Mach Learn (2010) 81: 179–205
DOI 10.1007/s10994-009-5143-5

On the infeasibility of modeling polymorphic shellcode

Re-thinking the role of learning in intrusion detection systems

Yingbo Song · Michael E. Locasto · Angelos Stavrou · Angelos D. Keromytis · Salvatore J. Stolfo

Received: 31 March 2008 / Revised: 28 July 2009 / Accepted: 7 August 2009 /
Published online: 29 October 2009
Springer Science+Business Media, LLC 2009

Abstract Current trends demonstrate an increasing use of polymorphism by attackers to disguise their exploits. The ability for malicious code to be easily, and automatically, transformed into semantically equivalent variants frustrates attempts to construct simple, easily verifiable representations for use in security sensors. In this paper, we present a quantitative analysis of the strengths and limitations of shellcode polymorphism, and describe the impact that these techniques have in the context of learning-based IDS systems. Our examination focuses on dual problems: shellcode encryption-based evasion methods and targeted "blending" attacks. Both techniques are currently being used in the wild, allowing real exploits to evade IDS sensors. This paper provides metrics to measure the effectiveness of modern polymorphic engines and provide insights into their designs. We describe methods to evade statistics-based IDS sensors and present suggestions on how to defend against them. Our experimental results illustrate that the challenge of modeling self-modifying shellcode

Editors: Pavel Laskov and Richard Lippmann.

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.

Y. Song (✉) · A.D. Keromytis · S.J. Stolfo
Department of Computer Science, Columbia University, New York, NY 10027, USA
e-mail: yingbo@cs.columbia.edu

A.D. Keromytis
e-mail: angelos@cs.columbia.edu

S.J. Stolfo
e-mail: sal@cs.columbia.edu

M.E. Locasto · A. Stavrou
Department of Computer Science, George Mason University, Fairfax, VA 22030, USA

M.E. Locasto
e-mail: mlocasto@gmu.edu

A. Stavrou
e-mail: astavrou@gmu.edu

by signature-based methods, and certain classes of statistical models, is likely an intractable problem.

Keywords Shellcode · Polymorphism · Metrics · Blending

1 Introduction

Code injection attacks, and their counter-measures, remain important research topics within the security community. Attacks such as buffer overflow and heap corruption exploit vulnerabilities in the way certain software handles input data, to inject foreign code into the execution context of target applications. At the memory layer, the injected code is composed of machine instructions known as "shellcode", whose etymology originates from their use in spawning command shells on the target. Until recent years, one of the most relied-upon methods to detect such attacks have been network layer signature matching, where specialized monitors issued alerts for suspicious strings observed within network traffic. Recently, however, standard practice in the attacker community have evolved to incorporate sophisticated obfuscation techniques which aim to render these detection schemes obsolete. The new paradigm, known as "polymorphic shellcode," utilizes self-modifying code that is delivered to the target in an encrypted form and performs dynamic self-decryption upon execution, revealing the actual attack code at the last moment. Evasion tactics against statistical sensors have also begun appearing, such as purposely skewing the distribution of bytes within shellcode—such methods represent the beginnings of a new challenge of targeted blending attacks.

In a previous paper, we examined the difficulty of modeling polymorphic shellcode from a machine learning perspective with experiments using real world obfuscation engines (Song et al. 2007). While emerging trends in polymorphic shellcode design include elements of both code obfuscation and targeted blending, our previous paper focused primarily on the former technique and had only briefly mentioned the latter. In this longer paper, we expand on our earlier work to provide a more comprehensive exploration of the problem of shellcode detection. This paper provides an examination of machine learning-based blending attacks which, given the current trends, we expect adversaries to utilize in the near future. This paper analyzes the complexity of these two problems, discuss their implications and presents evidence that signature-based modeling and certain statistical approaches are not likely to remain as practical solutions in the long term.

Organization This paper is organized as follows: Sect. 2 describes shellcode and polymorphism in detail. Section 3 discusses related works in this area. Code obfuscation techniques and our methods for measuring the strengths of polymorphic engines are presented in Sect. 4 while Sect. 5 covers the fusion of code obfuscation and simple targeted blending attacks. Section 6 presents our methodology for advanced higher order blending attacks. Section 7 explores the theoretical boundaries of shellcode polymorphism. We present our conclusions in Sect. 8.

2 Shellcode

Modern computer architectures are designed to follow the Von Neumann model, where instruction and data are stored together in memory, and it is largely up to the operating system to keep track of which is which. Memory layer exploits take advantage of software flaws such as bounds-checking errors, to break out of input data buffers and overwrite important system control information, such as stack pointers. This form of attack is exemplified by the classic technique known as "smashing the stack." In this scenario, when a function within a program is called (for simplicity, assume the program is written in C) the OS will create a frame record for the function. This record is allocated on the stack and encapsulates buffers for variables such as function arguments and local variables, along with control information such as the address of the calling function (EIP). The attacker may attempt to exploit this layout by providing a very long string as input, one that is much larger than what the buffer can hold. If the input is accepted and copied into the buffer without regard to proper boundary checking, it would overflow that buffer and overwite the EIP variable within that stack frame. Further, the attacker's input would be crafted so that the value which overwrote EIP would be an address value that pointed back into the input itself, which would be filled with the attacker's executable code. When the function returns, the control flow of the process passes into the injected code and the execution context of the process is hijacked.

It was "AlephOne" who first illustrated the basics of smashing the stack (AlephOne 2001). The virus writer "Dark Avenger" later introduced the "Dark Avenger Mutation Engine", a polymorphic engine for viruses that mutated each instance of the virus in order to throw off signature-based AV scanners. This later influenced the shellcoder "K2" to develop the initial research steps in shellcode polymorphism, which he implemented in the "ADMmutate" engine (K2 2003). "Rix" later advanced this field by demonstrating how to perform alphanumeric encoding (Rix 2001), which allowed shellcode to be written in alphanumeric characters. Later, "Obscue" described how to encode shellcode such that it would survive ASCII-to-Unicode transformations (Obscou 2003) and the "CLET" team developed the technique of "spectrum spoofing" which altered the statistical distribution of shellcode to throw off intrusion detection systems. They also expanded ideas on multilayer ciphering, as well as randomized register-selection (Detristan et al. 2003). More recently, the "Metasploit" project combined vulnerability probing, code injection, and shellcode polymorphism—among other features—into one complete system (Metasploit Developement Team 2006).

Shellcode typically contains three major components: (1) the "NOP sled" (2) the payload and (3) the "return address zone" which consists of a single jump target address, repeated in linear sequence. As mentioned previously, the goal of a simple stack exploit is to overwrite an existing address value on the stack with values contained within the return zone. This would cause the control flow to jump back into the input string that the attacker has sent. The NOP sled is designed to safely catch this jump, and then pass the execution into the payload portion of the attack.

Good payload code can be difficult to write and obfuscation is used as a method to prevent detection. Modern obfuscation techniques typically employ two ways of disguising shellcode: the first method attempts to automatically re-write code such that each instance differ syntactically, but retains the same operational semantics as previous samples. This approach is akin to code-"metamorphism," and has been shown to be decomposable to graph isomorphism (Spinellis 2003), which is NP-complete in terms of difficulty. A second, more tractable, approach is to use code-obfuscation, such as encrypting the shellcode with a randomly chosen key. Under this second scheme, a decoding routine is "semantically prepended" to the shellcode and executes before the payload is triggered. This so called "decoder" reverses the encryption dynamically at runtime, thus allowing the code to remain obfuscated while in transit but unveiled prior to execution. This latter technique has proven to be the most effective in practice, and has led to the development of a fairly standard form of shellcode, shown in Fig. 1.

[nop][decoder][encpayload][retaddr]

Fig. 1 Encrypted shellcode structure

```
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
```

Fig. 2 A 35-byte polymorphic decryption loop. *Left column*: position. *Center column*: byte values. *Right*: x86 assembly code. Note the five cipher operations: rol, xor, ror, sub, and add, that begin at 0x09. The working register for the cipher is EAX. Note the stop condition at 0x2B

With the payload encrypted, only the decoding routine needs to be made polymorphic, since it must remain in the clear. The return zone is vulnerability dependent, and can also be ofbuscated if needed, as described later. Rapid development of polymorphic techniques following this model has resulted in a number of off-the-shelf polymorphic engines such as ADMmutate (K2 2003), CLET (Detristan et al. 2003), Metasploit (Metasploit Develepoment Team 2006), Shellforge (Biondi 2006), Tapion (Bania 2009), Alpha2 (ported into Metasploit) and others. Modern self-ciphering techinques encode the payload using a reversible cipher operating such as “xor/xor”, “add/subtract”, “ror/rol”, etc. and will use several rounds of such cipherings for added diversity. Decoders for these types of obfuscations typically have a length of 30–50 bytes; the length of the overall shellcode sample will be several hundred to several thousand bytes.

With the payload encrypted, the effectiveness of the polymorphic transformation is now dependent on *how well the decoder can be hidden*, so that AV and IDS solutions cannot simply look for a decryption routine instead of a payload. Decoder obfuscation can be achieved in numerous ways: by rearranging and randomizing the order of the individual ciphers components, by using randomly chosen keys, by inserting junk instructions, and more, as this paper will describe. Figure 2 shows an example of a decoder that we extracted from a popular shellcode engine, it incorporates many of the features previous mentioned. Decoders provide an effective technique for rapid and simple dissemination of shellcode variants, allowing attackers to reuse old payloads in arbitrarily different forms—an invaluable tool for those looking to remain hidden. For the rest of the shellcode components, diversity can be injected into each component, as this section will show.

