Journal of Information Assurance and Security.
ISSN 1554-1010 Volume 6 (2011) pp. 512-521

Hybrid Multithreaded Pattern Matching Algorithm for Intrusion Detections Systems

Monther Aldwairi and Niveen Ekailan

College of Computer and Information Technology, Jordan University of Science and Technology,
Irbid, 22110, Jordan
munzer@just.edu.jo, nyekailan081@cit.just.edu.jo

***Abstract*: Intrusion Detection System (IDS) is a powerful tool to discover and counter malicious activities over the Internet. An IDS inspects the packet payload and header to identify attack signatures. Pattern matching over packet payload is the most expensive operation in the detection stage in terms of speed and memory. Therefore there is a real need to improve the existing pattern matching algorithms and reduce their memory requirements, run time and complexity. In this effort we propose a hybrid Aho-Corasick (AC) and Wu-Manber (WM) pattern matching algorithm for speeding up IDS. The premise is that AC performs better for short patterns while WM outperforms AC for longer patterns. In addition, AC performs better for clean traffic with constant worst case performance as opposed to WM which performs better for malicious traffic. A novel partitioning algorithm is developed to equally divide the attack signatures between AC and WM. Multiple threading configurations are investigated and the best configuration is identified. Over the entire new algorithm achieves a best case 73% speed improvement compared to four threads of WM, and 64% compared to the four threads of AC algorithm. In terms of memory requirement we achieve a best case reduction of 53% compared to four threads of WM, and 67 % compared to four threads of AC.**

***Keywords*:** Network Security, Intrusion Detection, Pattern Matching, Multithreading, Aho-Corasick, Wu-Manber**.**

I. Introduction

With the Internet growing popularity and speeds, more sensitive information is exchanged and with that comes new problems. Threats become more severe, an increasing the number of people are being victimized and sophisticated attacks are increasing in frequency. Therefore, Security has become a real challenge. This makes a case for fast, accurate and configurable network security tools that could handle this amount of malicious traffic in real time.

Intrusion detection systems aim to detect malicious activity by either matching packets against attack signatures or detecting suspicious behavior in network traffic. It looks for specific patterns or behavior anomalies that characterize malicious intents. An IDS is classified into either misuse or anomaly. Misuse, often referred to as signature based, intrusion detection system gathers information from the incoming traffic then analyzes it looking for attack signatures. The major advantages of this approach are speed and accuracy represented by less false positives compared to anomaly based approaches. However, the accuracy of misuse detection depends on the accuracy of the rules containing attack signatures. Unfortunately the rules are manually maintained which makes it efficient to mitigate only previously known attacks. In addition, the overall performance degrades as the number of signatures and amount of traffic increases. Anomaly IDS defines a normal baseline for the network behavior. It monitors the ongoing traffic then compares the behavior of the traffic to the predefined baseline. Any deviation from the baseline is considered anomalous. Anomaly IDSs are slow, require long training stage, and suffer from high false positives and negatives.

Intrusion detection systems can also be classified based on deployment into or host and network based detection. A host based IDS (HIDS) is software designed to check internal behavior of a machine. It inspects all host activities and actions such as system calls, running applications, and log files. HIDS uses database of normal states of all objects under monitoring. It is vital for this kind of IDS to keep the system database strongly protected. The attacker who targets HIDS tries to manipulate the system database to conceal the traces of their attack. Network based intrusion detection system (NIDS) on the other hand, is a processing device connected to a network ingress point to monitor its behavior. It captures and analyzes all packets in real time. Three modules are mainly used in NIDS. First, a filter module is used to analyze data in real time. It passes all known non malicious traffic and forwards the suspected traffic to the attack recognition module to be checked for attack signatures. Once an attack signature is found a response module is invoked to take a predetermined action specified by the rules.

Signature based IDS is more commonly used because it is faster and has less false positives. Signature based IDS simply inspects packets payload for attack signatures which makes pattern matching the most critical part of IDS. Mike Fisk and George Varghese showed that 31% of Snort (widely used signature based IDS) processing time is spent in pattern matching. They also showed that the cost of the matching operation increases rapidly in case of highly malicious traffic to reach 80% of the overall processing time [1]. In addition the number of attacks and potential threats increases rapidly. Therefore, the signatures database grows significantly each year, imposing more restrictions on IDS speeds and memory

requirements [2]. Hence, it is highly important to reduce the amount of work and processing time spent on content matching operations. More efficient pattern matching algorithms for real time packet inspection is a very active research area.

This paper introduces a fast efficient pattern matching algorithm for intrusion detection systems. We propose a hybrid pattern matching algorithm, based on two well-known multiple pattern matching algorithms Wu-Manber, and Aho-Corasick. The new algorithm matches long patterns against packet payload using WM and short patterns are matched using AC. A new partitioning algorithm is designed to divide the signatures between the two algorithms to keep the workloads balanced for optimal performance. Multiple threads are used to match the packets against different sets of signatures.

The rest of this paper is organized as follows. Section II will present the background necessary to understand the problem. It explains Snort, pattern matching for IDS and different parallelism types. Section III presents the state of the art IDS systems. Section IV presents the partitioning algorithm and describes the hybrid algorithm in details. Finally, Section VI presents a complete theoretical and experimental analysis.

II. Background

In order to understand the hybrid algorithm we must first study the properties of the IDS rules, attack signatures and pattern matching algorithms. The following subsections will introduce Snort and its rule format. Next a detailed description of pattern matching algorithms and more specifically the Aho-Corasick and Wu-Manber algorithms is brought forward. Finally, detailed examples from Snort extracted signatures are thoroughly explained.

A. Snort

Snort is a popular open source IDS that relies on a rule based engine to detect attacks [2]. A Snort rule specifies an action to be taken when a packet matches the rule header and options. The rules header specifies the IP address and port duple of a packet source and destination. Rules may contain one or more options consisting of two parts: a keyword and an argument. Rule options contain extra information for matching packets payload such as the *content* keyword which is used to find attack signatures in the packet. Contents or attack signatures could be presented as binary data in hexadecimal or in the form of ASCII character string. The *uricontent* keyword is similar to the content keyword except it checks only the Uniform Resource Identifier (URI) part of http packets. The *offset* keyword is used to start the search at a certain offset from the beginning of the packet payload. The *nocase* keyword is used to perform case insensitive search of signatures within the packet payload while the *ttl* keyword tests the IP header's TTL field value. The *sid* keyword specifies the signature identification number and *rev* keyword indicates the rules database revision number. The *msg* keyword indicates the message to be logged [3].

