
00000002  59              pop ecx
00000003  31D2            xor edx,edx
00000005  B220            mov dl,0x20
00000007  8B01            mov eax,[ecx]
00000009  C1C017          rol eax,0x17
0000000C  35892FC9D1      xor eax,0xd1c92f89
00000011  C1C81F          ror eax,0x1f
00000014  2D9F253D76      sub eax,0x763d259f
00000019  0543354F48      add eax,0x484f3543
0000001E  8901            mov [ecx],eax
00000020  81E9FDFFFFFF    sub ecx,0xfffffffd
00000026  41              inc ecx
00000027  80EA03          sub dl,0x3
0000002A  4A              dec edx
0000002B  7407            jz 0x34
0000002D  EBD8            jmp short 0x7
0000002F  E8CEFFFFFF      call 0x2
00000034  FE              db 0xFE
...
payload follows
```

Figure 1: A 35 byte polymorphic decryption loop. From left to right, the columns contain the address or offset of the instruction, the byte value of the instruction, and an x86 assembly representation. Note the five cipher operations, `ror xor ror sub add`, that begin at `0x0C`. The working register for the cipher is %eax. Note the stop condition at `0x2B`.

1.2 Shellcode Polymorphism

Polymorphic techniques no longer consist of simply disguising the payload; attackers frequently conceal other sections of malcode.

• [NOP]: The most basic design of a nop-sled is a buffer of NOP instructions {x90,x90,...,x90} inserted ahead of the decoder to safely capture a future change in the value of the instruction pointer. Many signature–based systems rely on this artifact for detection. Attackers, however, have introduced various innovations to make the NOP sled polymorphic. The sled need not consist of actual NOP instructions — it only has to pass the flow of execution safely into the decoder without causing instability. K2 described at least 55 different ways to write such single byte benign instructions [19] and implemented this method in the ADMmutate engine. This technique provides the potential for 55^n unique NOP sleds (where n is the sled length).

The CLET polymorphic engine [13] employs a more advanced NOP sled design. This method discovers 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 enters the 2-byte instruction or if it lands one byte to the right since that position will hold another equally benign instruction. Recursive use of this method to additional depths finds longer benign instruction sequences for a NOP sled. To the best of our knowledge, no analysis of the potential of this method exists, but it serves as a useful polymorphic technique because modeling this type of sled may amount to modeling random instructions.

• [RETADDR]: Without address space randomization, the location of the stack and stack variables on most architectures remains consistent across program executions. Thus, the attacker has a basis for guessing the appropriate value for an injected return address to redirect the instruction pointer into the malcode. Generating signatures that use these specific address values to filter out malcode seems possible for certain types of code injection attacks. An attacker can, however, achieve return address polymorphism by modifying the lower order bits [19]. This method causes control flow to jump into different positions in the stack. As long as it lands somewhere in the 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 v^m possible variations.

• `Spectrum shaping & byte padding`: In polymorphic blending attacks [17], exploits attempt to appear similar to benign traffic in terms of the n-gram content distribution. The CLET team's polymorphic engine [13] is an example of such a technique. Their engine alters the shellcode to take on the form:

[NOP][DECODER][ENC PAYLOAD][PADDING][RETADDR]

The engine adds junk bytes in the new padding area to ensure the 1-gram distribution of the shellcode appears different. In addition, the shellcode itself is ciphered with different length keys. These keys exhibit a variety of byte distributions that reshape 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 content anomaly detectors [40].

Perhaps the most worrisome threat is that these individual techniques are interchangeable and can be combined into a single polymorphic engine. Section 3 shows that this engine is simple to implement. Furthermore, the structure used by modern shellcode (*i.e.,* [NOP][DECODER][ENC PAYLOAD][RETADDR]) is really just a conventional design that happens to work. Nothing prevents the attacker from modifying the sections between the sled and the return address. With some additional `jmp` instructions, it is not hard to imagine seeing future shellcode of the forms:

[NOP][ENCRYPTED PAYLOAD][DECODER][RETADDR]
[NOP][DECODER 1][ENC. PAYLOAD][DECODER 2][RETADDR]
[NOP][PADDING][ENC. PAYLOAD][PADDING][DECODER][RETADDR]

and so on. These types of encoding will present difficulty for both static and dynamic code analysis.

1.3 Contributions

Conventional wisdom has held that attackers retain a significant advantage by using polymorphic tactics to disguise their shellcode. To the best of our knowledge, however, there exists no quantitative analysis of this advantage. Our work provides empirical evidence to support this folk wisdom and helps improve understanding of the polymorphic shellcode problem in the following ways:

- We illustrate the ultimate futility of 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 there is a strong case for investigating other protection paradigms.

- As a corollary, we show that given any normal statistical model, there is a significant probability that an attacker can craft successful targeted attacks against it.

- We propose metrics to gauge the relative strengths of polymorphic engines, and we use these to examine some of the current state-of-the-art engines. We believe our methodology is novel and helps provide some insight in a space that has generally been lacking in quantitative analysis.

- We show how to augment existing polymorphic engines and demonstrate this process by presenting our implementation of a proof-of-concept engine.

The problem we address can be stated as follows:

PROBLEM DEFINITION *Given n bytes, there can be a set of 256^n possible strings. The specific class of x86 code of length n that corresponds to decoders is a subset of this superset, i.e., spanning a subspace within this larger space. How difficult is it to model this subspace — in other words, what is the magnitude of this span? What are polymorphic threats we can expect to see in the immediate future? Finally, what are the theoretical limits?*

We combine a number of methods to answer this question. First, we introduce a set of measures to assess the randomness of a population of samples. We employ these measures to analyze a pool of decoders generated by existing polymorphic engines. Next, we demonstrate our improvements to existing polymorphic techniques. Finally, we analyze the theoretical limits of polymorphism by examining a fixed size space of n-bytes. We explore this space using efficient genetic algorithms to characterize the span of all x86 code that exhibits polymorphic behavior. Along the way, we 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 not only present a challenge for signature systems, but also presents a significant challenge for statistical approaches to model malcode.

