Journal of Digital Forensics,
Security and Law

Volume 11 | Number 4 Article 5

12-31-2016

Bloom Filters Optimized Wu-Manber for Intrusion Detection

Monther Aldwairi
Network Engineering and Security Department, Jordan University of Science and Technology

Koloud Al-Khamaiseh
cDepartment of Electrical Engineering, Tafila Technical University

Fatima Alharbi
College of Technological Innovation, Zayed University

Babar Shah
College of Technological Innovation, Zayed University

Recommended Citation
Aldwairi, Monther; Al-Khamaiseh, Koloud; Alharbi, Fatima; and Shah, Babar (2016) "Bloom Filters Optimized Wu-Manber for Intrusion Detection," *Journal of Digital Forensics, Security and Law*: Vol. 11 : No. 4 , Article 5.
DOI: https://doi.org/10.15394/jdfsl.2016.1427
Available at: https://commons.erau.edu/jdfsl/vol11/iss4/5

BLOOM FILTERS OPTIMIZED WU-MANBER FOR INTRUSION DETECTION

Monther Aldwairi[a,b,*], Koloud Al-Khamaiseh[c], Fatima Alharbi[b] and Babar Shah[b]
[a]Network Engineering and Security Department, Jordan University of Science and Technology, Irbid 22110, Jordan
[b]College of Technological Innovation, Zayed University, Abu Dhabi, P.O. Box 144534, U.A.E
[c]Department of Electrical Engineering, Tafila Technical University, Tafila 66110, Jordan
[*]Corresponding author. E-mail: munzer@just.edu.jo

ABSTRACT

With increasing number and severity of attacks, monitoring ingress and egress network traffic is becoming essential everyday task. Intrusion detection systems are the main tools for capturing and searching network traffic for potential harm. Signature-based intrusion detection systems are the most widely used, and they simply use a pattern matching algorithms to locate attack signatures in intercepted network traffic. Pattern matching algorithms are very expensive in terms of running time and memory usage, leaving intrusion detection systems unable to detect attacks in real-time. We propose a Bloom filters optimized Wu-Manber pattern matching algorithm to speed up intrusion detection. The Bloom filter programs the hash table into a vector, which is quickly queried to exclude unnecessary searches. On average hash table searches are avoided 10.6% of the time. The proposed algorithm achieves a best-case speedup of 66% and worst-case speedup of 33% over Wu-Manber at the cost of 0.33% memory usage increase.

Keywords: network security, intrusion detection systems, pattern matching, Wu-Manber, Bloom filters

1. INTRODUCTION[1]

Internet connectivity has become essential and a basic requirement for any kind of business. More applications require Internet connectivity to install, update, run or function correctly. In addition, cloud computing, social networks and Internet of things have placed a lot of private data at risk of being exposed to the increasing number of high skilled attackers. The number and complexity of attacks have been increasing at an alarming rate against banks, hospitals, governments and other companies. New vulnerabilities, exploits and attacks are announced daily at an alarming rate (Roberts, 2000).

Intrusion detection systems (IDS) are a widely-used control measure to inspect network traffic in order to detect and sometimes block attacks. IDSs are classified based on deployment into two main types: network and host-based systems. Network-based intrusion detection systems (NIDS) scan all ingress and egress traffic looking for attacks and are generally seen in the form of dedicated IDS appliance. While host-based intrusion detection systems (HIDS) on the other hand, are a software agent running on a specific host that monitors all activity looking for malicious events. Hardware IDS appliances cost

[1] Proceedings of the 2nd World Symposium on Web Applications and Networking (WSWAN'2015) (2015), IEEE.

hundreds of thousands of dollars, that is why software-based IDS running on PC workstations remain widely deployed. However, software-based systems are unable to keep up with the ever-increasing Internet speed (Zheng, Cai, Zhang, Wang, & Yang, 2015). Unlike IDSs, Intrusion prevention systems (IPS) are a new breed of proactive IDS that are deployed inline to detect malicious activity in real-time and take corrective action. IPSs can log the activities, alarm administrators, or drop connections. They have not been widely adopted due to users not favoring automatic dropping of sessions or packets.