Figure 1 shows a snippet of Snort rule number 1659. This rule indicates that an alert should be triggered for any HTTP packet from any external network to any HTTP server/port given that the packet tries to execute /sendmail.cfm'.

```
alert tcp $EXTERNAL_NET any -> $HTTP_
SERVERS $HTTP_PORTS (msg:"WEB-COLD
FUSION sendmail.cfm access"; uri
content:"/sendmail.cfm"; nocase;
classtype:attempted-recon; sid:1659;
rev:6)
```

Figure 1. Snippet of Snort rule number 1659

Snort is composed of several components with the detection engine performing the core operations of pattern matching. It reads the signatures into a data structure and matches them against incoming packets. The complexity of the work at the detection engine is affected by several factors: mainly the number of rules that should be evaluated against the received traffic, the speed of the network interface, the speed of the hardware running the IDS and the quality of the pattern matching algorithm. Snort spends at worst case 80% of its processing time doing pattern matching [1] and 87% of the 2003 Snort rules contain signatures to match against [4].

Snort implements Boyer-Moore, Aho-Corasick and Wu-Manber pattern matching algorithms in different stages of the detection process. The pattern matching operation remains the bottleneck of any IDS regardless of the algorithm.

B. Pattern matching for IDS

Pattern matching algorithms are categorized into either single or multiple pattern matching algorithms. Single pattern matching algorithms search for one string at a time, while multiple pattern matching algorithms search for all patterns at once.

1) Aho-Corasick

Aho-Corasick is a multiple pattern matching algorithms [5]. A preprocessing procedure is performed over the set of strings, and then all packet contents are searched at once in the search phase. In preprocessing, a deterministic finite state machine automata (DFA) is built from the strings as follow. The algorithm starts in the idle state as the root state of the automata. Then the characters of the first pattern are added to trie one by one. The same process is repeated for all patterns. A failure function is used to add links that point to partial string matches. The failure link is used in case of mismatch, where the current searched string is a substring of many patterns.

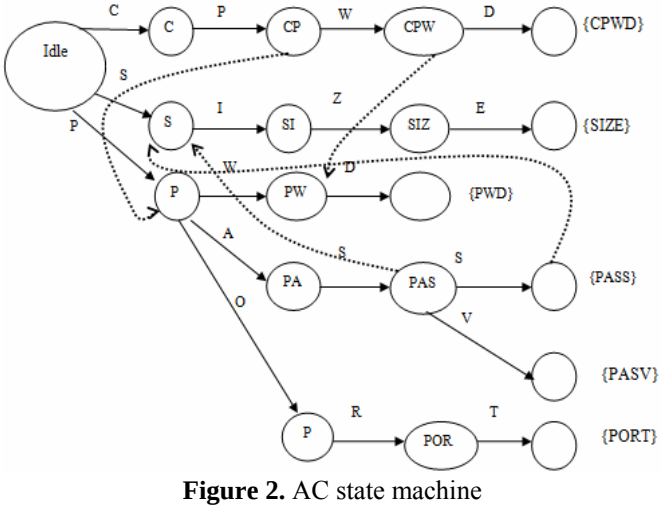

Figure 2. AC state machine

In the search process, the text is passed through the state machine character by character. If there is a match the goto function is invoked to proceed and match the next character. Otherwise the fail function is used to match the text against other patterns if possible or return no match idle state.
Figure 2 shows the trie for a set of string extracted from actual Snort rules, P = {CPWD, PWD, PASS, PORT, SIZE, PASV}. Solid arrows represent goto function and dotted arrows represent failure links.

2) Wu-Manber

Wu-Manber is a high performance multiple pattern matching algorithm. The matching process in WM is divided into two main stages: preprocessing and scanning stages. In the preprocessing stage WM builds three tables: SHIFT, HASH, and PREFIX [6].

In preprocessing stage, WM computes the minimum length of patterns, *m*. WM considers only the first *m* characters of each pattern in the SHIFT table computation. Only the entries with 0 shift values will be a candidate entry to HASH and PREFIX tables.

The SHIFT table contains a shift value for each possible substring of size *B*. In other words, it specifies the number of characters to skip when matching a substring of size *B*. To construct the SHIFT table take *X*, which is a substring representing the last *B* characters of Snort signatures of size *m*. Given *X*, there are two possibilities:

1. *X* does not exist in any patterns, then the maximum safe shift without missing any matches is *m-B+1*, which is the default value stored into *SHIFT[X]*.
2. *X* exists in some patterns. Find the rightmost incident of *X* in all patterns, call it *q*. Then the maximum safe shift without missing any matches is *m-q* and it will be stored into *SHIFT[X]*.

The HASH table contains groups of patterns of the same substring that have a zero shift value in the SHIFT table and it is accessed using the same hash function used to access the SHIFT table. The HASH table is needed to reduce the cost of searching for a substring that matches several patterns. The first *B* characters of all patterns are mapped into the PREFIX table as well to help search strings with different prefixes. When the search for the SHIFT table returns 0, the HASH and PREFIX tables are consulted to determine if there is a match.

In the scanning stage the following steps will be performed on a sliding window of size, *w>B,* over the packet.

1. Compute the HASH table entry *h*, based on the last *B* characters of the current sliding window.
2. If *SHIFT[h] >0*, shift the current sliding window and return to 1, else go to 3.
3. Compute the prefix hash value, *p*.
4. Check for *p* given that *HASH[h] ≤ p < HASH[h+1]* where *PREFIX[p] = text prefix*. If TRUE, match the pattern against the text.

