PIRANHA: FAST AND MEMORY-EFFICIENT PATTERN MATCHING FOR INTRUSION DETECTION

S. Antonatos[1], M. Polychronakis[1], P. Akritidis[1], K.G. Anagnostakis[2] and E.P. Markatos[1]

[1]*Institute of Computer Science Foundation for Research and Technology Hellas, P.O Box 1385 Heraklio, GR-711-10 Greece {antonat, mikepo, akritid, markatos}@ics.forth.gr;* [2]*Distributed Systems Laboratory, CIS Department, Univ. of Pennsylvania, 200 S. 33rd Street, Philadelphia, PA 19104 anagnost@dsl.cis.upenn.edu*

Abstract: Network Intrusion Detection Systems (NIDS) provide an important security function to help defend against network attacks. As network speeds and detection workloads increase, it is important for NIDSes to be highly efficient. Most NIDSes need to check for thousands of known attack patterns in every packet, making pattern matching the most expensive part of signature-based NIDSes in terms of processing and memory resources. This paper describes Piranha, a new algorithm for pattern matching tailored specifically for intrusion detection. Piranha is based on the observation that if the rarest substring of a pattern does not appear, then the whole pattern will definitely not match. Our experimental results, based on traces that represent typical NIDS workloads, indicate that Piranha can enhance the performance of a NIDS by 11% to 28% in terms of processing time and by 18% to 73% in terms of memory usage compared to existing NIDS pattern matching algorithms.

Key words: network security; intrusion detection; pattern matching; network monitoring; network performance.

1. INTRODUCTION

Network Intrusion Detection Systems (NIDSes) provide a powerful mechanism to defend against well-known attacks on a computer network or detect network abuse. NIDSes are mainly divided into two major categories: *signature-based* and *anomaly detection*. Anomaly-detection NIDS try to spot

abnormal behavior on network based on statistics like rate of connections, traffic overload or unusual protocol headers. On the contrary, the detection mechanism of a signature-based NIDS is based on a set of *signatures*, each describing a known attack. As an example, a signature taken from latest Snort, is

```
alert tcp any any -> HTTP_SERVER 80 (content:"/root.exe"; nocase;)
```

This signature instructs that if "/root.exe" is found inside the payload of a TCP packet that is originating from any host and any source port and is destined to an HTTP server on port 80, then an attack on the web server is taking place. While this signature requires full packet inspection, there exist simpler signatures that require only header lookups. Pattern matching inflicts a significant cost to the performance of a NIDS. Previous research results suggest that 30% of total processing time is spent on pattern matching[13], while in some cases, like Web-intensive traffic, this percentage raises up to 80%[6]. Apart from processing time, memory demands of a NIDS may reach at high levels due to rule-set growth. Although algorithms with low memory demands have been developed, their performance in comparison with algorithms that consume more memory is still poor. Given the fact that link speed increases every year, pattern matching evolves to a highly demanding process that needs special consideration. Minimizing the demands of pattern matching leaves headroom for further heuristics to be applied for intrusion detection, like anomaly detection or sophisticated preprocessors.

In this paper, we present Piranha, a pattern-matching algorithm designed for and applied to a NIDS. Our experiments with Piranha implemented in *Snort v2.2* indicate that Piranha is faster than existing algorithms by up to 28% in terms of processing time, and requires up to 73% less memory. This improvement relies on the small number of collisions and the compact memory footprint of the algorithm.

The rest of the paper is organized as follows: in Section 2 a description of existing state-of-the-art algorithms is provided, Section 3 depicts the Piranha algorithm, while Section 4 presents the performance of Piranha compared to other algorithms in various traffic scenarios and hardware platforms. Finally, our concluding remarks are discussed in Section 5.

2. BACKGROUND

In this section we describe how a content matching NIDS operates and summarize the key characteristics of pattern matching algorithms that have been recently used in intrusion detection.

2.1 Basic NIDS model

A NIDS is usually designed as a passive monitoring system that reads packets from a network interface through standard system facilities, such as *libpcap*[10]. After a set of normalization passes (e.g., IP fragment reassembly, TCP stream reconstruction, etc.) each packet is checked against the NIDS rule-set. Some -rather old- NIDS organize their rule-set as a two-dimensional data-structure chain, where each element, often called a *chain header*, tests the input packet against a packet header rule. When a packet header rule is matched, the chain header points to a set of signature tests, including payload signatures that trigger the execution of the pattern matching algorithm. Pattern matching is the single most expensive operation of a NIDS in terms of processing cost. Latest versions of *Snort* (above version 2.0) organize the rules in groups. Rules that check for the same destination port belong to the same group[14]. When a packet arrives, its destination port is used to find the appropriate group. Afterwards, multi-pattern matching is performed on patterns of the group in order to extract a set of rules that possibly match. Each rule of this set is then examined separately.