- [NOP]: Recall that the attacker needs to redirect the process execution flow into the injected payload code during exploitation (i.e. when overwriting EIP). In reality, it is rare that the exact beginning address of the injected code is known, making it difficult to specify exact addresses in the return zone. The NOP sled is a large contiguous array of NOP instructions, such as `{x90,x90,...,x90}`, that is prepended to the decoder to safely catch the execution jump, which can land in any position within the sled. The NOP sled transfers the execution flow into the decoder without leaving the system in a volatile state. As we would expect, many signature-based systems rely on this artifact for detection, for example, scanning for large blocks of `x90` characters. As such, various innovations have been introduced to make the sled polymorphic as well. Such methods leverage the fact that the sled does not need to consist of actual "NOP" machine instructions (such as x90 for the x86 architecture). Rather, such instructions need only be equivalent to code which did nothing, but can be invoked at any location. In the ADMmutate documentation, K2 described the discovery of 55 different ways to write single-byte NOP equivalent instructions. Thus, a NOP-sled of length N implies potentially 55^N unique sleds.

  "Recursive NOP sleds" have been proposed by the CLET team (Detristan et al. 2003). This technique for building sleds 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 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 that can be combined to create the NOP-sled. To the best of our knowledge, no analysis of the potential of this method exists, but it serves as a very useful polymorphic technique because modeling this type of sled may amount to modeling random instructions. While such effective ways at disguising the NOP-sled exist in practice, often times they are not even needed, as described by Foster et al. (2005). For many Windows exploits, the exact location of the target can be recovered due to the inherent homogeneity of the operating system and its applications.
- [RETADDR]: Attempting to model the return address zone to detect shellcode presents no less of a challenge. Without address space randomization, the locations of the stack and stack variables on most architectures remain consistent. The homogeneity of Windows systems means that, in most cases, at least for the same version of the OS, the same applications exist in identical locations in memory. Thus, the attacker has a firm basis for recovering a very good estimate for the location of the injected shellcode, which is then used to craft the return address section. Return address zone polymorphism is trivially achievable when using a NOP sled, by modifying the lower order bits within the address elements. This method causes the control-flow to jump into different positions in the NOP sled, but as long as it still lands somewhere in the sled then exploit will work. Thus, if the return address zone consists of the address target, repeated an m number of times, and if each value can be modified v ways (where v is some tolerable variance in the jump target) then a total of v^m possible variations exist. In addition, in certain classes of attacks, when the vulnerable function stores the location of the data input in a register, then no memory location guessing is required at all (thus no NOP sled or return zone are needed), other tricks may be employed to recover the *exact* target payload address. A notable study was

[nop][decoder][enc payload][PADDING][retaddr]

Fig. 3 Shellcode with blending section

[nop][encrypted payload][decoder][retaddr]
[nop][decoder 1][enc.payload][decoder 2][retaddr]
[nop][padding][enc.payload][padding][decoder][retaddr]
... etc.

Fig. 4 Other potential structures

done by Crandall et al. (2005b) who refer to this technique as a "register spring." For off-by-one exploits, a return zone is also not needed (since only one byte is over-written).

- `Spectrum shaping and byte padding`: Recent research has demonstrated the feasibility of polymorphic "blending" attacks where the shellcode is actually crafted to appear similar to benign traffic in terms of the n-gram content distribution as described by the worksof Fogla and Lee (2006) and Kolesnikov and Lee (2006). The CLET polymorphic engine (Detristan et al. 2003) uses this technique, to some extent. They accomplish this by changing the structure of the shellcode that it generates to take on a new form (Fig. 3).

In CLET shellcode, extra bytes are added to the "PADDING" area, shown in Fig. 3, to skew the 1-gram distribution of the shellcode. In addition, the payload itself is ciphered with different length keys, each of which is randomly generated. The entropy from these keys propagate into the shellcode through the cipher operations.

Perhaps the most troublesome threat is that these individual techniques are highly independent, and can be combined into a single engine. Later, we show that this is, in fact, not difficult to accomplish. Section 5 describes an engine that we designed as a proof of concept. Moreover, the structure described previously, in Fig. 3, is merely a convention; it is equally tractible to modify the individual sections between the NOP sled and the return address, to inject additional randomness. With additional jump instructions, new shellcode of the following form are also easily achievable (Fig. 4).[1]

This paper studies these dual evasion tactics, of self-cipher-based obfuscation and targeted blending attacks, and metrics are proposed to quantify the effectiveness of these obfuscations. Our attention is focused on analyzing the *decoder* portion of the shellcode, since this is the most constrained portion of the attack. Other than the higher order bits of the return address zone, this is the only section that cannot be trivially disguised, since it must hold executable machine code. At the same time we also study the challenge of making shellcode appear similar to normal traffic, in order to evade certain classes of statistical anomaly detection-based sensors.

3 Related work

Though it was the virus writers who first introduced self-decryption and self-unpacking-based polymorphism as simple, yet effective, ways of evading signature-based AV scanners,

[1] We demonstrate that these variations are indeed possible in a proof-of-concept engine described in a separate work-in-progress paper.

it did not take long before shellcode writers adopted this technique. Recently, polymorphism has become a standard tool for web-based attacks as well, where exploits are delivered in the form of interpreted code such as Javascript or PHP. The MPack (Panda Labs 2007) toolkit, currently being traded in the malware underground and responsible for tens of thousands of recent website compromises, obfuscates Javascript attacks with several layers of self-encryption and randomization. The Metasploit team, already famous for their shellcode exploit engines, has announced the release of their "eVade o'Matic Module (VoMM)," a polymorphism engine for Javascript based exploits.

Countering such polymorphic attacks is a difficult problem. Researchers have proposed a variety of defenses against polymorphic shellcode, from artificial diversity of the address space (Bhatkar et al. 2003) or instruction set (Kc et al. 2003; Barrantes et al. 2003) to compiler-added integrity checking of the stack (Cowan et al. 1998; Etoh 2000) or heap variables (Sidiroglou et al. 2005) and "safer" versions of library functions (Baratloo et al. 2000). Other systems explore the use of tainted dataflow analysis to prevent the use of untrusted network or file input (Costa et al. 2005; Newsome and Song 2005) as part of the instruction stream. 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 Development Team (2009). Many signature generation schemes focus on detection heuristics such as traffic characteristics (Singh et al. 2004; Kim and Karp 2004) (e.g., frequency of various packet types) or identification of the NOP sled (Toth and Kruegel 2002), while others derive a signature from the actual exploit code (Liang and Sekar 2005; Locasto et al. 2005) or statistical measures of packet content (Wang and Stolfo 2004; Wang et al. 2005; Newsome et al. 2005), including content captured by honeypots (Yegneswaran et al. 2005).

3.1 Traffic content analysis

Snort Development Team (2009) is a widely deployed open-source signature-based detector. Exploration of automatic exploit-signature generation has been the focus of a great deal of research (Kim and Karp 2004; Singh et al. 2004; Newsome et al. 2005; Liang and Sekar 2005; Yegneswaran et al. 2005; Locasto et al. 2005; Anagnostakis et al. 2005). To create a signature, most of these systems either examine the content or characteristics of network traffic, or instrument the host (through some degree of virtualization) to identify malicious input. "Host-based" approaches filter traffic through an instrumented (virtualized) version of the application to detect malcode. If detection is confirmed then the malcode is dissected to dynamically generate a signature in order to stop similar attacks. Abstract Payload Execution (APE) (Toth and Kruegel 2002) examines network traffic and treats packet content as machine instructions. Instruction decoding of packets can identify the NOP sled. Krugel et al. (2005) detect polymorphic worms by learning a control-flow graph for worm binaries with similar techniques. Convergent Static Analysis (Chinchani and Berg 2005) aims to reveal the control flow of a random sequence of bytes. The SigFree (Wang et al. 2006b) system adopts similar processing techniques. Statistical content anomaly detection represents another key direction of research. The PayL (Wang et al. 2005) sensor models 1-gram distributions for normal traffic, and detects anomalies by using the Mahalanobis distance to gauge the normality of incoming packets. Anagram (Wang et al. 2006a) 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—packets with a high amount of unrecognized content generate alerts.

3.2 Proactive defense

Preventing intrusions by attempting to remove the weaknesses of current execution environments is another active area of research. Data Execution Prevention (DEP) is used in the latest versions of the Windows operating system to flag certain memory areas as non-executable in order to prevent code injection. Similar features exist such as the W^X feature in BSD kernel. StackGuard and similar techniques (Cowan et al. 1998; Etoh 2000) inject control variables into the stack to detect unauthorized modifications, such as those stemming from a buffer overflow. Program shepherding (Kiriansky et al. 2002) 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. (2005) 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 by making the host a “moving target.” Address space randomization for example prevents return-into-libc type attacks. Instruction set randomization (Kc et al. 2003; Barrantes et al. 2003) has been proposed to frustrate an attacker’s ability to write executable code for a target system.

3.3 Countering polymorphism

Shield (Wang et al. 2004) represents recent work which calls into question the ultimate utility of exploit-based signatures, and research on vulnerability-specific protection techniques (Crandall et al. 2005a; Brumley et al. 2006; Joshi et al. 2005) (and especially Dynamic Taint Analysis Costa et al. 2005; Newsome and Song 2005) 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: i.e., the control flow path). Cui et al. (2007) discuss combining tainted dataflow analysis (similar to that used in the Vigilante system Costa et al. 2005) and protocol or data format parsing to construct network or filesystem level “data patches” to filter input instances related to a particular vulnerability.

