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

***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 text, *i.e.*, printable ASCII characters only. We show that the existing worm detectors often either do not examine the ASCII stream or may not be suited to *efficiently* detect the worm in the ASCII stream due to the structural properties of the ASCII payload. In this paper, we methodically analyze the potentials and constraints of the ASCII worms vis-a-vis their binary counterpart, and formulate detection techniques that would exploit their inherent limitations. We introduce DAWN, a novel ASCII worm detection strategy which 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

Among various malware, computer worms interest the security analysts immensely due to their ability to infect millions of computers in a very short period of time. In recent years, both sophistication and damage potential of worms have increased tremendously. Thus, we have observed the plain, single-packet, unencrypted worms getting transformed into complex, stage-loaded, encrypted entities and finally emerging as polymorphic (or metamorphic), self-mutating worms.

ASCII worms, which consist of entirely printable ASCII characters (or *text*), come as one of the latest generations of the self-mutating worms. They are very appealing since they can sneak in places where a worm is not expected to be able to get in under normal circumstances. To elaborate the point, there are cases where the server expects certain kind of traffic to be strictly text, as is the case with many important applications working with HTTP and XML. Examples include the URL in a HTTP request, or the email traffic. To ensure that only the text characters get in at times when text is expected, these servers usually employ ASCII filters[5] which drop any binary input (by binary, we mean containing both printable and non-printable characters). This filtering results in a beneficial side effect of eliminating worms that exploit vulnerability in the execution paths for processing text, since worms are *usually* binary. There are other cases too where the ASCII stream is bypassed not by the server but by the worm detector guarding it. For example, SigFree [12] usually keeps the ASCII filter on, since processing the ASCII traffic degrades its performance significantly. 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[9], and later Eller[5] 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 to these servers, 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, certain payload-based detection mechanisms may not be adequately suited for efficiently detecting an ASCII worm due to the structural properties of ASCII data. We 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[10]), and 2) that detects by looking at the frequency distribution and other statistical properties of the payload (e.g. PAYL[11]). There are two potential problems with the former scheme. First, almost *any* ASCII string translates into a syntactically correct sequence of instructions, which means checking for syntactic validity is of little value for detecting ASCII worms. Second, since *most* of the branch opcodes are already ASCII, the *proportion* of branch instructions for ASCII data is significantly higher than that of binary data. 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 doing instruction disassembly. In other words, to ensure quick detection of worms in ASCII data, one must find novel ways to prune the number of the paths to be inspected. Regarding the other detection approach of examining the frequency distribution and other statistical properties of the payload, there have been instances where ASCII worms have been shown to have successfully evaded such detectors. For example, Kolesnikov *et al* [6] have shown 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 commercial malware detectors and no alarms were raised. Therefore, we conclude that the threat of ASCII worm is real, and we can ignore them only at our own peril.

Our main contribution in this paper is twofold. First, we analyze the structural and behavioral traits of ASCII worms and come up with an efficient detection strategy for them. Our second contribution is in providing the mathematical foundation of the notion of *MEL* (*Maximum Executable Length*) of a valid stream of instructions – a concept that has been used by a number of existing worm detection schemes (e.g. APE[10], STRIDE[2]) previously without explaining the mathematical rationale behind it. To the best of our knowledge, we are the first to provide a solid probabilistic model to this concept, and we show how the MEL thresholds can be calculated *mathematically* (as we do in our detection strategy) rather than obtaining them *experimentally*.

The structure of the rest of the paper is as follows. Section 2 provides the details of ASCII worm. We focus on the constraints and weaknesses of the ASCII worm in Section 3 and devise a scheme to detect it. In Section 4, we describe the underlying probabilistic model that not only forms the solid foundation of our own detection strategy but also explains the mathematical rationale behind some existing detection schemes. In Section 5, we lay down the implementation details of our detection method and evaluate the results in Section 6. In Section 7, we compare our work with others, and finally in Section 8, we discuss the limitations and resilience to evasion of our scheme, concluding with remarks regarding future work.

II. Inside the ASCII Worm

In this section, we start with a formal definition of the ASCII worm and then proceed towards discovering their advantages and limitations. We also discuss the typical construction of an ASCII worm. The treatment of the ASCII worms in this section forms the foundation of the detection strategies detailed in the next section.

A. Definitions and Terminologies

We use the following definitions throughout this article:

- **ASCII data**: data consisting of only *text*, or *printable* ASCII characters (0x20 through 0x7E). This is the same set of characters which can be entered using a keyboard. A worm whose payload consists of entirely ASCII data is called an ASCII worm.
- **Binary data**: character stream consisting of printable as well as non-printable characters. Worms whose payload contains these characters are called Binary worms.
- **Valid (or invalid) instruction**: an instruction that will not (or will) cause the running process to abort.
- **Maximal valid instruction sequence**: the longest sequence of consecutively-executed valid instructions in an ASCII (or binary) stream. Its length is denoted as **LMVI** (for *Length of Maximal Valid Instruction sequence*).

The concept of valid instruction was introduced in [10] for detecting the buffer-overflow binary worms. The definition there is narrower than ours; it considered an instruction invalid only when it has a memory operand accessing an illegal address, which is a special case of our definition. Moreover, the paper did not present *specific* method to determine which instructions are valid and which are invalid. Finally, it addressed the problem in the context of classical binary worms rather than ASCII worms (as we do in this paper), which has many intriguing properties that we can successfully exploit.