In order to understand the interaction between pattern matching algorithm, rule-set and experimental workload, we briefly present some of the pattern matching algorithms that are commonly used in intrusion detection systems.

2.2 Pattern matching algorithms

A number of algorithms have been proposed for pattern matching in a NIDS. The performance of each algorithm may vary according to the case in which it is applied. The multi-pattern approach of Boyer-Moore is fast for a few rules, but does not perform well when used for a large set. On the contrary, Wu-Manber behaves perform well when used with large rule-sets. On the contrary, Wu-Manber behaves well on large sets, but its performance starts to degrade when short patterns appear in rules. E^2xB is based on the idea that in most cases we have a mismatch and tries to filter out patterns that do not match. However, E^2xB introduces additional preprocessing cost per packet, which is amortized only after a certain number of rules. In the following subsections a more detailed description for each algorithm is provided.

2.2.1 The Boyer-Moore algorithm

The most well-known algorithm for matching a single pattern against an input was proposed by Boyer and Moore[4]. The Boyer-Moore algorithm

compares the search pattern with the input, starting from the rightmost character of the search pattern. This allows the use of two heuristics that may reduce the number of comparisons needed for pattern matching (compared to the naive algorithm). Both heuristics are triggered on a mismatch. The first heuristic, called the *bad character heuristic*, works as follows: if the mismatching character appears in the search pattern, the search pattern is shifted so that the mismatching character is aligned with the rightmost position at which the mismatching character appears in the search pattern. If the mismatching character does not appear in the search pattern, the search pattern is shifted so that the first character of the pattern is one position past the mismatching character in the input. The second heuristic, called the *good suffixes heuristic*, is also triggered on a mismatch. If the mismatch occurs in the middle of the search pattern, then there is a non-empty suffix that matches. The heuristic then shifts the search pattern up to the next occurrence of the suffix in the pattern. Horspool[8] improved the Boyer-Moore algorithm with a simpler and more efficient implementation that uses only the bad-character heuristic. Fisk and Varghese[6] recently developed Set-Wise Boyer-Moore (SWBM), an algorithm based on Boyer-Moore concepts and operating on a set of patterns. SWBM was integrated in *Snort* and tested using a single traffic trace from an enterprise Internet connection.

2.2.2 The E^{2}xB algorithm

E^{2}xB is a pattern matching algorithm designed for providing quick negatives when the search pattern does not exist in the packet payload, assuming a relatively small input size (in the order of packet size)[2,9]. As mismatches are by far more common than matches, pattern matching can be enhanced by first testing the input (i.e., the payload of each packet) for *missing* fixed-size sub-strings of the original signature pattern, called *elements*. The collisions induced by E^{2}xB, i.e., cases with all fixed-size sub-strings of the signature pattern showing up in arbitrary positions within the input, can then be separated from the actual matches using standard pattern matching algorithms, such as Boyer-Moore[4]. The small input assumption ensures that the rate of collisions is reasonably small -experiments have shown collision rates of 10% in the worst case-. In the common case, negative responses can be obtained without resorting to general-purpose pattern matching algorithms. The E^{2}xB algorithm was evaluated with traffic traces from diverse environments, including traces containing attacks, traces with normal web traffic, and WAN traffic traces from a local ISP.

2.2.3 The Wu-Manber algorithm

The most recent implementation of *Snort* uses a simplified variant of the Wu-Manber multi-pattern matching algorithm[16], as discussed by Snort developers[14]. The "MWM" algorithm is based on a bad character heuristic similar to Boyer-Moore, but uses a one or two-byte bad shift table constructed by pre-processing all the patterns instead of only one. MWM performs a hash on the two-character prefix of the current input to index into a group of patterns, which are then checked starting from the last character, as in Boyer-Moore. The performance of MWM was originally measured using text files and various sets of patterns. The first attempt to measure MWM as the basic algorithm for pattern matching in a NIDS was performed in recent Snort implementation[14]. The results of previous studies[14] show that *Snort* is much faster than previous versions that used Set-Wise Boyer-Moore and Aho-Corasick[1].