2. POLYMORPHIC ENGINE ANALYSIS

This section explains the details of our approach for analyzing the range of decoders generated by any given engine. We apply our methods to analyzing six of the state-of-the-art polymorphic engines used in the wild: ADMmutate, CLET, and four engines from Metasploit: Shikata Gai Nai[1], Jumpcall additive, Call4dword and fnstenv mov. Previous research on automatic generation of exploit signatures from polymorphic code [22, 28] reports successful detection of exploits from many existing engines, some of which are from Metasploit. Our work makes it easy to visually observe the artifacts that some of these engines leave in each shellcode instance that they generate: artifacts which can be taken advantage of for detection. We show, however, that 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. We designed our measures[2] around the following parameters:

VARIATION STRENGTH: Given sequences of length n, the variation strength of an engine measures that engine's ability to generate sequences of length n that span a sufficiently large portion of n-space. This metric is meant to offer some insight into the magnitude of the set of signatures that may be needed to accurately encapsulate all decoders generated by a particular engine.

PROPAGATION STRENGTH: For the sequence of decoders, $\mathbf{x}_1,\ldots,\mathbf{x}_N$ that an engine can generate, the propagation strength of the engine characterizes the efficacy of the engine in making any two samples $\mathbf{x}_i$, $\mathbf{x}_j$ for all $i, j = 1 \ldots N$, look different from one another. The purpose of this metric is to quantify the amount of information gain obtained by isolating a few samples from a particular engine.

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

[2]Notations used in this paper: all variables in bold text font such as $\mathbf{x}$ and $\mathbf{y}$ denote column vectors. We use $\mathbf{x}_i$ to denote the i^{th} vector of a set of vectors and we use $\mathbf{x}(i)$ to denote the i^{th} component of the vector $\mathbf{x}$.

Using these metrics, we analyze six current polymorphic engines and provide some measurements for their relative polymorphic strengths, yielding a scaled score for each engine which we call the "relative polymorphism strength score" or *p-score*. We use this score to compare the samples generated by the polymorphic engines to sequences that we generate at random. In addition, we leverage the concept of a spectral image, which allows easy visualization of the amount of distortion in a sample pool. We combine this technique with the above metrics to derive our results and confirm the folk wisdom that the class of x86 polymorphic shellcode is too random to model.

Spectral Image

Given any polymorphic engine, we can use it to generate a set of D decoders, each of length N. For non-fixed length decoders, we can add padding at the end to make the lengths equal so that they can be displayed. We next sort these decoders and stack them together row-wise into a $D \times N$ matrix, then display this matrix as an image, considering the i^{th} byte of decoder j as the intensity value for the $(i, j)^{th}$ pixel of the image. A byte value of 0x00 produces a black pixel; 0xFF produces a white pixel. Values within this range exhibit a shade of gray. This representation helps us visualize the randomness of a set of generated decoders. Salient bytes – bytes that exist in the same places within all generated decoders – are easily identifiable artifacts since they show up as visible columns within the spectral image. Figure 2 shows the spectral images of the six engines we examined. They were generated by taking a single shellcode sample and encrypting it with each engine 10,000 times to generate 10,000 unique shellcode sequences for each engine. We extracted the decoder portions from these sequences, sorted them, down–sampled (so that the number of samples used is on the order of the dimensions of the samples), then generated the images.

Notice how Shikata Ga Nai generates roughly three subclasses of decoders. The same blocks of code exist in the engine but not always at the same place. The weaknesses of the `c4d` and `fnstenv mov` engines are apparent as the vertical columns show that these engines always embed large artifacts in every decoder. These artifacts can be used as signatures and are easily recovered using current techniques [22, 28]. As the images show, even though these engines perform the same basic actions to decode a string within a small distance of itself in memory, these invariants do not hold across different engines. For example, the vertical band for CLET represents clearing of registers (we confirmed this by reading their documentation).

Minimum Euclidean distance

Any string $\mathbf{x}$ of fixed length n can be considered as a single point embedded in n-space, *i.e.,* $\mathbf{x} \in \mathbb{R}^n$. For $n = 2$, we can imagine a 2-D plane – the string "ab", where the ASCII character "a" is 97 and "b" is 98, can be considered as a single point in this 2-D plane, embedded at (97,98). The string "yz" would likewise be embedded at (121,122). The Euclidean "distance" between these two strings quantifies the length of the line drawn from (97,98) to (121,122) and is calculated using the Euclidean norm, denoted $||\cdot||$ and defined as: $||\mathbf{x}|| = \sqrt{\sum_{i=1}^{n}(\mathbf{x}(i))^2}$. Without loss of generality, we can see that this extends for strings up to higher order n-space for any arbitrary n. We can therefore consider each decoder string a single point within this n-space of all strings of length n.

The minimum Euclidean distance between two strings is defined as the normalized Euclidean distance between the strings under arbitrary byte–level rotation. We find this definition useful because we expect decoders to employ forms of polymorphism that retain the same ciphering methods but shift the order of operations.

$$\delta(\mathbf{x}, \mathbf{y}) = \min_{r=1\ldots n}\left[\frac{||\mathbf{x} - rot(\mathbf{y}, r)||}{||\mathbf{x}|| + ||\mathbf{y}||}\right] \qquad (1)$$

$rot(\mathbf{y}, r)$ means rotate the string $\mathbf{y}$ to the left by r-bytes, with wraparound. We divide by $||\mathbf{x}|| + ||\mathbf{y}||$ to transform the metric into a ratio of the distance between two vectors with respect to the sum of their individual lengths. This normalizes the metric and removes the number of dimensions (length) of a string as a factor in the distance. This distance measure plays an important role in our metrics, described more fully in the following sections.

Variation Strength

Following our previously described notion of viewing decoders as embedded points in n-dimensional space, we can conceptually visualize a set of decoders generated by a particular engine as a cloud of points in this n-space. The magnitude of the space covered by the span of these points is what we refer to as the *variation strength* of the engine. The magnitude and complexity of the span is directly proportional to the difficulty of modeling the engine, *i.e.* the number of signatures in the case of signature based methods or model complexity in the case of statistical models. We present a method to bound this magnitude, making use of the covariance matrix, which is defined as the following:

$$\Sigma = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T \qquad (2)$$

This gives us a symmetric matrix with dimensions $n \times n$ for decoder sequences of dimensionality n. Here, $\mathbf{x}_i$ is a decoder sample and $\mu = \frac{1}{N}\sum_i^N \mathbf{x}_i$ is the sample mean of the set of decoders. $\mathbf{x}$ and μ are column vectors and T denotes the vector transpose operator.