Block	Shift	Block	Shift
PW	1	OR	1
WD	0	RT	0
AS	1	SV	0
SS	0	Others	2

Table 1. Wu-Manber SHIFT table

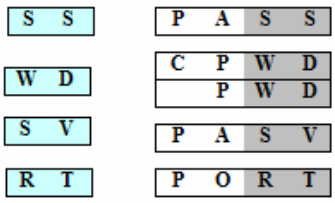

Figure 3. WM HASH table

Step	A	P	C	W	D	P	W	D	Shift	Match
1		↑							2	
2				↑					2	
3					↑				2	
4							↑		0	PWD

Figure 4. WM scanning process

To search for these patterns {CPWD, PWD, PASS, PORT, PASV} using the WM algorithm, we first program the SHIFT table as shown in Table 1. Minimum pattern length is *m=3* and we select a block of size *B=2*. The default shift value *3-2+1=2* is listed under {others}. To calculate the shift for {PW} we locate the right most occurrence of the block in any string, *q=2* for signature {CPWD}. Therefore the shift value is *3-2=1*. On the other hand, for {WD} there are two matches: {CPWD} with occurrence index of 3 and {PWD} with occurrence index of 2. We pick the rightmost to calculate the shift = *3-3=0*.

Figure 3 shows the HASH for the same signatures. It contains, for each *B* character with SHIFT table value of 0, a linked list with all strings containing that substring. Figure 4 shows the search process for the text {APCWDPWD} with a sliding window of three characters. The arrows point to the substring to be used to access the SHIFT table in each step. The first window is {APC}; check the SHIFT table for block {PC}, which falls under "Other", with a shift value of 2. The window is shifted by two to {CWD}. Check the SHIFT table for block {WD} which has a shift of 0 indicating a possible match. Next the HASH table is checked and no match is returned. The sliding window is shifted by one to become {WDP}. Step three retrieves a shift of 2 from SHIFT table for block {DP} which makes the next window {PWD}. In step four block {WD} has a shift of 0 indicating a possible match. The HASH table indicates a match for {PWD}.

III. IDS Parallel Techniques

Distributed IDS is different from parallel IDS in that distributed IDS performs monitoring at different locations of the target network while parallel IDS is the system that processes traffic in a parallel fashion or performs other processing tasks at the same time. In general, a parallel IDS monitors traffic on a single ingress or egress point [7]. The main function of distributed IDS is to improve the detection process and minimize the risk of malicious traffic.

Parallelism on the other hand can help reduce the time consumed in pattern matching and increases the throughput. Parallelism techniques can be classified into two categories: data and function parallel. Data parallel techniques aim to divide the processing load over several threads or nodes. Although the actual processing time will not be greatly improved, the key advantage of those techniques is to decrease the arrival time and increase the throughput. In

function parallel techniques each node or thread performs different work than the other threads. That is each thread performs different part or task of the processed job. This parallelism technique reduces the processing time of each packet on a single node.

Parallelism can be deployed at three levels in IDS: subcomponent, component, and node level. Deploying parallelism at one level will not affect the behaviour or the performance of other levels. Parallelism of both types may be deployed at one, two, or at all levels [8]. The following subsections will shed some light on the two classes of parallelism: data and function parallel as well as provide a brief description of the three parallelism deployment levels.

A. Data parallel techniques

Data parallel techniques offer three possible scenarios for dividing the load of signature-based intrusion detection systems. First, pattern division, the attack signatures are divided over several threads. In this scenario each thread matches all packets against a subset of the signatures dataset. Second, packets division where each packet is divided to several fragments and each fragment is forwarded to a thread. Each thread matches a complete set of patterns against the received packet fragment. The third scenario is a hybrid of both.

Several attempts were made to make use of data parallel techniques to improve IDS performance. Wheeler et al. introduced a data parallel algorithm, called dual algorithm [9] based on the idea of gaps in the rules set. It divides the rules set into two groups: one and two bye patterns. If the group is small, Snort's default Boyer-Moore (BM) algorithm [10] is used for the search process. Otherwise, if the group is large a special one character version of WM is used. Christopher Kopek proposed Divided Data Parallel (DDP) content matching algorithm [11]. He divided the incoming packet across *n* processors and studied the performance enhancement given the added division overhead. The overhead is caused by the overlap between packets fragments which must be larger than longest signature in order not to miss any attack signatures by splitting them into two different packet fragments. DDP provided each processor with WM HASH, SHIFT, and PREFIX tables programmed with all rules. The results show a 1.94 speed gain in case of two processors, and 3.48 in case of four. Because of the increasing maximum signature length this algorithm will suffer because the penalty is magnified.

B. Function parallel techniques

There are variant ways to implement function parallelism depending on the pattern matching algorithm. Zhang et al. [12] proposed a function parallel architecture to parallelize the WM algorithm. They proposed to spread the HASH table entries on a number of threads. By using this technique the search time per thread will be reduced, because reducing the number of HASH table entries will decrease the number of distinct patterns to be compared per thread. Consequently, each thread has less patterns to match against packet payload, by doing this it is most probable that each processing thread will skip larger segment of the searched text when no match returned by the search process. On the other hand, decreasing the number of entries in the HASH table and increasing the shift values means that the number of calls to the HASH table decreases, so the overall performance of the search process will be optimized. The practical test of this technique did not show much optimization because the number of rules increases rapidly, which makes the effect of spreading the HASH table entries meaningless. In one hand the time needed to perform a search with 30 HASH table entries is almost the same as searching a HASH table with 120 entries. On the other hand, the overhead of spreading the HASH table and handle all nodes will be an extra charge. Therefore, the search time each thread needs will be almost the same time needed if it has a complete rule set.

C. Component level parallelism

Component level parallelism is generally a function parallelism with some functionality isolated in separate processing units. In IDS there are two candidates for component level parallelism: preprocessing and content matching. In practice component level parallelism could be implemented as lightweight processes or threads, where each single component uses a single process or thread to be executed. Another implementation is to use a single processor for each IDS component. This technique main advantage is that the instruction remains in the cache memory of its processing unit, therefore, the performance is improved [9].

D. Subcomponent level parallelism

In this technique some components of the detection system are parallelized. In practical we can classify the preprocessing stage into critical and non-critical as argued in [11]. Practical implementation can be either by spreading malicious traffic rules on a group of processing units, so each processing unit serves only network traffic with certain characteristics applied to a set of rule group or by grouping the rule set and processing these groups in parallel fashion. One of the practical applications of this technique is in matching web traffic where pure-content rules and rules containing uri-content implemented in two isolated structures. Aldwairi et al. divided the traffic between parallel engines based on the port [4].