3. IMPLEMENTATION

The Piranha algorithm is based on the idea that if we find the rarest 4-byte substring of a pattern inside the packet payload, then we assume that this pattern matches. Each pattern is now represented by its least popular 4-byte sequence, where popular reflects the number of times that a specific substring exists in all patterns. For all the instances of the rare substring, *Snort* is instructed to check the corresponding rule. Piranha itself can only handle patterns with length greater or equal to 4. For completeness, patterns with length less than 4 are handled separately.

3.1 Preprocessing

Piranha treats every byte-aligned pattern as a set of 32-bit sub-patterns. For example, the pattern *"/admin.exe"* (R1) is considered as the set of its 32-bit byte-aligned sub-patterns, i.e.,*"/adm"*, *"admi"*, *"dmin"*, *"min."*, *"in.e"*, *"n.ex"* and *".exe"*. The 32-bit partitioning was chosen as the use of integers results to faster operations. Pattern matching can then be formulated in terms of an AND operation. Every pattern is represented by a gate. The gate has as many inputs as the number of its 32-bit sub-patterns. Each input represents whether the 32-bit sub-pattern has appeared in the payload or not. The gate for pattern R1 can be seen on the top-right part of Figure 1 with all its sub-patterns constituting the inputs of the gate. Initially, all inputs are set to zero, and are being switched on based on the sequences seen on the packet. However, the output must not be regarded as an exact match.

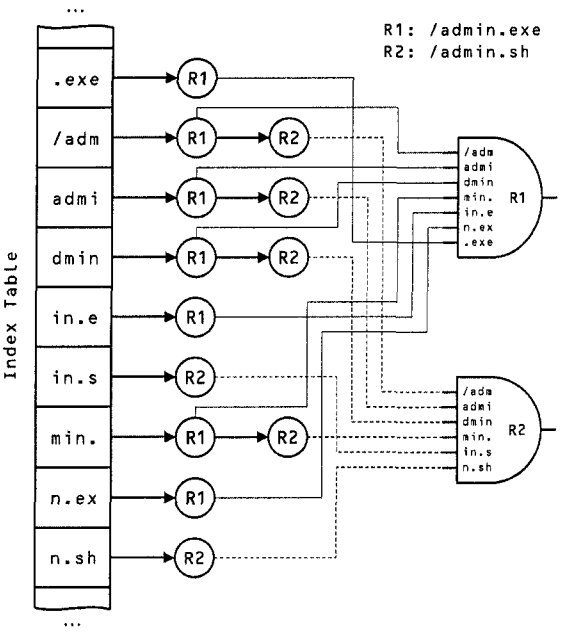

Figure 1. An example of index table and gates for two patterns. When all the inputs of a gate are switched on, then the pattern is possibly matched

For example, if the packet payload is "*/admAAAdmin.exe*", then, despite the fact that all 4-byte sequences for R1 have appeared, the pattern itself does not match. Each time the output of the gate is switched on, we consider it as collision and *Snort* is instructed for further inspection. In order to find fast which inputs to switch on, an index table is maintained. The index table keeps for all 4-byte sequences a list of all patterns that contain them. For example, if we assume that we only have the patterns "*/admin.exe*" (R1) and "*/admin.sh*" (R2), a view of the table is displayed in Figure 1. Sequences "*/adm*", "*admi*", "*dmin*", and "*min.*" appear in both patterns, while "*.exe*", "*in.e*", and "*n.ex*" exist only in R1, and "*in.s*" and "*n.sh*" only in R2. Each time a node of index table is reached then the appropriate input is switched on. As an example, if the payload is "*min.exe*", we first access the "*min.*" entry of index table and we switch on the "*min.*" inputs for R1 and R2, afterwards the "*in.e*" entry and switch on the "*in.e*" input for R1, then we access the "*n.exe*" entry and switch on the input for R1 and finally the "*.exe*" entry is traversed. The performance of Piranha for a subset of our packet traces, in terms of running time and collisions per packet, is displayed in Table 1 under the "full gates" column.