B. Opcode Availability for ASCII Worms in Intel Architecture

The available Intel opcodes in the ASCII data are:

- Dual-operand register/memory manipulation opcodes: *sub*, *xor*, *and*, *inc* and *imul*

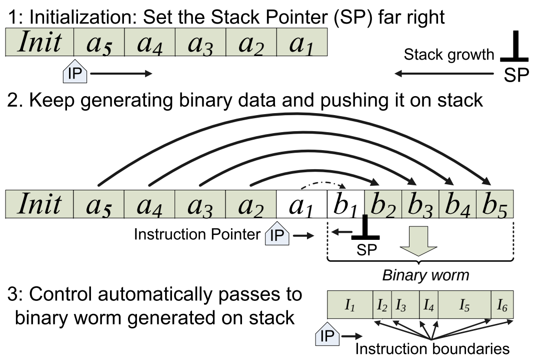

Fig. 1. Creation of a binary worm from ASCII code on stack

- 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*

Therefore, we observe that there is a reasonable number of useful opcodes that are available in the ASCII data. Using these ASCII opcodes intelligently, one can also simulate many other non-available opcodes. For example, although the opcode *move* is not available, one can easily simulate *move eax, ebx* by doing *push ebx* followed by *pop eax*. In the next subsection, we show how a typical ASCII worm can be constructed.

C. Construction of a Typical ASCII Worm

While it is impossible to enumerate all the possible ways to create an ASCII worm, it is easily seen that in order to create a potent worm (that uses system calls etc. in order to open sockets for propagation), one needs to have opcodes which consist of non-printable characters. Therefore, the only way to create such an opcode is to generate one on-the-fly (preferably on the stack). A typical construction method (known as the “stack” method) is described as follows. Suppose the actual n-word binary worm that is to be generated is given by $B = b_1 b_2 ... b_n$, and the ASCII code segment generating the binary word b_i is denoted by a_i. Since the Instruction Pointer *IP* and Stack Pointer *SP* move in opposite directions, first b_n is generated by a_n, then b_{n-1} is generated by a_{n-1}, and so on. Finally, b_1 is generated by a_1 to its immediate right, which means now the control can automatically pass on to the binary worm. Thus, a typical ASCII worm looks like $I a_n a_{n-1} ... a_3 a_2 a_1$, where I denotes the *initialization* code.

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*, an instruction sequence (also in printable ASCII) 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 Strategies for ASCII Worms

In this section we start with a description of the *constraints* that are imposed on an ASCII worm, and then identify the resulting behavioral characteristic that it has in order to overcome these constraints. Once we obtain the behavioral characteristic, we proceed towards devising a practicable detection scheme exploiting that characteristics.

A. Constraints of an ASCII Worm

- **Opcode Unavailability:** We have 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, we see that 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, 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, we observe that since the ASCII domain is a *proper* subset of the binary domain, we cannot have a *one-to-one* correspondence between the two. Therefore, if we are encrypting one byte of binary data into ASCII, the size of the output will be definitely *more* than one byte, which means that the size of the encrypted payload will increase. This somewhat defeats the 3rd goal. Moreover, without the one-to-one correspondence the decryption logic will be more complex, thus resulting in a larger decrypter.
- **Control Flow Constraints:** Usually decryption routines are 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, using *jump* statements, one can only go forward in an ASCII worm but not backwards. As a result, the option of having a small decrypter routine is precluded. While it is theoretically possible to overcome this difficulty by generating the negative displacement dynamically, that would most likely make the decrypter more complicated and increase its size (we will discuss this issue in greater detail in Section 8).

Therefore, we observe that ASCII worms have the following weakness:

> *Due to opcode unavailability, if an ASCII worm is to attain the same potency of a binary worm, then it* must *create the corresponding binary code by* ***self-mutating****, i.e., by dynamically producing the binary code on-the-fly and writing the code in appropriate memory locations. While self-mutation is only an* ***optional*** *obfuscation tool for* ***binary worms****, it is a* ***mandatory constraint*** *for the* ***ASCII worms****.*

Due to the weakness mentioned above, an ASCII worm has to self-mutate to produce binary code so that it can utilize the opcodes required for its propagation. So, this binary code (say of size n words) must be produced and written in the memory. Since instructions work on at most one word at a time, the decrypter has to produce the binary code word by word. To do that, we have two options for the construction of the decrypter: 1) It consists of sequentially n "hardcoded" ASCII instruction blocks, each producing one binary word in the final code, or 2) it contains a loop over a single ASCII instruction block that is executed for n times, each time for one word. However, we have already observed that, without the availability of negative displacements, an ASCII worm does not contain a loop,[1] which means the second option is ruled out. This leaves us with the first option, which results in a relatively large decrypter with hardcoded instructions; the length of such a decrypter is $O(n)$ words.

B. Formulating the Detection Strategy for the ASCII Worm

Because a long decrypter means a long execution path, we posit that an ASCII worm will have a long sequence of valid intructions. These instructions may not occur contiguously in the instruction stream, but they must be executed one after another without raising error. For example, in an instruction stream $i_1i_2i_3i_4i_5i_6i_7$, $i_1i_2i_4i_7$ could be a valid instruction sequence if i_2 and i_4 turned out to be branch instructions pointing to i_4 and i_7 respectively. Now, can we design an ASCII worm detection scheme by disassembling the incoming ASCII stream and checking the maximum length of consecutive *syntactically-correct* and *executable* instructions against a threshold value? To answer that, this approach is viable *only* when invalid instructions frequently appear in benign ASCII data, fragmenting valid instruction sequences into much smaller blocks. Fortunately, such a property is true for benign ASCII data, as we will establish by instruction analysis below, by probability analysis in Section 5, and by experiments in Section 6. To identify the real gap between LMVI of malicious data and that of benign data, we must aggressively explore the properties of ASCII data after it is disassembled, particularly, the cases where invalid instructions are produced. To the best of our knowledge, no prior work has studied this problem in the ASCII domain.

