Network Intrusion Detection with Semantics-Aware Capability

Walter Scheirer and Mooi Choo Chuah

Lehigh University
Dept. of Computer Science and Engineering
Bethlehem, PA 18015 USA
{wjs3, 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 propose a network intrusion detection system (NIDS) with semantics-aware capability. Our NIDS segregates suspicious traffic from the regular traffic flow, extracts binary code from the suspicious traffic, and performs semantic analysis on it to identify potential threats. Our contributions in this work are threefold: (a) we believe our prototype is the first NIDS that provides semantics-aware capability, (b) our implementation is more efficient than what is reported in [5], (c) our designed templates can capture polymorphic shellcodes with added sequences of stack and mathematic operations.

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 [1]. In 2003, this number jumped to 55 billion dollars [1]. 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 [2] and Bro [3] 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 [4], 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 describe a prototype system with semantics-aware capability that we have built to automatically identify threats from some unknown malicious network traffic. This work is an extension of the approach presented in [5]. The semantics-aware malware detection algorithm of [5] 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 [5], 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 work, we have built a full-featured network intrusion detection system with semantic-aware capability that 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 [5].

The rest of the paper is organized as follows: In Section 2, we describe some related work and discuss how one particular work motivates this research. In Section 3, we describe the motivation for the semantic analysis of malicious code, and discuss how binary exploits work. In Section 4, 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 5. Finally, we summarize our findings and discuss some future work that we intend to explore in Section 6.

2. Related Work

Much research has been devoted to intrusion detection in recent years. Two enormously popular open source tools, Snort [2] and Bro [3], 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 [13], 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 [10], 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 [7], [8], the authors show that byte-level analysis of packet payloads can yield useful signatures for worm detection. The premise being some portions of the worm code will be invariant. At first glance, these approaches looked promising, however, in practice, they generate far too many signatures, with a sometimes-undesirable accuracy rate. A recent paper [12] also addresses the polymorphic worm detection problem in a similar manner. They advocate using disjoint data signatures. With [14], 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 [5] and [6], 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 [6]. In this work, we built upon the approach described in [5]. Our contributions 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 [5], (c) our designed templates can capture polymorphic shellcodes with added sequences of stack and mathematic operations.

3. Semantics-Aware Malware Detection Methodology

Current IDSs often use static signatures. However, 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 1 shows a simple decryption routine and two obfuscated variants of that same decryption routine. The decryption routine shown in Figure 1(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 1(b) makes several changes to the code in Figure 1(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 1. Three equivalent code routines.

These seemingly minor changes are good enough to fool a pattern matching IDS. Figure 1(c) improves on 1(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.

The authors of [5] 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 [5], "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.

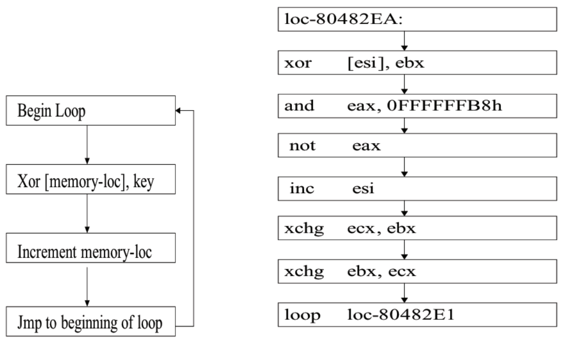

Figure 2. A template and matching assembly code segment.

In Figure 2, 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 1. 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 [5] 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 [18], ASProtect [19]) that obscure binaries with simple encryption routines as a form of copy protection. Locating a decryption loop (the primary test in [5]) 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 in [5] 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 in [5].

4. NIDS with Semantic-Aware Capability

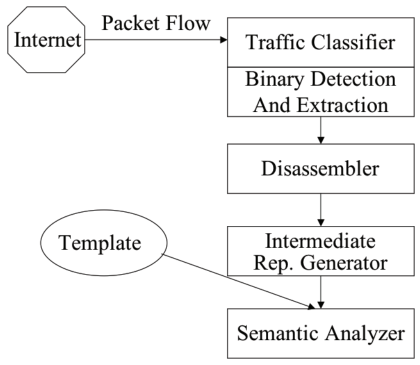

Figure 3. The semantic-aware NIDS architecture.

In this work, we develop a full NIDS that 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 3 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.

4.1. 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 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 [16]). 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.

4.2. 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.

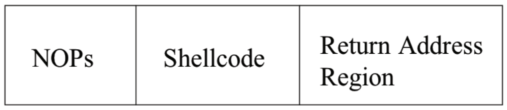

Figure 4. Format of buffer overflow exploits.

Traditional buffer overflow exploits (Figure 4) 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 4 encapsulated within it. By noting what is expected in a protocol request, and what is abnormal, we can often locate malicious binary content. Figure 5 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 5), 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?XXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXX%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 5.Code Red II exploit portion.

4.3. Semantic Analysis

Because we have chosen a specific commercial product, IDA Pro [17], 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 [5] that we have described in Section 3. 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.

5. 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 [11] and Clet engine [9].

5.1. Linux Shell Spawning

In this first test, we selected eight different remote exploits, which can spawn a shell in a machine running the Linux operating system. A template was created (Figure 6) 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.

