Syntax vs. Semantics: Competing Approaches to Dynamic Network Intrusion Detection

Walter Scheirer*
Department of Computer Science,
University of Colorado at Colorado Springs, CO 80918, USA
E-mail: wjs3@vast.uccs.edu
*Corresponding author

Mooi Choo Chuah
Department of Computer Science and Engineering,
Lehigh University, PA 18015, USA
E-mail: chuah@cse.lehigh.edu

Abstract: Malicious network traffic, including widespread worm activity, is a growing threat to Internet-connected networks and hosts. In this paper, we consider two competing approaches to dynamic network intrusion detection: syntax based and semantics based approaches. For the syntax driven approach, we propose two sliding window based schemes to generate potential worm signatures automatically. Since syntax based approaches cannot cope well with sophisticated polymorphic and metamorphic worms, the semantics-based approach is a better alternative. Our contribution in this work is threefold: (a) our syntax-based scheme that uses variable-length partition with multiple breakmarks can detect many polymorphic worms, (b) we believe our semantic-based prototype is the first NIDS that provides semantics-aware capability and our system is more efficient than what is reported by Christodorescu et al (2005), (c) our designed templates can capture polymorphic shellcodes with added sequences of stack and mathematic operations.

Keywords: network security; computer security; intrusion detection; semantics.

Biographical Notes: Mooi Choo Chuah received the first class honors Bachelor's degree in electrical engineering from University of Malaya, Malaysia and the Master's and Ph.D. degrees from the University of California, San Diego. She is an associate professor of Computer Science and Engineering Department at Lehigh University. Her current research interests include Internet and wireless security, system and protocol design for Disruption Tolerant Networks, adhoc and sensor network design.

Walter Scheirer received his Bachelor's and Master's degrees in computer science from Lehigh University. His primary research interest is computer security, however, he has also participated in research in the fields of bioinformatics, networking, and computer vision. Walter is currently pursuing a Ph.D. in computer science at the University of Colorado at Colorado Springs.

1 INTRODUCTION

In recent years, computer intrusion has been on the rise. The popularity of the Internet and the widespread use of homogeneous software provide an ideal climate for infectious programs. The cost of viruses and worms in 2002 was estimated to be 45 billion dollars (Reuters 2003). In 2003, this number jumped to 55 billion dollars (Reuters 2003). Much money has to be spent on researching techniques that can fend off intrusion attempts such that computer systems can operate effectively. A popular technology called the Intrusion Detection System (IDS) has emerged to identify and block intrusion attempts. Popular network IDS (NIDS) systems such as Snort (Roesch 1999) and Bro (Paxson 1998) utilize a signature-based approach to detect malicious network traffic. In these systems, static signatures of known attacks are used to identify attack packets. A major drawback of this approach is that unknown attacks cannot be detected – the ones, which conceivably will cause the most damage.

Typically, new attacks are detected in an ad hoc fashion through a combination of intrusion detection systems

alerting potential attacks, and skilled security personnel manually analyzing traffic to generate attack characterization. Such an approach is clearly not sufficient since it may take hours to generate a new worm signature. In recent studies by Moore et al. (2004), the authors suggest that if the attack traffic is indicative of a worm outbreak, effective containment may require a reaction time of well under sixty seconds. Thus, new techniques that can help to identify threats from unseen worms or exploit packets need to be devised.

In this paper, we discuss two competing approaches for dynamic network intrusion detection, namely the syntax-based and semantic-based approaches. Sliding window schemes as presented by Kim et al. (2004), Singh et al. (2004), and Newsome et al. (2005) are syntax-based approaches that partition suspicious worm payloads to generate worm signatures. They are based on the premise that some portion of the malicious codes will inevitably be invariant, despite attempts to obscure their true natures for detection avoidance. In this paper, we describe two sliding window schemes. One scheme uses fixed-length partition while the other uses variable-length partition. To minimize the number of signatures that are retained, we use similar threshold-based unique source/destination IPs approach described in Kim et al. (2004). We also use clustering algorithm to include similar signatures so that the false negative rate can be reduced. Via extensive traffic analysis, we demonstrate that one of the schemes called the variable length partition with multiple breakmarks (VPMB) scheme is effective in detecting several polymorphic worms.

While the syntax-based approaches can catch many polymorphic worms, they will still miss certain categories of polymorphic worms. A worm author may craft a worm that changes substantially its payload on every successive spreading attempt, and thus evades matching by any single substring signature that does not also occur in innocuous traffic. This motivates us to propose another NIDS with semantics-aware capability. The prototype system that we have built can potentially identify threats from some unknown malicious network traffic. This work is an extension of the approach presented in Christodorescu et al. (2004). The semantics-aware malware detection algorithm of Christodorescu et al (2005) is an extremely powerful tool for program profiling. Based on the observation that certain malicious behaviors appear in all variants of a certain kind of malware, the authors propose using template-based matching to detect malware. Their approach looks for a match of program behaviors rather than program syntax matching. In this manner, *polymorphic* and *metamorphic* code instances can be identified right along with their static counterparts. However, in Christodorescu et al. (2004), the authors only perform experiments on a non-networked host with standalone virus samples as well as evaluating their templates against a set of benign programs. As most threats to end-systems now emanate from the Internet, much in the form of self-propagating network code, network enabled detection is critical. Thus, in this paper, we report on a full-featured semantics-aware network intrusion detection system we have built. Our system can detect not only viruses, but remote exploits, including worm traffic. Through rigorous testing, we show that semantic detection is an extremely powerful tool for identifying static and polymorphic network exploits. Our system can perform more efficiently than the system presented in Christodorescu et al. (2004).

The rest of the paper is organized as follows: In Section 2, we describe some related work and discuss how several pieces of work motivate this research. In Section 3, we describe the motivation for three sliding window schemes that we propose for the syntax-based approach. In Section 4, we present our experiments and results we obtained using the syntax-based approach. In Section 5, we describe the semantic analysis of malicious code, and discuss how binary exploits work. In Section 6, we present the system architecture of the NIDS we have built and describe in detail how different stages of the system work. We describe our experiments and the results we obtained in Section 7. Finally, we summarize our findings and discuss some future work that we intend to explore in Section 8.

