DAWN: A Novel Strategy for Detecting ASCII Worms in Networks

Parbati Kumar Manna   Sanjay Ranka   Shigang Chen
Department of Computer and Information Science and Engineering, University of Florida
{pkmanna, ranka, sgchen}@cise.ufl.edu

***Abstract*—While a considerable amount of research has been done for detecting the binary worms exploiting the vulnerability of buffer overflow, very little effort has been spent in detecting worms that consist of only text, *i.e.*, printable ASCII characters. We show that the existing worm detectors often either do not examine the ASCII stream or are not well suited to *efficiently* detect worms in the ASCII stream due to the structural properties of the ASCII payload. In this paper, we analyze the potentials and constraints of the ASCII worms vis-a-vis their binary counterpart, and devise a detection technique that would exploit those limitations. We introduce DAWN, a novel ASCII worm detection strategy that is fast, easily deployable, and has very little overhead. Unlike many signature–based detection methods, DAWN is completely signature-free and therefore capable of detecting zero-day outbreak of ASCII worms.**

I. Introduction

Computer worms interest the security analysts immensely due to their ability to infect millions of computers in a very short period of time. Among various worms, ASCII worms (consisting of text, *i.e.* entirely printable ASCII characters) are very appealing since they can sneak in places where a worm is not expected to be able to get in under normal circumstances. For example, in many cases the server expects certain kind of traffic to be strictly text, as is the case with many important applications working with HTTP and XML. To ensure that only the text characters get in at times when text is expected (like the email traffic or the URL in a HTTP request), these servers usually employ ASCII filters [1] which drops or mangles any binary input. Since worms are presumed to be binary, the act of filtering alone (without subjecting the text stream for malware detection) gives a false sense of security against worms. Even when the ASCII stream *does* undergo detection for malware, there are cases where it can still be bypassed internally by the detector itself (e.g. SigFree [2], in order to avoid significant performance degradation). Thus, we observe that there are cases where the ASCII traffic effectively does not undergo any kind of worm detection. Therefore, if there is a way to have worms that are completely ASCII, these servers will be immediately vulnerable to worm attacks. Rix [3], and later Eller [1] showed a few years ago that it is indeed possible to convert any binary worm into ASCII, and if required, even alphanumeric. This implies that the ASCII traffic *does* pose a real threat, and bypassing it altogether from the worm detection mechanism may not be a good idea.

Next we demonstrate that even when the ASCII traffic is *not* bypassed, payload-based detection mechanisms for binary worms may still not be adequately suited for efficiently detecting ASCII worms. We specifically consider two such schemes: 1) that detects by disassembling the input into instructions and then checking for the validity and executability of instruction sequences (e.g. APE [4]), and 2) that detects by looking at the frequency distribution and other statistical properties of the payload (e.g. PAYL [5]). There are two potential problems with the disassembly-based scheme. First, nearly all ASCII strings translate into syntactically correct sequences of instructions, which means checking for syntactic validity is of little value for detecting ASCII worms. Second, since most of the branch opcodes are ASCII, the proportion of branch instructions for ASCII data is significantly higher than that for binary. Since each branch instruction forks the current execution path into two directions, having a lot of them exponentially increases the total number of paths to be inspected by a detector that employs pseudo-execution. Therefore, for disassembly-based detectors, one must find novel ways to prune the number of the paths to be inspected to ensure quick detection of worms in ASCII data. Also, the detection approach of examining the frequency distribution and other statistical properties of the payload is not foolproof either, as there have been instances where ASCII worms have been shown to have successfully evaded such detectors. For example, Kolesnikov *et al* [6] showed the way to create an ASCII worm that follows normal traffic pattern to the extent that it can evade even a robust payload-based detector like PAYL [6]. Finally, we scanned the ASCII worms that we used for our testing using a commercial malware detector and no alarms were raised. Thus, we conclude that the threat of ASCII worm is real, and we can ignore them only at our own peril.