E. Node level parallelism

Node level parallelism consists of a number of independent systems that performs their tasks separately. Packets are delivered to the processing nodes using packet splitter or traffic duplicator based on the behavior of the system. Each processing node could be a processor unit, multiprocessor desktop, uniprocessor desktop, or any other system. Each node at this level of parallelism operates as a black box system that works on data or function parallel manner [9]. A one significant advantage of node level parallelism is that packet pre-processing performed once before splitting over the processing nodes. On the other hand, the need for a sophisticated traffic splitter is almost the main drawback of this parallel system [11].

Fulp et al. [13] introduced a novel firewall architecture that performs packet matching under heavy traffic loads, higher traffic speeds, and high quality of service requirements. The proposed architecture runs multiple firewalls in parallel, where each single firewall runs a subset of the original policy set. This system checks the incoming packets in parallel fashion; each packet is checked by all the firewalls. The proposed firewall architecture has lower time complexity and higher throughput. It provides 74% speedup with four firewall architecture. Furthermore this technique has a major

advantage, because it provides state full packet verification, which is vital to identify certain types of attacks.

IV. Related Work

Several works are introduced in pattern matching research area to reduce the time, complexity and memory requirements in both single and multiple pattern matching algorithms. Kim et al. [14] proposed a multiple string pattern matching algorithm based on compact encoding. They assume that the bit representation of a single pattern needs one word to be stored. The proposed algorithm shows high performance in matching large number of patterns simultaneously. The algorithm performs better than *gerp* and *agerp* in many test cases. They used hashing to handle the problem of having a large number of patterns without affecting the performance of the matching process.

For single pattern matching algorithms, Liu et al. [15] proposed double single pattern matching algorithms. Each single matching algorithm consists of two structures: a finite state automation and smallest suffix automation. The proposed algorithm shows better performance in terms of time in average and it over performs BM in matching short and long patterns. In addition, by comparing the performance of the proposed algorithm with RF and Long Derivative Matching (LDM) [16], it provides lower time complexity than both in matching short patterns.

There exist many efficient pattern matching algorithms for IDS suitable for hardware implementation. Rafla et al. [17] proposed a Finite State Machine (FSM) pattern matching algorithm for hardware implementation. The proposed design is RAM based and is reconfigured on the fly through altering memory contents. For reconfiguration they used embedded processing unit. The results of their experiments show that the proposed system can be restructured to handle new patterns. The performance evaluation of this algorithms shows that every search iteration took one clock cycle to finish, and that pattern length did not affect the clock frequency. In [4] Aldwairi et al. proposed a reconfigurable memory based accelerator for intrusion detection. The memory system consists of high speed DMA and multi-port SRAM. In order to increase pattern matching speed they experimented with a number of configurable accelerators. The proposed system consists of two components: a software executed on a VLIW core and a hardware accelerator for pattern matching for FSM construction. The software creates an FSM and builds the state tables, and then the FSM performs pattern matching using the original AC algorithm. The authors stored the AC state tables in SRAM using different techniques, which improves the overall throughput of the system. The experimental evaluation of the proposed technique reported throughput of up to 14 Gbps using 8 parallel FSMs. On the other hand, the throughput degrades with increasing the number of patterns, because the number of characters to be matched increases and the number of states increases as well [18].

Weinsberg et al. [19] introduced a multiple string matching algorithm based on a standard RAM and Field Programmable Gate Arrays (FPGAs). The algorithm optimizes the detection speed while false positives and negatives remain the same. In [20] Woods et al. introduced a real time pattern matching architecture for hardware implementation, the proposed architecture monitors network stream and analyses it in real-time. This system consists of a complex event processor connected with FPGAs. It is optimized for small packets (generated by financial applications) and was able to handle a saturated 1Gbps link. The system is reconfigurable by introducing a rich pattern description language.

Many researchers introduced modified versions of AC and WM algorithms. Tuck et al. [21] proposed a modified version of AC. They optimized the memory required in the matching operation, and provide efficient solutions to improve the efficiency of the matching process on hardware implementation by using two techniques: applying bitmap compression on each node and apply path compression to AC to achieve compact storage and worst case performance. Zhang et al. [12] propose a modified version of WM algorithm that improves the performance of WM in short patterns matching. The proposed algorithm groups the malicious patterns based on length, then each group is processed independently. A single data structure for each group is built. This technique has lower time complexity than the basic WM algorithm. Yang Dong Hong et al. [22] merged Quick Search (QS) [23] and WM algorithms. They maximized the shift values by making use of the mismatch states in the matching phase. They modified the preprocessing phase of WM algorithm by adding a HEAD table to hold the first two characters of each pattern. The performance evaluation shows lower time complexity compared to regular WM with short patterns.

Researchers additionally, made use of artificial intelligence algorithms for pattern matching. Lu et al. [24] used genetic algorithm to learn a set of rules from anomalous network traffic. They demonstrated potential to generate new rules using GA to detect unknown attacks. Despite the low false positives and negatives, the use of random crossover and mutations resulted in modest results in terms of accuracy.

Moraru et al. introduced a modified version of Rabin-Karp string search algorithm with feed-forward Bloom filter [25]. They take into consideration that very large number of patterns generates small number of matches. They used the Bloom filter to discard all corpuses that does not return hits in the filter because they cannot contain a match. They benefit from a memory hierarchy to attain a speedup between 2 and 30 times and memory reduction with almost 50 percent compared to *gerp*.

The problem of speeding up pattern matching remains open and a wealth of new research projects is being conducted. More recently, Prasad et al. introduced an approximate multiple pattern matching algorithm [26] that finds all occurrences of a set of patterns in the searched text with minimum false negatives. It does not require verification and can handle long patterns efficiently. Pendlimarri et al. proposed a multiple pattern matching algorithm in [27] by designing a new version of the dynamic programming-based (DP) algorithm. They enhanced the algorithm to work as a multiple instead of single pattern matching algorithm. The proposed algorithm preprocesses the searched text once before starting the search phase. The performance of this algorithm is better than the BM algorithm with almost 33%-91%, and 37%-85% than Quick Search algorithm.

Parallelism has sparked the attention of researchers in the field. Thinh et al. parallelized Cuckoo hashing on an FPGA to achieve up to 8.8 Gbps for four characters sliding window [28]. They report area savings of 30% with high flexibility to

add new patterns without hardware reconfiguration. Kim et al. partitioned the patterns into subgroups based on balancing the patterns length for each matching unit [29]. Later Kim et al. used a bit-split parallel string matching to reduce the number of states and state transitions [30]. The use of the balanced length pattern portioning resulted in homogeneous matching units and an overall 47.8% reduction in memory usage.