2 RELATED WORK

Much research has been devoted to intrusion detection in recent years. Two enormously popular open source tools, Snort (Roesch 1999) and Bro (Paxson 1998), have shown that static signature based IDSs can be quite successful in the face of known attacks. Combined with automatic monitoring and incident response, system administrators have a powerful tool against network attacks. In Locasto et al. (2004), the authors present the case for collaborative intrusion detection system where intrusion detection nodes cooperate to determine if a network attack is taking place and take corrective actions if it does. Others have sought to use statistical approaches to detect worm outbreaks. In Gao et al. (2004), the authors propose a method to identify a worm victim by observing if the number of scans per second it performs exceeds a certain threshold. The numbers of worm victims observed in successive windows are then compared to the numbers predicted using a typical worm spread model and if they match, then a worm outbreak is declared.

In Kim et al. (2004) and Singh et al. (2004), the authors show that byte-level analysis of packet payloads can yield useful signatures for worm detection. We referred to this as the syntax-based sliding window approach. Such sliding window schemes are based on the premise that some portion of malicious code will inevitably be invariant despite attempts at obscuring its true nature for detection avoidance. In this approach, the payload of a packet is partitioned into multiple chunks when a hosen breakmark is detected and Rabin fingerprints of these data chunks are generated. There is no comparison as to whether a fixed or variable partition works equally well. Thus, we proposed and evaluated two schemes that compare between fixed and variable-length partitions. In addition, we also extend the work in Kim et al. (2004) by using a set of breakmarks rather than a single

breakmark as described in Singh et al. (2004). Our multiple breakmark approach is similar to a recent paper Newsome et al. (2004). The authors in Newsome et al. (2004) advocate using disjoint data signatures.

At first glance, these syntax-based approaches looked promising, however, in practice, they generate far too many signatures, with a sometimes-undesirable accuracy rate. With Newsome et al. (2004), we begin to see a research trend towards using semantics knowledge for potential worm detection. Here, the authors observe that invariant byte positions may be disjoint (a result of advanced polymorphic techniques), but will be present nonetheless as they are integral to functionality. With Christodorescu et al. (2004) and Yegneswaran et al. (2006), the application of semantics is introduced. Non-binary attacks, such as URL based web server exploits, are analyzed and clustered in a data-mining scheme in Yegneswaran et al. (2006). In this work, we built upon the approach described in Christodorescu et al. (2004). Our contributions in the semantic aware related approach are three fold: (a) our prototype is a complete NIDS that provides semantic aware capability, (b) our implementation is more efficient than what is reported in Christodorescu et al. (2004), (c) our designed templates can capture polymorphic shellcodes with added sequences of stack and mathematic operations.

3 SYNTAX BASED SLIDING WINDOW SCHEMES

In this section, we describe two sliding window-based schemes that we proposed to automatically generate worm signatures. As in Kim et al. (2004) and Singh et al. (2004), we divide the payload of a packet into multiple chunks either using fixed-size window or variable-length sliding window until a chosen breakmark is detected. Rabin fingerprints (Broder 1992), (Rabin 1981) of these data chunks are then generated.

3.1 Fixed Partition Sliding Window Scheme (FPSW)

The FPSW scheme incorporates a fixed window size and a one-byte window sliding. The premise is simple – a series of fingerprints is generated as the window slides down the payload of a packet. Common signatures will be seen among different packet payloads if their contents are identical or similar. If the window size is small enough, common data portions can be isolated, despite the variation in the overall payloads. This is useful for the dynamic detection of new worm variants (e.g. W32.Blaster versus W32.Blaster.H). Figure 1 shows the operation of the sliding window using FPSW. The segments in $f0$, $f1$, and $f2$ will all be fingerprinted.

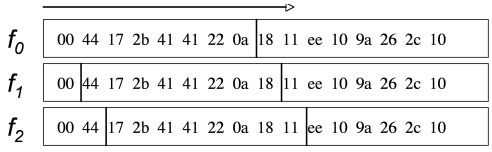

Figure 1 *Three instances of an 8-byte sliding window, beginning at '0' for the FPSW scheme*

One important decision related to this approach is choosing a proper window size. If the window size is very small (just a few bytes), the false positive rates will be higher. Certain short sequences are bound to appear in benign traffic as well as in malicious code. For example, "GET /" is typically at the beginning of a basic web request, but could also be followed by malicious exploit code. A 5-byte window would match both to the same fingerprint. In addition, the amount of signatures generated is always related to the window size. Smaller windows will produce more fingerprints, thus placing a higher burden on storing and searching.

3.2 Variable-length Partition with Multiple Breakmarks Scheme (VPMB)

In a polymorphic exploit, we often see a static region initiating a request (for example, a web based exploit may begin with a normal HTTP GET request), followed by a region of instructions that function as NOP equivalents (K2 1998). Thus, our VPMB scheme incorporates a one-byte sliding window approach until a series of breakmarks is reached. The breakmarks are chosen to be NOP-like instructions. In this method, using a look-ahead window of size w bytes, we search and see if all the bytes in this window can be found in a set of 76 breakmarks that have been identified. By using an adjustable look-ahead size to match these NOP-like instructions, we can reliably generate consistent fingerprints for the static regions preceding the NOP-like instructions. If they are, then we will generate a fingerprint using all the bytes that appear before this look-ahead window. After that, we begin a new search using a new window that begins 1 byte after the previously matched position. Figure 2 shows the operation of VPMB with three different window sizes: 5 bytes, 10 bytes, and 15 bytes. The same initial byte region is isolated in all three, producing one, consistent signature.

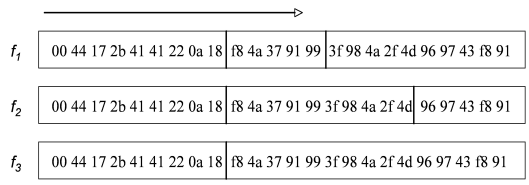

Figure 2 *Example of VPMB with three different window sizes*

An appropriate choice of look-ahead size is required to reduce the false positives. Similar to the FPSW's dilemma, choosing a smaller size will increase false positives. But how small is too small? Through testing (as will be shown later), it has been determined that a size of 20 bytes reduces false positives to a minimum. Tested cases with values greater than 20 did not result in further reductions of false positive but incurred additional processing cost. Insight as to why 20 is the "magic" look-ahead size is the following - the probability of finding a grouping of NOP-like instructions in benign traffic drops considerably as the

window size is increased. But in actual exploits, NOP regions tend to be larger than 20 bytes (so guessing an address back into the stack is a simpler process). Thus, 20 represents the point at which false positives drop to a minimum, and true positives don't require excess processing time.