C. Formulating the Detection Strategy for the ASCII Worm

Here, we describe why a benign ASCII traffic *cannot* have a very long sequence of valid instructions.

Prevalence of Privileged Instructions: The characters '*l*', '*m*', '*n*' and '*o*', which occur frequently in text [8] 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. 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 can't

[1]Dynamically created loops will have their own problems that are discussed in Section 9.

have 1 in their MSB, register-register instructions are ruled out. Therefore, to manipulate a register, the value must come from memory (other than *inc dec*, or *pop*). While accessing the memory, a violation 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 and most likely it will be outside of the allocated memory block for this process, which means any attempt to access that address would cause a protection exception.
- **Wrong Segment Selector:** For memory-accessing dual-operand instructions, if the instruction is preceded by a segment override prefix, and the selector in the segment register points to an area in the Global Descriptor Table which is outside the bounds for this process, then it will raise a protection exception. We have observed during our experiments that FS and GS segment prefixes usually result in a protection exception error. The reason that, if these segment registers are not needed, then they could very well contain junk data, which is likely to out-of-bound memory.
- **Explicit Memory Address:** For certain ASCII ModR/M bytes ('%', '–' , '=', '&', '.', '6', '>'), the memory address is expressed as an explicit 4-byte or 2-byte displacement. Linux randomizes the start address of each program, and there is research work [3] on randomizing static libraries in Windows, too. Therefore, it is very likely that the usage of an explicit memory address will raise a memory violation.

Based on the above analysis, we conclude that a benign stream of ASCII data is likely to have many "invalid" (error-raising) instructions dispersed in it, which is not true for malicious data because an ASCII worm has to avoid these instructions to ensure a crash-free execution. As our mathematically analysis will demonstrate, unlike ASCII worms, it is extremely unlikely for benign ASCII data to carry a long sequence of "valid" instructions. Therefore, a threshold on LMVI (length of maximal valid instruction sequence) can be used to determine whether an ASCII stream is malicious or benign. It should be stressed that our major contribution in this paper is *not* about this rather straightforward approach of setting an MEL threshold, which has appeared in previous works [10], [12], [2] for classical binary worms. Our contribution is to 1) devise new ASCII-specific techniques for identifying invalid instructions and to 2) establish analytically and experimentally the fundamental reasons for why these new techniques will enable the LMVI-threshold approach to detect ASCII worms. Below we perform a probabilistic analysis, which lays the mathematical rationale for the design of our DAWN strategy (to be described in Section 6) against ASCII worms.

IV. Probabilistic Analysis

The detection strategy we propose is as follows:

1) Find the length of the maximal valid instruction sequence (say x) for the input data
2) Compute the probability p_x that the LMVI for the given input is greater than or equal to x, and
3) If p_x is less than certain pre-set false positive limit α, then we deduce that the probability of the input data having such a high LMVI purely by chance is very less, and hence the input data must contain a worm.

We map this practical problem to the classical coin-tossing experiment as follows:

- **Head:** The event that the instruction is invalid
- **Tail:** The event that the instruction is valid
- ***p***: the probability of a head (invalid instruction)
- ***n***: the total number of trials (or instructions)
- ***N***: random variable denoting the total number of heads (or invalid instructions), which automatically also denotes the number of valid instruction sequences.

We know that a maximal valid instruction sequence will be ended by an invalid instruction, as otherwise we would have simply included that instruction in the maximal sequence. Thus, if there are N invalid instructions in an instruction sequence, then there would be an equal number of valid instruction sequences. Each valid sequence (say of of length l) would contain $l-1$ valid instructions followed by a single invalid instruction. Thus, the problem of finding the maximal valid instruction sequence is equivalent of finding two invalid instruction that are the furthest apart, with the property that all the intermediate instructions being valid ones. In other words, the probability of finding a maximal valid instruction sequence length of τ or more is answered by the following question: "If we toss a coin n times (with probability of head p), then what is the probability that the maximum inter-head distance (in number of coin tosses) is greater than or equal to τ?" We also answer the following inverse question: "what should be the appropriate threshold τ if we are willing to accept a false positive rate no more than a certain $\alpha\%$?". To answer these questions, we need to characterize the underlying distribution of events first.

A. Assessment of Randomness of the Character Stream and the Instruction Stream

We observe the fact that while the ASCII traffic is not purely random, the range of the patterns and correlations (which can be viewed as deviations from randomness) in different traffic data is *huge*. Although it is nearly impossible to account for *all* the correlations and trends for all possible types of ASCII traffic, making *reasonable* assumptions about the data can result in a much simplified model that is realistic yet easier to manage. For example, in our case we simplify the traffic model by assuming the characters in the traffic to be independently distributed. In our model, the individual probability of occurrence for each character directly follows from the overall frequency distribution of characters in the incoming character stream. Since the instructions are formed by combining a variable number of contiguous characters (which we hypothesized to be independently occurring), we further assume that the instructions also occur independently. As will be demonstrated in the experimental results section, the probabilistic behavior of our simplified model turns out to match the actual traffic behavior almost perfectly.