Intrusion detection systems are categorized based on the technique into: signature and anomaly-based (Aldwairi, 2006). Signature-based intrusion detection systems detect known attacks by searching network traffic for attack signatures. They generally use traditional pattern matching algorithms and yield better speed and accuracy compared to anomaly detection. The signatures are manually written after security analysts study the captured attack or malware code looking for invariant parts. The manually developed signatures are a big disadvantage in terms of signatures accuracy and the fact that it takes a considerable amount of time to provide a signature after a new attack is detected (Jirachan, & Piromsopa, 2015). On the other hand, anomaly detection builds a profile of the normal system behavior during the training phase. It uses common machine learning classifiers to extract features from new traffic and classify them into benign or malicious. The profiles are based on statistical analysis to capture specific behavior patterns such as system calls. Proprietary rule based languages are used to capture those profiles in isolated setup. It is true that anomaly-based IDSs detect new attacks, however they are considerably slow and generate more false positives and negatives as opposed to signature-based (Aldwairi, Khamayseh, & Al-Masri, 2015).

Signature-based IDSs continue to dominate the market, with Snort being one of the most commonly deployed systems (Roesch, 1999). Snort (2016) has been the target of numerous studies and became the de facto among researchers working to speed up pattern matching algorithms for IDS. Simply, Snort inspects network traffic trying to match packets against predefined rules. It has many other capabilities such as packet capture and reassembly (Lam, Mitzenmacher, & Varghese, 2010). However, this work is concerned only with pattern matching, which dominates Snorts performance. Antonatos, Anagnostakis, and Markatos (2004) found that pattern matching algorithms consume up to 70% of Snort running time. To make matters worse, as new attacks arise, the number of signatures grows exacerbating the performance issue. Snort rules examine the packets header and search the packets payload for attack signatures (Aldwairi, & Alansari, 2011). However, the majority of the rules contain one or more signatures. Almost 87% of Snort rules contain signatures to match against (Aldwairi, Conte, & Franzon, 2004). Therefore, there is still a need to speedup pattern matching for intrusion detection (Gharaee, Seifi, & Monsefan, 2014).

There is surge of studies to improve pattern matching for intrusion detection whether in hardware or software. Dharmaprikar, Krishnamurthy, Sproull, & Lockwood, (2004) proposed hardware parallel Bloom filters to exclude benign packets. But because Bloom filters only work with fixed length signatures, they were forced to use many parallel Bloom filters. Bearing in mind that Snort signatures lengths can be over *1000* characters, this solution ends up being very expensive in terms of memory. We will show later that each Bloom vector can grow up to

1MB in size, having thousands of those is not quite efficient. It is worth pointing out that Bloom filters are used in a more efficient way in this paper. We program only the *B* character prefixes of the sparse hash table to avoid unnecessary expensive hash table searches. Consequently, one Bloom filter is used as opposed to one for each signature's length in case of Dharmaprikar et al. (2004).

Yang, Xu, & Cui (2006) improved Wu-Manber (QWM) using Quick Search (QS) algorithm (Sunday, 1990) and mismatch information, to increase the shift values. Quick Search is basically used to find if a packet contains a prefix of an attack signature. If a prefix is found QWM then uses Wu-Manber (WM) to verify the match. To achieve that a fourth table is added, the HEAD table. The table decides if the first two characters of a matching window are the prefix of a pattern. QWM was designed to outperform WM for Chinese texts with large alphabets as opposed to network traffic with limited character set. In addition, a considerable memory overhead is added due to the additional HEAD table.

WM+ by Xunxun, Binxing, Lei, and Yu (2005) merged Aho-Corasick (AC) and Wu-Manber algorithms to improve the shift table. WM+ algorithm derived a prefix automata scanning from AC instead of the ordinary hash table based pattern matching. In addition, a filtering algorithm was used along with the finite automata to skip the bad characters in order to speed up the search. Unfortunately, for longer patterns lengths the memory consumption of WM+ is significantly larger than WM. On top of that, the finite automata construction adds a considerable overhead.