4 EVALUATION OF SYNTAX BASED APPROACHES

To compare between the two schemes described in Section 3 and to evaluate the false positive/false negative rates of such a worm detection system, each of these algorithms has been implemented and applied to two one-hour traces that are extracted from a whole day's traffic trace that was kindly made available by Pang et al (2004). In this whole-day trace, traffic was observed from two /16 subnets (16K addresses) on two adjacent class B networks. Traffic from only one class C network contained within this trace is considered in this study. The total packet count for each 1-hour trace is 23,554 and 6,834 respectively. Each 1-hour trace is further divided into 5-minute intervals. Signature generation is performed on the traffic obtained in every 5-minute interval. To minimize the number of packets that the signature generation module needs to process, some simple filtering is performed on the trace: (i) only incoming packets destined for the target network are considered, (ii) only TCP packets with the PUSH flag set are taken for fingerprint generation. The methods, however, can be used for other attack packets (i.e. UDP-based attacks) as well. All data in each packet is considered for analysis (i.e., no static SNAPLEN is utilized). We have chosen to examine contiguous blocks of time over random time samples in order to reflect realistic attack detection. In practice, sustained scanning/propagation activity from a single host operating at a particular time interval is common. Thus, we wanted to get a sense of how this detection would operate in a real-time environment.

4.1 FPSW

As previously mentioned, because of the potential large number of fingerprints that can be generated using the FPSW scheme, it is desirable to find ways to reduce the number of signatures to be retained for future intrusion detection purposes. To accomplish this, the simple IP address dispersion algorithm proposed by Yegneswaran et al. (2005) has been implemented. This algorithm is well suited for detecting rapidly spreading worms, by observing the frequency of distinct source and destination IP addresses of packets carrying a particular fingerprint. If a single fingerprint is sent from at least n distinct source IPs, and is destined to at least n distinct destination IPs, then, it is retained. A further trimming of the fingerprint pool is performed for each test by discarding fingerprints generated from data chunks with a high prevalence of NULL bytes (i.e. {00, 00, 00, 00, 00, 00, 00, 80}). These fingerprints are far too general, and have little value for detecting malicious traffic.

To cope with polymorphic worms, we want to include additional signatures among those not retained but were created by payloads that bear similar resemblance to those with retained signatures. To do so, we use Levenshtein Edit Distance algorithm (Levenshtein 1965) to find similar fingerprints (amongst those that have not been retained) to the ones that have been retained for each 5 minute interval.

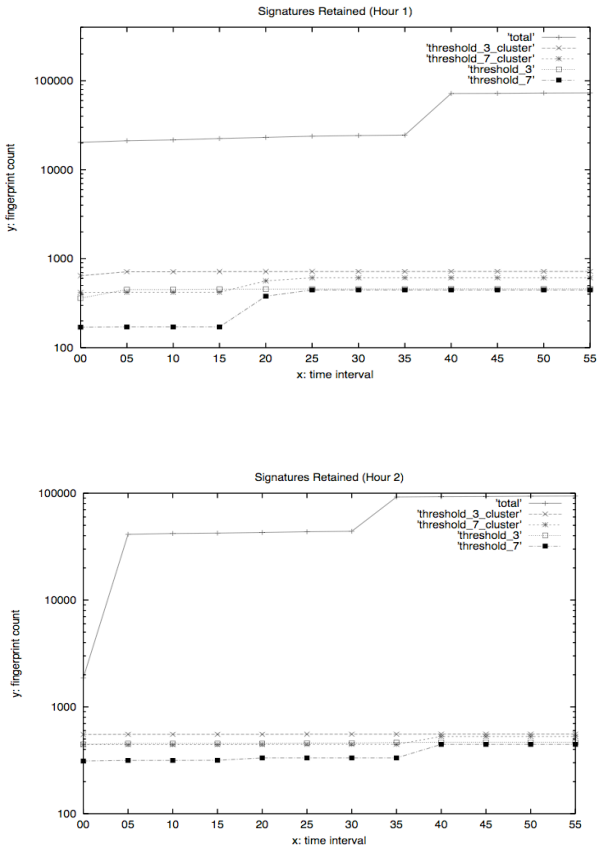

Figure 3 *Signature counts for 2 1-hour traces*

Figure 3 contains two graphs representing the application of classification and clustering against the FPSW scheme for the two one-hour capture intervals. Each graph interval (a five minute portion from the trace) increases cumulatively, with only new signatures being added to the total pool. Classification (using the algorithm described above) was performed at two different thresholds (T: $T = 3$ and $T = 5$. From the graphs, we see that thousands of signatures are discarded at $T = 3$, while only a slight *decrease* occurs from $T = 3$ to $T = 7$. When the clustering algorithm (also described above) is applied at both thresholds, we see a slight *increase* in the signature pool, as expected. Via clustering, a total of 274 signatures are added to the signature pool, with about 23 signatures added per 5-minute interval in hour 1, and 111 signatures with about 9 signatures added per 5-minute interval in hour 2. Thus, we

are able to tune the signature pool accordingly, with the maximum amount of useful signatures retained.

From what has been shown thus far, it is clear that FPSW has the potential to generate a large number of signatures. So, in response, we are interested in knowing the minimum amount of fingerprints needed to recognize malware variants. Thus, we mold the FPSW process into a multi-stage, computational pipeline, which will output a minimum set of fingerprints for suspected malicious traffic. These fingerprints will be stored for future detection use. Figure 4 shows the process of this pipeline. The first three stages, which have already been briefly described, pull traffic from the network, generate Rabin fingerprints, and classify the prints for retention. The fourth stage generates a minimum fingerprint set by finding the minimum intersection of fingerprints that covers all connection flows between set time intervals (in our testing, 5 minute windows in an hour) per destination port. After K windows (in our testing, K=12), we have M signatures retained. We check how many suspicious packets (those from flows which are retained due to the IP address dispersion rule) match each signature and retain the top W (W sets to 3). Fingerprints that are not specific enough (those with many '00' groupings) are immediately discarded.

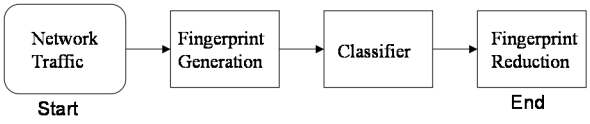

Figure 4 *FPSW Pipeline*

With fingerprint reduction, we can get an idea of the utility of the FPSW scheme. Tables 1 and 2 display the results of this process for the WebDAV Search exploit. The first column in each table indicates the real instances of the exploit in the sample data (determined by snort IDS analysis). Two different classification thresholds are shown (T = 3 and T = 7), with total unique prints increasing as time progresses. As expected, we retain fewer prints as the classification threshold is increased. Finally, the last column in each table represents the minimum number of fingerprints needed to identify all malicious flows per time interval.