Luchaup et al. exploited data parallelism by dividing the packet into independent chunks [31]. They scanned multiple byes using speculation to guess the next state of the DFA. The incorrect speculations are corrected later and on average the reported a small memory usage and throughput improvements on commodity processors. In order to benefit from the speculation they later used Cell architecture to achieve a 1.6 speedup and a 1.33 Gbps throughput using 8 threads [32]. Their solution still suffers from the limited storage available on the RISC based processing units in the Cell architecture and the extra overhead added by speculation.

V. The Algorithm

In this paper we aim to reduce the pattern matching cost by dividing the pattern matching task over several threads. To this end, we partition IDS signatures into several classes based on two factors: pattern length, and percentage of occurrence of each length in the signatures database. Most of the previous algorithms relied on the number of signatures which is misleading because of the variation in signatures lengths. Rather we use the the number of characters which represents the actual workload. The goal is to have equal workloads for each thread. Threads run different algorithms depending on their assigned signatures class properties. Before we can explain the hybrid algorithm we have to divide the signatures between AC and WM algorithms.

A. *Signatures partitioning algorithm*

We study attack signatures extracted from Snort 2.8.4.1 database published in 2009 [33]. Figure 5 plots the average pattern length and the percentage of occurrence of each pattern length. We observe that the majority of the signatures lie in length interval of 3 to 20. The area under the curve represents the number of characters in that range. We approximate the area by calculating the area of rectangles as opposed to calculating the integral. Algorithm 1, partitioning algorithm, divides the signatures based on the area under the curve as marked by colored rectangles shown in Figure 5. The first interval was chosen based on, experimental AC performance measurement and the number of threads to be used. The adopted partitioning was corroborated by experimental results reported in previous work. Faro et al. [34] split the signatures based on theoretical assumptions into four classes: very short ($m \leq 4$), short ($4 < m \leq 32$), long ($32 < m \leq 256$) and very long patterns ($m > 256$). A simple examination of Figure 5 shows that this partitioning does not even come close to load balancing. In fact, 74.6% of signatures have a length less than twenty characters and would fall under the first two classes: very short and short.

Algorithm 1 partitions the signatures into 7 classes with almost equal workload and average area under the curve of 14. The remaining signatures are summed up in class 7 as shown in Table 2.

B. *The hybrid algorithm*

The premise is to exploit the fact that WM is inefficient for short strings and AC performance degrades for longer strings. WM performance depends on several factors. The main factor is the maximum shift length which is bound by the length of the shortest pattern in the signatures database. The maximum shift determines the maximum text skipped without search. The longer the strings the better WM will perform. On the other hand, AC performance depends on the number of states to be traversed. The number of states grows exponentially with the number of strings or characters. Therefore AC performs better for shorter strings.

The Hybrid algorithm uses AC threads to match pattern classes with short signatures length, and WM threads to match patterns with relatively longer length. In other words, the hybrid algorithm tries to make the best of both. Each packet will be processed by all threads. The number of threads is determined experimentally as shown next.

VI. Experimental Results

We bring forward a comprehensive set of experiments to evaluate the performance of the proposed algorithm. Experimental time and memory measurements are carried out using actual traffic traces. Subsection A explains the simulation environment and performance metrics used for evaluation. Subsection B presents the various configurations in search of the perfect combination of AM and WM threads. Subsection C evaluates the relative performance for the best hybrid multithreading configuration. Finally, Subsection D evaluates how the algorithm performance scales with increasing number of threads.

A. *Simulation environment and performance metrics*

We implement the hybrid algorithm using C++ under Windows operation system and MS Visual Studio 2010. The simulator reads packets and signatures form text files, performs the matching using the target algorithm and measures time and memory to be compared against the original AC and WM algorithms. Numbers reported in following subsections are the average of five runs. Simulations are conducted on a workstation with Intel Core 2 Quad 2.60GHz processor and 4GB of memory. To measure the execution time, we use CodeProject ExecutionStopwatch class and we use the GlobalMemoryStatusEx function to measure the memory usage.

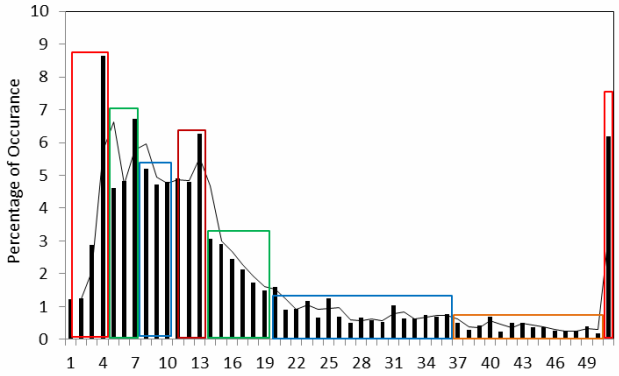

Figure 5. WM scanning process

Algorithm 1 *Patterns Partitioning*

```
1:  procedure Partition ()
2:  Input: patterns file
3:  Output: subsets of patterns with equal number of character
4:  nI = no. of patterns of length I; initial =1
5:  % of occurrence of length i = nI / (total no. of patterns)
6:  i = initial +1
7:  Sum= % of occurrence of patterns of length1
8:  While(!EOF)
9:     Sum = Sum + % of occurrence of patterns of length i
10:    Average % of occurrence= Sum / (i-initial)
11:    Area = Average % of occurrence×( i-initial)
12:    If Area =14 then
13:        interval={initial, i}
14:     initial= i+1
15:     i=i+1
16: end While
17: end procedure
```

Class No.	Length Interval	Class No.	Length Interval
0	1-4	4	14-19
1	5-7	5	20-36
2	8-10	6	37-50
3	11-13	7	>50

Table 2. Snort signatures partitions

B. Traffic and signatures extraction

To evaluate the algorithm we use the extracted Snort signatures partitioned earlier [33]. As far as packet traces are concerned we adopt the good, bad and ugly traces studied in our previous work [35]. The classification is based on the maliciousness of packets defined as the percentage of Snort signatures found in the trace. For the worst case performance evaluation we use the bad and ugly traces extracted from the 2009 Capture the Flag (CTF) competition held at DEFCON17 conference [36]. Out of the 78 attack infested traces we pick the most malicious traces 51 and 52 as the ugly traces, and the least malicious traces 1 and 22 as the bad traces.