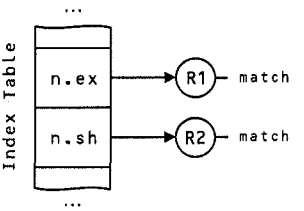

Figure 2. Optimized view of index table

Although gates present a low rate of collisions, their performance is poor as a lot of steps and transitions are needed in order to take a decision whether a pattern matches or not. In a typical case, the index table is firstly accessed, then the appropriate input is switched on and then the whole gate is checked if all inputs are switched on. In our effort to reduce the number of steps, and consequently, memory accesses, an optimization phase takes place. The optimization phase involves the procedure of selecting one input for each gate, a representative sequence. The rarest sequence is chosen as representative. It is defined as the sequence found in the least number of rules and can be found through the index table by counting the number of rules that is contained in. All other inputs are removed from the gate as well as the corresponding nodes from the index table. For example, trying to optimize our previous example we keep sequence "*n.ex*" as representative for pattern R1 and "*n.sh*" for R2. The optimized view of the index table is illustrated in Figure 2. After the optimization phase, every gate has only one input, and thus, it can totally be removed (output is equal to input), as we can use the index table for the searching phase -if a node of the index table is reached then a possible match is triggered-.

The effect of optimization is shown in Table 1, in terms of running time and collisions. The "*full gates*" column represents the unoptimized case of Piranha, and the "*representative sequence*" refers to the optimized case. Although collisions per packet increase as now only one input triggers possible match, the performance increases due to decrease of steps and compactness of memory footprint. Performance is increased by up to 36% even if collisions are two to three times more.

Table 1. Effect of optimizing gate inputs. Collisions increase but running time decreases as less steps and memory are required

	Full gates		Representative sequence	
	Running time	Collisions	Running time	Collisions
forth.web	37.71	0.61	24.06	1.65
forth.tr	36.14	0.29	27.60	0.67
forth.tr2	34.58	0.29	27.04	0.63
ideval2	12.07	0.33	9.58	1.06
ideval3	13.16	0.25	10.68	0.93

With further optimization during the searching phase as it is described in Section 3.2, collisions and running time drop significantly.

3.2 Searching

The searching phase of Piranha is straightforward. For each 4-byte sequence of the packet payload, the index table is consulted in order to find the patterns that contain this sequence. All these patterns are then sent to *Snort* for further inspection. Following our previous example, if the payload is "*/login.sh*", we have to check sequences "*/log*", "*logi*", "*ogin*", "*gin.*", "*in.s*" and "*n.sh*". According to the index table, "*n.sh*" is found in pattern R2, so we assume that R2 is matched. The rest of the sequences are not contained in any pattern so no checks are necessary. Such an approach would trigger further inspection multiple times for each packet, as shown in Table 2 ("*No check*" case). We observe that, in the average case, in an unoptimized search we trigger one rule per packet, which is prohibitive in terms of performance. In our effort to reduce collisions, we perform a trivial check before the decision that a pattern is matched. The last two characters of the pattern are checked against the corresponding two characters in the payload, and if the check succeeds then further inspection is triggered. The effect of this optimization is summarized in Table 2. In some cases, up to 75% of triggers are eliminated while the minimum reduction reaches 50%.

4. EXPERIMENTS

We evaluated the performance of Piranha against E^2xB and MWM algorithms in *Snort 2.2* using a set of packet traces. All Snort preprocessors were disabled.

Table 2. Collisions per packet without and with checking last 2 bytes of pattern against payload

	No check	Check last 2 bytes
forth.web	1.65	0.62
forth.tr	0.67	0.24
ideval2	1.06	0.32

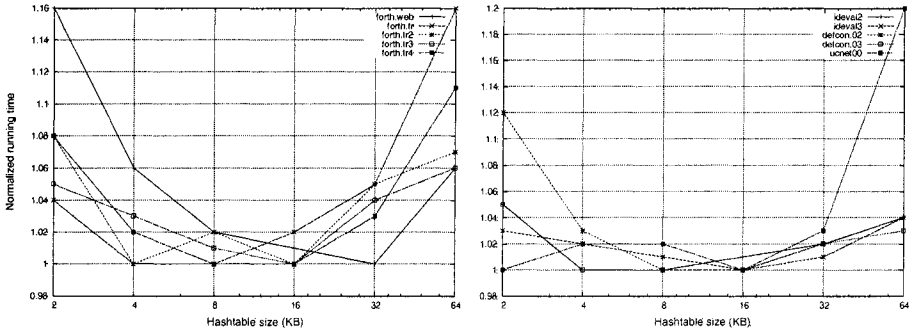

Figure 3. Effect of hash-table size on running time

4.1 Environment

All the experiments were conducted on a machine equipped with a Pentium 4 processor running at 2.80GHz, 8KB of L1 cache, 512KB of L2 cache, and 1GB of main memory. The host operating system was Linux (kernel version 2.4.0, Redhat 9.0). We used five sets of packet traces from diverse environments. The first set consists of a full packet trace containing Web traffic (*forth.web*), generated by concurrently running a number of recursive *wget* requests on popular portal sites from a host within the FORTH network. The second set contains two full packet traces (*forth.tr*) and *forth.tr2*) collected in a local area network at Institute of Computer Science inside FORTH. The third set includes a full-packet trace from the DEFCON "capture the flag" data-set (*defcon.02*). This trace contains numerous intrusion attempts. The fourth set consists of two full-packet traces (*ideval2* and *ideval3*) which were collected during the DARPA evaluation tests at MIT Lincoln Laboratory. Finally, a header-only trace with uniformly random payload (*ucnet00*) collected on the OC3 link connecting the University of Crete campus network (UCNET) to the Greek academic network (GRNET)[5] was used.