Time	Hour 1 Real Instance	Hour 1 T = 3 Common to 00	Hour 1 T = 3 Actual	Hour 1 T = 7 Common to 00	Hour 1 T = 7 Actual	Prints Needed
*00	8	148	148	147	147	1
05	14	148	153	147	148	3
10	7	148	153	147	148	1
15	10	148	158	147	148	1
20	8	148	158	147	148	3
25	15	148	158	147	148	2
30	5	147	160	147	148	1
35	9	148	160	147	148	1
40	8	148	161	147	148	3
45	4	147	161	147	148	1
50	8	148	161	147	148	3
55	10	148	161	147	148	1

Table 1 *WevDAV Hour 1*

Between both hours, a final total of three unique prints are needed to identify all real instances of WebDAV present. In order to assess the false positive rate, we processed an entire month's worth of benign network traffic, which resulted in 498,020 total fingerprints. Out of this, 125 instances matched the 3 fingerprints isolated for WebDAV.

Time	Hour 2 Real Instance	Hour 2 T = 3 Common to 00	Hour 2 T = 3 Actual	Hour 2 T = 7 Common to 00	Hour 2 T = 7 Actual	Prints Needed
*00	20	298	298	165	165	23
05	20	297	303	165	169	23
10	18	297	303	165	169	23
15	14	23	303	165	169	23
20	20	297	303	165	186	23
25	16	192	303	165	186	23
30	19	298	303	165	186	23
35	18	292	309	165	186	23
40	23	297	309	165	298	23
45	19	297	309	165	298	43
50	22	284	309	165	298	23
55	19	298	309	165	298	23

Table 2 *WevDAV Hour 2*

We also conducted tests against known instances of the Welchia worm. Such testing produced interesting results regarding malware variants. In table 3, we see that the maximum number of fingerprints needed to identify all instances in each flow per interval is 23, with a complete total of 23 unique fingerprints needed for the whole hour trace. In table 4, this number increases to 43. Specifically, at time interval 45, we note a variation of Welchia that was not present in the first hour. Despite this final number of fingerprints being larger than WebDAV's 3, it is still a significant improvement over 298 prints retained when T = 7. As with WebDAV, the false positive rate is also rather low. Applied to the same month of benign traffic as WebDAV, we find 681 instances of false matches to Welchia's fingerprints.

Time	Hour 1 Real Instance	Hour 1 T = 3 Common to 00	Hour 1 T = 3 Actual	Hour 1 T = 7 Common to 00	Hour 1 T = 7 Actual	Prints Needed
*00	13	212	212	23	23	23
05	6	189	297	23	23	1
10	14	212	297	23	23	23
15	5	189	297	23	23	23
20	17	212	297	23	231	23
25	20	212	297	23	297	23
30	23	212	297	23	297	23
35	16	212	297	23	297	23
40	22	212	297	23	297	23
45	16	212	297	23	297	23
50	16	212	297	23	297	23
55	15	212	297	23	297	23

Table 3 *Welchia Hour 1*

Time	Hour 2 Real Instance	Hour 2 T = 3 Common to 00	Hour 2 T = 3 Actual	Hour 2 T = 7 Common to 00	Hour 2 T = 7 Actual	Prints Needed
*00	11	149	149	146	146	1
05	8	147	152	146	147	3
10	10	147	152	146	147	3
15	8	147	152	146	148	1
20	7	149	152	146	148	1
25	13	147	156	146	148	1
30	9	146	156	146	148	1
35	7	147	156	146	148	1
40	14	147	158	146	148	1
45	9	147	158	146	148	1
50	5	147	171	146	161	1
55	6	146	171	146	161	1

Table 4 *Welchia Hour 2*

4.2 VPMB

To test this method, multiple polymorphic versions of the Blaster worm, Welchia worm, and WebDAV Search exploit were created using the ADMmutate (K2 1998) kit. Another experiment was devised with two intents: (a) to see if consistent signatures are produced between different polymorphic versions of the same exploit, and (b) to see if the false positive rate will increase if more fingerprints are retained. In this experiment, two polymorphic worm packets of each type were injected into every 5-minute interval of the two hour-long traces and the VPMB scheme is used to see how many signatures are retained and how many worm packets of these 3 types are retained.

Tables 5 and 6 show the success and false-positive rate of the VPMB scheme. Using the 20-byte look-ahead window size, the VPMB scheme was able to identify all the worm packets that belong to these three types of malicious traffic. In testing, all six additional pieces of malicious traffic were detected, with only a single signature being generated for each distinct type. The performance of VPMB is worse with a 10-byte or a 5-byte look-ahead window.

Time	signatures L-A = 5	false pos. L-A = 5	signatures L-A = 10	false pos. L-A = 10	signatures L-A = 20	false pos. L-A = 20
00	8	0.025	3	0.0	3	0.0
05	9	0.036	3	0.0	3	0.0
10	9	0.042	3	0.0	3	0.0
15	9	0.031	3	0.0	3	0.0
20	9	0.036	3	0.0	3	0.0
25	10	0.034	4	0.005	3	0.0
30	11	0.038	5	0.010	3	0.0
35	11	0.048	5	0.012	3	0.0
40	23	0.066	5	0.007	3	0.0
45	23	0.120	5	0.012	3	0.0
50	23	0.131	5	0.013	3	0.0
55	23	0.132	5	0.013	3	0.0

Table 5 *Hour 1: VPMB, false positives - only three signatures are expected (signatures are cumulative)*

Time	signatures L-A = 5	false pos. L-A = 5	signatures L-A = 10	false pos. L-A = 10	signatures L-A = 20	false pos. L-A = 20
00	4	0.004	3	0.0	3	0.0
05	13	0.010	3	0.0	3	0.0
10	13	0.050	3	0.0	3	0.0
15	13	0.059	3	0.0	3	0.0
20	13	0.042	3	0.0	3	0.0
25	14	0.055	4	0.005	3	0.0
30	14	0.049	4	0.004	3	0.0
35	30	0.079	4	0.003	3	0.0
40	30	0.101	4	0.004	3	0.0
45	30	0.119	4	0.004	3	0.0
50	31	0.125	5	0.009	4	0.004
55	31	0.120	5	0.009	4	0.004

Table 6 *Hour 2: VPMB, false positives - only three signatures are expected (signatures are cumulative)*

Additional packets are classified as "worm" packets because the generated signatures are not specific enough; for each time interval, only three signatures are expected. In Table 5, the false positive rate is the highest with a 5-byte look-ahead window, yet, even at this low look-ahead size, the worst interval, at '40', only adds 13 false signatures. With a 10-byte window, only one false signature is added at times '25' and '30'. Finally, with a 20-byte window, all false positives are eliminated. Table 6 follows closely to Table 5, with the exception of one false signature added at time '50' for the 20-byte look-ahead series.