B. Modeling the Max. Length of Valid Instruction Sequences

We use Bernoulli trials for modeling the length of the maximal valid sequence for an instruction stream. We consider an instruction to be *invalid* when it marks the end of execution for the current instruction sequence. In our model, while classifying an instruction as invalid, we restrict ourselves to the ones which we can see to be invalid on their own (without considering their neighboring instructions). Thus, we designate the following instructions to be invalid: a) privileged instructions and b) memory-accessing instructions that would cause segmentation-faults due to a wrong Segment prefix. We do not consider the unpopulated-register case since that requires semantic study of the previous instructions. We define the term *valid instruction sequence* to be a sequence of zero or more valid instructions (*failure*s) followed by a *single* invalid instruction (*success*). This way, one can visualize the whole of the instruction stream as a collection of contiguous valid instruction sequences ($S_1 S_2 S_3 ... S_N$), with each sequence S_i ending with a single invalid instruction (except the last one). We will be using the following notations: If p denotes the probability of success (i.e. the probability of an invalid instruction), then we observe that the *p.m.f.* of N is given by $\binom{n}{N} p^N (1-p)^{n-N}$. We define the random variable X_i to denote the length of S_i. X_i follows the Geometric distribution, which is the number of trials required to obtain the first success. The *p.m.f.* and *c.d.f.* of X_i are known to be $p(1-p)^{x-1}$ and $1-(1-p)^x$ for $x = 1, 2, 3, ...\infty$. However, for our detection purpose, we consider only the *maximal* sequence (i.e., the *longest* among *all* such sequences), and denote its length by X_{max} (which itself is another random variable given by $max\{X_1, X_2, X_3, ..., X_N\}$). Due to the lack of memory property of the Geometric distribution, we can treat each $X_i (1 \leq i \leq N)$ to be occurring independently. In our proposed method, we compare X_{max} to a threshold τ to decide if an instruction stream contains malicious data or not.

C. Characterization of the Maximum Length of the Valid Instruction Sequences

Given that there are total of n instructions with Prob[invalid instruction] being p, the random variable N denoting the number of instruction sequences, and the length X_i of each valid instruction sequence following Geometric distribution with the same parameter p, we proceed to characterize X_{max} by calculating its *p.m.f.* and *c.d.f.* as follows. First we *assume* that there are N sequences and calculate the *c.d.f.* of X_{max} accordingly. However, since this *cd.f.* is *conditional* upon the number of sequences being N, we sum up the total probabilities for *all* possible values of N. Because of the lack of memory property of Geometric distribution, we treat can each X_i independently. Although we relax the condition that $X_1 + X_2 + ... + X_N = n$, it will be shown later that for realistically big values of n, the effect of this relaxation is almost unrecognizable.

For a single instance of having exactly N sequences, the *cd.f.* of X_{max} is given by

The conditional *c.d.f.* of X_{max} , i.e., Prob $[X_{max} \leq x]$

$$\begin{aligned}
&= \text{Prob } [max(X_1, X_2, X_3, ..., X_N) \leq x] \\
&= \text{Prob } [(X_1 \leq x) \text{ and } (X_2 \leq x) \text{ ...and } (X_n \leq x)\,] \\
&= \text{Prob } (X_1 \leq x) * \text{Prob } (X_2 \leq x) * ... * \text{Prob } (X_n \leq x) \\
&\quad \text{(because of lack of memory property)} \\
&= [1-(1-p)^x] * [1-(1-p)^x] * ... * [1-(1-p)^x] \\
&= [1-(1-p)^x]^N
\end{aligned}$$

With the conditional *c.d.f* for X_{max} for a single instance of N calculated above, we derive the *overall c.d.f* of X_{max} for all possible values of N (we observe that N is upper bounded by n).

The *c.d.f.* of X_{max}

$$\begin{aligned}
&= \sum_N \text{Prob}[n_{sequences} = N] \times \text{Prob}[max(X_1, ..., X_N) \leq x] \\
&= \sum_N \tbinom{n}{N} p^N (1-p)^{1-N} \times [1-(1-p)^x]^N \\
&= \sum_N \tbinom{n}{N} [p(1-(1-p)^x)]^N (1-p)^{1-N} \\
&= [p(1-(1-p)^x + (1-p)]^n \quad \text{(using binomial theorem)} \\
&= [1-p(1-p)^x]^n
\end{aligned}$$

The *p.m.f.* of X_{max} (using above *c.d.f.* formula)

$$\begin{aligned}
&= \text{Prob } [X_{max} = x] \\
&= \text{Prob } [X_{max} \leq x] - \text{Prob } [X_{max} \leq x-1] \\
&= [1-p(1-p)^x]^n - [1-p(1-p)^{x-1}]^n
\end{aligned}$$

D. Verifying our Model Using Monte-Carlo Simulation

We test the correctness of our model by running Monte-Carlo simulation. We toss a coin with Prob[Head]=p for n times. We examine all the sequences that comprise of successive Tails from the very beginning and ends with a single Head (which is the first and the only head of that sequence). We note down the maximum length of all such sequences. We repeat the procedure for 1000 rounds and plot the distribution of the maximum length for different values of n (1000, 5000 and 10000) and p (0.125, 0.175 and 0.3). We plot the *p.m.f* of the maximal length from our Monte-Carlo distribution against the *p.m.f* of X_{max} as obtained from our probabilistic model on the same graph in Figure 2. We observe a near-perfect match between the two for all the different instances, which justifies the correctness of our probabilistic model for characterizing X_{max}, and vindicates our assumption of independence among the X_i's. We also see that the threshold τ corresponding to a fixed α increases with the size of the input character stream (n) and decreases with p, which are in line with our expectation.