4.2 Effect of hash-table size

A complete index table of 32-bit-long patterns would normally contain 2^{32} entries, an outrageous number in terms of memory usage. In order to keep the memory footprint as small as possible, the index table was implemented as a hash-table. Since the memory footprint and locality of accesses is critical to the performance of the algorithm, we determined the

optimal size of the hash-table by obtaining the running time for different sizes and for all available traces.

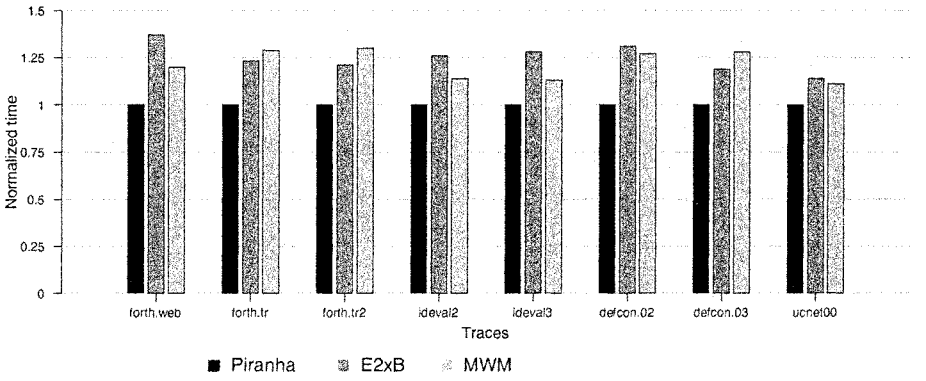

Figure 4. Running time for E2xB, Piranha and MWM for patterns with length greater or equal to 4

Results are summarized in Figure 1. Running times for each set are presented normalized to the lowest value. The time was measured using the *time* facility of the operating system. Small hash-tables suffer from conflicts and consequently longer chains have to be traversed in order to find the correct index. A large hash-table, on the contrary, has fewer conflicts but for every access a performance penalty is paid due to poor cache behavior. We observe that optimal size of the hash table for most of the traces is around 16KB and this is the size we used for all our experiments presented in the paper.

4.3 Comparison against other algorithms

We compared Piranha against MWM[14] and E^2xB[2,9] on all available traces. In our experiments, we measured running time in user space (kernel time was negligible). Results are presented in Figure 4. Times are presented normalized against the running time of Piranha algorithm.

The performance of Piranha is consistently better compared to other algorithms. Improvement ranges between 10 and 23.50%, with the results remaining the same for the rest of the traces that are not displayed in Figure 4. We also compared our algorithm with AC-Banded[11], an optimized implementation of Aho-Corasick[1], but running time of AC-Banded was two to four times the time of our algorithm. Results in Figure 4 are for patterns with length greater or equal to four, as four is the length that can be natively handled by Piranha. For completeness reasons, the case of small pattern was

also implemented. Small patterns impose a performance bottleneck for Piranha and MWM as well as E^2xB. MWM can natively handle patterns with length greater or equal to two while patterns with length one are examined separately. The overhead that small patterns impose in terms of running time can be seen on Table 3. In average case, running time was decreased by 25% for Piranha and 20% for MWM. The effect on E^2xB is smaller as it is not dependent to pattern length but proportional to the number of patterns. In the last two columns of the table we can observe the performance benefit of Piranha against MWM and E^2xB for all pattern lengths. Despite the performance bottleneck, our algorithm still performs better for all available traces, except the case of *defcon.02* trace where improvement is marginal. However, our main contribution is focused on patterns with a fair enough large size as only 3% of patterns have length less than four.