The covariance matrix describes the shape of an n-dimensional ellipsoid in n-space. Therefore, recovering the covariance matrix for a set of decoders recovers hyper-ellipsoidal Θ bound on the data set. Calculating the span of the set is a problem of measuring the radii of the principle axes of the ellipsoid, which is an eigenvector decomposition problem. Recall that eigenvector decomposition finds a new set of basis vectors that spans a space defined by any given symmetric matrix. The new basis vectors are called eigenvectors, and their corresponding eigenvalues reveal the scale of these vectors. Thus, we recover $\mathbf{v}$ and λ such that $\Sigma\mathbf{v} = \mathbf{v}\lambda$, where $\mathbf{v}$ is the set of n eigenvectors and λ is the set of n eigenvalues. We now define the variation strength of a polymorphic engine as:

$$\Psi(engine) = \frac{1}{n}\sum_{i=1}^{n}\sqrt{\lambda_i} \qquad (3)$$

The square roots of the eigenvalues are taken to *whiten* the distribution, and we take the average of the eigenvalues since we are interested in the relative scatter of the decoders in n-space; higher dimensions should not increase the score. The utility of the normalization procedures is shown in Table 1, where the distribution spanning [0..128] is shown to exhibit half of the "randomness" of one that spans [0..256].

To analyze the variation strength of an engine, we encoded a shellcode sample 10,000 times and extracted the corresponding decoder sequences. After generating the covariance matrix according to Equation 2, we recovered the eigenvalues and obtained the score using Equation 3. The larger the number of samples used to generate the covariance matrix, the more accurate the estimate will be. In practice, around a few hundred samples is usually enough[3].

[3]Defining a full ranked covariance matrix requires more samples

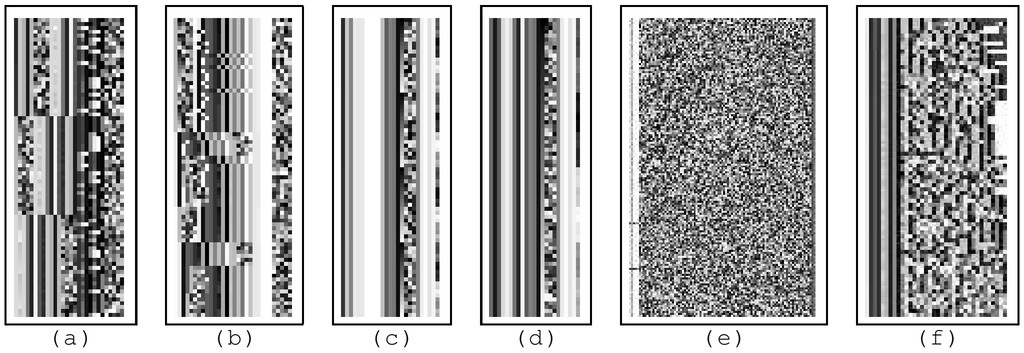


Figure 2: Spectral images to show variation strength (a) Shikata Na Gai (b) jcadd (c) call4dword (d) fnstenv mov (e) ADMmutate (f) CLET. Each pixel row represents a decoder from that engine and each individual pixel value represents the corresponding byte from that decoder. A column of identical intensities indicates an identifiable artifact left by the engine.

PROPAGATION STRENGTH

If the true decoder distribution happens to exhibit a large span but lies on a lower-dimensional "manifold" in n-space where the significant dimensions of the manifold is much less than n (in the worst case, imagine a hollow n-dimensional sphere with large radii) then the variation strength might overestimate the bound on the n-space scatter since the decoders do not exist in the space between the sphere and the origin. This is why we introduce a second component to the engine strength metric based on the expected distances between decoders. To visualize this metric, imagine a fully connected graph where each node is a decoder sample and the edge weight is the distance between any two nodes. The average of the edge weight is then proportional to the scale of the graph. We call this metric the *propagation strength* because of its close relationship to the problem of connecting any two decoder samples together.

$$\Phi(engine) = (1 - \frac{\eta}{n}) \int \int p(\delta(\mathbf{x}, \mathbf{y}))\delta(\mathbf{x}, \mathbf{y}) \, d\mathbf{x} \, d\mathbf{y} \quad (4)$$

$\delta(\mathbf{x}, \mathbf{y})$ is a function that returns the distance between any two decoder sequences. Flexibility in choosing the δ function allows us to fine tune this metric. For our experiments we set delta as Equation 1, which is rotation invariant. If the engine performs a simple shift in the different layers of cipher operations, then the bytes are decoupled from one another, and the variance in the samples would be great. The propagation strength, however, would be very low since δ is shift invariant, thus lowering the overall score. In addition, we introduce the η variable which is defined as the number of salient bytes within all of the decoder samples generated by an engine. This parameter is used as a scaling factor to decrease the strength of engines that leave consistent artifacts in their decoders which signature–based IDS implementation can lock on to. If prior information is available in the form of probability density function (pdf) for $p(\delta(\mathbf{x}, \mathbf{y}))$ such as a Gaussian then the above equation is solvable in closed form and can act as a regularizer for this metric. If not we can use a uniform prior *i.e.* $p(\delta(\cdot)) = 1$ and the result can be approximated by generating the matrix D such that $D_{i,j} = \delta(\mathbf{x}_i, \mathbf{x}_j)$ and taking the average of this matrix. Since $\delta(\cdot)$ is symmetric ($D_{i,j} = D_{j,i}$), we only find the average of the upper diagonal of the matrix. We use this simpler estimation procedure to derive the results presented in this paper.

A polymorphic engine might have a restricted span, but if the sequences that it generates 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. Our propagation metric is proportional to how long an engine's generated shellcode can propagate before a detector can be properly trained.

than the dimensionality of the data sample.

OVERALL STRENGTH

We define the overall strength of a polymorphic engine $\Pi(\cdot)$ to be the product of the variation strength and propagation strength since they are positively correlated.