The structure of the rest of the paper is as follows. Section II provides the details of ASCII worm. We focus on the constraints and weaknesses of the ASCII worm in Section III and devise a scheme to detect it. In Section IV, we lay down the implementation details of our detection method and evaluate the results. In Section V, we show why binary detectors are unsuitable for ASCII, and finally in Section VI, we conclude with the limitations of our detection strategy.

II. Inside the ASCII Worm

In this section, we start with a formal definition of the ASCII worm and discuss a typical construction of an ASCII worm. The treatment of the ASCII worms here forms the foundation of the detection strategy detailed in the next section.

A. Definitions and Terminologies

We use the following definitions throughout this article:

- **ASCII data**: data consisting of only *text*, i.e. keyboard-enterable, or in other words, printable ASCII characters (0x20 through 0x7E). A worm whose payload consists of entirely ASCII data is called an ASCII worm.
- **Binary data**: character stream consisting of text as well as non-text characters. A worm whose payload contains binary data is called a binary worm.
- **Valid (or invalid) instruction**: an instruction that will not (or will) cause the running process to abort by raising an error during its execution.
- **MEL (Maximum Executable Length)**: length of the longest sequence of consecutively-executed valid instructions in an instruction stream (binary or ASCII).

The concept of MEL was introduced in Abstract Payload Execution (APE) [4] for detecting the binary worms. However, it will be shown in section V why APE does not work for ASCII worms (or for any *current* binary worms for that matter) and how our work is different from APE.

B. Construction of a Typical ASCII Worm

It is well known that a worm needs to make system calls and perform similar activities (like opening sockets for propagation). However, the opcodes required for those are not available in ASCII. The only available Intel opcodes and instruction prefixes in the ASCII data are:

- Dual-operand register/memory manipulation or comparison opcodes: *sub*, *xor*, *and*, *inc*, *imul* and *cmp*
- Single-operand register manipulation opcodes: *inc*, *dec*
- Stack-manipulation opcodes: *push*, *pop*, and *popa*
- Jump opcodes: *jo*, *jno*, *jb*, *jae*, *je*, *jne*, *jbe*, *ja*, *js*, *jns*, *jp*, *jnp*, *jnge*, *jnl* and *jng*
- I/O operation opcodes: *insb*, *insd*, *outsb* and *outsd*
- Miscellaneous opcodes: *aaa*, *daa*, *das*, *bound* and *arpl*
- Operand and Segment override prefixes: *cs*, *ds*, *es*, *fs*, *gs*, *ss*, *a16* and *o16*

It is evident that in order to create a potent ASCII worm, the required non-ASCII opcodes must be dynamically generated using ASCII opcodes at runtime. It is possible to generate any binary byte using ASCII data only, *e.g.*, the binary character 0 can be generated by doing `'a'`$\oplus$`'a'`. While it is impossible to enumerate all the possible ways to create an ASCII worm, a typical construction method known as the "stack" method [3] is shown in Figure 1.

In this paper, we refer to the process of turning an ASCII worm into binary code as *decryption*. The worm itself must carry a *decrypter*, a cleartext ASCII instruction sequence that performs the decryption. In many cases, the whole worm is a decrypter, as is the case with the worm shown in figure 1.

III. Detection Strategy for ASCII Worms

In this section we start with a description of the constraints that are imposed on an ASCII worm. Next we characterize how

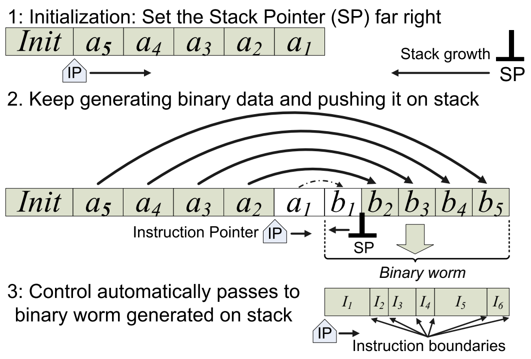
1: Initialization: Set the Stack Pointer (SP) far right
Init a5 a4 a3 a2 a1
IP
Stack growth
SP
2. Keep generating binary data and pushing it on stack
Init a5 a4 a3 a2 a1 b1 b2 b3 b4 b5
Instruction Pointer IP
SP
Binary worm
3: Control automatically passes to binary worm generated on stack
I1 I2 I3 I4 I5 I6
IP
Instruction boundaries