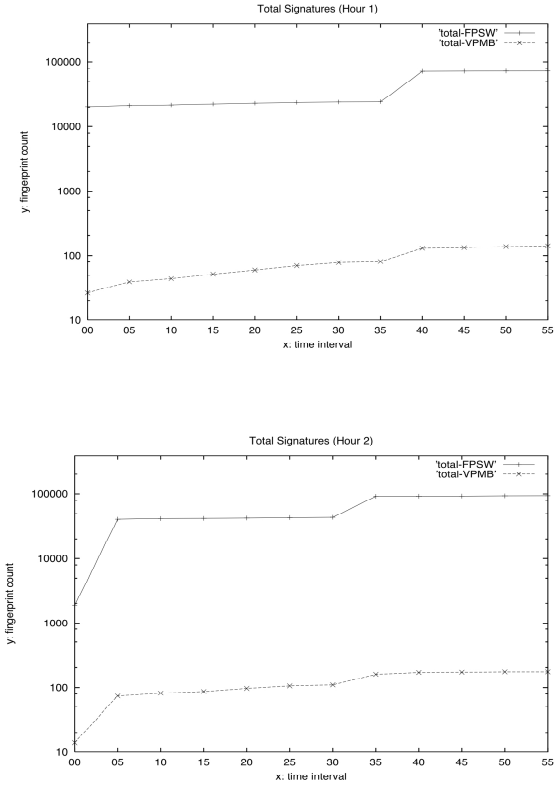

Figure 5 *FPSW vs. VPMB*

In Figure 5, the two graphs represent the cumulative fingerprint counts for VPMB and FPSW. With this comparison, we see that far fewer fingerprints are generated overall in the VPMB scheme. Both hours show roughly 100 unique fingerprints added by the end of each hour. Because VPMB is more suited toward buffer overflow detection, if it doesn't find an appropriate breakmark (i.e. a series of NOPs) then it will take a fingerprint over the entire packet. This may not have the specificity of fingerprints generated by FPSW.

5 SEMANTICS-AWARE DETECTION METHODOLOGY

New malware or worms that have appeared recently indicate that the authors of such malicious code often use code obfuscation to evade IDSs that use static signatures. There are two forms of code obfuscation: *polymorphism* and *metamorphism*. Traditional polymorphism has taken the form of an encrypted body of code with an attached (and often obfuscated) decryption routine. The encryption technique used is good enough to fool pattern-matching IDSs. Metamorphic code relies on the obfuscation of the entire code base, including code transposition, equivalent instruction substitution, jump insertion, NOP insertion, garbage instruction insertion, and register reassignment. Figure 6 shows a simple decryption routine and two obfuscated variants of that same decryption routine. The decryption routine shown in Figure 6(a) consists of a loop that performs an *xor* of a memory location against a static key, followed by an increment of the memory address to the next location. Figure 6(b) makes several changes to the code in Figure 6(a), including obscuring the key by adding *mov* and *add* instructions that work with a register. The *inc* instruction is also substituted with an *add* instruction.

```
decode:
        xor   byte ptr [eax], 95h
        inc   eax
        loop  decode
```

(a) Simple xor based decryption routine

```
decode:
        mov   ebx, 31h
        add   ebx, 64h
        xor   byte ptr [eax], ebx
        add   eax, 1
        loop  decode
```

(b) 2nd instance of xor decryption routine

```
decode:
        mov   ecx, 0
        inc   ecx
        inc   ecx
        jmp   one
two:    add   eax, 1
        jmp   three
one:    mov   ebx, 31h
        add   ebx, 64h
        xor   byte ptr [eax], ebx
        jmp   two
three:  loop  decode
```

(c) obfuscated instance of xor decryption

Figure 6 *Three equivalent code routines*

These seemingly minor changes are good enough to fool a pattern matching IDS. Figure 6(c) improves on 6(b) by adding garbage instructions, and changing the code order while preserving the execution sequence with *jmp* instructions. One can think of a plethora of equivalent programs – thus, we must rely on the *meaning* of the code, and not its syntax, for reliable detection.

Christodorescu et al. (2005) reduce the problem of semantic equivalency to a *template* matching problem. In essence, if we can create a template describing the expected behavior of a piece of code, we can match it to an actual code routine to see if the tested code exhibits the same behavior. Stated formally in Christodorescu et al. (2005), "A program *P satisfies* a template *T* (denoted as $P \models T$) iff *P* contains an instruction sequence *I* such that *I* contains a behavior specified by *T*." A template will consist of a sequence of instructions, along with its associated variables and symbolic constants.

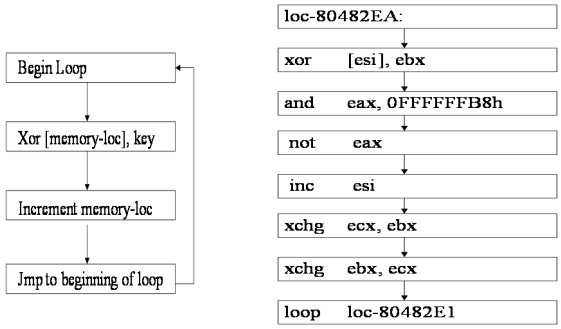

Figure 7 *A template and matching assembly code segment*

In Figure 7, we show an instance of a template on the left, and a matched assembly code segment on the right. The template shown is designed to match the decryption routine described in Figure 6. Each template is simply a description of the *behavior* we expect from a known routine – not the exact syntax that will show up in a code fragment. By looking at the assembly code segment on the right, we see that the code segment does not have a one-to-one correspondence with the template but the behavior defined by the template is present in the code routine. Thus, we can construct an algorithm to locate patterns defined in templates in real assembly code segments.