Brumley et al. (2006) 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. (2005a) discuss generating high quality vulnerability signatures via an empirical study of the behavior of polymorphic and metamorphic malcode. They outline the difficulty of identifying enough features of an exploit to generalize about a specific vulnerability. One way to counter the presence of the proof-of-concept engines which we propose in later sections to use Anomaly Detection (AD) sensors to shunt suspect traffic (that is, traffic that does not match normal or white-listed content) to a heavily instrumented replica to confirm the sensor’s initial classification. The intuition behind our 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.

In fact, Anagnostakis et al. (2005) propose such an architecture, called Shadow Honeypot. A shadow honeypot is an instrumented replica host that fully shares state with the production application, that 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 to drop future instances of that attack, and provides positive confirmation to the anomaly detector. If the detector mis-classified the traffic, the only impact will be slower processing of the request (since the shadow shares full state with the production application).

4 Polymorphic engine analysis

As previously mentioned, our study is focused on the most constrained portion of the shellcode: the decoder, which is the only section that is delivered in clear-text. Six modern polymorphic engines are examined in this paper: ADMmutate, CLET, and four engines from Metasploit: Shikata Ga Nai, Jumpcall Additive, Call4dWord and Fnstenv Mov. Previous research on automatic generation of exploit signatures from polymorphic code (Kim and Karp 2004; Newsome et al. 2005) reports successful detections of exploits from many existing engines, some of which are from Metasploit. We explore the reasons why some signature-based methods are successful, and why they might not work in the future.

This section explores methods to measure the usefulness of a polymorphic engine. We demonstrate how to easily visualize the artifacts that some engines leave in their shellcode instances, artifacts that can be used as signatures for string-matching-based detection (the type that AV scanners use.) We also show that these artifacts appear differently across engines—an indication that they cannot be used as general indicators for polymorphic code, outside of their training class. The metrics presented here are built upon two main factors, or "strengths." What we call the VARIATION STRENGTH is a quantitative measure of an engine's ability to generate high spatial variance in their decoder samples, on a per-byte basis. The PROPAGATION STRENGTH is an information divergence measure, conditioned on pairwise decoder distances, and is meant to be a way of measuring the minimum expected distance between samples drawn from the same engine. These metrics are positively correlated though not exactly equivalent, but when used together, can provide good empirical estimates to support real world observations.

For each of the six engines, our metric yields a scaled score which we call the "relative polymorphism strength score" or *p-score* for short. This normalized metric allows multiple engines to be evaluated with respect to each other, as well as with respect to random distributions. The concept of a "spectral image" is also proposed, which allows one to view the distortion in a collection of decoder samples, and is used in combination with the above metrics to derive our ultimate results. These results lend strong empirical support to the long held, yet unproven, belief that the class of x86 polymorphic shellcode is too random to model.

Spectral image It is often desirable to have an easy and intuitive way to understand the effectiveness of a particular engine, for analysis and reverse engineering. For this purpose we propose the concept of viewing an engine's "Spectral Image." This image can be generated as follows: given a polymorphic engine, produce a set of N decoders, each of length d—for non fixed-length decoders, padding can be appended to make the lengths equal. These decoders should then be stacked together, yielding a $N \times d$ matrix which can then be rendered as an image where the ith byte of decoder j is the intensity value for the (i, j)th pixel of the image. The byte-value of 0x00 would produce a black pixel, 0xFF, a white pixel, and everything in between would fall within a shade of gray. This representation is most useful as a quick way to observe bytes which exist in the same positions within all decoders from a particular engine—such salient bytes would show up clearly as *columns* within the image.

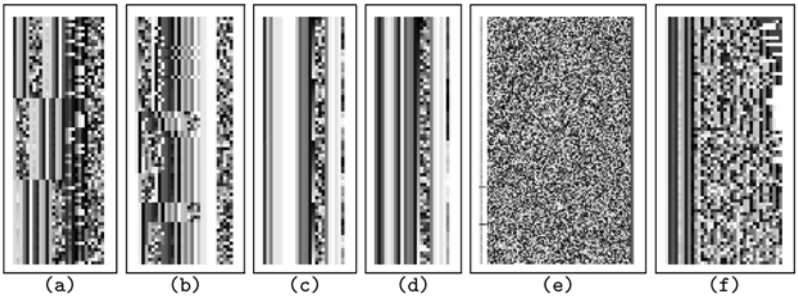


Fig. 5 Visualization of shellcode variations. (**a**) Shikata Ga Nai (**b**) Jcadd (**c**) Call4dWord (**d**) Fnstenv Mov (**e**) ADMmutate (**f**) CLET. Each pixel row represents a decoder from that engine and each individual pixel value represents the corresponding byte from that decoder. A column of identical intensities indicates an identifiable artifact left by the engine

Figure 5 shows the images for six engines. These were generated by taking a single shellcode sample and generating 10000 unique samples through as many independent obfuscations, per engine. From these sequences the decoder portion were extracted, down-sampled, then stacked. The values are then row-sorted and displayed as previously described. Note how Shikata Ga Nai generates roughly three subclasses of decoders—similar 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 indicate that these engines always embed large artifacts in every decoder. The vertical band for the CLET engine represents register-clearing operations. It is evident that though these engines perform the same basic actions (to decode a string within a small distance of itself in memory) there lacks a single invariance across these different engines.

Construction of a spectral image is a simple way to visualize decoder variation but a metric that can quantify the observed distortion is desired—for this we can take a parametric approach and assume that each decoder sample is a d-dimensional random variable embedded an integer space, with each dimension bounded to the range of values representable with a single byte $\mathbf{x} \in \mathbb{B}^d$ where $\mathbb{B} \in \{0, 1, \dots, 255\}$. The metric assumes that the samples are independent and identically-distributed (i.i.d.) and examination of the moments of the sample distribution provides a way to measure the "scatter" of the samples generated by different engines. We note that this metric is, at best, an approximation since the geometry of the x86 instruction space does not admit a simple Euclidean space embedding. x86 instructions are variable-length, and matters are complicated by the fact that polymorphic instructions may actually "overlap," such as the case of recursive NOP sleds. The same engine may generate decoders of different lengths and "scatter" identical decoders within $\mathbb{B}^d$ by using different keys, or swapping the order of cipher components, all without changing the operational semantics of the code. The metrics that we propose simply remaps this complex decoder manifold to the $\mathbb{B}^d$ cube, and measures the scatter of the distribution in this space, treating these polymorphic transformations as shifts and translations.

In practice, not just decoders but all data, will inhabit this $\mathbb{B}^d$ space when information is represented as byte sequences. In the domain of IDS solutions, false positives represent the primary inhibiting factor and for most applications a 0.1% false positive rate could be considered unusable for modern gigabit-rate traffic networks, since such a rate would entail hundreds of alerts per second when modeling packet-level traffic. Given that benign data and polymorphic shellcode both share the same bounded space, the fact that we cannot bound arbitrary benign data when we design generic IDS systems, and that any confusion between the two would render an IDS solution unusable, we define "strength" as the ability of an engine to increase the scatter of its decoders within this space (to envelope normal content), such that a larger scatter entails a stronger rating. The ideal polymorphic engine is able to yield a uniform distribution of decoders throughout $\mathbb{B}^d$. Such an engine is unlikely to be reliably detected under a 0.1% FP constraint. Under this assumption, the remain part of this section describes our approach at quantifying this strength.

Variation strength To quantify strength as a measure of variance, so that a single score may be assigned, a function can be constructed, in the most general way, by assuming a Gaussian distribution, and then utilizing the spectral norm of the covariance matrix.

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

$\mathbf{\Sigma} \in \mathbb{R}^{d\times d}$ for sequences of length d where we have N total samples. $\mathbf{x}_i$ is a single decoder sample, generated by the engine that we are examining, and $\mu = \frac{1}{N}\sum_i^N \mathbf{x}_i$ to be the sample mean. Both $\mathbf{x}$ and μ are column vectors. The construction of our metric relies on evaluating the spectral norm of this covariance matrix. The solution is simply $\mathbf{v}$ and λ such that $\mathbf{\Sigma v} = \mathbf{v}\lambda$. $\mathbf{v}$ is the set of d eigenvectors and λ are the square of the corresponding d eigenvalues. Summing over the square roots of $\{\lambda_1, \ldots, \lambda_d\}$ gives us the variation strength of an engine:

$$\Psi(\text{engine}) = \frac{1}{d}\sum_{i=1}^{d}\sqrt{\lambda_i} \tag{2}$$

To analyze the strength of a single polymorphic engine, we run 10000 independent encoding operations on a seed shellcode sample and extract the same number of unique decoder sequences. These sequences are used to generate the covariance matrix according to (1) and recover the eigenvalues to solve for the score using (2).