Fig. 1. Creation of a binary worm on stack from ASCII code. For generating a n-word binary worm $B = b_1b_2...b_n$, the corresponding ASCII worm code is $Init\ a_na_{n-1}...a_2a_1$, where $Init$ denotes some initialization code and the ASCII code block a_i dynamically generates corresponding binary word b_i. Observe the reverse order of binary word genration so that control is automatically passed to the binary worm once it is fully created.

an ASCII worm must behave in order to overcome these constraints. Once we pinpoint the behavioral characteristic (high MEL), we proceed towards devising a practicable detection scheme exploiting that characteristics.

A. Constraints of an ASCII Worm

The following are the inherent limitations of ASCII worms:

- **Opcode Unavailability:** It has been stated earlier that in order to propagate to other host, a worm must perform certain actions (like opening a socket, etc.) that require making system calls, for which opcodes are unavailable in ASCII data. Therefore, the ASCII worms are constrained to *generate* these opcodes *dynamically*.
- **Difficulty in Encryption:** In oder to avoid detection, it is a common practice to convert a cleartext worm into an encrypted payload preceded by a *small* cleartext decrypter. In order to encrypt a binary worm into ASCII efficiently, one has to realize the following goals: 1) both the decrypter and the encrypted payload should be ASCII, 2) the size of the decrypter should be small, and 3) there should not be a significant size discrepancy between the encrypted payload and the cleartext, as the vulnerability can very well have a size constraint. However, there are a few difficulties with realizing these goals. First, since the ASCII domain is a proper subset of the binary domain, it is not possible to have a one-to-one correspondence between the two. Therefore, when one byte of binary data is encrypted into ASCII, the size of the output is more than one byte. This has the undesirable effect that the size of the encrypted payload is larger than the original cleartext by a *factor* rather than by a *constant*. Finally, without the one-to-one correspondence the decryption logic is more complex, thus resulting in a significantly larger decrypter.
- **Control Flow Constraints:** Decryption routines are customarily implemented using loops, which involves control redirection using opcodes mainly belonging to the *loop* or *jump* family followed by a negative displacement byte (to "go back" to the beginning of the decrypter loop). However, since all the printable ASCII characters have

0 in their most significant bit, it is not possible to have a negative displacement byte. Therefore, by using *jump* statements, one can only go forward in an ASCII worm but not backwards. As a result, the option of re-executing the same decrypter for each word of the encrypted payload is precluded – for a n-word encrypted payload, one must have $O(n)$ decrypter blocks where each decrypter block will decrypt one individual word. While it is theoretically possible to overcome this difficulty by generating the negative displacement dynamically, that would very likely make the decrypter more complicated and increase its size and MEL (this issue will be discussed in greater detail in Section VI).

To summarize, due to the opcode constraints, an ASCII worm must decrypt itself to generate the actual binary worm at runtime. The encryption and control flow constraints imply that the decryption process would involve a relatively large cleartext decrypter, which will have a high MEL. Therefore, if an ASCII stream has a high MEL, it can be reasonably presumed to contain an ASCII worm.

B. Benign ASCII Stream Tends to Have Low MEL

Here, it is shown that the benign text does not tend to have a high MEL. This is because benign ASCII stream is frequently interspersed with invalid (error-raising) instructions that prune the execution path into smaller segments, and thus result in a lower MEL. The errors are raised due to the following reasons:

Prevalence of Privileged Instructions: The characters '*l*', '*m*', '*n*' and '*o*', which occur frequently in text correspond to the opcodes `insb`, `insd`, `outsb` and `outsd` respectively. These are privileged I/O instructions that cannot be invoked from any user-level application without generating an error [7]. Thus, benign ASCII data may have these instructions, but a worm will never have them in its execution path.