E. Automated Determination of the Threshold Corresponding to a Pre-set Rate of False Positive

For a given p and n, the false positive rate α corresponding to a threshold τ = Prob$[X_{max} > \tau] = 1-$Prob$[X_{max} \leq \tau] = 1-(1-p(1-p)^\tau)^n$. Conversely, we can derive a threshold τ corresponding to a given probability of false positive (α) as

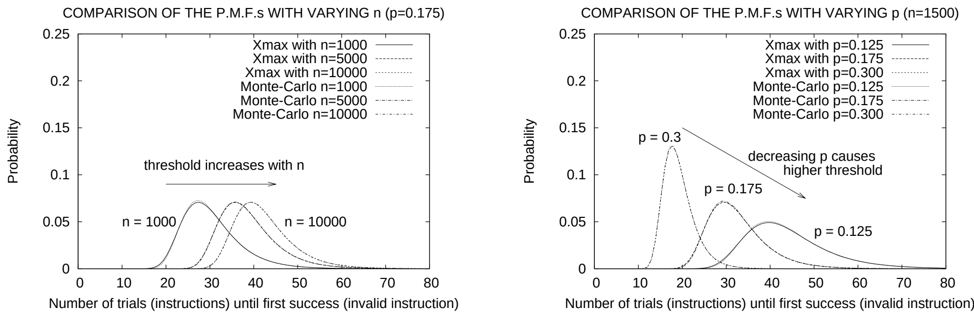

Fig. 2. Juxtaposition of the *p.m.f.*s for the length of the maximal valid instruction sequence from the probabilistic model and from the Monte-Carlo simulation. A near-perfect match can be observed.

$\tau = \frac{\log(1-(1-\alpha)^{\frac{1}{n}})-\log p}{\log(1-p)}$. Therefore, we see that for a given input character stream (n) and a character frequency table (yielding p), we can *directly* calculate the value of the threshold τ corresponding to a user-given false positive rate of α. This is the *biggest* strength of our detection scheme, as we do not need to *tune* any parameter to get the desired result – all the parameters are calculated *automatically* without any human intervention ever – there is no "parameter tuning".

Bolstered with the mathematical backbone of our model, we come up with a detection strategy that exploits the differentiators between benign and malicious ASCII data to catch the ASCII worms.

V. Implementation of DAWN

In this section, we lay down the details of DAWN, the detection strategy for ASCII worms. Briefly, DAWN operates into two main stages: instruction disassembly and instruction sequence analysis. First we disassemble the ASCII input from every possible position and then we attempt to see if any such disassembly could potentially lead to a malicious code. If we detect a long sequence of valid instructions (longer than a certain automatically-calculated threshold) that involves memory-writing operations, then we raise an alert. The individual steps are delineated in more detail in the following two subsections.

A. Step 1: Instruction Disassembly

Since we cannot predict the entry point of the worm in the input stream, we need to disassemble the input (say of length n) from all possible n entry points. Although it appears to be a $O(n^2)$ process, actually it is linear. It has been shown [4] that if we start 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, a process that is clearly linear.

B. Step 2: Instruction Sequence Analysis

The main purpose of this stage is to find out how long an instruction sequence (which may start anywhere within the ASCII data) may proceed without generating an error. The error may come either from using a privileged instruction, or from a memory access violation (please refer to section 4.3 for details). As we proceed, we keep track of which registers have been initialized properly. When an uninitialized register is used to address memory (as source or destination), we consider it to be the end of that sequence. For a control flow bifurcation (like *jump*), we recursively consider both the possible routes (*jump target* as well as the *fall-through* instructions) and choose the longest path among the two. We also observe that since there are only forward jumps in ASCII data, there is no chance of "looping around" in the code endlessly. If the length of the longest instruction sequence exceeds a certain threshold (considering all n possible entry points), then we raise an alarm.

The sketch of the detection algorithm implementing the above ideas is given in the next page.

VI. Evaluation

The task of evaluating the effectiveness of DAWN in an experimental setup consisted of the following steps: 1) Creation of the test data, 2) determining the appropriate thresholds from the created test data, 3) running the detection algorithm with the thresholds determined in the previous step, and 4) assessment of the effectiveness of the detection strategy by observing the false positive and false negative rates. The tests were run on an Intel(R) Pentium-IV 2.40 GHz CPU with 1 GB of RAM in a Linux machine.

A. Creation of the Test Data

For creating the ASCII worm, the frameworks provided by RIX [9] and [5] convert multiple binary buffer overflow programs into their ASCII counterpart and more than one hundred ASCII worms were created out of it. The effectiveness of the ASCII worms were tested by actually running the vulnerable program and then by observing the spawning of the shell. For creating the benign dataset, nearly 0.5 MB of real web traffic were collected using Ethereal. After stripping off the headers, 100 cases, each containing nearly 4K printable ASCII characters, were selected to serve as the benign data.