Propagation strength Some trivial byte-level transformations might yield the same codeblock signatures but in different positions within the decoder, such as shifting the order of a group of identical instructions—a method that does not necessarily speak much for the engine's obfuscation capabilities. Simply adjusting the padding length, for example, might cause a sequence of bytes within the decoder to be positionally shifted. Such trivial transformations are commonly used, yet they induce large variances in the covariance matrix due to the fact that they are interpreted as complex translations within $\mathbb{B}^d$. To compensate, we introduce an additional term based on the *minimum* distances between the individual decoders. We refer to this quality as the engine's "propagation strength." This second metric is defined as the expected divergence between any two distinct decoders. One way to deal with trivial transformations, such as the padding example, is to use information divergence metrics such as Bhattacharyya affinity over the 1-byte distributions. This is useful if a blending attack is not involved. More stringent metrics such as string kernels are also possible but one must keep in mind that these sequences are non-i.i.d, and maliciously generated by actual adversaries—it is unwise to use overly-specific assumptions about the underlying parametric form. For our work, we found that simply using a minimum string-alignment distance was sufficient to obtain a good estimate. We use the following distance:

$$\delta(\mathbf{x}, \mathbf{y}) = \min_{r=1,\ldots,d}\left[\frac{\|\mathbf{x} - \text{rot}(\mathbf{y}, r)\|}{\|\mathbf{x}\| + \|\mathbf{y}\|}\right] \tag{3}$$

where $\mathrm{rot}(\mathbf{y}, r)$ indicates a rotation of the string $\mathbf{y}$ to the left by r-bytes, with wrap-around. 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 norms. This relates more to the *angle* between samples than their spatial distances, and removes the number of dimensions (decoder length) as a factor in the distance. The final form for the propagation strength is given as:

$$\Phi(\text{engine}) = \frac{2(1-\eta/d)}{N(N-1)} \sum_{i=1}^{N} \sum_{j=i+1}^{N} \delta(\mathbf{x}_i, \mathbf{y}_j) \tag{4}$$

As before, $\mathbf{x}_i$ is the ith decoder generated by an engine, N is the total number of decoders, and d is the length of the decoders. $\delta(\mathbf{x}, \mathbf{y})$ is the previously defined (3). This is a divergence metric for decoder sequences as a whole rather than byte-level components. In addition, an η parameter is introduced which measures the number of salient bytes within the samples generated by an engine. This is used as a scaling factor to decrease the strength of engines that leave consistent artifacts in their decoders; artifacts which can be exploited by signature based IDS implementations. For the case where an engine generates variable length decoders, the average length (over all sequences) can be used to compare any two sequences. Padding that exhibit the same statistical properties can be added to the shorter sequence and the longer sequence should be truncated to minimize the effect of distortion based on length.

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

$$\Pi(\text{engine}) = \Phi(\text{engine}) \cdot \Psi(\text{engine}) \tag{5}$$

Table 1 shows the relative strengths of these engines based on our metrics. $rand_{128}$ refers to a set of randomly generated strings with each byte value between [0, 128], encapsulating the range of ASCII printable characters. $rand_{256}$ refers to random strings with byte values between $[0, 256]$.

The minimum Euclidean distance metric that we use is not differentiable, therefore a normalization constant can not be calculated in closed form. Empirically, we can simply apply the metric to a uniformly random distribution, then divide the strengths of each engine by this value to generate a normalized score. We call this final scalar value the "relative polymorphism score" or *p-score*, for short. Table 1 shows that this metric is normalized since the strength for $rand_{128}$ is half of that for $rand_{256}$. The p-score, used in conjunction

Table 1 The decoder polymorphism strengths of various engines under our metric. Also shown as the scores for random distributions of strings within range 128 and range 256. Compare with Fig. 5

Engine	Prop. St.	Var. St.	Overall St.	p-score
Shikata	0.27	53.24	14.38	**0.67**
Jcadd	0.22	44.62	9.82	**0.46**
C4d	0.12	14.62	1.75	**0.08**
Fnstenv	0.14	15.70	2.20	**0.10**
Clet	0.28	53.00	14.84	**0.69**
Admmutate	0.30	68.76	20.62	**0.96**
$rand_{128}$	0.29	36.90	10.70	**0.50**
$rand_{256}$	0.29	73.74	21.38	**1.00**

with the spectral images, can be used to measure the effectiveness of polymorphic engines relative to each other, as well as to random noise. This method provides some usefulness in predicting detection success rates for new unseen engines. Three of the engines from Metasploit are not fully polymorphic, according to their documentation, and it is easy to see which ones these are. While the novelty and efficiency of CLET 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 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 more creative engines that we have seen, this particular artifact lowers the level of polymorphism in the decoders; this fact is even acknowledged in the engine's documentation. CLET also allows the user to specify arbitrary layers i.e., *xor* then *sub* then *add* etc. For our experiments, we tested on the default setting of five instruction operations.

5 A hybrid engine: combining polymorphism and blending

The polymorphism techniques described in earlier sections can be easily combined. To demonstrate this, we took the best of aspects of the CLET and ADMmutate engines and put them into a single engine. CLET's decoder leaves noticeable artifacts but it does have a useful spectral ciphering technique which allows the shellcode to blend to a target byte distribution by ciphering the payload with specially chosen keys. ADMmutate cannot perform blending but it can generate very random looking decoders, and produce recursive NOP-sled. We simply use CLET to cipher the shellcode, then hide CLET's decoder with ADMmutate and use ADMmutate's advanced NOP sled generator. Section 2 described techniques used to make the other sections polymorphic, such as return address randomization; we employ these tactics in our engine design.

The combination of all of these features makes the generated shellcode instances not only difficult to model but also allows them to blend into a target host's normal-traffic model. Every section of Shellcode can be made polymorphic, leaving only a padding section, known as the "blending section", exposed—as demonstrated in Fig. 6. Here, bytes were added into each of the padding sections for each sample so that, when stacked together, the independent samples render a picture of the BSD daemon. Each row of the three spectral images shown in

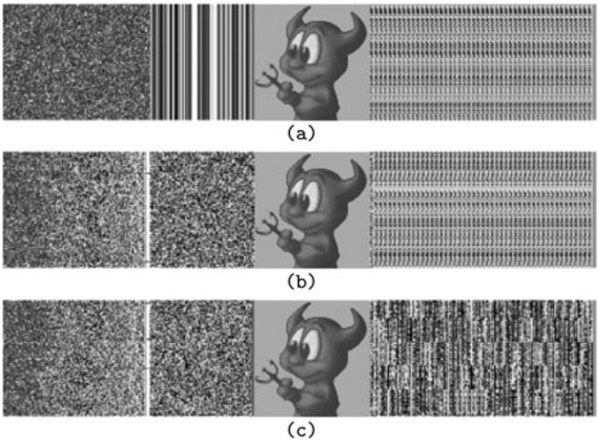


Fig. 6 Spectral images. (**a**) A single CLET mutated exploit is stacked row-wise 100 times (note the *vertical bands*). CLET leaves a blending section which is never reached in the execution. We fill this section with bytes that stack together to display a random picture. (**b**) CLET's decoder and exploit is hidden by ADMmutate, leaving only the blending bytes exposed. The repeating columns represent the [RETADDR] section which is shown morphed in (**c**) through application of random offsets

Fig. 6 represents a 512-byte *fully working* shellcode instance that was tested and confirmed to have executed successfully. In practice, this section can be enlarged and “normal looking” n-grams can be placed within this padding section, to make the shellcode blend into normal traffic and evade certain classes of statistical IDS sensors.

ADMmutate’s encoding scheme uses two layers of ciphering on the payload with 16-bit random keys. This allows the formerly detectable CLET decoder samples to be scattered across $\mathbb{B}^d$. The padding section can carry any arbitrary combination of bytes since the exploit code exists semantically in front of this section, and executes before the control flow can pass into the blending section. For 1-gram blending attacks, a simple method is to sequentially fill the section with new characters in proportion to the target frequency, and then randomly permuting the section. The permutation prevents signatures from being derived but, in terms of the 1-byte distribution, the section has not changed. In a later section, we show how to blend n-grams with $n > 1$. In our results, 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, can easily be enlarged. The only change that needs to be made is to increment the values in the [RETADDR] section to “aim a little higher” to compensate for the larger shellcode.

The [RETADDR] section is shown as the series of repeated columns, seen to the right of the padding section in Figs. 6(a), (b) (notice the periodicity). As mentioned in the previous section, the [RETADDR] is not easily modeled—this section normally has variable length, is permutable by adding random offsets and is both platform and vulnerability dependent. This mutability feature is demonstrated in Fig. 6(c) where we have mutated the [RETADDR] section by aiming `EIP` (the value in the return addresses) at the center of the NOP-sled and adding a random 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 return address is repeated. The base of the exponent (100) can be larger or smaller, depending on how large the NOP sled section is.