Older Snort versions implemented Aho-Corasick, and a lot of researches were performed on optimizing AC automata. Liu, Chen, Wu, and Wu (2011) proposed a finite automata with extended character set to reduce the number of states, which is the main disadvantage of AC. They used auxiliary variables to compress the number of states while maintaining one memory access per byte.

Newer versions of Snort opted out to implement a modified Wu-Manber (MWM) (Beale, Baker, Esler, & Northcutt, 2007). WM is more attractive because of the smaller memory requirements and better performance for longer strings. That is possible because WM is conservative in that the maximum shift possible is *m–B+1*, which depends on the minimum string length. MWM examines the suffixes of the block in order to change the default shift value. The modified WM can have a larger shift equivalent to the block size if the no pattern contains any block suffixes.

To overcome the degrading performance as the number of signatures increases, Peng, Wang, and Xue (2014) proposed a new enhanced Wu-Manber. They optimized WM by minimizing number of candidate patterns in the HASH table and using binary search to look for candidate patterns in the index table to cut the searching time. Experimental results showed that in case of large pattern sets ($> 3\times10^5$), the enhanced algorithm is more efficient than the classical WM, MWM, and TFD algorithms. This is due to the fact that in the enhanced algorithm, the hash table was well balanced and the binary search helped reduce the search time.

Zhang (2016) modified WM to suit matching short bit streams for wireless communication protocols. The algorithm added a new GSSHIFT table to determine the shift distance when the SHIFT table returns zero. They achieved speedup, over WM, of 1.6 times for 5 bit patterns. However, the algorithm scaled very poorly with string's length, with no improvement for strings longer than 64 bits.

Finally, Lee, Woo, and An (2016) modified WM using multiple sub-patterns on multi-core CPU. However, the modified algorithm had poor performance for large number of signatures and did not improve time proportional to the number of cores used.

This paper presents Exhaust: a modified version of Wu-Manber with negligible overhead. Exhaust is designed specifically to speed up pattern matching for intrusion detection systems to match higher network speeds. The main contribution is to insert only one Bloom filter to wither out unnecessary hash table searches (Aldwairi, & Al-Khamaiseh, 2015). It results in a considerable improvement on the overall performance with minimal overhead. The rest of the paper is organized as follows. Section 2 explains the basic knowledge required to understand the problem. It explains Snort rules in full details, pattern matching algorithms, Wu-Manber and Bloom filters theory. Section 3 describes Exhaust inner workings and details the initialization and search phases. Section 4 brings forward a complete formal and experimental validation of the proposed algorithm using actual traffic traces and attack signatures.

2. BACKGROUND

This section explains Snort and its rules format. An example of actual Snort rules and attack signatures is presented. Subsection 2 presents a pattern matching algorithms overview and provides a thorough WM explanation with preprocessing and search examples using real Snort signatures. Finally, a brief introduction to Bloom filters is set forwards.

2.1 Snort

Snort is a popular open source IDS from Sourcefire which has recently been acquired by CISCO. We're mostly concerned with Snorts' rules that contain attack signatures. The rules are in plaintext and describe set of conditions for the packet's header/payload to match. The rules' headers field specifies the action to be taken and provides values for the protocol type, source and destination IP addresses and port numbers. The options field contains more than twenty-four keyword and value pairs, such as: *msg* for the alert message, *sid* for signature identification number, *priority* gives rules' severity level, and *class-type* to categorize rules. The rules options also contain several *content*, *uricontent* and *pcre* keywords that specify attack signatures (Beale, Baker, Esler, & Northcutt, 2007).

Figure 1 shows a redacted Snort v2.8 rule from ddos.rules rule set. You can easily extract the attack signature, "gOrave", from the content keyword. The rule is very easy to read: fire an alert if any external TCP packet going to any local machine on port *27665* while containing the string "gOrave". This rule detects a well-known old DDoS attack called, Trin00 (Dittrich, 2015).