Illegal Memory Access: Since ASCII characters have 0 in their most significant bits, direct register-register instructions are ruled out. Thus, to manipulate a register, the value must come from memory (other than *inc dec*, or *pop*). Violations during the memory access can happen in the following ways:

- **Uninitialized Register:** If an uninitialized register is used to address a memory location, the memory address pointed to by the register will be an unpredictable one, with a high probability of being outside of the allocated memory block for this process. This implies that any attempt to access that address would cause a protection exception error.
- **Wrong Segment Selector:** For memory-accessing dual-operand instructions, if the instruction is preceded by an arbitrary segment override prefix, it could lead to a memory address which is outside the bounds for this process and thus raise a protection exception. During our experiments, it was observed that FS and GS segment prefixes resulted in a protection exception error.
- **Explicit Memory Address:** For certain ASCII ModR/M bytes ('%', '–' , '5', '=', '&', '.', '6' and '>'), the memory address is expressed as an explicit 4-byte or 2-byte displacement. Linux randomizes the start address of each program, and similar porpositions have been made to randomize the static libraries in Windows [8]. Thus, it is very likely that using an explicit memory address will raise a memory violation.

Based on the above analysis, we conclude that a benign stream of ASCII will have a low MEL, and hence a threshold on MEL can be used to determine whether an ASCII stream is malicious or benign.

IV. Implementation of DAWN

This section describes DAWN, the proposed detection strategy for ASCII worms. Briefly, DAWN operates in two main stages: instruction disassembly and instruction sequence analysis. First it disassembles the ASCII input from every possible position. Next, by performing pseudo-execution of the instruction stream, it attempts to see if any such disassembly could potentially lead to a malicious code. If it detects a long sequence of valid instructions (longer than a certain threshold), then an alert is raised. The individual steps are delineated in more detail in the next two subsections, followed by the experimental results.

A. Step 1: Instruction Disassembly

Since it is not possible to predict the entry point of the worm in the input stream, the input (say of length n bytes) needs to be disassembled from all possible n entry points. It has been shown [9] that if one starts interpreting the same instruction stream from two adjoining bytes, the instruction boundaries of the two instruction sequences tend to get aligned within 6 instructions (max 78 bytes) with a very high probability. Thus, for every entry point we need to disassemble an average of 6 instruction before we can re-use the instruction sequence that has been already disassembled. Therefore, although the disassembly is technically a $O(n^2)$ process, it is linear from a practical standpoint.

B. Step 2: Instruction Sequence Analysis

The main purpose of this stage is to ascertain how long an instruction sequence (which may start anywhere within the ASCII data) may execute without generating an error. The error may result either from using a privileged instruction, or from a memory access violation. As DAWN proceeds with the pseudo-execution of the instruction sequence, it keeps track of which registers have been initialized properly. When an uninitialized register is used to address memory (as source or destination), it is considered to be the end of that sequence. For a control flow bifurcation (like *jump*), DAWN recursively considers both the possible routes (*jump target* as well as the *fall-through* instructions) and chooses the longest path between the two. It should be noted that as there are only forward jumps in ASCII data, there is no chance of DAWN "looping around" in the code endlessly. If the length of the longest executable instruction sequence exceeds a certain threshold (considering all n possible entry points), then an alarm is raised.

The sketch of the detection algorithm implementing the above ideas is given below.