$$\Pi(engine) = \Psi(engine) \cdot \Phi(engine) \quad (5)$$

To normalize the metric, we find the "strengths" of completely random distributions of data we generated, then divide the strengths of each engine by these strengths to generate a scaled score, which we call the "relative polymorphism score", or (*p-score*) for short. Non–linear combinations (such as adding exponents to weight the two strengths differently) of the $\Phi(\cdot)$ and $\Psi(\cdot)$ metrics are possible. Althougth we could attempt to find tighter bounds, our main goal was to introduce this particular dual approach to quantifying the capacity of polymorphic engines, allowing them to be ranked relative to one another. We therefore keep the metric in its basic setting and leave open problems such as what is the most appropriate δ function.

Engine	Prop. St.	Var. St.	Overall St.	p-score
Shikata	0.14	53.24	7.24	0.62
Jcadd	0.11	44.62	4.87	0.42
C4d	0.06	14.62	0.83	0.07
Fnstenv	0.07	15.70	1.05	0.09
Clet	0.14	53.00	7.37	0.63
Admmutate	0.15	68.76	10.59	0.91
$rand_{128}$	0.16	36.90	5.83	0.50
$rand_{256}$	0.16	73.74	11.61	1.00

Table 1: Decoder polymorphism strengths of various engines under our metric (the first four engines are from Metasploit). Also shown are the scores for random distributions of strings within range 128 and range 256. Compare with Figure 2.

Table 1 shows the strengths of these engines based on our metrics. The latter two rows in the table, $rand_{128}$ and $rand_{256}$, refer to a set of randomly generated strings with each byte values between [0..128] and [0..256], respectively. This overall *p-score*, used in conjunction with the spectral images, can be used to gauge the effectiveness of polymorphic engines relative to each other as well as to noise. This comparison provides some utility for predicting detection success rates of various IDS systems for newly released engines. For example, IDS solutions that cannot detect CLET samples have no hope against ADMmutate. The *p-score* is also useful in determining identifiability. Engines with scores higher than a certain threshold would generate decoders which cannot be traced back to the same engine, as we can see from the spectral image for ADMmutate. The value of this threshold is the subject of our ongoing work.

Some of the engines we examined can be adjusted to obtain better scores. For example, CLET allows the user to specify an arbitrary number of decoding operations *i.e., xor* then *sub* then *add*, and so on. Our experiments used the default setting of five instruction operations. CLET's main weakness derives from the fixed way in which it clears registers before decoding. Therefore, testing different levels of ciphering would not yield significantly different results. Note that three of the engines from Metasploit are not entirely polymorphic (according to the Metasploit documentation) and it is easy to see which ones these are.

While CLET's cleverness and efficiency was on par with that of ADMmutate in terms of disguising its payload, we found that all decoders generated by CLET contained a unique 9-byte signature string that 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 CLET is one of the more creative engines that we have seen, this particular feature makes the decoders easier to detect and identify than the other engines, thus explaining the lowered score. The CLET team acknowledged as one of their weaknesses this static structural layout [13]. This weakness is not difficult to address, and we expect future versions of CLET will eliminate these artifacts.

3. A HYBRID ENGINE: FULL SPECTRUM POLYMORPHISM AND BLENDING

Previous sections illustrated how polymorphism works in various engines and how efficient certain engines are at hiding their payloads. In this section, we show how one can extend existing polymorphic methods by combining two powerful engines: CLET and ADMmutate. While CLET's decoder leaves some noticeable artifacts, it has very useful spectral ciphering techniques that allow the shellcode to blend to a target byte distribution. ADMmutate cannot perform blending attacks, but it generates very random looking decoders as well as a recursive NOP sled. We simply use CLET to cipher the shellcode, then hide CLET's decoder with ADMmutate. We also take advantage of ADMmutate's advanced NOP sled generator. Section 1.2 outlines some of the techniques used to make the other sections polymorphic, and we employ these tactics in our engine design.

The combination of these engines makes the shellcode not only impossible to model but also allows the exploit instance to blend in with normal network traffic. Every section of the shellcode can be made polymorphic, leaving only the blending section exposed, as demonstrated in Figure 3. Here, we have added bytes into each of padding sections of the shellcode samples, so that when stacked together, the shellcode shows the ACM SIGSAC logo. Each row of the three spectral images shown in Figure 3 represents a 512-byte **fully working** shellcode sample that was tested and confirmed to execute successfully.

The polymorphic capabilities employed by ADMmutate, which uses two layers of ciphering on the payload using 16-bit random keys, allows the payload to be scattered across n-space and thus avoids being detected by signature detectors. The padding section

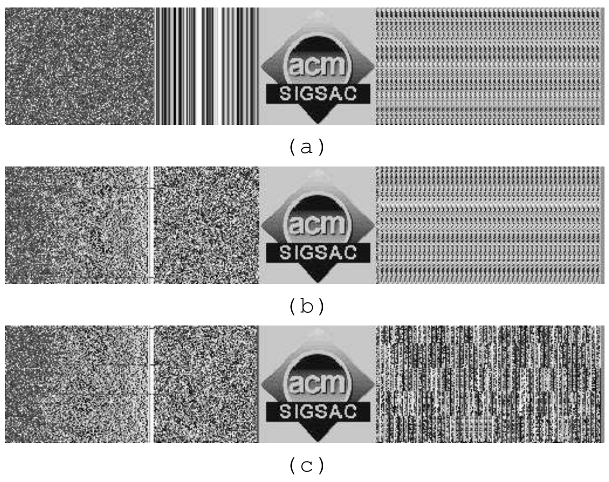


Figure 3: Spectral images. (a) A single CLET mutated exploit is stacked row-wise 100 times (note the vertical bands). Next to it, CLET's polymorphic blending ability leaves a padding area open for arbitrary filler bytes which are never reached in the execution. We fill it with the ACM logo. (b) CLET's decoder and exploit is hidden by ADMMutate, leaving only the blending bytes exposed. The repeating columns represent the [RETADDR] section, which is shown morphed in (c) using the random offset method.