The only real weakness we see from ADMmutate is a white column, which represents a 4-byte salient artifact generated by the engine and is present in all of the decoders. This is obviously is too small to use as a signature or statistical feature. In terms of statistical features, Fig. 7(a) shows the 1-gram distribution of the decoders generated by our engine. This plot was calculated by finding the *average* 1-byte histogram of the decoders, then normalizing by dividing by the variance along each dimension. This is a Fisher-score-motivated

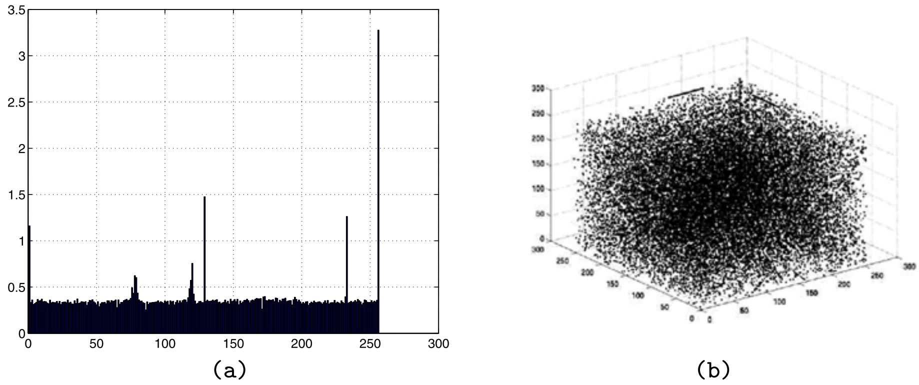


Fig. 7 adm + clet engine. (**a**) 1-gram distribution (**b**) 3-gram scatter. Comparing this to Fig. 15, we can see that this engine is equally difficult to model

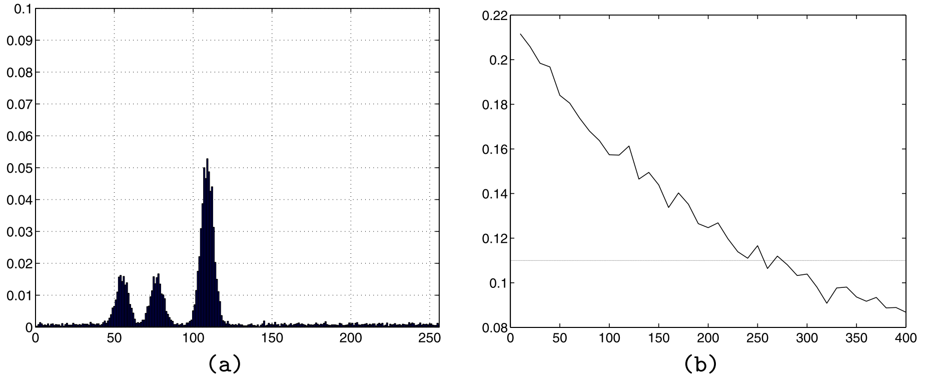


Fig. 8 adm + clet engine executing a blending attack. (**a**) Target distribution (**b**) distance to target given padding section size

operation, and normalizes the values so that we obtain their discriminative scores—if a feature is consistently present then it would have low variance. Thus, dividing by its variance increases the prominence of that feature. Conversely, if a feature exhibits very high variance, then it's discriminative power is low. From Fig. 7(a), we see that there is little to no signal from the 1-byte distribution. Figure 7(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 full spectrum spread.

Furthermore, we also added an automated blending function into our ADM + CLET engine. Figure 8 shows a simulation of a blending attack. Here, we first generate an artificial target distribution which is shown in Fig. 8(a). We chose the target distribution to be 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 multiple clear-text transfers. We also added binary noise to account for binary transfers, such as images and video. Each centroid has a variance of 15.

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

The Mahalanobis distance, shown in (6), is used in statistical IDS sensors such as PayL (Wang and Stolfo 2004). μ is the 1-byte target distribution and we set $\Sigma = 0.1\mathbf{I}$ where $\mathbf{I}$ is the identity matrix. The estimates are chosen exactly as described in the original PayL paper. Figure 8(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 shellcode instance 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 the equation provided above.

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 practice, the target distribution will have to be estimated by capturing traffic corresponding to normal interactions with the target host, using tools such as `tcpdump`. Then, using the captured data, estimate the parameters of the target models. In addition to this, we can also make use of *binary-to-text encryption* to transform the

Fig. 9 Function 2-Blend($\mathcal{D}$, L) returns an L-length blending string whose 1-gram transition probabilities mirror that of D

```
function 2-Blend($\mathcal{D}$, $L$)
    Estimate $p(x_i|x_j)$ from training data $\mathcal{D}$
    $x_1 \sim rand(1..256)$
    for i $\leftarrow$ 2 to $L$
        $x_i \sim p(x_i|x_{i-1})$
    end
return $x$
```

decoder portion of the shellcode into a string that consists of only printable characters by using techniques such as the ones provided in the ShellForge engine (Biondi 2006). There are also techniques that will allow the shellcode to survive sanitization functions such as `to_upper()` and `to_lower()`. For additional discussion on the topic of 1-gram blending, as well as some more detailed comments on potential attack scenarios where such tactics would most likely be effective, we refer the reader to the work by Fogla et al. (2006) who also present some interesting complexity arguments by appealing to NP-completeness. The next section of this paper extends this work to higher order blending with larger gram sizes.

6 *N*-gram blending

This section describes an approach towards blending attacks against statistical AD sensors which uses n-gram models. Given a sequence such as “http://”, 2-gram tokens include “ht”, “tt”, “tp”, *etc*. A description of a simple 2-gram blending attack in Sect. 6.1 sets up the motivation for our methodology, which is then generalized to the n-grams case in the remaining section.

6.1 2-gram blend

To generate a blending attack against 2-gram feature-counting-based anomaly detection models, we can recover the 1-gram transition probabilities and use a Markov random walk approach to generate the bytes that are to be placed in the blending section. These bytes, combined, will hereafter be referred to as the “blend-string.” Let $x \sim p(\Theta)$ denote drawing a sample x from an arbitrary distribution p that operates over a set of parameters Θ. For 2-grams, the probability distribution can be the 1-gram transition probabilities—that is, the probability of any gram to immediately follow any other gram. Under this model we have a conditional probability specified by $p(x_t|x_{t-1})$ where x_t denotes the character in position t. The simple model is *convex* in the data and parameters and therefore admits parameter estimation by Maximum Likelihood, which entails counting the number of gram-to-gram transitions from the training data, then normalizing the resulting transition matrix so that each row sums to 1.

Any sampling algorithm can be used for the $x \sim p(\Theta)$ step. For our experiments, we used random draws from a “bag-of-grams” model. Our method draws from a pool which contains different numbers of characters in proportions specified by a 256-dimensional distribution vector $\theta \in \mathbb{B}^{1\times 256}$ ($\mathbb{B}$ denotes the set of integers between 0 and 255.) Θ, in this case, is the full transition matrix that contains the set of all 256 different discrete character-to-character transition probabilities.

Figure 10 shows as we increase the size of the blending string and allow more artificially crafted data to be present, the L2 norm distance between the transition matrix of the generated samples to the target transition matrix decreases (note the 10^{-3} scale). This method can

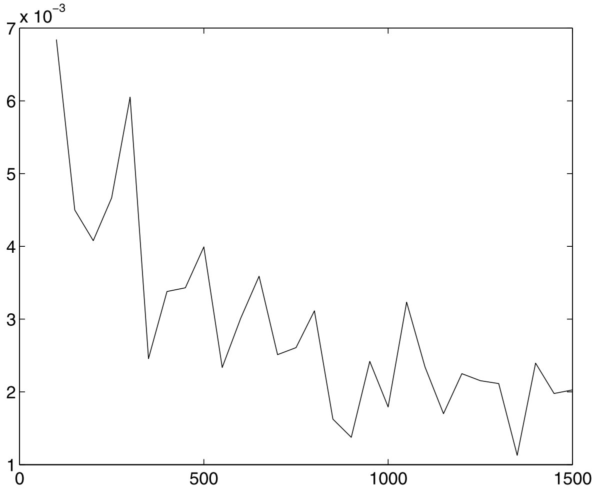


Fig. 10 2-gram blending attack. x-axis shows the size of the blending string, y-axis shows L2-norm distance between the transition matrix of the blending string and the target transition matrix. Compare with Fig. 8(b)

be viewed as an approximation algorithm for an *exact* 2-gram blend, where the transition proportions in the generated samples are identical to the target distribution. Generating such an exact attack was recently shown to be NP-complete by Fogla et al. (2006) which analyzes the complexity of 2-gram blending from an algorithmic complexity motivation. Note that our algorithm is, in fact, *linear* in both runtime and space requirements relative to the gram size.

6.2 N-gram blend

The 2-gram blend is generalizable to n-gram models if we define the random walk over an $(n-1)$-dimensional sample space. The probability of observing the nth gram would then be conditioned on the probability of observing grams $n-1, n-2, \ldots, 1$. This n-gram model subsumes models for all models for grams sizes smaller than n, therefore, there is no need for more complicated factorizations to account for sub-gram models, although it is worth noting that using larger gram sizes carry the potential for dramatic over-fitting to the training data, as is the standard caveat when increasing model complexity. An exact sampling for n-grams, under our previously described methodology, would require *exponential* storage space and runtime since the transition matrix will be of size $(n-1)^{256} \times 256$. This is due to the strong coupling of the $n-1$ individual variables $p(x_n|x_{n-1}, x_{n-2}, \ldots, x_1)$. For smaller gram sizes, the blending attack can be brute forced, in which case the only change to the previous algorithm is the input of a larger transition matrix. However a linearly-complex algorithm is possible if we relax the model to assume pair-wise independence between the individual variables i.e. $x_i \perp x_j | x_n$, where $i, j < n$, and only look at products of pair-wise conditional probabilities. This factorization yields the following for a 5-gram model: $p_5(x_5|x_4, \ldots, x_1) = p(x_5|x_4)p(x_5|x_3)p(x_5|x_2)p(x_5|x_1)$. More generally, for any gram size G we have:

$$p_G(x_n|x_{n-1}, \ldots, x_{n-G+1}) = \prod_{i=1}^{G-1} p(x_n|x_{n-i}) \tag{7}$$

Given this model, only $n-1$ transition matrices (each of dimensionality 256 × 256) needs to be kept in memory. The factorization also yields a computational advantage—as

```
function $n$-blend($\mathcal{D}$, $L$)
    Estimate $n-1$ transition matrices from training data $\mathcal{D}$
    $x_{1..n-1} \sim rand(1..256)^{1\times n-1}$
    for i $\leftarrow n$ to $L$
        $x_i \sim p(x_i|x_{i-1})p(x_i|x_{i-2})\cdots p(x_n|x_{i-n-1})$
    end
return $x$
```

Fig. 11 Function 2-blend($\mathcal{D}$, L) returns an L-length string whose transition probabilities mirror that of training samples D

```
k2105/web_exp/jumble.plindars/grapeoplendamenders/
calenaleoraphtheor/persandateorl/caiendors/peory/
pen/arsondets
```

Fig. 12 Sample blend string generated by a model focused at mimicking URL structure. Note the word transitions (overlaps)

we compute the probability of subsequent grams, only the conditional probability related to the previous gram needs to be calculated, the other $n-2$ probabilities do not change and can be kept in memory. Note that the model remains convex, thus the maximum likelihood solution requires only estimation of $n-1$ individual transition matrices, each in the manner as mentioned previously. Similar to the previous example, the algorithm for n-gram blending is shown in Fig. 11.