Algorithm 1 DetectWormASCII (printable ASCII stream A)

```
1: $D$ = disassembleInstructionsFromEveryEntryPoint($A$);
2: for startPoint $s = 1$ to size($A$) do
3:     $v \Leftarrow 0$;          // maximum length of valid instructions
4:     $\Pi \Leftarrow \emptyset$;          // set of properly populated registers
5:     RecursiveDetect($D$, $s$, $v$, $\Pi$);
6:     if $v > threshold$ then
7:         Raise Worm Alert;
8:     end if
9: end for
```

Algorithm 2 RecursiveDetect(disassembled instructions D, entry point s, max valid length v, populated register set Π)

```
1: $i_s \Leftarrow D[s]$;        // instruction starting at byte $s$
2: $s_{next} \Leftarrow s + length(i_s)$;        // location of the next instruction
3: if status($i_s$) $\in$ (invalid, truncated) then
4:     return;
5: else if $i_s$ is a privileged instruction or accesses memory with an
   inappropriate segment override prefix then
6:     return;
7: else if $i_s$ is a single-register or register→stack instruction (e.g.
   inc, dec, push) then
8:     $v \Leftarrow v + 1$ ;
9:     RecursiveDetect($D$, $s_{next}$, $v$, $\Pi$);
10: else if $i_s$ is a immediate→register or stack→register instruction
    (e.g. pop, popa) then
11:     $v \Leftarrow v + 1$ ;
12:     $\Pi \Leftarrow \Pi \cup$ (destination registers);        // Initialization
13:     RecursiveDetect($D$, $s_{next}$, $v$, $\Pi$);
14: else if $i_s$ is a register–memory operand instruction (e.g. xor) then
15:     $\Sigma \Leftarrow$ memory-accessing registers;
16:     if $\Sigma \nsubseteq \Pi$ then
17:         return;
18:     else
19:         $v \Leftarrow v + 1$ ;
20:         $\Pi \Leftarrow \Pi \cup$ (destination registers);        // Initialization
21:         RecursiveDetect($D$, $s_{next}$, $v$, $\Pi$);
22:     end if
23: else if $i_s$ is control-flow instruction (e.g. jne, jae etc.) then
24:     $v_{target} \Leftarrow v_{fallthrough} \Leftarrow v$;
25:     $\Pi_{target} \Leftarrow \Pi_{fallthrough} \Leftarrow \Pi$;
26:     RecursiveDetect($D$, $s_{target}$, $v_{target}$, $\Pi_{target}$);
27:     RecursiveDetect($D$, $s_{fallthrough}$, $v_{fallthrough}$, $\Pi_{fallthrough}$);
28:     $v \Leftarrow max(v_{target}, v_{fallthrough})$;   // Also set $\Pi$ accordingly
29: end if
30: return;
```

C. Evaluation

The effectiveness of DAWN was evaluated by running the experiments in a Linux machine with an Intel(R) Pentium-IV 2.40 GHz CPU with 1GB of RAM. For creating the ASCII worms, the frameworks provided by RIX [3] and Eller [1] were used to convert binary buffer overflow programs into their ASCII counterparts. For creating the benign dataset, approximately 0.5 MB of real web traffic were collected using Ethereal. After stripping off the headers, 100 cases, each containing approximately 4K printable ASCII characters, were selected to serve as the benign data.

The MEL threshold was chosen to be 40 by observing the MEL distribution in the benign data set. In our limited empirical experiments, this threshold caught *all* the worms but no benign stream got wrongly classified, thus yielding zero false positive and zero false negative rates. The MELs for benign and malicious test data are compared in Figure 2. While none of the MELs exceed 40 (our estimated threshold) for the benign data, for the ASCII worm data all the MEL figures are greater than 120, thereby marking a clear differentiator.