can carry an arbitrary byte combination since the exploit exists to the left of the section and triggers immediately before the execution flow ever passes into the padding section. "Normal-looking" n-grams, placed within this padding section, can thus allow the shellcode to blend into normal traffic — slipping by statistical IDS methods as well. It is also non–trivial to model the blending bytes section; one simply takes as input a distribution model and for every byte feature i, multiply it with the length of the section to find how many of these bytes to use. The section is then filled up with accordingly with the appropriate number of bytes for each `0x00...0xFF` value, then this section is *randomly permuted* (for example, rearranging the order of the bytes). This alteration prevents the derivation of signatures. Statistically speaking, the section has not changed since all of the bytes are still present in their corresponding frequencies. We implemented this technique in our engine and report results later in the section.

For this particular demonstration, we have chosen a padding section of size 100 bytes, out of a total shellcode size of 512 bytes. Of course, this section and the entire shellcode sample can be enlarged. The only change that needs to be made is to increment the values in the [RETADDR] section to "aim a little higher" into memory to compensate for the larger shellcode.

The [RETADDR] section is the series of repeated columns seen to the right of the padding section in Figure 3(a) and (b) — notice the periodicity. As mentioned above, the [RETADDR] is not easily modeled. This portion normally has variable length, is mutable, and is both platform and vulnerability dependent. This mutability feature is demonstrated in Figure 3(c) where we have mutated the [RETADDR] section by aiming the instruction pointer at the center of the NOP sled and adding a random byte offset of approximately 50 bytes in each of the repeated return addresses. This gives us about 100^m possible unique sequences for the [RETADDR] section, where m is the number of times the target address is repeated. The base of the exponent (100) can be larger or smaller, depending on

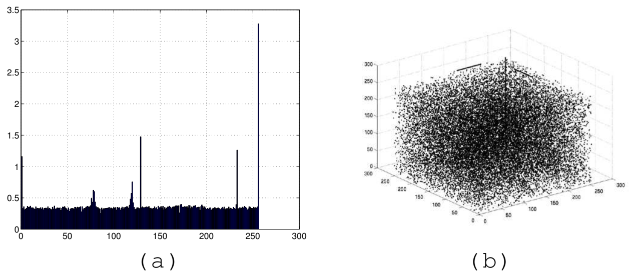


Figure 4: adm+clet engine. (a) 1-gram distribution (b) 3-gram scatter. Comparing this to Figure (7), we can see that this engine is equally difficult to model.

how large the NOP sled section is. The only real weakness we see from ADMMutate is a white column, representing a 4-byte salient artifact generated by the engine, which is too small to use as a signature or statistical model.

In terms of statistical features, Figure 4(a) shows the 1-gram distribution of the ADMmutate decoder section of the above engine, which was calculated by finding the average 1-byte histogram of these decoders, then normalizing it by dividing by the variance along each dimension.

Dividing by the variance normalizes the values so that we obtain their discriminative scores. For example, if a feature is consistently present, it has low variance. Therefore, dividing by its variance will increase the prominence of that feature. Conversely, if a feature exhibits very high variance, then its reliability in statistical modeling is correspondingly low. From Figure 4(a), we see that there is little to no signal from the 1-byte distribution. Figure 4(b) shows the 3-gram scatter of these 100 decoders, showing us the range of 3-grams present. As we can see, for 3-grams, it is a full spectrum spread. If 3-space is saturated, then so is 2-space since it is a subspace within 3-space. Having the decoder bytes spread across n-

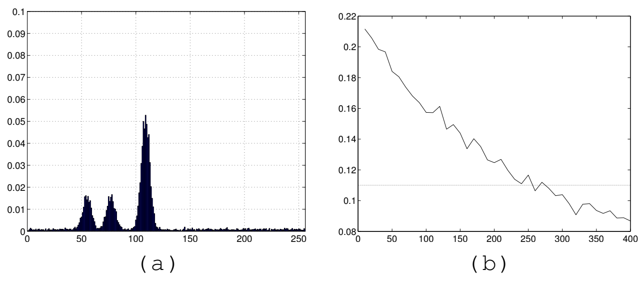


Figure 5: Our combined adm+clet engine executing a blending attack. Image (a) shows target distribution, while image (b) shows distance to target given padding section size.

space means we have some freedom to perform blending attacks, since it means the engine can generate decoders which do not appear "too binary." We implemented a blending attack function into our adm+clet engine.

Figure 5 displays a simulation of a blending attack using our engine. We first artificially generate a target distribution, shown in Figure 5(a). We created a distribution from a mixture of three Gaussians with centroids at **55** (ASCII character "7"), **77** (ASCII character "M") and **109** (ASCII character "m") in order to simulate the network traffic distribution of a server hosting many clear text transfers. We also added some binary noise to account for binary transfers (*e.g.,* images and video). We set the variance of each of the three Gaussians to 15. Next, we implemented the Mahalanobis distance classifier:

$$Mahdist(\mathbf{x}, \mu|\Sigma) = (\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu) \tag{6}$$

Here, μ is the 1-byte target distribution, and we set $\Sigma = 0.1\mathcal{I}$ where $\mathcal{I}$ is the identity matrix. Statistical IDS systems such as PayL [41] employ the Mahalanobis distance classifier. We chose our estimates in the same manner.

Figure 5(b) shows the engine's blending attack converging on the target distribution. The Y-axis shows the Mahalanobis distance as a function of the size of the blending section. For each size, we generate a new malcode sample with a blend section of that size. Next, we fill the section with bytes generated using the method we outlined at the start of this section, then calculate the 1-byte distribution of the shellcode with these new bytes in place and find the Mahalanobis distance to the target distribution using Equation 6.

We see that a padding section of around 200 bytes is needed to blend an executable shellcode sample generated from our engine into the given target distribution, under our chosen threshold value. In our example, we have given our engine the correct target distribution. In practice, the target distribution will have to be estimated in some way. We included this demo only to demonstrate the potential of combining many different attack vectors into the same engine. Our blending attack is not meant to be the most advanced; we refer the reader to related work on this topic [17].

We could also make use of *binary to text encryption* to transform the decoder portion of the shellcode into a string that consists of only printable characters by using techniques such as the ones provided in ShellForge engine [6]. This technique could improve the blending strength of the engine with relatively little implementation effort. In addition, there are other techniques that will allow the shellcode survive sanitization functions such as `to_upper()` and `to_lower()`.