B. Determining Appropriate Thresholds for the Test Data

Since we can express the threshold τ as a function of the probability of false positive α (as $\frac{\log(1-(1-\alpha)^{\frac{1}{n}})-\log p}{\log(1-p)}$) with parameters p (probability of an invalid instruction) and n (total number of instructions), we needed to first calculate the values of those *parameters* p and n for our test data. For the calculations, we use only the following two entities: 1) the length of the instruction stream (in number of characters)

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:     $\Pi \Leftarrow \phi$;          //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 i, max valid length v, populated register set Π)

```
1: $i_s \Leftarrow D[i]$        // instruction starting at byte $s$
2: if status($i_s$) $\in$ (invalid, truncated) then
3:     return;
4: else if $i_s$ is privileged or accesses memory with an inappropriate segment override prefix then
5:     return;
6: else if $i_s$ is a single-register or register$\rightarrow$stack instruction (e.g. inc, dec, push) then
7:     $v \Leftarrow v + 1$ ;
8:     RecursiveDetectWorm($D$, $i_{next}$, $v$, $\Pi$)
9: else if $i_s$ is a immediate$\rightarrow$register or stack$\rightarrow$register instruction (e.g. pop, popa) then
10:    $v \Leftarrow v + 1$ ;
11:    $\Pi \Leftarrow \Pi \cup$ (destination registers)        // Initialization
12:    RecursiveDetect($D$, $i_{next}$, $v$, $\Pi$)
13: else if $i_s$ is a register–memory operand instruction (e.g. xor, and, sub) then
14:    $\Sigma \Leftarrow$ memory-accessing registers
15:    if $\Sigma \not\subseteq \Pi$ then
16:        return;
17:    else
18:        $v \Leftarrow v + 1$ ;
19:        $\Pi \Leftarrow \Pi \cup$ (destination registers)        // Initialization
20:        RecursiveDetect($D$, $i_{next}$, $v$, $\Pi$)
21:    end if
22: else if $i_s$ is control-flow instruction (e.g. jne, jae etc.) then
23:    $v_{target} \Leftarrow v_{fallthrough} \Leftarrow v$;
24:    $\Pi_{target} \Leftarrow \Pi_{fallthrough} \Leftarrow \Pi$;
25:    RecursiveDetect($D$, $i_{target}$, $v_{target}$, $\Pi_{target}$)
26:    RecursiveDetect($D$, $i_{fallthrough}$, $v_{fallthrough}$, $\Pi_{fallthrough}$)
27:    $v \Leftarrow max(v_{target}, v_{fallthrough})$   //Set $\Pi$ accordingly
28: end if
29: return;
```

and 2) character frequency table (indicating probability of occurrence for each character), which can either be pre-set (from experience) or can be obtained by a linear sweep of the character stream in case no pre-set data exists (like our test condition). We do not need to disassemble any data for determining p or n.

Determining *p*: As mentioned earlier, p denotes the probability of an instruction that causes the execution to abort. However, among the different kinds of instructions that we identified to be causing the abort, the first two kinds (privileged instructions (I/O) and wrong-Segment-override-memory-operand instructions) are *always* invalid, irrespective of their relative position in any instruction sequence. Hence the probability of them occurring can be *directly* measured from the character frequency table. The third kind of invalid instruction (illegal memory access due to unpopulated memory-addressing register) requires evaluation of prior instructions and thus, it cannot be determined standalone whether it will cause an abort or not. Therefore, we calculate the p by summing up the Prob [I/O instruction] and Prob [wrong-Segment-override memory-accessing instruction] (which are 18.5% and 4.2% according to the frequency distribution of our test data), and ignoring the third component since that can only be estimated and not measured probabilistically. While it is our understanding that the third component will indeed cause a lot of instructions to be invalidated (and thus help our cause), we still take the conservative view and use the *lower bound* of p obtained from first two components only. Thus, p turns out to be 0.185 + 0.042 = 0.227 in our case.

Determining *n*: Since we do not perform any kind of disassembly for *estimating* this parameter, n is not known to us initially. Therefore, we derive n by dividing C (the total number of input characters) with the expected length (in number of characters) of an instruction. Now, E[length of instruction] = E[length of prefix chain] + E[length of *actual* instruction] (by *actual* instruction, we mean the rest of the instruction *after* the prefix chain, starting with the instruction opcode). Now, if *scp* denotes the probability that a character is one of the instruction-prefix characters (0.16 in our case), then E[length of prefix chain] = $\sum_{i=0}^{\infty} i \times$ Prob[length(prefix chain) $= i$] $= \frac{scp}{(1-scp)} = 0.19$. Similarly, E[length of actual instruction] = $\sum_i$ length[instr(i)] $\times$ Prob[instr(i)], which was found to be 2.4 for us (instr(i) represents an ASCII instruction like `xor`, `sub`, `inc`, `pop` etc. Thus, E[length of instruction] turns out to be 0.19 + 2.4 $\approx$ 2.6 bytes per instruction. Since C = 4000 in our case, $n = \frac{C}{E[\text{instruction size}]} = \frac{4000}{2.6} \approx 1540$. For these $n = 1540$ and $p = 0.227$, *c.d.f.*s of the length of a *single* valid instruction sequence (X_i) and the *maximum* length of *all* such sequences (X_{max}) are shown in the Figure 3 in the next page.

Determining the threshold τ: For a preset error (false positive) rate of $\alpha = 1\%$, we calculate the threshold $\tau = \frac{\log(1-(1-0.01)^{\frac{1}{1540}})-\log 0.227}{\log(1-0.227)} = 40$ for $n = 1540$ and $p = 0.227$. In our limited empirical experiments, this threshold of 40 catches *all* the worms and does not result in any false positive.