Piranha does not only perform better in terms of processing time but also in terms of memory usage. While MWM requires 45MB of memory to process the full rule-set, AC-Banded 96MB and Aho-Corasick 140MB, Piranha consumes only 37MB. Efforts have been made recently in order to develop algorithms with low memory consumption. Tuck et al.[15]; have developed two modified versions of Aho-Corasick, AC-Bitmap and AC-Path, that reduce memory usage. AC-Bitmap needs 20MB memory while AC-Path only 15MB. However, such algorithms present very high processing time. Comparing Piranha with AC-Bitmap and AC-Path, we observed that they need, in average, three to four times more processing time. *Snort* also comes with *SFKSearch*, an algorithm that requires only 14MB of memory, but its performance compared to others is poor - three to four times more processing time against Piranha -. The tradeoff between memory usage and processing time can be seen on Figure 5. Algorithms with low memory usage need three to four times more processing time, while algorithms with high memory usage present high processing capacity. Although the assumption that low memory means high processing time cannot be generalized, there are strong indications that this tradeoff might hold for other algorithms that are not discussed here.

4.4 Evaluation on different architectures

We evaluated the performance of Piranha on different hardware architectures. Our testing environment, besides the machine described in Section 4.1, consists of a Pentium Xeon 2.4 GHz with 8KB L1 cache, 512KB L2 cache and 512MB main memory, an AMD Athlon MP 1.8GHz with 128KB L1 cache, 256KB L2 cache and 512MB main memory and a Pentium 3 running at 600 MHz with 8KB L1 cache, 256KB L2 cache and

Table 3. Effect of small-patterns on running time

	Piranha		MWM		E2xB			
	pattern length		pattern length		pattern length		Piranha vs. MWM	Piranha vs. E^2xB
	>=4	all	>=4	all	>=4	all	%	%
forth.web	21.05	30.17	25.32	33.59	28.86	34.12	10.18	11.57
forth.tr	23.78	30.78	30.80	35.65	29.80	31.18	13.66	1.28
forth.tr2	26.55	30.37	30.23	36.12	29.91	30.46	15.91	0.29
ideval2	8.49	11.36	9.68	12.70	10.84	13.25	10.55	14.26
ideval3	9.88	12.89	11.26	14.58	12.69	15.25	11.59	15.47
defcon.02	7.06	9.91	8.99	9.97	9.42	9.96	0.60	0.50
defcon.03	7.20	8.74	8.59	9.20	8.18	8.99	5.00	2.78
ucnet00	3.11	3.59	3.48	4.21	3.59	3.81	14.72	5.77

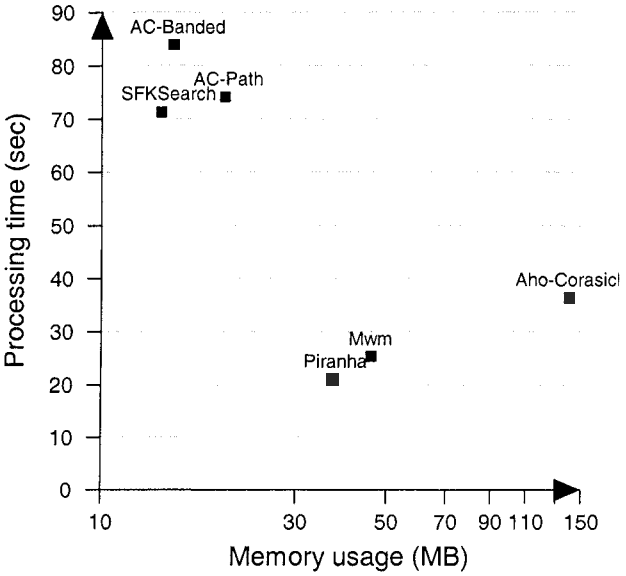

Figure 5. Memory usage against processing time

512MB main memory. Results are presented in Figure 6. Running time is normalized against the time of Piranha running on P4 at 2.8GHz.

Independent of the underlying hardware platform, Piranha performs better for all traces. As processor clock speed decreases, performance of both algorithms decreases as expected. However, the performance gap seems to decrease with the clock speed for specific traces while for others it remains constant. On Pentium Xeon 2.4GHz, improvement waves between 7.8% and 18.8% while on Pentium 3 600MHz between 10.86% and 14.83% (leaving out the *ucnet00* trace where improvement is marginal). Similar results apply to the AMD Athlon architecture, where improvement is ranged between 7.32% and 18.21% (again *ucnet00* trace is omitted).