V. Binary Detectors Ineffective For ASCII

In this section, it is shown that even though ASCII is a subset of binary, binary worm detectors (especially disassembly-based and frequency-based ones) do not work well for ASCII worms. For disassembly-based detectors, particular attention is paid to the Abstract Payload Execution (APE [4]) method because it introduced the concept of MEL. APE detected binary worms by finding their NOP sled, which had a high MEL. However, sleds are almost obsolete now [10], and as a result, APE's effectiveness is severely dwindled today. Without the sled, APE is unlikely to detect the binary worm by catching the decrypter either, because a binary decrypter can be very short and thus have a low MEL. While an ASCII decrypter does have a high MEL, it is observed that APE, in its current form, is not effective for detecting that either. This is because APE, which was designed for binary worms, did not exploit the ASCII-specific properties. The definition of invalid instruction in APE is narrower than ours; APE considered an instruction invalid only when it is either incorrect or has a memory operand accessing an illegal address. This is a special case of our definition; we introduce new ways to invalidate more instructions in text (like I/O instructions etc.). Moreover, the APE paper [4] did not present any *specific* method to determine which instructions are valid and which are invalid. Finally, APE runs on random samples of data, while we examine the full content. We implemented an APE-like algorithm (which we refer to as APE-L) that did not exploit the ASCII-specific criteria that we presented in this paper, and compared the detection sensitivity in table I and figures 2 and 3. As expected, the ranges of MEL for malicious and benign are distinct for DAWN but overlapping for APE-L.

The high frequency of jump instructions in ASCII data is another reason why disassembly-based binary worm detectors do not work well for ASCII. Since every branch forks an execution path into two, having too many of them increases the number of execution paths to be searched exponentially. So, unless the ASCII-specific criteria is used to invalidate instructions to prune this search space, detectors may take very large time to run for ASCII data. This finding is corroborated by the observation that compared to DAWN, APE-L runs much slower for ASCII (see table II), to the extent that for some cases APE-L does not even terminate for hours.

Frequency-based detectors are not suitable against ASCII either, as Koleshnikov *et al* [6] showed how an ASCII worm could easily evade a powerful and robust detector like PAYL [5]. However, running DAWN on the same portion of

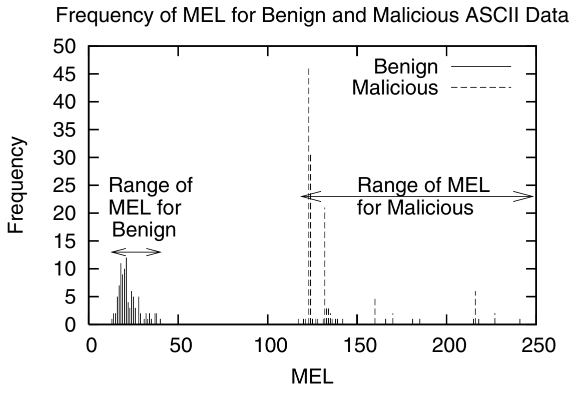

Fig. 2. Comparison of frequency charts of maximum valid instruction sequence length for benign and malicious ASCII traffic in DAWN

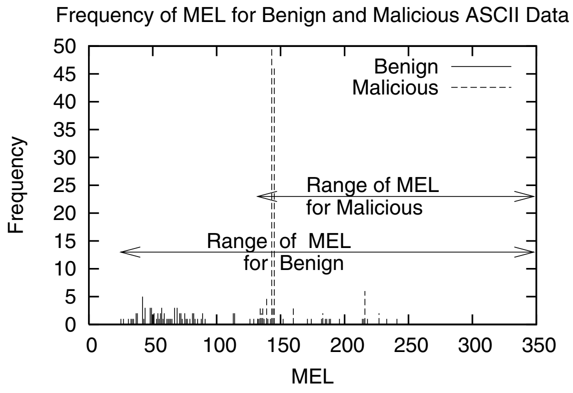

Fig. 3. Comparison of frequency charts of maximum valid instruction sequence length for benign and malicious ASCII traffic in APE-L

the worm as shown in their paper [6] resulted in a high MEL of 78, and an alarm was raised. Finally, the malicious ASCII data used in our experiments were also run past commercial malware detector McAfee but no alarm was raised.