C. Experimental Results and Assessing the Effectiveness of the Detection Methodology:

Our detector catches *all* the ASCII worms and not a single benign payload gets termed as malicious, thus yielding zero false positive and zero false negative rates. We compare the maximum length of the valid instruction sequence for benign and malicious test data in Figure 4(a). For the benign data, the maximum length hovers around 20, and none above 40 (the calculated threshold), which matches our expectations very well. On the other hand, for the ASCII worm data, the maximum length figure goes well above 120, thereby marking a clear differentiator. Thus, we observe from Figure 3(b) that the gap between the false positive boundary (p value of 0.227 corresponding to LMVI of 40) and false negative boundary (p value of 0.073 corresponding to LMVI of 120) is quite

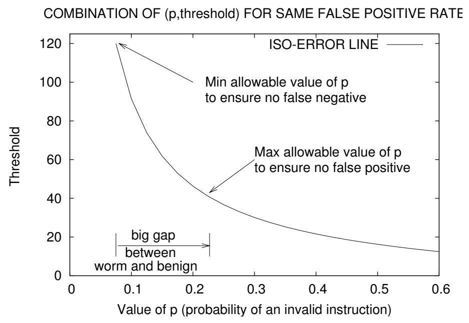

Fig. 3. Correlation between τ and p for maintaining same error (false positive) rate $\alpha = 1\%$.

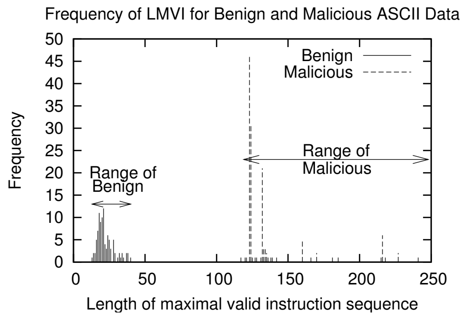

Fig. 4. Comparison of frequency charts of maximum valid instruction sequence length for benign and malicious ASCII traffic in DAWN

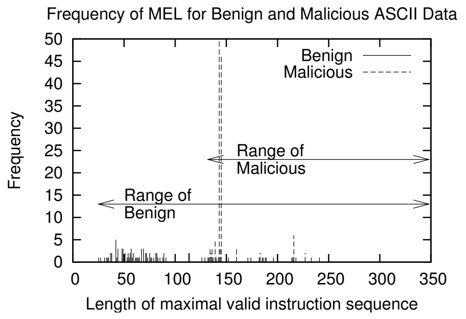

Fig. 5. Comparison of frequency charts of maximum valid instruction sequence length for benign and malicious ASCII traffic in DAWN

large, which means even if the estimated p changed by a small margin, we would still have been able to distinguish the worms from the benign data. Also, the average instruction length from our actual experiment (2.65) was found to be very close to our expected value (2.6) assuming character and instruction independence.

VII. Comparing Our Work with Others

Since our detection mechanism is completely payload-based, we only compare our work with other schemes that are payload-based employing MEL-like strategies. We contrast our method with 1) SigFree[12], 2) Abstract Payload Execution[10] and briefly consider the evasion strategy provided by [6].

SigFree[12] is a zero-day buffer-overflow detector that detects the worm by counting the length of only the *useful* instructions in an instruction sequence, while our approach counts all executable instructions, irrespective of whether they are useful or not. SigFree incurs significant computational overhead in order to examine the ASCII traffic, as a result of which it *usually* bypasses the ASCII traffic to enhance performance. However, in our case, processing the ASCII stream is very fast. Finally, one of the criteria for measuring the *usefulness* of an instruction in SigFree is that it must not have any data anomaly, as for example by checking if the sources have been properly populated or not. However, we show in the following example that it may be possible to make SigFree think that the data anomalies have happened while actually none happened. One of the data anomalies that SigFree reports are *undefine-reference*, which happens when a variable, which is not yet *defined* (*i.e.*, not populated properly), is *referenced* (used as source) again. SigFree posits that the state of an undefined variable remains undefined when its value is reset with an undefined value. However, we show that even by using the undefined variable as a source, one can properly define, i.e., initialize a variable. The following example would make it clear. Suppose the register variable `eax` initially contains junk value, which implies `eax` is in *undefined* state. Now if we do `and eax, 0x20202020`, followed by `and eax, 0x40404040`, according to SigFree `eax` will still remain in *undefined-reference* state since the source register referenced in this case, `eax`, was in *undefined* state. However, the two instructions mentioned above actually sets `eax` to zero irrespective of its previous content, which means it is possible to reach a *defined* (initialized) state even with an undefined reference. In DAWN, we overcome this flaw by taking the conservative approach that any undefine-reference is also a potentially defined state, thus increasing the detection probability.

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

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

TABLE I
Comparison of the LMVI (for DAWN) and MEL(for APE-L) figures for their sensitivity of detection and runtime analysis

Abstract Payload Execution (APE) [10] first introduced the concept of MEL for detecting the binary worms exploiting buffer-overflow. However, it did not provide any mathematical foundation of the underlying model, which we do in this paper. As a result, any MEL thresholds in APE is obtained *experimentally*, while in our case the threshold is calculated automatically by the model – there is no "parameter tuning". Moreover, in APE the detection method is run on random samples of data as the focus is on finding the sled, 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 and runtime in the table on the right. As expected, the range of LMVI (or MEL) for benign and malicious is distinct for DAWN but *not* for APE-L. However, without using the ASCII-specific criteria APE-L is much slower. In fact, for some cases APE-L does not even terminate for hours due to the exponential state space it has to search because of the excessive number of *jump*s in ASCII. Therefore, it is imperative that we exploit as many ways as possible to prune the search space.