While Christodorescu et al. (2005) formalizes the template matching problem rather nicely, it presents a somewhat limited engineering approach to intrusion detection. The system that the authors built currently assumes that malware samples are available as inputs to their system. In order for the semantics-aware approach to be useful in a NIDS, a classifier needs to be provided so that semantic analysis is only performed on a small percentage of suspicious traffic. In addition, we believe that false positives are bound to emerge unless a good classifier is provided. For example, during the course of this research, we identified several legitimate programs (Crypkey 2006), (ASPack 2006) that obscure binaries with simple encryption routines as a form of copy protection. Locating a decryption loop (the primary test by Christodorescu et al. (2005)) within a program protected by one of these applications will signal a false

alert. As copy protection schemes begin to incorporate methods reminiscent of code circulating in the computer underground, we expect the false positive rate of the detection scheme based on purely checking installed binary programs on an end-host as described by Christodorescu et al. (2005) to grow accordingly. However, it is highly unlikely for copy protected program to be embedded in a web request sent by a scanning source, thus, one can easily differentiate between the two scenarios using a smart traffic classifier. Thus, we incorporate (a) a traffic classifier, and (b) a binary data identification and extraction module in our prototype. The combination of these features, and the semantic analysis allow the NIDS system we have built to be more effective than other NIDSs that are based on syntactic pattern matching approaches. In addition, our NIDS is more efficient than that reported by Christodorescu et al. (2005).

6 SEMANTIC-AWARE NIDS

Motivated by the work in Christodorescu et al. (2005), we developed a full NIDS with semantic-aware capability. Our NDIS segregates suspicious traffic from regular traffic flow, extracts binary data from suspicious traffic and performs semantic analysis on the binary data in order to identify potential threats. Such a NIDS does not rely on fingerprints or other syntax based methods. Figure 8 shows the system architecture of our NIDS. It consists of five major components, namely (a) traffic classifier, (b) binary data identification and extraction module, (c) disassembler, (d) intermediate representation generator, (e) semantic analyzer. This NIDS can be deployed on a standalone machine connected to the network.

6.1 Traffic Classification

Traffic classification is necessary to determine which packets are "interesting" and require further analysis. While it is possible to pass all traffic directly to the "Binary Detection and Extraction" module, it is more efficient to prune the traffic sent to the later stages, as they are very CPU-intensive. Currently, two classification schemes are implemented in our prototype system. The first is a simple

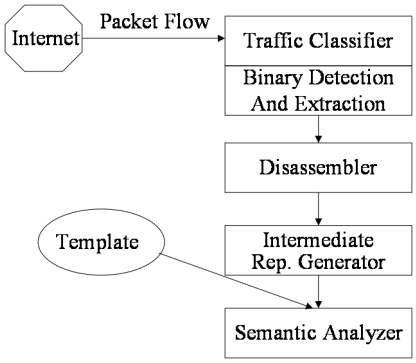

Figure 8 *The semantic-aware NIDS architecture*

and effective *honeypot* scheme. When the system is initialized, it is given a list of decoy hosts that exist for no other purpose than to attract unsolicited traffic (the effectiveness of honeypots has been explored in-depth by the Honeynet Project (2006)). Any sending host emitting traffic destined for a honeypot address is considered suspicious; and any packets sent by such a host will be analyzed.

The second scheme is a bit more complicated, and is useful for the detection of widespread worm traffic. Initially, we note the un-used IP address space in our network, with the premise that any traffic repeatedly destined to the un-used address space may be indicative of malicious scanning. If a host sends an initial packet to an un-used address, a count n is initialized. If we continue to observe this host sending additional packets to other un-used addresses, the count will be incremented until it reaches a threshold t, at which point, packets emanating from that suspicious host will be considered for further analysis.

6.2 Binary Detection and Extraction

In this work, we are interested in examining binary threats primarily in the form of buffer overflow exploits (we do not currently support detection of textual web attacks, brute force password attacks, etc.). Thus, we need a way to identify binary data within packet payloads. To accomplish this task, we need to understand how buffer overflow exploits are constructed and presented to a victim host.

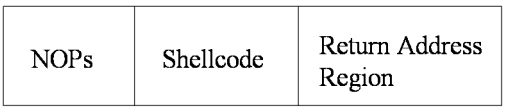

Figure 9 *Format of buffer overflow exploits*

Traditional buffer overflow exploits (Figure 9) have taken the following form: a region of NOP instructions at the lowest address region on the stack, followed by the instructions the attacker wishes to execute, followed by a series of return addresses that will overwrite the return pointer of the subroutine and point back into the stack. Historically in IDS, it has been easy to detect the NOP region, as it was only composed of a repeating series of the same instruction (i.e. 0x90 for the x86 architecture). However, this is no longer the case – polymorphic exploit generators can use a whole host of instructions that have "NOP-like" behavior, thus making the NOP region variant. This leaves us with the return address region as a possible place to observe some invariant data. Only the least significant byte can be varied, since the return address must point back to a valid address in the buffer.

In practice, we observe network buffer overflow exploits to consist of a well-formed initial application layer protocol request, with exploit content usually resembling (but not necessarily matching exactly) Figure 9 encapsulated within it. By noting what is expected in a protocol request, and

what is abnormal, we can often locate malicious binary content. Figure 10 displays the content of the Code Red II worm exploit. Here, a well-formatted HTTP GET request is made to a module of the IIS webserver. A stream of repeated 'X' characters initiates the overflow, and these characters are followed by the Unicode data. Our module has the ability to distinguish between acceptable protocol usage and suspicious repetition. Thus, we can locate the approximate region where we believe the binary content is located, and extract it. In the case of Unicode data (as is observed in Figure 7), we translate it into an appropriate binary form, for further analysis. This process will yield some binary data that is benign, but it dramatically cuts down on the amount of data that must be processed by the disassembler which is the slowest stage in our system. This binary identification and extraction process can be bypassed but it will result in a system with much degraded performance.

```
GET /default.ida?XXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXX%u9090%u6858%cbd3
%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%u
cbd3%u7801%u9090%u9090%u8190%u00c3%u0003%u8b
00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0
```

Figure 10 *Code Red II exploit portion*

6.3 Semantic Analysis

Because we have chosen a specific commercial product, IDA Pro (DataRescue, 2006) for our disassembler stage, our NIDS can only disassemble x86 code at the present. The binary detection and extraction stage produces special binary frames (binary data extracted from network packets) in a format that can be processed by the disassembler. Once an assembly code representation is generated by the disassembler, we prune the code to include only the instructions we are interested in. Any excess code from the program frame is discarded.