To evaluate the best case performance we use the good traces extracted from normal traffic. The traces are extracted as follows. SIP is a video and audio streaming trace from 2009 TechTraces [37], Web is email traffic taken from 2008 TkuIM Mail [38], LC is a live chat trace and GD is a good download traffic from 2007 v8 Laura's Lab Kit [39].

C. Optimal combination

First we start with only four threads (two signature classes per thread) with the intention to find the best hybrid combination of AC and WM threads. The first experiment, named, 1TAC_3TWM, starts with four threads: one thread running AC (short patterns in classes 0 and 1) and the other three running WM for the remaining signature classes. The name, 1TAC_3TWM, is read as follows: One thread running AC and 3 threads running WM. We run experiments for all possible combinations: 4T_AC, 1TAC_3TWM, 2TAC_ 2TWM, 3TAC_1TWM and 4T_WM. In addition, we simulate two more configurations with only one thread of AC and one thread of WM, they are named: 1T_AC, 1T_WM, respectively. The last two experiments represent the classical no threading runtime for the original algorithms.

For all seven configurations outlined earlier, Table 3 shows the matching runtime in seconds and Table 4 shows the memory consumption in MBs. It is evidently clear that 1TAC_3TWM has the smallest runtime for all traces: the good, bad and the ugly. In terms of memory usage 1TAC_3TWM is the best when it comes to worst case bad and ugly traces. For the good traces a one thread WM slightly tops 1TAC_3TWM. In general you can see from columns one and two that one thread WM outperforms AC in terms of run time and memory for all traces. It can be also observed that the memory consumption for WM is almost steady when we move to more malicious traces while AC memory requirements increase sharply. Finally, the last two columns show that the overhead caused by threading in the cases of four AC and WM threads renders multithreading useless. Those conclusions confirm the original thinking that AC and WM perform very well for short and long signatures, respectively.

D. A closer look

Next we analyze the performance of the best combination, that is, 1TAC_3TWM. The column chart in Figure 6 compares the runtime for 1TAC_3TWM to the one and four threads original algorithms for all traces. While the column chart in Figure 7 compares the memory requirements for 1TAC_3TWM to the one and four threads original algorithms for all traces. Both figures show a significant reduction in both runtime and memory usage. On average for all traces, 1TAC_3TWM reduces the runtime by 41% and 33% compared to 1T_AC and 1T_WM, respectively. Best case runtime reduction is reported for GD traces of 50% and 40% compared to 1T_AC and 1T_WM, respectively. Additionally, on average for all traces, 1TAC_3TWM reduces the memory usage by 42% and 9% compared to 1T_AC and 1T_WM, respectively. However if we compare to 4T_AC and 4T_WM the average reduction in runtime is 50%, 55% and the average the reduction in memory usage is 54% and 37%, respectively. The best case runtime reduction compared to 4T_AC and 4T_WM becomes 63% and 73%.

		Experiment						
		1T_ WM	1T_ AC	1TAC_ 3TWM	2TAC_ 2TWM	3TAC_ 1TWM	4T_ AC	4T_ WM
Good	GD	15.06	18.34	9.11	13.53	12.97	19.68	33.50
	LC	11.83	11.81	7.31	9.23	8.31	12.66	23.97
	WEB	14.12	18.31	9.76	10.51	11.44	13.66	30.67
	SIP	13.99	17.59	9.46	10.27	10.65	13.63	30.93
Bad	1	78.41	81.76	51.89	67.10	138.81	142.19	88.96
	22	77.83	84.13	52.01	66.92	138.67	141.59	87.22
Ugly	51	79.23	88.75	52.71	67.42	139.83	141.41	88.73
	52	78.83	88.27	58.83	67.34	139.20	141.70	88.68

Table 3. Execution runtime in seconds

		Experiment						
		1T_ WM	1T_ AC	1TAC_ 3TWM	2TAC_ 2TWM	3TAC_ 1TWM	4T_ AC	4T_ WM
Good	GD	25.04	36.56	25.21	35.555	40.05	44.97	33.44
	LC	26.32	37.08	25.56	35.328	41.68	43.99	33.06
	WEB	26.62	36.56	26.75	35.602	41.71	44.07	33.81
	SIP	26.55	36.46	26.69	35.218	42.07	46.28	33.06
Bad	1	33.83	55.77	27.73	65.238	83.59	84.37	55.35
	22	33.84	56.11	28.37	63.106	82.46	85.11	56.36
Ugly	51	33.62	69.11	27.64	60.83	84.90	83.68	58.85
	52	34.29	69.99	27.61	60.916	83.48	83.95	58.91