4. EXPLORING N-SPACE

We have so far focused our attention primarily on understanding and extending existing engines and techniques. This section investigates the extent to which polymorphic code exists in n-space. More specifically, given n-space, we have 2^{8n} possible strings. We want to explore the entirety of this space and find all of the sequences that "behave" like polymorphic code. Since completing this search is intractable for large n (*e.g.,* $n > 4$), we restrict our attention to byte strings of length 10 in order to make our search feasible. From the structure of the decoder, we know that it must contain two components: (1) a modification operation (*e.g.,* `add, sub, xor,` *etc.*), and (2) some form of a loop component *e.g.,* `jmpz`, that sweeps the cipher across the payload. Figure 1 shows that real full decoders are longer and more complex than these simple requirements. For example, they contain maintenance operations such as clearing registers, multiple cipher operations, and some exotic code to calculate the location of the executable. We believe, however, that our restrictions retain the most critical operations for examining decoding behavior.

Our restricted, 10-byte examination reduces the search space to 2^{80} strings. This problem remains intractable if we plan to explore the space one unique string at a time. Starting at {`0x00 0x00...0x00`}, testing to see if it exhibits polymorphic behavior, proceeding to {`0x00 0x00...0x01`}, testing that string, and so on until we reach {`0xFF 0xFF...0xFF`} represents a significant dedication of time and resources with little reason to suggest that such a complete procedure conveys substantially more meaningful data than a more intelligent and judiciously directed search. Instead, we make use of genetic algorithms [33] to perform the search in a directed manner by choosing to explore areas were existing polymorphic code resides as a form of local search. To satisfy these requirements, we define a function that accepts a string as input and determines whether that string represents x86 code that exhibits polymorphic behavior.

4.1 Decoder Detector

We designed our "decoder detector" and implemented it as a process monitoring tool within the Valgrind emulation environment. Valgrind's [27] binary supervision enables us to add instrumentation to a process without modifying its source code or altering the semantics of the process's operations. Most importantly, Valgrind provides support for examining memory accesses, thus allowing us to track what parts of memory a process touches during execution. Our tool detects "self-modifying code," which we define as code that modifies bytes within a small distance of itself. We restrict our attention to instruction sequences that modify code within two hundred bytes of itself in either direction in memory — that is, we sandwich the code within two NOP sleds of two hundred bytes each. The GA-search framework compiles and executes a buffer overflow exploit in the emulation environment and checks for any polymorphic behavior.

The following polymorphic behaviors are of interest: we define *self–write* as writing to a memory location within two hundred bytes of the executing instruction. We define *self–modify* as reading from a memory location within two hundred bytes and then, within the next four instructions, performing a write to the same location, simulating the behavior of in-place modification operations effected via instructions such as `xor, add, sub`. Of course, some polymorphic techniques may not replace code in–place, but any such examples further saturate n-space.

4.2 Genetic Algorithms

Genetic algorithms is a classic optimization technique from AI and have proved most useful in problems with a large search space domain and where closed formed solutions are not available or directly optimizeable. Instead, various solutions are represented in coded string form and evaluated. A function is used 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*. For a more detailed discussion, we refer the reader to [33].

The fitness function used for our GA search framework is the decoder detector described above. We score 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`-style decoder behavior. 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 relax our GA optimization constraint since we do not need to find the "best" decoder. Instead, we have a low limit for polymorphic behavior and will admit any string that passes that threshold into the population. 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 very few self-modifications; we wanted to capture strings that exhibited a significant amount of polymorphic behavior (*i.e.,* it encapsulated some form of a loop construct)[4]. We stored all unique strings that met the polymorphic criteria in what we term the *candidate decoder pool*. We observed that the average fitness value reached into the hundreds after a few hundred epochs.

Genetic algorithms perform intelligent searching by restricting their attention to searching the space surrounding existing samples. Therefore, this form of local search needs quality starting positions to achieve reasonable results. We seeded our search engine with two decoder strings extracted from *ShellForge* [6] and roughly 45,000 strings from Metasploit [26] in order to obtain a good distribution of starting positions. We implemented a standard GA-search framework using some common evolution strategies, listed here:

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. It is used mainly to introduce some entropy into the population and is not useful by itself since the likelihood of finding executable x86 code with self modification and an inner loop at random is low.

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. An alternative search procedure would parameterize the actual `x86` instruction set into a genetic algorithm search package and dynamically write decoders. This is the subject of our ongoing work. This technique bears a strong similarity to work done by Markatos *et al.* [31]. Whereas they implemented their tool as a detector, dynamically filtering network content through the detector to search for the presence of decryption engines, we use our decoder detector in an offline manner where we generate the strings ourselves in order to precompute a set of byte strings that perform self–modification.

4.3 GA-Search results

This evaluation aims to assess the hypothesis that the class of self-modifying code spans n-space where n is the length of the decoder sequence. Our GA-search framework found roughly *two million* unique sequences after several weeks of searching and shows no signs of diminishing returns. The results that we derive show that the class of n-byte self-modifying code not only spans n-space but saturates it as well.

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

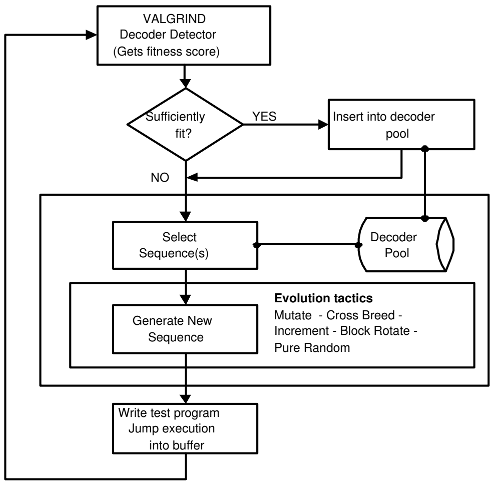
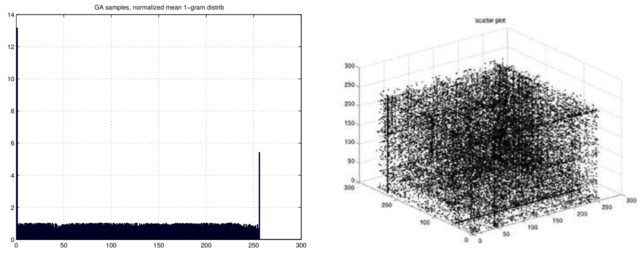


Figure 6: *Decoder search engine flow chart.* We construct our library of decoders using a feedback loop that creates candidate decoders, confirms that they exhibit sufficient decoding behavior, and generates more samples from them.