At this point, we have a sequence of instructions that we can analyze semantically. The semantic analyzer uses the template matching scheme Christodorescu et al. (2005) that we have described in Section 5. The templates that we built have the ability to handle out of order code, NOP insertion, junk instruction insertion, and register reassignment. If a piece of code matches one of our templates, an alert is generated, and further action may be taken against the offending IP address.

7 SYSTEM EVALUATION

We have conducted an extensive evaluation of our semantic NIDS, against real malware samples and captured network traffic. All of our tests were performed on an Intel P4 2.8Ghz system with 512MB of memory. One of our primary goals with this work is to establish a reliable method for detecting polymorphic exploit instances. Thus, we evaluate two popular toolkits for polymorphic exploit generation, along with a publicly available exploit known to contain polymorphic shellcode. We also test a month's worth of benign traffic, with classification disabled (all packet payloads are analyzed). Our preliminary results are extremely promising: we observe no false positives when we analyzed the benign traffic and we can nearly detect all polymorphic versions of malicious contents generated using ADMmutate (K2 1998) and Clet engine (CLET Team 2003).

7.1 Linux Shell Spawning

In this first test, we select eight different remote exploits, which can spawn a shell in a machine running the Linux operating system. A template is created (Figure 11) to match the relevant system calls associated with this behavior. It can detect shells created as an immediate instance of the exploit, and, with an extension, those that are bound to a separate network port. In our experiment, we built an exploit generator tool that sends exploit packets to a honeypot machine registered with the NIDS. All eight exploits are successfully detected as spawning a shell, while the two that bind the shell to a different port are also noted as such.

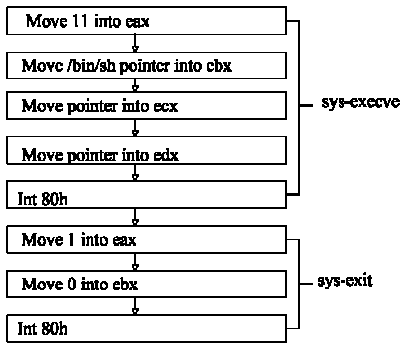

Figure 11 *template for Linux shell spawning code*

The results for this first set of experiments are tabulated in Table 7. The running time for these eight instances ranges from 2.36 seconds to 3.27 seconds. The average binary code size is less than 10Kbytes for these exploits. As a comparison, we ran two variants of the Netsky virus with an average code size of 22 Kbytes through our program and it takes about 6.5 seconds each time. The time reported in Christodorescu et al. (2005) is about 40 seconds.

Exploit Name	Spawn Shell	Bind Shell	Running Time
bncex.c	X	X	2.621s
imap3.c	X		2.485s
imapx.c	X		3.270s
mod-mylo.c	X		2.264s
rdC-LPRng.c	X		2.355s
arksink2.c	X		2.670s
qpop-linux.c	X		2.777s
SDI-bnc.c	X	X	2.679s

Table 7 *Linux shell spawning buffer overflow exploits*

7.2 Polymorphic Shellcode Detection

To detect polymorphic code, we create a template that captures the decryption loop functionality described in Section 5. Then, we create a tool that can generate numerous exploits towards a honeypot machine that was registered with the NIDS. The first test we perform is to verify that our system can detect the iis-asp-overflow.c exploit based on the template we designed. This particular exploit has a decryption routine prefixed to an encoded shell-spawning region of code. The shellcode is encoded to evade detection by IDSs that employ pattern-matching techniques. Using the template we design, our system was able to detect the decryption routine. The running time for this test is 2.14 seconds.

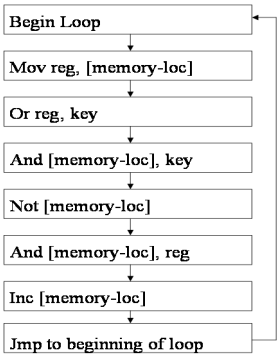

Figure 12 *template for alternate ADMmutate decryption loop*

The ADMmutate kit (K2 1998) is a popular polymorphic shellcode generation toolkit. It incorporates NOP-like instruction insertion, garbage instruction insertion, equivalent instruction replacement, and out-of-order code sequencing to obscure its decryption routine. For testing, 100 instances of polymorphic payloads were generated, and inserted into a generic network buffer overflow exploit. As shown in Table 8, the first test which uses the template in Fig. 7 yielded only a 68% detection rate. Further manual inspection of the assembly code generated by our NIDS led us to establish that ADMmutate incorporates one of two distinct methods for its decryption routine. The first is the *xor* decryption our template can match, while the second is a decoding scheme involving a sequence of *mov*, *or*, *and*, and *not* instructions that perform operations on a single memory location and register pair. Once we developed an enhanced template (shown in Figure12) that can match such behaviors, we achieve 100% detection of all shellcodes generated by ADMmutate.

Polymorphic Instance	Decryption Routine	Avg. Run Time
ADMmutate, test one	68%	2.179s
ADMmutate, test two	100%	2.510s
Clet	100%	2.666s

Table 8 *Polymorphic shellcode detection*

The Clet engine (CLET Team 2003) is another popular tool for generating polymorphic shellcode. It relies on obscuring an *xor* based decryption routine in a fashion that will defeat data mining approaches to IDS. Thus, it incorporates many of the same features as ADMmutate, but Clet can also score the feature distribution probabilistically, so that the packet can appear to be "normal traffic." Our *xor* decryption template matched all 100 shellcode instances that Clet generated.

7.3 Code Red II Worm Detection

A template is devised to match the initial exploitation vector of the Code Red II worm. We test this template against 12 5-minute traces collected from two Class B production networks, each with a total packet count of over 200,000. Before evaluation, we note the correct number of instances of Code Red II within each capture. The results are tabulated in Table 9. From Table 9, one can note that every instance is classified and matched correctly by our NIDS.