The sampling algorithm can still use the bag-of-grams concept as before, except now one needs to take into account $n-1$ distinct $\theta_i \in \Theta$ models and factor in the product probabilities accordingly. Figure 12 shows an example of a typical blend-string generated by our engine, which was trained to mimic URL tokens. As we can observe, the model yields many recognizable n-gram tokens which transition smoothly into each other, such as how "grape" and "people" are both present in "grapeople". Our sampling algorithm minimizes the number of undesired tokens, such as those which would be generated when words were simply concatenated, i.e. a concatenation to form "grapepeople" would yield the undesired "pepe" token.

6.3 The role of blending engines

Blending methods such as those described in this paper are aimed primarily at exploiting IDS sensors which use feature-counting-based models that measure the existence of certain artifacts (in specific proportions) in the data such as the sensors described in Wang et al. (2005, 2006a), Kruegel and Vigna (2003). The adversarial environment is similar to spam detection. However, it is important to note the distinction that network-situated anomaly detectors typically see traffic at significantly higher rates (often orders of magnitude higher). This must be taken into consideration when setting thresholds, i.e. even a 0.01% false positive rate on a gigabit connection could mean the user is bombarded with thousands of false positive alerts per minute, depending on the type of network. Statistical IDS sensors are therefore often prevented from applying tight detection thresholds. The role of blending is to train models which can generate n-grams for whatever portion of the data-request (be it packet level or higher) that the adversary selects to mimic.

In a blending attack against a port 80 web sensor, for example, the adversary might generate pseudo-random URL-like tokens and place them as arguments within portions of an

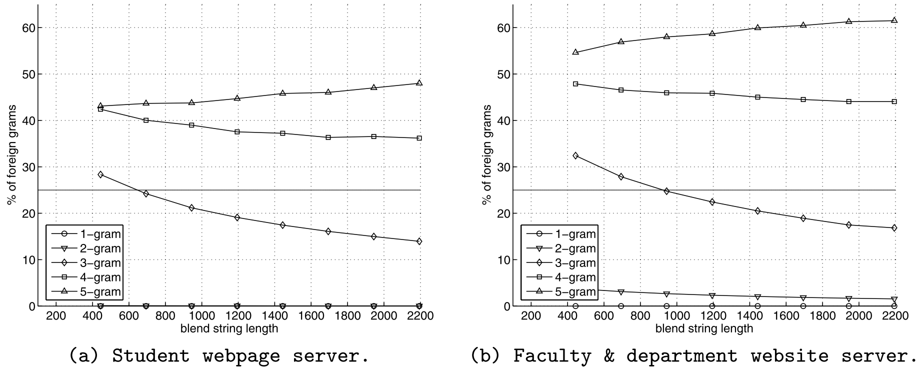
HTTP request which do not affect the exploit itself. In the case of an attack that uses HTTP GET requests such as IISWebDav (SANS 2004c), IISMedia (SANS 2004a) or the notorious CodeRed (CERT 2001) and Santy (SANS 2004b) worms, blend-strings can be placed in the message body or in the adjacent protocol fields. Common server applications such as Apache will not recognize the extra argument strings and will ignore them by default, thus allowing the exploit to execute unaffected. Network layer IDS sensors which are protocol-aware might be able to discriminate such inputs but in general, NIDS do not always have the protocol specifications for the applications available on the network.

6.4 N-gram blending evaluation

We evaluated blending attacks against 1-gram models in the previous section (blending against PayL). We now evaluate blending attacks against n-gram models using these new algorithms. We tested these methods against `Anagram` (Wang et al. 2006a), an n-gram based detection algorithm from our lab. The Anagram sensor, first introduced by Wang et al. (2006b) is a hash-dictionary based *anomaly detector*, that is, it only trains on normal traffic. Since exact modeling of n-grams is an ill-posed problem due to the exponential growth in the sample space as a function of n, the sparsity of training samples prevents exact parameter estimation. Instead, anagram chooses to simply store every instance of the *positive* training data into a dictionary by storing the hashes of the *distinct* n-grams observed in the data. More specifically, the algorithm presented by Wang et al. makes use of Bloom filters—binary arrays where elements are indexed by the hash values of n-grams. Observed n-grams have corresponding Bloom filter elements set to 1; all other elements of the filter are set to 0. Given a test sample, Anagram compares hash collisions of *distinct* n-grams extracted from the data with existing Bloom filters for "normal traffic", if the amount of "foreign" n-grams (i.e. those that do not exist in the Bloom filter) exceeds some proportional threshold, an alert is thrown.

For our evaluation, we craft an attack by first using binary-to-text encryption to hide the shellcode (as described in earlier sections). This done to reduce the footprint of the shellcode, we then add a blend-string generated by our engine and craft the attack by putting the string into different protocol fields as mentioned in the previous subsection. The entire HTTP request is then given to Anagram to test for normality. The dataset we used consists of packets we captured on our university network over the course of a day. Two unique hosts are considered: one which serves faculty and department web-pages and the other serves student pages. The former server hosts several hundred distinct pages as well as a technical papers database, on-site conference pages, and other web material one might expect to find on a computer science department's website. The latter serves only student pages.

We used "tcpdump" (Tcpdump 2009) to capture packets corresponding to *normal requests* that are sent to our target. These include requests for certain websites, file-downloads, etc. For an actual attacker, recording normal transactions with a target webpage would suffice to as a way to collect normal traffic. This data was captured over the course of a single day and contained roughly 5500 requests for the student server, and 3700 for the department server. It took roughly one second to generate a 2200-character-length blend-string on a 1.8-GHz processor using MATLAB. Note that simply replicating a single good token will not work since the distribution of the n-grams would be skewed. Using stochastic random draws to find these unique tokens is the a more reliable solution. Figures 13(a) and (b) show the normality scores that Anagram (Wang et al. 2006a) generates as we increase the size of the blend-string. Depending on the type of traffic modeled, Anagram can use thresholds up to 25%. As we can see from the figure, it is possible to blend an attack such that it passes



(a) Student webpage server.

(b) Faculty & department website server.

Fig. 13 URL token blending attack for various gram-sizes. For models of up to 3-grams, the shellcode eventually blended underneath the threshold. At 5-grams the models begin to diverge instead

under this threshold, at least for smaller gram sizes. As we increase the size, the models begin to under-fit, larger grams entails additional complexity, and the models yield non-useful grams at the 5-gram level. This leads to divergence rather than blending. For larger n, the ill-posed nature of density estimation remains intractable even in this factorized model.

6.5 Probabilistic dictionary attack

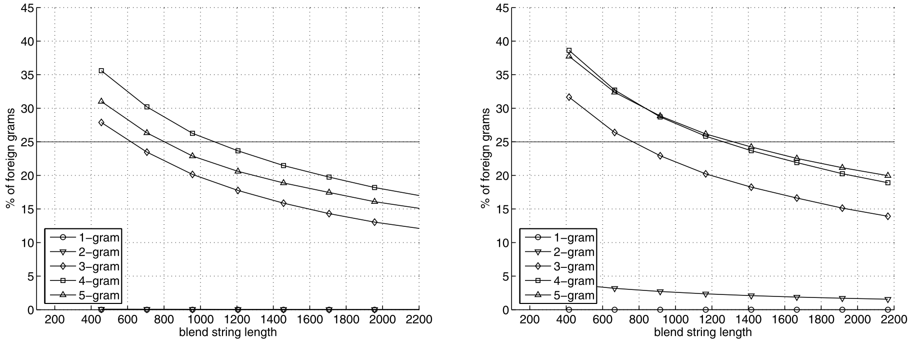
This section shows how we can modify the use of the model to yield an alternative blending strategy which is more effective and suffers less from under-fitting problems. Instead of sampling from the probability model directly, we can use the models to build a dictionary of blending tokens that can then be concatenated into a blend-string. The model estimation process is the same as before, except the models are now used on the training data to find the existing tokens which are most likely to be present in the n-gram model for normal traffic on the target site. Normalization is achieved by finding the arithmetic mean in log-space:

$$\begin{aligned}\log p_G(x_1,\dots,x_N) &= \log\left(\prod_{i=G}^{N} p_G(x_i|x_{i-1},\dots,x_{i-G+1})\right)\\ &= \log\left(\prod_{i=G}^{N}\prod_{j=i-G+1}^{i} p_G(x_i|x_j,\dots,x_{i-1})\right)\\ &= \sum_{i=G}^{N}\sum_{j=i-G+1}^{i} \log p_G(x_i|x_j,\dots,x_{i-1})\end{aligned}$$

Dividing by the length of the string N recovers the likelihood function $p_G^*(x_1,\dots,x_N)$ for a string $x_1,\dots,x_N$ when using gram size G. This yields the new likelihood function:

$$p_G^*(x_1,\dots,x_N) = \frac{1}{N}\sum_{i=G}^{N}\sum_{j=i-G+1}^{i} \log p_G(x_i|x_j,\dots,x_{i-1}) \tag{8}$$

This normalizes the likelihood so that a longer string does not lower the overall likelihood, and corresponds to finding the geometric mean of the likelihood of subsequent characters



(a) Student webpage server.