In **[6]**, Koleshnikov *et al* displayed part of an ASCII worm that evaded the APE[10]. However, running DAWN on the same portion of the worm resulted in an LMVI of 78, which was way higher than the benign traffic and thus DAWN was able to catch it easily.

VIII. Limitations, Extensions and Conclusions

In this section we discuss some of the limitations of our detection strategy. We reiterate the basic principle of our detection method: 1) An ASCII worm must self-mutate to generate binary opcodes, 2) this mutation requires a lots of memory-writing instructions, 3) due to difficulties in encryption (including unavailability of loops) the size of a decrypter is relatively big for ASCII worm, 4) due to randomness property, benign ASCII data does not have such a long *executable* instruction sequence, and 5) the length of the maximal valid instruction sequence can thus be used to differentiate between benign and malicious data. A possible argument against this logic (along with our counterarguments) is as follows:

One can contend that it might be possible for an ASCII worm to 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 doing this 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 multilevel encryption, which does ensure one-to-one correspondence, is discussed below.

As mentioned above, to achieve the coveted one-to-one correspondence (which makes the task of encryption and decryption much simpler), one may also propose using the Russian doll architecture in the following manner. First, convert the binary worm into ASCII, and then encrypt this ASCII worm in such a way that the output is yet again ASCII. We observe that in the second step, we are doing encryption within the *same* ASCII domain, which signals the possibility of a one-to-one correspondence. On the surface, this approach appears to have merit since 1) the final encrypted ASCII data will show very little trend of an ASCII worm, and 2) because of the one-to-one correspondence, one may be able to use *simple* decryption schemes, which means a short decrypter. We put forth our rebuttal to this argument by demonstrating the case of using `xor`, which is usually a favorite choice for encryption. First of all, 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. This is because the ASCII data (0x20–0x7E) occupies a somewhat *odd* slot in the original ASCII table, and `xor`-ing two characters from ASCII data often yields a result that is not ASCII. To show that, we divide the 95-char ASCII domain into three almost equal-sized parts (viz. 0x20–0x3F, 0x40–0x5F, and 0x60–0x7E). We see that if we `xor` *any* two bytes from the *same* part, then the output will belong to the non-ASCII domain 0x00-0x1F. This means that, in order to use `xor`, we cannot use a *constant* decryption key for all of the ASCII cleartext. Consequently, the decryption logic will have to be more complex too, leading to a *not-so-small* decrypter.

We emphasize that while we offer a novel way to differentiate between benign and malicious ASCII traffic, this only means that we have merely made the task of an attacker significantly harder. As per our limited experiment, the difference between the maximum length of the valid instruction sequence between benign and malicious traffic is currently significantly large. To the best of our knowledge, no ASCII worms employing decryption to this date has been able to come up with a decrypter smaller than our current threshold. However, as security is a cat-and-mouse game, in future we will invariably see such worms, and we must strive to find more exploits to counter that.

We would like to reiterate that while our approach is similar to some other existing schemes[10], [12], [2], some fundamental differences exist. Probably the biggest difference is that, in our detection scheme all the parameters and thresholds are obtained purely from the statistical properties of ASCII traffic rather than obtaining them experimentally. This provides a concrete foundation of our model.

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. Finally, the facts that its design has got a solid mathematical foundation and that it does not require any parameter tuning make it extremely robust.

References

[1] Aleph One. “Smashing The Stack For Fun And Profit”. Phrack, 7(49), November 1996.

[2] Akritidis, P., Markatos, E. P., Polychronakis, M. and Anagnostakis. K. Stride: “Polymorphic sled detection through instruction sequence analysis”. In Proceedings of the 20th IFIP International Information Security Conference (May 2005).

[3] Bhatkar,S., Sekar, R., and DuVarney, D.C.“Efficient Techniques for Comprehensive Protection from Memory Error Exploits”. In Proceedings of the 14th USENIX Security Symposium (July 2005).

[4] Chinchani, R., and Berg, E. V. D. “A fast static analysis approach to detect exploit code inside network flows”. In Proceedings of *RAID* (September 2005).

[5] Eller R. “Bypassing msb data filters for buffer overflow exploits on intel platforms.” http://community.core-sdi.com/∼juliano/bypass-msb.txt, 2003.

[6] Kolesnikov, O., and Lee,W. “Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic”. Technical report, Georgia Tech, 2004.

[7] obscou. “Building IA32 ’Unicode-Proof’ Shellcodes”. http://www.phrack.org/show.php?p=61&a=11.

[8] Compact Oxford dictionary. http://www.askoxford.com/asktheexperts/faq/aboutwords/frequency?view=uk

[9] RIX. “Writing ia32 alphanumeric shellcodes”. http://www.phrack.org/show.php?p=57&a=15, 2001.

[10] Toth, T., and Kruegel, C. “Accurate buffer overflow detection via abstract payload execution”. In Proceedings of 5th International Symposium on RAID (October 2002).

[11] K.Wang and S.Stolfo. Anomalous payload-based network intrusion detection. In Proceedings of RAID (2004)

[12] Wang, X., Pan, C., Liu, P.P. and Zhu, S. “A Signature-free Buffer Overflow Attack Blocker.” In Proceedings of 15th USENIX Security Symposium (July 2006).