Table 4. Memory usage in MBs

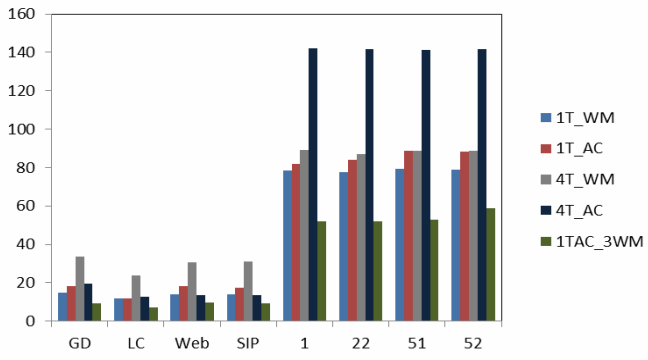

Figure 6. 1TAC_3TWM runtime vs. original algorithms for all traces

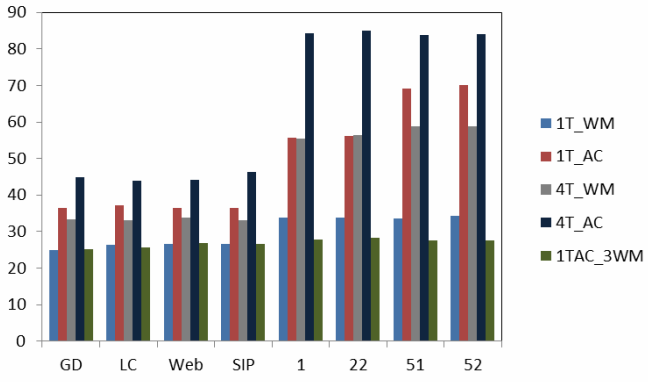

Figure 7. 1TAC_3TWM memory usage vs. original algorithms for all traces

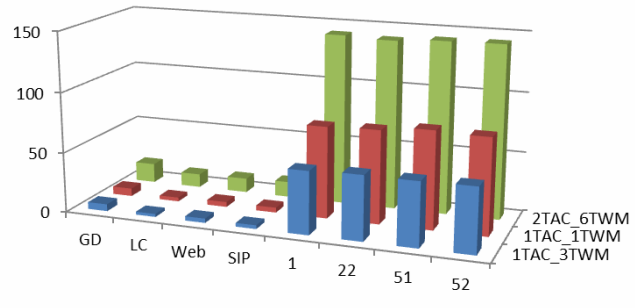

Figure 8. 1TAC_3TWM runtime for varying thread configurations

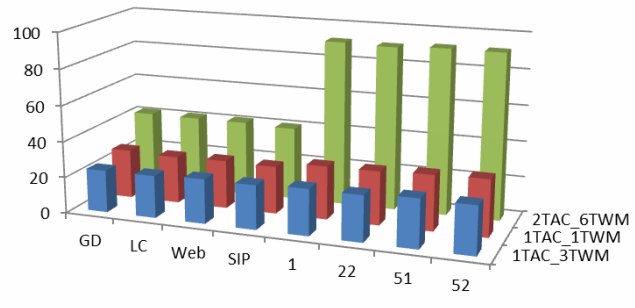

Figure 9. 1TAC_3TWM memory usage vs. original algorithms for all traces

E. Performance scaling

Finally, we vary the number of threads to check how the performance of the hybrid algorithm scales for increasing numbers of threads. Unfortunately, we do not have access to a workstation with 8 cores to better simulate eight threads. The 3D column charts in Figures 8 and 9 compare the runtime and memory requirements for 1TAC_1TWM, 1TAC_3TWM and 2TAC_6TWM for all traces. In both cases increasing the number of threads hurts performance because of the increasing number of copies of the packet. 1TAC_3TWM remains the favorable best performing combination of AC and WM threads.

VII. Conclusions

Network intrusion detection system inspects the incoming traffic to find attack signatures. The most intensive operation in the detection process is pattern matching in terms of CPU and memory requirements. In this paper we propose a hybrid AC and WM multithreaded algorithm to improve the runtime memory usage of the matching operation. We propose a novel partitioning algorithm used to balance the load between the threads to maximize the performance of the hybrid algorithm. In terms of average search time the hybrid algorithm provides 41% and 33% reduction compared to 1T_AC and 1T_WM, respectively. Best case runtime reduction is reported for GD traces of 50% and 40% compared to 1T_AC and 1T_WM, respectively.

References

[1] M. Fisk and G. Varghese. “An analysis of fast string matching applied to content-based forwarding and intrusion detection”. *Technical Report CS2001-0670*, University of California, San Diego, 2002.

[2] M. Roesch. “Snort – Lightweight Intrusion Detection for Networks”. In *Proceedings of USENIX (LISA’99)*, pp. 229-238, Seattle, WA, 1999.

[3] R. Rehman. “*Intrusion Detection Systems with Snort Advanced IDS Techniques Using Snort*”, Prentice Hall PTR, New Jersey, 2003.

[4] M. Aldwairi, T. Conte and P. Franzon. “Configurable String Matching Hardware for Speeding up Intrusion Detection”, *ACM SIGARCH Computer Architecture News*, 33(1), pp. 99-107, 2005.

[5] A. Aho and M. Corasick. “Efficient string matching: An aid to bibliographic search”, *Communications of the ACM*, 18(6), pp. 333-340, 1975.

[6] S. Wu, U. Manber. “A Fast Algorithm for Multi-Pattern Searching”. *Technical Report TR-94-17*, University of Arizona, Tuscon, 1994.

[7] S. Snapp, J. Brentano, G. Dias, T. Goan, L. Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal and D. Mansur. “DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an early Prototype”. In *Proceedings of the 14th National Computer Security Conference*, pp.167-176, Oct 1991.

[8] P. Wheeler and E. Fulp. “Taxonomy of Parallel Techniques for Intrusion Detection”. In *Proceedings of ACM 45th Southeast Regional Conference*, pp. 278–282, 2007.

[9] P. Wheeler. “Techniques for Improving the Performance of Signature-Based Network Intrusion Detection Systems”. *MS Thesis*. University of California Davis, CA, 2006.

[10] R. Boyer and S. Moore. “A Fast String Searching Algorithm”, *Communications of the ACM*, 20(10), pp. 762-772, 1977.

[11] C. Kopek. “Parallel Intrusion Detection Systems For High Speed Networks Using The Divided Data Parallel Method”. *MS Thesis*, Wake Forest University, Winston-Salem, NC 2007.

[12] B. Zhang, X. Chen, X. Pan and Z. Wu. “High Concurrence Wu-Manber Multiple Patterns Matching Algorithm”. In *Proceedings of the 2009 International Symposium on Information Processing (ISIP’09)*, Huangshan, , pp. 404-409, P. R. China, Aug 2009.

[13] E. Fulp and R. Farley. "A Function-Parallel Architecture for High-Speed Firewalls". In *Proceedings of the IEEE International Conference on Communications (ICC'06)*, pp.2213-2218, Jun 2006.

[14] S. Kim and Y. Kim. “A Fast Multiple String-Pattern Matching Algorithm”. In *Proceedings of the 17th AoM/IAoM International Conference on Computer Science*, San Diego, 1999.

[15] C. Liu, D. Liu and D. Li. “Two Improved Single Pattern Matching Algorithms”. In *Proceedings of the 16th International Conference on Artificial Reality and Telexistence-Workshops*, pp.419-422, Hangzhou, China, 2006.

[16] Y. Takahashi, H. Tanaka, A. Shio and S. Ohtsuka. "Log-derivative Matching Method for Pattern Comparison", In *Proceedings of the SPIE 3653*, pp.956-965, San Jose, CA, 1998.