Sensitivity	MEL Avg		MEL Range	
	DAWN	APE-L	DAWN	APE-L
Benign	22.5	73.7	13 − 46	25 − 359
Malicious	138.1	152.9	117 − 327	132 − 353

TABLE I
COMPARISON OF DAWN AND APE-L FOR DETECTION SENSITIVITY

Performance	Runtime Avg		Runtime Range	
	DAWN	APE-L	DAWN	APE-L
Benign	0.58s	22.0s	0 − 1s	0 − 3hr
Malicious	0.23s	0.3s	0 − 1s	0 − 2s

TABLE II
COMPARISON OF PERFORMANCE (RUNTIME) FOR DAWN AND APE-L

VI. LIMITATIONS AND CONCLUSIONS

Here we discuss some of the limitations of our detection strategy. It can be contended that an ASCII worm may have a relatively short decrypter by *dynamically creating* a loop. To achieve that in an ASCII worm, one needs to first insert the *loop* opcode or a negative displacement for a *jump* statement dynamically, and then proceed to enjoy the benefits of a small decrypter. However, we envisage two problems with this argument. *First*, we predict that the dynamic insertion itself would increase the number of valid instructions in the decrypter significantly due to the lack of the opcodes in ASCII. However, the bigger problem is that in absence of a one-to-one correspondence between the binary domain and ASCII domain, the decryption logic will *not* be simple, which again would increase the size of the decrypter code. The issue of obtaining one-to-one correspondence through multilevel encryption (Russian doll architecture) is discussed below.

Suppose the binary worm is first converted into ASCII, and then this ASCII worm is re-encrypted in such a way that the output is yet again ASCII. Since the second encryption is happening *within* the ASCII domain, it can be envisaged to use a short decrypter employing one-to-one correspondence. Although it is impossible to consider all encryption schemes, we show why encryption even *within* ASCII domain is complicated by demonstrating the case of using `xor`, which is a favorite choice for encryption. We observe that there is no *single* decryption key (an ASCII byte) with the property that `xor`-ing it with any other ASCII byte will still yield ASCII data. In fact, when two ASCII characters are `xor`-ed, for 33% of all possible cases the result is not ASCII. So, in order to use `xor`, a *constant* decryption key cannot be used for all of the ASCII cleartext. Consequently, the complexity of the decryption logic will increase, leading to a larger decrypter.

To conclude, we have proposed and successfully implemented DAWN, a worm detector specifically for the ASCII worms. It is signature-free, capable of detecting zero-day polymorphic ASCII worms, fast and easily deployable.

REFERENCES

[1] R. Eller, "Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel platforms," *http://community.core-sdi.com/~juliano/bypass-msb.txt*, 2003.
[2] X. Wang, C. Pan, P. Liu, and S. Zhu, "A Signature-free Buffer Overflow Attack Blocker," *In Proc. of* 15^{th} *USENIX Security Symposium*, July 2006.
[3] RIX, "Writing IA32 Alphanumeric Shellcodes," *Phrack*, 2001. [Online]. Available: http://www.phrack.org/issues.html?issue=57&id=15#article
[4] T. Toth and C. Kruegel, "Accurate buffer overflow detection via abstract payload execution," *In Proc. of* 5^{th} *International Symposium on RAID*, October 2002.
[5] K. Wang and S. Stolfo, "Anomalous Payload-based Network Intrusion Detection," *RAID 2004: In Proc. of* 7^{th} *Internation Symposium on Recent Advances in Intrusion Detection*, September 2004.
[6] O. Kolesnikov and W. Lee, "Advanced polymorphic worms: Evading ids by blending in with normal traffic," *Technical report, Georgia Tech*, 2004.
[7] "Intel Architecture Software Developers Manual, Basic Architecture," vol. 1, 1999.
[8] S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient Techniques for Comprehensive Protection from Memory Error Exploits," *In Proc. of the* 14^{th} *USENIX Security Symposium*, July 2005.
[9] R. Chinchani and E. V. D. Berg, "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows," *In Proceedings of RAID 2005*, September 2005.
[10] J. Crandall, S. Wu, and F. Chong, "Experiences Using Minos as A Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities," *In Proc. of DIMVA*, July 2005.