Capture	Packet Count	Capture Size	Binary Content	Expected Instances	Detected Instances	Total Run Time	Avg. Run Time
1	262,342	13.7Mb	140	1	1	195.66s	1.398s
2	256,638	13.3Mb	186	3	3	638.74s	3.434s
3	254,587	13.5Mb	93	3	3	420.36s	4.520s
4	272,127	13.7Mb	121	0	0	191.94s	1.586s
5	254,874	13.6Mb	136	2	2	500.34s	3.679s
6	255,657	12.8Mb	96	5	5	368.29s	3.836s
7	282,239	17.2Mb	547	5	5	1956.54s	3.836s
8	276,420	16.3Mb	131	1	1	450.00s	3.442s
9	273,336	13.3Mb	169	0	0	267.33s	1.582s
10	254,280	14.1Mb	262	3	3	401.08s	1.531s
11	256,283	14.9Mb	227	7	7	359.04s	1.587s
12	252,000	13.1Mb	201	5	5	278.21s	1.384s

Table 9 *Detection of the Code Red II worm*

7.4 False Positive and Negative Evaluation

For a final test, we disabled traffic classification on the NIDS, and examined every packet's payload in a month's worth of traffic captured from two Class C networks (a total capture of 566MB). Most of the packets in this trace are legitimate web traffic. The traffic was examined beforehand, to ensure none of the threats we are attempting

to detect with our current template set (decryption routines, shell spawning, Code Red II memory addressing) were present. No false positives were reported from our template matching module; this is consistent with the findings of Christodorescu et al. (2005), though now confirmed in the network scenario.

Our experiments so far show no false negatives. This is not to say that our current templates will be able to handle all future buffer overflow exploits. We anticipate that in future, one may conceive of new ways to exploit buffer overflows using a different format from what is shown in Figure 9. This would drive up the false negative rate if the current set of templates could not match the new behaviours. As new attack classes emerge, researchers must respond with templates to match the attack behaviours. Thus, we have not eliminated completely the need for human intervention in IDS, but have managed to reduce the threat posed by polymorphic and metamorphic variants of known attack classes.

7 CONCLUSION AND FUTURE WORK

In this paper, we have described both syntax-based and semantic based approaches for dynamic network intrusion detection. For syntax-based approaches, we evaluated a fixed-partition and variable-length partition sliding-window scheme for automatic worm generation. Our results indicate that the variable length partition scheme is more flexible and can handle several types of polymorphic worms. To deal with more sophisticated polymorphic and metamorphic worms, we propose a semantic-aware approach. We have designed and built a NIDS with semantic analysis capability. We have performed extensive tests on our prototype system. Our results show that using high quality templates, our system is able to detect a wide variety of code exhibiting the same behavior, as opposed to the same formal syntax. Our experimental evaluation shows that our system does not produce any false positives when tested against a network trace of benign traffic. In the near future, we intend to classify more exploit behaviors so that we can generate additional useful templates that can be used in our NIDS to detect additional families of malicious traffic (i.e. email worms). Moreover, we will continue to advance our understanding of polymorphic behavior in malicious software. One can envision a multitude of encryption schemes used in tandem to obscure the behavior of a malicious payload. A template that detects a simple loop may be sufficient for detection in a suspicious context (i.e. a web request with x86 executable content). Finally, we also intend to optimize our implementation so that it can run even faster than what has been achieved.

REFERENCES

ASPack Software. (2006) 'ASProtect', *Published online at http://www.aspack.com*, Last accessed on 6 Jan. 2006.

Broder, A. (1992) 'Some applications of Rabin's fingerprinting method', *In Renato Capocelli, Alfredo De Santis, and Ugo Vaccaro editors, Sequence II: Methods in Communications, Security, and Computer Science*, Springer-Verlag, pp. 143-152.

Christodorescu, M., Jha S., Seshia S., Song, D., and Bryant, R. (2005) 'Semantics-aware malware detection', *IEEE Security and Privacy Symposium*, May.

CLET Team. (2003) 'Polymorphic shellcode engine using spectrum analysis', *Phrack Magazine*, 11(61).

CrypKey. (2006) 'CrypKey', *Published online at http://crypkey.com*, Last accessed on 6 Jan. 2006.

Datarescue. (2006) 'IDA Pro – interactive disassembler', *Published online at http://www.datarescue.com/idabase,* Last accessed on 6 Jan. 2006.

Gao, L., Wu, S., Vangala, S., and Kwiat, K. (2004)) 'An effective architecture and algorithm for detecting worms with various scan techniques', *Proceedings of NDSS*.

The Honeynet Project. (2006) 'Project Homepage', *http://project.honeynet.org*, Last accessed on 6 Jan. 2006.

K2 (1998) 'ADMmutate 0.8.4', *Published online at http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, Last accessed on 6 Jan. 2006.

Kim, V. and Karp, B. (2004) 'Autograph: toward automated, distributed worm signature detection.', *Proceedings of the 13th USENIX Security Symposium.*

Levenshtein, V. I. (1965) 'Binary Codes capable of correting spurious insertions and deletions of ones', *Problems of Information Transmission*, 1:8-17.

Locasto, M., Parekh, J., Stolfo, S., Keromytis, A., Malkin, T., and Misra, V. (2004) 'Collaborative distributed intrusion detection', *Tech Report CUCS-012-04, Department of Computer Science, Columbia University*.

Moore, H.D., Shannon, C., Voelker, G., and Savage, S. (2003) 'Internet quarantine: requirements for containing self-propagating code.', *Proceedings of the 2003 IEEE Infocom Conference*, April.

Newsome, J., B. Karp., and Song, D. (2005) 'Polygraph: automatically generating signatures for polymorphic worms', *Proceedings of the IEEE Symposium on Security and Privacy*.

Pang, R., Yegneswaran, V., Barford, P., Paxson, V., and Peterson L., (2004) 'Characteristics of Internet background radiation', *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, October.

Paxson, V. (1998) 'Bro: a system for detecting network intruders in real-time', *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January.

Rabin, M.O., (1981) 'Fingerprinting by Random Polynomials', *Tech Report TR-15-81, Center for Research in Computing Technology, Harvard University*.

Reuters. (2004) 'Virus damage estimated at $55 billion in 2003', *Published online at http://msnbc.msn.com/id/3979687/.* Last accessed on 6 Jan. 2006.

Roesch, M. (1999) 'Snort – lightweight intrusion detection for networks', *Proceedings of the 13th USENIX conference on System Administration (LISA '99)*, Seattle, Washington, pp. 229-238.

Singh, S., Estan, C., Varghese, G., and Savage, S. (2004) 'Automated worm fingerprinting', *Proceedings of the 6th USENIX Symposium on Operating System Design and Implementation.*

Stolfo, S., and Wang, K. (2004) 'Anomalous payload-based network intrusion detection', *Proceedings of Recent Advances in Intrusion Detection (RAID)*, September.

Yegneswaran, V., Griffin, J., Barford, P., and Jha, S. (2005) 'An architecture for generating semantic-aware signatures', *14th USENIX Symposium on Security*, August.