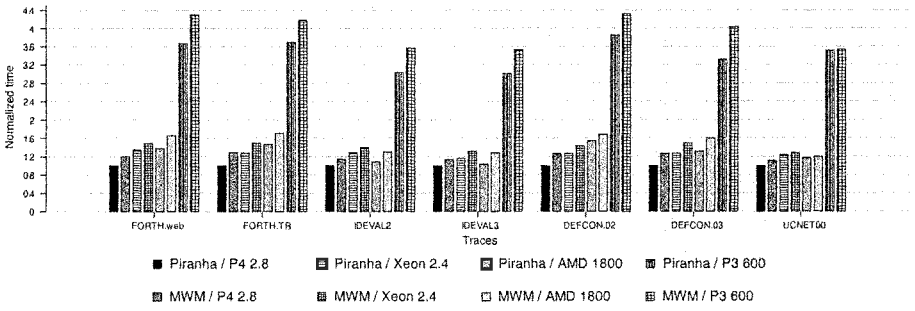

Figure 6. Performance of Piranha and MWM on different architectures

4.5 Performance under attack

Intrusion detection systems are themselves subject to being attacked. Some types of attack try to evade NIDS by exploiting weaknesses in protocol handling, like IP defragmentation or TCP reassembly[7,12]. Other attacks aim at overloading the detection engines by exploiting weaknesses in the internal algorithms used, in our case pattern matching. The attacker sends packets with carefully crafted payload in order to force the pattern matching engine to spend more processing time than it would require for an innocent packet. Most of the traffic is then dropped by the NIDS, including packets containing attack, giving the attacker the chance to evade detection. Our previous work on such attacks has shown that the processing time of *Snort* can be raised by up to 25 times[3]. Although the worst case scenario for each algorithm and the *Snort* itself is extremely difficult to be generated, we provide some hints on how a NIDS can be heavily overloaded. For performance reasons, *Snort* firstly performs the multi-pattern matching and then for all possible matches the whole rule is checked: header processing and exact string matching for all patterns that the rule contains[14]. Examining the groups of rules that are processed during packet inspection, it can be observed that rule

alert tcp any any → any any (ack:0; flags:SFU12; content:"AAAAAAAAAAAAAAAA"; depth:16;)

is found in all groups as it applies to all source and destination ports. That means that for all packets examined, *Snort* will try to locate the pattern ``*AAAAAAAAAAAAAAAA*" and for all possible matches will check the rest of the rule. In our example, after the pattern matching phase the

acknowledgment number and the TCP flags will be verified. We constructed an attack trace by taking the headers of the *forth.web* and placing only "A" in the payload. In that way, in every offset of the payload *Snort* finds that pattern and checks for the rest of the rule. However, the header of the packets is normal (no special TCP flags are turned on and acknowledgment number in non-zero) and thus the rule is never matched.

Forcing *Snort* to generate matches and checks in every offset is very expensive as it can be seen on Table 4. We observe that processing time is raised by 3 to 15 times and that all algorithms are subject to payload attacks, as the way *Snort* performs detection is exploited and not the nature of the algorithms. Such overload factors can provide the attacker the ability to hide his attack among legitimate traffic. Other payloads were also crafted, like payload including only "a". As the packet payload is capitalized, possible matches are also generated and the overloading still takes place. In the case of MWM, running time is increased further as there are some patterns that start with "aa" and trigger more inspections on the internal structures of MWM. The Aho-Corasick-like algorithms try -as an optimization- to verify their match by calling *memcmp()* for pattern against the payload before forcing Snort to check the whole rule. The cost of memory-comparing is increasingly high as in each offset a comparison is performed. However, there are some cases where a specific payload can cause Piranha to generate collisions in most of the payload offsets but Aho-Corasick-like algorithms are not affected. This payload can be made by replacing the last character of "*AAAAAAAAAAAAAAAA*" pattern with another character, like "B". Piranha decides that pattern matches only by seeing the appearance of an "*AAAA*" but the whole pattern is not really matched. Aho-Corasick algorithm detect that the whole pattern cannot be matched so their time remains practically the same. As Table 4 shows, only Piranha and MWM suffer from this payload attack. Focusing on the worst overall performance (the "*worst overall*" column) among all attacks described above, Piranha needs 3 times less running time than other algorithms.