First let us look at the (rounded) mean and variances of the generated sample pool of 10-byte sequences, shown in decimal for each reading:

Mean: {90,66,145,153,139,127,123,138,134,126}
Standard deviation: {72,71,86,78,80,84,86,82,75,76}

The mean exists near the center of n-space (in this case, n-space is a vector of 10 entries each of value 128). The high variance along each dimension shows that the samples are widely scattered. Statistical IDS detectors typically operate under the assumption



(a) (b)

Figure 7: Results. (a) 1-gram distribution - note the uniform byte distribution. (b) 3-gram scatter plot - each dot represents a 3-gram, note the 3-space saturation.

that the class of malcode being modeled exhibit a certain n-gram distribution. This "byte-spectrum" can be modeled and used to design a classifier to separate malcode from normal traffic ($n = 1$ in the case of PayL [40] and $n = 3, 4, 5, 6, 7$ in the case of Anagram [21]).We examined our generated samples to see if such a signal existed. For each sequence in our sample pool, we compute the 1-byte distribution, then find the average for all sequences, normalized by dividing by the variances along each dimension, as we did in Section 3. Figure 7(a) shows the average 1-bytes distribution. We can see that the sample pool contains no distinguishable distribution but rather is closer to white noise (with the exception of the {x00} and {xFF} values, which are likely to be padding artifacts). For 3-space, Figure 7(b) shows the 3-gram scatter plot of all 3-grams extracted from all the candidates in the pool. This plot shows that, for 3-grams, the space is well saturated. Since it is a subspace of 3-space, 2-space also saturated. The p-score of these samples was close to 1.00. This result can be expected as "polymorphic code" is less constrained than the full decoders we have worked with in the previous sections. Nevertheless, our results show that there is a significant degree of variance in x86 code that performs operations that we can associate with self–decryption routines.

4.4 Results Discussion

Our results show that the span of polymorphic code likely reaches across n-space. The challenge of signature–based detection is to model a space on the order of $O(2^{8 \cdot n})$ signatures to catch potential attacks hidden by polymorphism. To cover thirty-byte decoders requires $O(2^{240})$ potential signatures, for comparison there exist an estimated 2^{80} atoms in the universe. We would much sooner run out of atoms than attackers run out of decoders. Current signature schemes work only because of advances in rapid isolation and generation of signatures. This strategy may work for the short term; however, our work indicates that defenders cannot capture the initiative from the attacker under this reactive defense strategy. Somewhat troubling is the additional implication that regardless of what a normal model of traffic for a particular site may be, there exists a certain probability that a range of decoders would fall within the span of that normal model because sequences which exhibit polymorphic behavior span most of n-space.

5. RELATED WORK

Countering attacks and malcode is a hard problem. Spinellis showed that identification of bounded length metamorphic virii is NP-complete [37] by decomposing the problem into one of graph isomorphism. In addition, Fogla *et al.* [17] showed that finding a polymorphic blending attack is also an NP-complete problem.

TRAFFIC CONTENT ANALYSIS

Snort [36] 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, 35, 28, 24, 44, 43, 25, 2]. 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) [38] treats packet content as machine instructions. SigFree [42] adopts similar techniques. 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.* [23] detect polymorphic worms by learning a control flow graph for the worm binary. *Convergent static analysis* [8] also aims at revealing the control flow of a random sequence of bytes.

Statistical content anomaly detection is another avenue of research, and PayL [40] 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 compares ratios of seen and unseen grams to determine normality.

COUNTERING POLYMORPHISM

Recent work [39] calls into question the ultimate utility of exploit-based signatures, and research on *vulnerability–specific* protection techniques [11, 7, 18] 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. Cui *et al.* [12] combine dataflow analysis and protocol or data format parsing to construct network or file system level "data patches" to filter input instances related to a particular vulnerability.

Brumley *et al.* [7] supply an initial exploration of some of the theoretical foundations of *vulnerability–based signatures*. Vulnerability signatures help classify an entire set of exploit inputs rather than a particular exploit instance. As an illustration of the difficulty of creating vulnerability signatures, Crandall *et al.* [11] discuss generating high quality vulnerability signatures via an empirical study of the behavior of polymorphic and metamorphic malcode. The authors present a vulnerability model that explicitly considers that malcode can be arbitrarily mutated. They outline the difficulty of identifying enough features of an exploit to generalize about a specific vulnerability. For example, the critical features of an exploit may only exist in a few or relatively small number of input tokens, and if the attacked application is using a binary protocol, telltale byte values indicating an attack may be common or otherwise unextraordinary values. For example, the Slammer exploit essentially contains a single "flag" value of `0x4`. For other protocols, detecting that the exploit contained the string "HTTP" or some URL typically does not provide enough evidence to begin blocking arbitrary requests — or if it does, our analysis indicates that such exploits can be arbitrarily mutated, thus vastly increasing the signature database and the processing time for benign traffic.

One way to counter the presence of the engine we propose in Section 3 is to use an anomaly detection (AD) sensor to shunt suspect traffic (that is, traffic that does not match normal or whitelisted content) to a heavily instrumented replica to confirm the sensor's initial classification. In fact, Anagnostakis *et al.* [2] propose such an architecture, called a "shadow honeypot." A shadow honeypot is an instrumented replica host that shares state with a production application and receives copies of messages sent to a production application — messages that a network anomaly detection component deems abnormal. If the shadow confirms the attack, it creates a network filter for that attack and provides positive confirmation to the anomaly detector. If the detector misclassified the traffic, the only impact will be slower processing of the request (since the shadow shares full state with the production application). The intuition behind this approach is that the normal content model for a site or organization is regular and well–defined relative to the almost random distribution representative of possible polymorphic exploit instances. If content deemed normal is put on the fast path for service and content deemed abnormal is shunted to a heavily protected copy for vetting, then we can reliably detect exploit variants without heavily impacting the service of most normal requests.