[17] N. Rafla, I. Gauba. “A Reconfigurable Pattern Matching Hardware Implementation using On-Chip RAM-Based FSM”, In *Proceedings of the 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS),* pp.49-52, Seattle, WA, Aug 2010.

[18] S. Fide, S. Jenks. “A Survey of String Matching Approaches in Hardware”. *Technical Report*, University of California, Irvine, 2006.

[19] Y. Weinsberg, S. Tzur-David, D. Dolev and T. Anker. “One Algorithm to Match Them All: On a Generic NIPS Pattern Matching Algorithm”. In *Proceedings of the High Performance Switching and Routing Conference*, pp.1-6, Brooklyn, NY, 2007.

[20] L. Woods, J. Teubner, G. Alonso. “Real-Time Pattern Matching with FPGAs”. In *Proceedings of the 27th International Conference on Data Engineering (ICDE), Demonstration Track*, pp.1292-1295, Hannover, Germany, Apr 2011.

[21] T. Tuck, T. Sherwood, B. Calder, and G. Varghese. “Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection”. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM),* pp.2628-2639, Hong Kong, Mar 2004.

[22] Y. Hong, X. Ke and C. Yong. “An Improved Wu-Manber Multiple Patterns Matching Algorithm”. In *Proceedings of the 25th IEEE International Performance, Computing, and Communications Conference (IPCCC'06),* Phoenix, AZ, Apr 2006.

[23] D. Sunday. “A Very Fast Substring Search Algorithm”, *Communications of the ACM*, 33(8), pp.132-142, 1990.

[24] W. Lu and I. Traore. “Detecting New Forms of Network Intrusion using Genetic Programming”, *Computational Intelligence*, 20(3), pp.475–494, 2004.

[25] I. Moraru, David and D. Andersen. “Exact Pattern Matching with Feed-Forward Bloom Filters". In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX11)*, Jan 2011.

[26] R. Prasad, A. Sharma, A. Singh, S. Agarwal and S. Misra. “Efficient bit-parallel multi-patterns approximate string matching algorithms”, *Scientific Research and Essays*, 6(4), pp.876-881, 2011.

[27] D. Pendlimarri, P. Petlu, R. Satrasala. “Novel Devaki-Paul Algorithm for Multiple Pattern Matching”, *International Journal of Computer Applications*, 13(3), 37-42, 2011.

[28] T. N. Thinh and S. Kittitornkun. "Massively Parallel Cuckoo Pattern Matching Applied for NIDS/NIPS", In *Proceedings of the Fifth IEEE International Symposium on Electronic Design, Test and Application (DELTA '10)*, pp.217-221, Ho Chi Minh City, Vietnam,13-15 Jan. 2010.

[29] H. Kim and S. Kang. "A Pattern Group Partitioning for Parallel String Matching using a Pattern Grouping Metric", *IEEE Communications Letters*, 14(9), pp.878-880, Sept 2010.

[30] H. Kim, H. Kim and S. Kang. "A Memory-Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems", *IEEE Transactions on Parallel and Distributed Systems*, 22(11), pp.1904-1911, Nov 2011.

[31] D. Luchaup, R. Smith, C. Estan and S. Jha. "Speculative Parallel Pattern Matching", *IEEE Transactions on Information Forensics and Security*, 6(2), pp.438-451, Jun 2011.

[32] C. Radu, C. Leordeanu, V. Cristea and D. Luchaup. “Using Cell Processors for Intrusion Detection through Regular Expression Matching with Speculation”, In *Proceedings of the 2011 International Conference Complex, Intelligent and Software Intensive Systems (CISIS),* pp.203-210, Seoul, Korea, Jun 30-Jul 2 2011

[33] Snort rules, http://www.snort.org/snort-rules/, last access in April, 2011.

[34] S. Faro, T. Lecroq. “The Exact String Matching Problem: a Comprehensive Experimental Evaluation”, *CoRR*, abs/1012.2547, 2010.

[35] Monther Aldwairi and Duaa AL-ansari,“Exscind: Fast Pattern Matching For Intrusion Detection Using Exclusion and Inclusion Filters”. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices (NWeSP 2011)*, Salamanca, Spain, Oct 2011.

[36] DEFCON17 traffic traces, accessed in April, 2011, http://www.defcon.org.

[37] TechTraces sample captures, accessed in April, 2011, http://techtraces.com/sample_captures/.

[38] TkuIM mail, accessed in April, 2011, http://mail.im.tku.edu.tw/~miller.lai/pcap/pcapList.php.

[39] Laura's Lab Kit v.8, accessed in April, 2011, http://demeter.uni-regensburg.de/Lauras_Lab_Kit_v8/ AutoPlay/ trace_files_llk8/.

Author Biographies

Monther Aldwairi was born in Irbid, Jordan in 1976. Aldwairi received a B.S. in electrical engineering from Jordan University of Science and University (JUST) in 1998, and his M.S. and PhD degrees in computer engineering from North Carolina State University (NCSU), Raleigh, NC in 2001 and 2006, respectively.

He is an Assistant Professor of Network Engineering and Security Department at Jordan University of Science and Technology, where he has been since 2007. He is also the Vice Dean of Faculty of Computer and Information Technology since 2010 and was the Assistant Dean for Student Affairs in 2009. In addition, he is an Adjunct Professor at New York Institute of Technology (NYiT) since 2009. He worked as Post-Doctoral Research Associate in 2007 and as a research assistant at NCSU from 2001 to 2006. He interned at Borland Software Corporation in 2001. He worked as a system integration engineer for ARAMEX from 1998 to 2000. His research interests are in network and web security, intrusion detection and forensics, artificial intelligence, pattern matching, natural language processing and bioinformatics. He published several well cited articles.

Dr. Aldwairi is an IEEE and Jordan association of engineers' member. He served at the steering and TPC committees of renowned conferences and he is a reviewer for several periodicals. He organized the Imagine cup 2011-Jordan and the national technology parade 2010.

Niveen Ekailan was born in Amman, Jordan on October 1981. Eng. Ekailan received the B.S. degree in Electrical and Computer Engineering from Palestine Polytechnic University, Hebron in 2004. She received her M.S. in computer engineering from Jordan University of Science and Technology, Irbid in Aug of 2011.