Table 4. Completion time and overhead factor (attack completion time / original completion time) for different attack payloads. "Time" denotes completion time and "factor" denotes overhead factor

	Packet payload							
	Origi-nal	AAAAA...		aaaaa...		AAA...B...		Worst overall
	Time	Time	Factor	Time	Factor	Time	Factor	
Piranha	21.94	120.01	5.46	118.50	5.40	91.47	4.16	120.01
MWM	25.91	233.73	9.02	376.72	14.53	204.88	7.90	376.72
AC	35.71	417.72	11.69	361.45	10.12	28.98	0.81	417.72
AC-path	81.59	357.84	4.38	212.62	2.60	78.81	0.96	357.84
AC-Bitmap	72.87	409.74	5.62	241.88	3.31	110.65	1.51	409.74

5. CONCLUDING REMARKS

We have presented the design of Piranha, a novel pattern matching algorithm for NIDS and evaluated its performance under various network traffic characteristics using a diverse set of packet traces. Our comparison against existing algorithms shows that an improvement of up to 28% can be achieved. The improvement is due to its quick decisions on which patterns may match and to its compact memory footprint which infers good cache behavior. Our results on different architectures indicate that Piranha performs consistently better, with the performance gain increasing along with processor speed. Furthermore, we have concluded to some general remarks for pattern matching on NIDS: small patterns inflict a significant performance overhead that needs to be examined carefully, and cache-conscious programming of a NIDS pattern-matching algorithm is a key element to its performance.

ACKNOWLEDGEMENTS

This work was supported in part by the IST project SCAMPI (IST-2001-32404) funded by the European Union and in part by the i-Guard GSRT Project (02-PRAXE-212) funded by the Greek General Secretariat for Research and Technology through PRAXE A. Work of K.G. Anagnostakis is also supported in part by ONR under Grant N00014-01-1-0795. E. P. Markatos, S. Antonatos, M. Polychronakis and P. Akritidis are also at University of Crete. Work of K. G. Anagnostakis was done while at ICS-FORTH. We would also like to thank Vasilis Siris for providing the UCnet traces.

REFERENCES

1. Aho and M. Corasick, Fast pattern matching: an aid to bibliographic search. Commun. ACM, 18(6):333-340, June 1975.
2. K. G. Anagnostakis, E. P. Markatos, S. Antonatos, and M. Polychronakis, E2xB: A domain-specific string matching algorithm for intrusion detection. In Proceedings of the 18th IFIP International Information Security Conference (SEC2003), May 2003.

3. S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, Generating realistic workloads for network intrusion detection systems. SIGSOFT Softw. Eng. Notes, 29(1):207-215, 2004.
4. R. Boyer and J. Moore, A fast string searching algorithm. Commun. ACM, 20(10):762-772, October 1977.
5. C. Courcoubetis and V. A. Siris, Measurement and analysis of real network traffic. In Proceedings of the 7th Hellenic Conference on Informatics (HCI'99), August 1999.
6. M. Fisk and G. Varghese, An analysis of fast string matching applied to content-based forwarding and intrusion detection, Technical Report CS2001-0670 (updated version), University of California - San Diego, 2002.
7. M. Handley, V. Paxson, and C. Kreibich, Network intrusion detection: Evasion, traffic normalization, and End-to-End protocol semantics, In Proceedings of USENIX Security Symposium, pages 115-134, 2001.
8. R. Horspool, Practical fast searching in strings. Software - Practice and Experience, 10(6):501-506, 1980.
9. E. P. Markatos, S. Antonatos, M. Polychronakis, and K. G. Anagnostakis, ExB: Exclusion-based signature matching for intrusion detection, In Proceedings of CCN'02, November 2002.
10. S. McCanne, C. Leres, and V. Jacobson, libpcap. Lawrence Berkeley Laboratory, Berkeley, CA, available via anonymous ftp to ftp.ee.lbl.gov.
11. M. Norton, Optimizing Pattern Matching for Intrusion Detection, July 2004. http://docs.idsresearch.org/ OptimizingPatternMatchingForIDS.pdf.
12. T. H. Ptacek and T. N. Newsham, Insertion, evasion, and denial of service: Eluding network intrusion detection, Technical report, Secure Networks, Inc., Jan. 1998.
13. M. Roesch, Snort: Lightweight intrusion detection for networks. In Proceedings of the 1999 USENIX LISA Systems Administration Conference, November 1999. http://www.snort.org/.
14. Sourcefire, Snort 2.0 - Detection Revisited. October 2002. http://www.snort.org/ docs/Snort_20_v4.pdf.
15. N. Tuck, T. Sherwood, B. Calder, and G. Varghese, Deterministic memory-efficient string matching algorithms for intrusion detection, In Proceedings of the IEEE Infocom Conference, March 2004.
16. S. Wu and U. Manber, A fast algorithm for multipattern searching, Technical Report TR-94-17, University of Arizona, 1994.