Since network traffic may look similar enough across sites, pretrained blending attacks such as the ones we explored in section 3 pose a real threat. Future statistical IDS techniques should take measures to hide the profiles of the normal content from the attacker. If we can force the attacker to guess where to aim his attack then perhaps we can turn the complexity of n-space to our favor.

6. CONCLUSIONS

Our empirical results demonstrate the difficulty of modeling polymorphic behavior. We briefly summarized the achievements of the shellcoder community in making their code polymorphic and examined ways to improve some of these techniques. We presented analytical methods that can help assess the capabilities of polymorphic engines and applied them to some state-of-the-art engines. We explained why signature–based modeling works in some cases and confirmed that the viability of such approaches matches the intuitive belief that polymorphism will eventually defeat these methodologies. The strategy of modeling malicious behavior leads to an unending arms race with an attacker. Alternatively, white–listing normal content or behavior patterns (perhaps in randomized ways in order to defend against blending attacks) might ultimately be safer than blacklisting arbitrary and highly varied malicious behavior or content.

7. REFERENCES

[1] ALEPH0NE. Smashing the Stack for Fun and Profit. *Phrack 7*, 49-14 (1996).

[2] ANAGNOSTAKIS, K. G., SIDIROGLOU, S., AKRITIDIS, P., XINIDIS, K., MARKATOS, E., AND KEROMYTIS, A. D. Detecting Targeted Attacks Using Shadow Honeypots. In *Proceedings of the* 14^{th} *USENIX Security Symposium.* (August 2005).

[3] BARATLOO, A., SINGH, N., AND TSAI, T. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference* (June 2000).

[4] BARRANTES, E. G., ACKLEY, D. H., FORREST, S., PALMER, T. S., STEFANOVIC, D., AND ZOVI, D. D. 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).

[5] BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium* (August 2003), pp. 105–120.

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

[7] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards Automatic Generation of Vulnerability-Based Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy* (2006).

[8] CHINCHANI, R., AND BERG, E. V. D. 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)* (September 2005), pp. 284–304.

[9] COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP)* (October 2005).

[10] COWAN, C., PU, C., MAIER, D., HINTON, H., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Symposium* (1998).

[11] CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. 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).

[12] CUI, W., PEINADO, M., WANG, H. J., AND LOCASTO, M. E. ShieldGen: Automated Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In

Proceedings of the IEEE Symposium on Security and Privacy (May 2007).

[13] DETRISTAN, T., ULENSPIEGEL, T., MALCOM, Y., AND VON UNDERDUK, M. S. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack 11*, 61-9 (2003).

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

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

[16] FERRIE, P., AND SZÖR, P. Zmist Opportunities. `http://pferrie.tripod.com/papers/zmist.pdf`, 2005.

[17] FOGLA, P., AND LEE, W. Evading network anomaly detection systems: Formal reasoning and practical techniques. In *Proceedings of the* 13th *ACM Conference on Computer and Communications Security (CCS)* (2006), pp. 59–68.

[18] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. 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] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10th *ACM Conference on Computer and Communications Security (CCS)* (October 2003), pp. 272–280.

[21] KE WANG, JANAK J. PAREKH, S. J. S. Anagram: A Content Anomaly Detector Resistant To Mimicry Attack. In *Proceedings of the* 9th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (2006).

[22] KIM, H.-A., AND KARP, B. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Conference* (2004).

[23] KRUGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 207–226.

[24] LIANG, Z., AND SEKAR, R. Fast and Automated Generation of Attack Signatures: A Basis for Building Self-Protecting Servers. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[25] LOCASTO, M. E., WANG, K., KEROMYTIS, A. D., AND STOLFO, S. J. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 82–101.

[26] METASPLOIT DEVELOPEMENT TEAM. Metasploit Project, 2006. http://www.metasploit.com.

[27] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science* (2003), vol. 89.

[28] NEWSOME, J., KARP, B., AND SONG, D. Polygraph: Automatically Generating Signatures for Polymorphic Worms. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2005).

[29] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the* 12th *Symposium on Network and Distributed System Security (NDSS)* (February 2005).

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

[31] POLYCHRONAKIS, M., ANAGNOSTAKIS, K. G., AND MARKATOS, E. P. Network-level polymorhpic shellcode detection using emulation. In *Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)* (2006).

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

[33] RUSSELL, S., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[34] SIDIROGLOU, S., GIOVANIDIS, G., AND KEROMYTIS, A. D. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the* 8th *Information Security Conference (ISC)* (September 2005), pp. 1–15.

[35] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm Fingerprinting. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)* (2004).

[36] SNORT DEVELOPMENT TEAM. Snort Project. http://www.snort.org/.

[37] SPINELLIS, D. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory 49*, 1 (January 2003), 280–284.

[38] TOTH, T., AND KRUEGEL, C. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the* 5th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (October 2002), pp. 274–291.

[39] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference* (August 2004), pp. 193–204.

[40] WANG, K., CRETU, G., AND STOLFO, S. J. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the* 8th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2005), pp. 227–246.

[41] WANG, K., AND STOLFO, S. J. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the* 7th *International Symposium on Recent Advances in Intrusion Detection (RAID)* (September 2004), pp. 203–222.

[42] WANG, X., PAN, C.-C., LIU, P., AND ZHU, S. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the* 15th *USENIX Security Symposium* (2006), pp. 225–240.

[43] XU, J., NING, P., KIL, C., ZHAI, Y., AND BOOKHOLT, C. Automatic Diagnosis and Response to Memory Corruption Vulnerabilities. In *Proceedings of the* 12th *ACM Conference on Computer and Communications Security (CCS)* (November 2005).

[44] YEGNESWARAN, V., GIFFIN, J. T., BARFORD, P., AND JHA, S. An Architecture for Generating Semantics-Aware Signatures. In *Proceedings of the* 14th *USENIX Security Symposium* (2005).