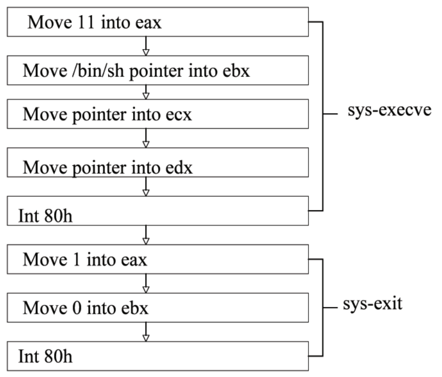

Figure 6. Template for Linux shell spawning code.

The results for this first set of experiments are tabulated in Table 1. 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 [5] 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 1. Linux shell spawning buffer overflow exploits.

5.2. Polymorphic Shellcode Detection

To detect polymorphic code, we created a template that captures the decryption loop functionality described in Section 3. Then, we created 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 designed, our system was able to detect the decryption routine. The running time for this test is 2.14 seconds.

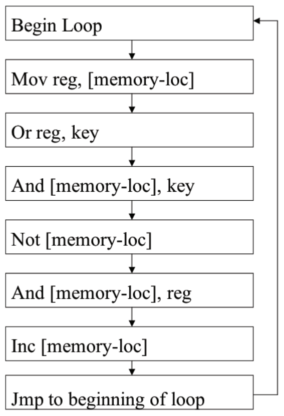

Figure 7. Template for alternate ADMmutate decryption loop.

The ADMmutate kit [11] 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. The first test 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 a template that can match such behaviors, we achieved 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 2. Polymorphic shellcode detection.

The Clet engine [9] 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.

5.3. Code Red II Worm Detection

A template was devised to match the initial exploitation vector of the Code Red II worm. We tested 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 noted the correct number of instances of Code Red II within each capture. The results are tabulated in Table 3. From Table 3, one can note that every instance was 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 3. Detection of the Code Red II Worm.

5.4. False Positive 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 [5], though now confirmed in the network scenario.

6. Conclusion

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). We also intend to optimize our implementation so that it can run even faster than what has been achieved.

Acknowledgements

We would like to thank Vinod Yegneswaran, Dr. Paul Barford, and the Wisconsin Advanced Internet Laboratory for their willingness to share several network traces that we have used in this paper.

7. References

[1] Reuters. Virus damage estimated at $55 billion in 2003. Jan. 2004, `http://msnbc.msn.com/id/3979687/`. Last accessed on 6 Jan. 2006.

[2] M. Roesch, Snort - lightweight intrusion detection for networks. LISA '99: Proceedings of the 13th USENIX conference on System administation, Seattle, Washington, 229-238, 1999.

[3] V. Paxson. Bro: a system for detecting network intruders in real-time. Computer Networks, Amsterdam, Netherlands, 31 (23-24): 2435-2463, 1999.

[4] H.D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet quarantine: requirements for containing a self-propagating code. Proceedings of the 2003 IEEE Infocom Conference, April 2003.

[5] M. Christodorescu, S. Jha, S. Seshia, D. Song and R. Bryant. Semantics-aware malware detection. IEEE Security and Privacy Symposium, May 2005.

[6] V. Yegneswaran, J. Griffin, P. Barford and S. Jha. An architecture for generating semantic-aware signatures. 14th USENIX Symposium on Security, August 2005.

[7] H. Kim and B. Karp. Autograph: toward automated, distributed worm signature detection. Proceedings of the 13th Usenix Security Symposium, 2004.

[8] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, 2004.

[9] CLET Team. Polymorphic shellcode engine using spectrum analysis. Phrack Magazine, 11(61), 2003.

[10] L. Gao, J. Wu, S. Vangala, and K. Kwiat. An effective architecture and algorithm for detecting worms with various scan techniques. Proceedings of NDSS, 2004.

[11] K2. ADMmutate 0.8.4. Published online at `http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz`. Last accessed on 6 Jan. 2006.

[12] S. Stolfo and K. Wang. Anomalous payload-based network intrusion detection. Proceedings of Recent Advances in Intrusion Detection (RAID), Sept. 2004.

[13] M. Locasto, J. Parekh, S. Stolfo, A. Keromytis, T. Malkin, and V. Misra. Collaborative distributed intrusion detection. Tech Report CUCS-012-04, Department of Computer Science, Columbia University, 2004.

[14] J. Newsome, B. Karp, and D. Song. Polygraph: automatically generating signatures for polymorphic worms. Proceedings of the IEEE Sysmposium on Security and Privacy, 2005.

[15] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of internet background radiation. IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, Oct. 2004.

[16] The Honeynet Project. Project homepage. http://project.honeynet.org. Last accessed on 6 Jan. 2006.

[17] DataRescue. IDA Pro – interactive disassembler. Published online at `http://www.datarescue.com/idabase`. Last accessed on 6 Jan. 2006.

[18] CrypKey. Published online at `http://www.crypkey.com`. Last accessed on 6 Jan. 2006.

[19] ASPack Software. ASProtect. Published online at `http://www.aspack.com`. Last accessed on 6 Jan. 2006.