(b) Faculty & department website server.

Fig. 14 Dictionary-based token-blending attack. The under-fitting problems are gone, now efficient blending at 5-gram level is possible. Blending attacks against two distinct servers (which see different types of HTTP resource requests) exhibit the same performance, demonstrating the generalization ability of this technique. No adjustments were made to the learning algorithm between the two different attacks

within a string for n-gram probability models, with gram size G. Equation (8) is the final equation used to score each input string when building the dictionary. Additionally, `argmax` sampling is used instead of randomized sampling. The distinction is that, in this case, we do not draw from a distribution, rather we select the character that yields the highest likelihood-increase at each position of the string. A hash is also used to prevent the selection of identical tokens given the uniqueness constraint. While we evaluated our algorithm on Anagram, our method is aimed at defeating any n-gram counting features based detectors by using judicious selection of known good tokens.

Figure 14 shows the effectiveness of the dictionary based blending tactic. As the figure shows, higher order blending up to 5-grams is possible given a large enough blend-string. Also note that smaller blend-strings are ineffective, as can be expected since the ASCII encrypted shellcode we used in this experiment is around 800 bytes long and represents the majority of the foreign grams detected by the Anagram model. The larger the blend-string the more diluted the anomaly score will be for the overall attack. To populate the dictionary with several thousand tokens, it takes on average 45 seconds, per dataset, on a 1.8-GHz processor running MATLAB code. Concatenating the resulting tokens into an actual attack takes negligible time. An attacker would need to populate only one dictionary per target. From Fig. 14, we can also see that in order to launch a successful blending attack, the attacker needs to use at least two packets, where the exploit is split nearly evenly between them with two separate blend-strings. This is because network layer, packet-level sensors such as Anagram examines each packet separately, and the Maximum Transmission Unit (MTU) limit for a network packet is at most 1500-bytes. One packet is not enough to hold both the exploit and a sufficiently large blend-string. There already exist certain methods which can deal with this challenge, among them is a `fragmentation attack` which we briefly describe in the next section.

6.6 Overlapping fragmentation attack

This section is meant to show that various techniques exist to make blending attacks feasible. Among the most well known is overlapping reassembly, which we describe here. Many network layer sensors work at packet-level, and when left in their naive setting could fall vulnerable to the "overlapping reassembly" packet fragmentation attack, a method to bypass network layer IDS sensors by distributing an attack into several interlocking pieces. For technical details we refer the reader to an earlier "Security Focus" tutorial (Siddharth 2005). To bypass the size limit issue mentioned in the previous section, overlapping reassembly can be used to craft an attack such that smaller portions of the exploit are exposed sequentially. This would work as follows: the first arriving packet would expose a small portion of an exploit, such as the first 10 bytes, followed by a large blend-string to make the entire packet appear normal. The second packet would contain the next sequential substring of the exploit, followed by another large blend-string, but this second packet would also have a fragmentation marker set to indicate that the payload it carries consists of bytes starting at the 11th position of the datagram. This will cause the TCP-layer to reassemble the two packets in such a way that the first packet is largely overwritten by the second, erasing the previous blend-string and leaving 20 bytes of the exploit exposed, followed by the remaining portion of the second blend-string. This process would be repeated until the exploit has been fully reassembled.

7 Exploring *N*-space

So far, this paper has focused primarily on empirical analysis of existing engines. This section discusses work on exploring the potential span of polymorphic shellcode within $\mathbb{B}^d$. More specifically, given n bytes, 2^{8n} possible strings exist and an interesting problem is to consider what portion of this space consists of code which "behaves" like polymorphic shellcode. The answer, or an estimate thereof, would provide insights into the future of shellcode detection. Since a complete search of this space is intractable for large n, our search methodology focuses on byte strings of length 10. From the structure of the decoder, it is known that two main components must exist: (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 2 shows that 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 critical operations for examining decoding behavior, and focus only on this small restricted portion. A 10-byte constraint reduces the search space to 2^{80} strings, which is an intractable problem if each string is to be evaluated. Instead, we make use of Genetic Algorithms (Russell and Norvig 2002) to perform the search in a directed manner by choosing to explore areas where existing polymorphic code are known to reside.

7.1 Decoder detector

To construct the GA search, a function is constructed that accepts a string as input and determines whether that string represents x86 code that exhibits polymorphic behavior. A "decoder detector" was implemented as a process monitoring tool within the "Valgrind" (Nethercote and Seward 2003) emulation environment. Valgrind's binary supervision enables us to add instrumentation to a process without modifying its source code or altering the semantics of the process operations. Most importantly, Valgrind provides support for examining memory accesses, thus allowing us to track the 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 (two hundred bytes) of itself. The GA-search framework compiles and executes this simulated buffer-overflow in the instrumented environment and checks for polymorphic behavior. In particular it looks for the following actions: *self-write*, when writing to a memory location within two hundred bytes of the executing instruction, *self-modify* when reading from a memory location within two hundred bytes of the instruction, and within the next four instructions, performs a write to the same location. This latter definition captures the machine level behavior of in-place modification operations, such as `xor, add, sub`, etc.

7.2 Genetic algorithms

Genetic Algorithms is a well-known 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 solve a closed form equation to directly optimize a solution. Instead, various solutions are represented in coded string form and evaluated. Users of a GA search define a function 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.

The fitness function used for our GA search framework is the decoder detector described above. We assign a score of 1 to each self-write operation and a score of 3 to each self-modify operation. The higher score for the latter operation reflects our interest in identifying instruction sequences that represent the `xor, add, sub, ...` 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.

Since the search is not looking for the "best" decoder, a lower limit for polymorphic-behavior can be set, and any string that passes this threshold can be admitted into the population. This dynamic threshold for minimum acceptable polymorphic behavior is set to be five percent of the *average* polymorphic score of the previously found sequences; we bootstrapped with an arbitrary constant of 6. The threshold was used in order to ignore strings which performed a small (one or two) number of modifications, since we wanted to capture strings that exhibited a significant amount of polymorphic behavior. Significant scores would indicate the presence of some form of a loop construct.[2] All strings which met the polymorphic criteria are stored in an associative array, to preserve uniqueness and for certain speed advantages. 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. The algorithms accomplish this search by taking existing samples and permuting them slightly using evolution strategies, then re-testing. Therefore, this form of local search needs good starting positions to achieve reasonable results. We seeded our search engine with two decoder strings extracted from *ShellForge* (Biondi 2006) and roughly 45000 strings from Metasploit (Metasploit Developement Team 2006) in order to obtain a good distribution of starting positions. We implemented a standard GA-search framework using some common evolution strategies, which we list below.

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] We used a four second runtime limit in our Valgrind decoder detector tool as we periodically find strings which run into infinite self-modification loops.

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 helps prevent the search from getting stuck on local max. It is used mainly to introduce 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.

This method is similar to work done by Polychronakis et al. (2006). While they had implemented their tool as a detector, dynamically filtering network content through the detector to search for the presence of decryption engines, our detector is used off-line to pre-compute a set of byte strings that perform self-modification.

7.3 GA-search results

This evaluation aims to assess the hypothesis that the class of self-modifying code *spans* $\mathbb{B}^d$ 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 currently shows no signs of slowing down. The results that we derive show that the class of n-byte self-modifying code not only spans $\mathbb{B}^d$ but saturates it as well. 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}
Std.: {72,71,86,78,80,84,86,82,75,76}

The mean is observed to be near the center of $\mathbb{B}^d$, as we would expect from Central limit theorem and the high variance along each dimension shows that the high degree of scatter within these samples.

For each sequence in our sample pool, a 1-byte distribution was computed to find the average for all sequences. This average is further normalized by dividing by the variance along each dimension, as was done in the previous section. Figure 15(a) shows the average 1-byte distribution. The sample pool contains no distinguishable distribution, and is closer to white noise—with the exception of the `{x00}` and `{xFF}` characters, which are likely to be padding artifacts. Moving on to 3-space, Fig. 15(b) shows the 3-gram 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. It follows that 2-space is saturated as well, since it is a subspace of 3-space. This result can be expected since arbitrary polymorphic code is less

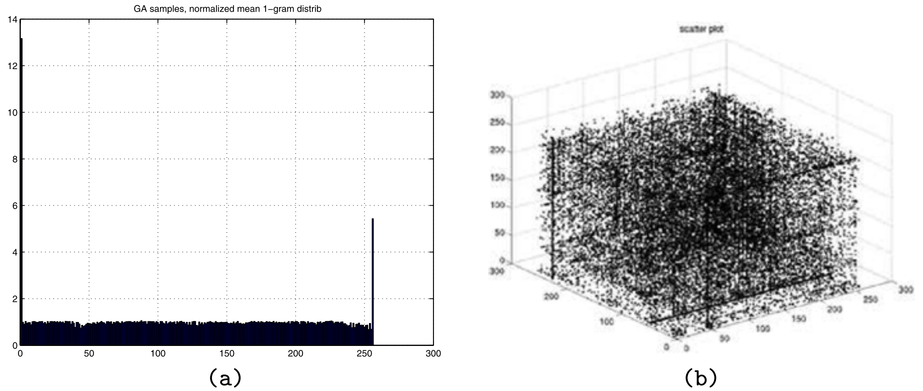


Fig. 15 Results. (**a**) 1-gram distribution—note the uniform byte distribution. (**b**) 3-gram scatter plot—each dot represents a 3-gram, note the 3-space saturation

constrained than the decoders that were examined in the previous sections. Nevertheless, our results show that there is a significant degree of variance in the range of x86 code that perform operations associated with self-decryption routines.

8 Conclusions

Our results show the difficulties we face when designing detectors given modern obfuscation techniques. We proposed metrics which can measure the strengths of these engines and have shown how targeted higher order blending attacks are possible, in an effort to forecast future threats. We explained how signature-based modeling can work in some cases, and confirmed that the long-term viability of such approaches is consistent with the long held, intuitive, belief that polymorphism will eventually defeat these methodologies.

We argue that while signature-based methods may work in the short term, evidence suggests that they cannot generalize to protect against future attacks. Similarly, feature-counting based statistical models also suffer from equally troubling deficiencies. In our discussion on blending, we assumed that we had access to labeled samples—in practice this is a luxury that is not always available, at least not for the defender. In contrast, the attacker can generate perfectly labeled data by querying the target host for positive samples, as we did in our experiments to find the normal traffic pattern.

Given these properties, any generative approach to modeling malicious code would appear to be threatened. At the same time, we must always keep in mind the stringent false-positive requirements that constrain all anomaly-detection sensors, and that we must expect the attackers to exploit this requirement. The strategy of modeling malicious content forces us to try to keep up with the adversary, and if polymorphic code is really distributed in the manner that our study indicates, then we might never achieve a generalizable defense by continuously refining a single representation of what the attackers may do. A positive-security model, one that focus on normal content, might be a more reliable solution.

References

Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005). Control-flow integrity: principles, implementations, and applications. In *Proceedings of the ACM conference on computer and communications security (CCS)*.

AlephOne (2001). Smashing the stack for fun and profit. *Phrack, 7*(49-14).

Anagnostakis, K. G., Sidiroglou, S., Akritidis, P., Xinidis, K., Markatos, E., & Keromytis, A. D. (2005). Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th USENIX security symposium*.

Bania, P. (2009). Tapion polymorphic engine. http://pb.specialised.info/all/tapion/.

Baratloo, A., Singh, N., & Tsai, T. (2000). Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX annual technical conference*.

Barrantes, E. G., Ackley, D. H., Forrest, S., Palmer, T. S., Stefanovic, D., & Zovi, D. D. (2003). Randomized instruction set emulation to distrupt binary code injection attacks. In *Proceedings of the 10th ACM conference on computer and communications security (CCS)*.

Bhatkar, S., DuVarney, D. C., & Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX security symposium* (pp. 105–120).

Biondi, P. (2006). Shellforge project. http://www.secdev.org/projects/shellforge/.

Brumley, D., Newsome, J., Song, D., Wang, H., & Jha, S. (2006). Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE symposium on security and privacy*.

CERT (2001). Code red I/II worm. http://www.cert.org/advisories/CA-2001-19.html.

Chinchani, R., & Berg, E. V. D. (2005). A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th international symposium on recent advances in intrusion detection (RAID)* (pp. 284–304).

Costa, M., Crowcroft, J., Castro, M., & Rowstron, A. (2005). Vigilante: end-to-end containment of Internet worms. In *Proceedings of the symposium on systems and operating systems principles (SOSP)*.

Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., & Zhang, Q. (1998). Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX security symposium*.

Crandall, J. R., Su, Z., Wu, S. F., & Chong, F. T. (2005a). On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM conference on computer and communications security (CCS)*.

Crandall, J. R., Wu, S. F., & Chong, F. T. (2005b). Experiences using minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In *Detection of intrusions and malware and vulnerability assessment (DIMVA)*.

Cui, W., Peinado, M., Wang, H. J., & Locasto, M. E. (2007). ShieldGen: automated data patch generation for unknown vulnerabilities with informed probing. In *Proceedings of the IEEE symposium on security and privacy*.

Detristan, T., Ulenspiegel, T., Malcom, Y., & von Underduk, M. S. (2003). Polymorphic shellcode engine using spectrum analysis. *Phrack, 11*(61-9).

Etoh, J. (2000). GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp.

Fogla, P., & Lee, W. (2006). Evading network anomaly detection systems: formal reasoning and practical techniques. In *Proceedings of the 13th ACM conference on computer and communications security (CCS)* (pp. 59–68). http://doi.acm.org/10.1145/1180405.1180414.

Fogla, P., Sharif, M., Perdisci, R., Kolesnikov, O., & Lee, W. (2006). Polymorphic blending attacks. In *Proceedings of the USENIX security conference*.

Foster, J. C., Osipov, V., Bhalla, N., & Heinen, N. (2005). *Buffer overflow attacks: detect, exploit, prevent*. Syngress.

Joshi, A., King, S. T., Dunlap, G. W., & Chen, P. M. (2005). Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the symposium on systems and operating systems principles (SOSP)*.

K2 (2003). ADMmutate documentation. http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz.

Kc, G. S., Keromytis, A. D., & Prevelakis, V. (2003). Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on computer and communications security (CCS)* (pp. 272–280).

Kim, H. A., & Karp, B. (2004). Autograph: toward automated, distributed worm signature detection. In *Proceedings of the USENIX security conference*.

Kiriansky, V., Bruening, D., & Amarasinghe, S. (2002). Secure execution via program shepherding. In *Proceedings of the 11th USENIX security symposium*.

Kolesnikov, A., & Lee, W. (2006). Advanced polymorphic worms: evading IDS by blending in with normal traffic. In *Proceedings of the USENIX security conference*.

Kruegel, C., & Vigna, G. (2003). Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on computer and communications security (CCS)*.

Krugel, C., Kirda, E., Mutz, D., Robertson, W., & Vigna, G. (2005). Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th international symposium on recent advances in intrusion detection (RAID)* (pp. 207–226).

Liang, Z., & Sekar, R. (2005). 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)*.

Locasto, M. E., Wang, K., Keromytis, A. D., & Stolfo, S. J. (2005). FLIPS: hybrid adaptive intrusion prevention. In *Proceedings of the 8th international symposium on recent advances in intrusion detection (RAID)* (pp. 82–101).

Metasploit Development Team (2006). Metasploit project. http://www.metasploit.com.

Nethercote, N., & Seward, J. (2003). Valgrind: a program supervision framework. In *Electronic notes in theoretical computer science* (Vol. 89).

Newsome, J., & Song, D. (2005). 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)*.

Newsome, J., Karp, B., & Song, D. (2005). Polygraph: automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE symposium on security and privacy*.

Obscou (2003). Building IA32 ‘Unicode-Proof’ shellcodes. *Phrack, 11*(61-11).

Panda Labs (2007). MPack uncovered. http://pandalabs.pandasecurity.com/.

Polychronakis, M., Anagnostakis, K. G., & Markatos, E. P. (2006). Network-level polymorhpic shellcode detection using emulation. In *Detection of intrusions and malware and vulnerability assessment (DIMVA)*.

Rix (2001). Writing IA-32 alphanumeric shellcodes. *Phrack, 11*(57-15).

Russell, S., & Norvig, P. (2002). *Artificial intelligence: a modern approach*. New York: Prentice Hall.

SANS (2004a). IISMedia Exploit. http://www.sans.org/newsletters/cva/vol2_21.php.

SANS (2004b). Santy worm. http://isc.sans.org/diary.html?date=2004-12-21.

SANS (2004c). Webdav exploit. http://www.sans.org/resources/malwarefaq/webdav-exploit.php.

Siddharth, S. (2005). Evading NIDS. http://www.securityfocus.com/infocus/1852.

Sidiroglou, S., Giovanidis, G., & Keromytis, A. D. (2005). A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th information security conference (ISC)* (pp. 1–15).

Singh, S., Estan, C., Varghese, G., & Savage, S. (2004). Automated worm fingerprinting. In *Proceedings of symposium on operating systems design and implementation (OSDI)*.

Snort Development Team (2009). Snort project. http://www.snort.org/.

Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., & Stolfo, S. J. (2007). On the infeasibility of modeling polymorphic shellcode. In *Proceedings of the ACM conference on computer and communications security (CCS)*.

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

Tcpdump (2009). http://www.tcpdump.org.

Toth, T., & Kruegel, C. (2002). Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th international symposium on recent advances in intrusion detection (RAID)* (pp. 274–291).

Wang, K., & Stolfo, S. J. (2004). Anomalous payload-based network intrusion detection. In *Proceedings of the 7th international symposium on recent advances in intrusion detection (RAID)* (pp. 203–222).

Wang, H. J., Guo, C., Simon, D. R., & Zugenmaier, A. (2004). Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM conference* (pp. 193–204).

Wang, K., Cretu, G., & Stolfo, S. J. (2005). Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th international symposium on recent advances in intrusion detection (RAID)* (pp. 227–246).

Wang, K., Parekh, J. J., & Stolfo, S. J. (2006a). Anagram: a content anomaly detector resistant to mimicry attack. In *Proceedings of the 9th international symposium on recent advances in intrusion detection (RAID)*.

Wang, X., Pan, C. C., Liu, P., & Zhu, S. (2006b). SigFree: a signature-free buffer overflow attack blocker. In *Proceedings of the 15th USENIX security symposium* (pp. 225–240).

Yegneswaran, V., Giffin, J. T., Barford, P., & Jha, S. (2005). An architecture for generating semantics-aware signatures. In *Proceedings of the 14th USENIX security symposium*.