(Kharbutli, Aldwairi, & Mughrabi (2012) identified pattern matching to locate the attack signatures in the packet payload, as the main bottleneck. Despite Snort using the fastest pattern matching available, it still lags behind increasing network access speeds.

```
alert tcp $EXTERNAL_NET any ->
$HOME_NET 27665 (msg:"DDOS Trin00
Attacker to Master default
password"; content:"gOrave";
classtype:attempted-dos; sid:234)
```

Figure 1. A sample Snort rule

Snort relies on exact pattern matching algorithms and does not use regular expressions for encoding signatures. Pattern matching is classified into either single or multiple pattern matching. Single pattern matching must scan the packet once for each signature in the dataset, which makes it counterproductive. They are not used in IDS, but it is a good introductory example to pattern matching. Boyer-Moore (BM) is one of the most common single pattern matching algorithms. In an effort to locate a match, it places the pattern and packet side by side and shifts the pattern by one position in the case of mismatch. In the event of a match it moves to match the next character from the pattern with subsequent character from the packet. The algorithm is fairly simple and inefficient, because the search time grows linearly with the packets' and patterns' lengths. Several improvements over BM exist such as good and bad character heuristics as well as Boyer-Moore-Horspool (Boyer, & Moore, 1977).

On the other hand, multiple pattern matching algorithms preprogram all patterns into a table or tree and match all patterns at the same time. The additional preprocessing stage is the obvious drawback, but this pales in comparison to the savings attained from traversing the packet once. Aho-Corasick and Wu-Manber are two of the fastest multiple pattern algorithms to date.

In the preprocessing phase, AC builds a trie based state machine from the set of patterns to be matched (Aho, & Corasick, 1975). AC search time is linearly proportional to the searched packet length and is not affected by the number of characters in the signatures. However, AC preprocessing time and complexity increases exponentially with the number of characters, which makes it ideal only for short signatures. Moreover, the state machine needs to be rebuilt every time a new pattern is added to the signatures database. Unfortunately, AC memory requirements scale exponentially with increasing number of signatures. Wu-Manber algorithm on the other hand, is based on hash tables, which makes it more attractive option compared to AC for longer signatures (Wu, & Manber, 1994).

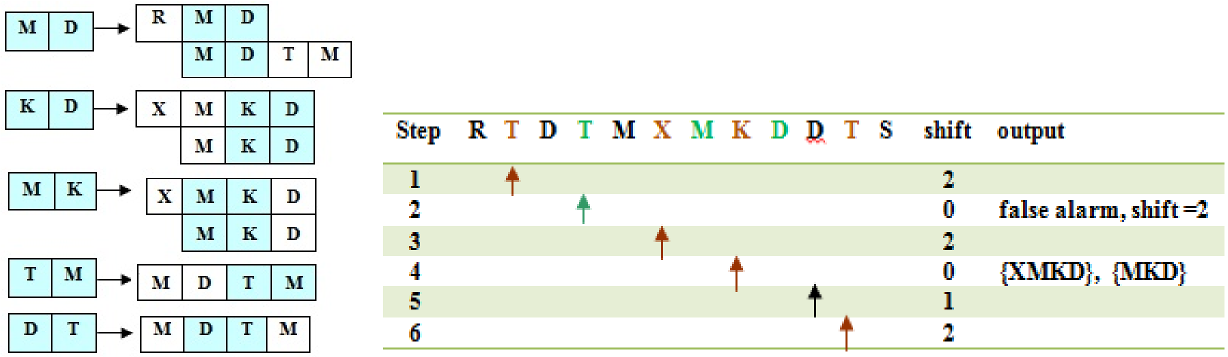
2.3. Wu-Manber algorithm

Wu-Manber relies on the same principles used in Boyer-Moore algorithm (Boyer, & Moore, 1977), but adds a block of B characters and new data structure for more efficient matching. Like all multiple pattern matching algorithms, WM has two stages: preprocessing and search. The preprocessing stage of the algorithm starts by computing the minimum length m of all patterns that are available beforehand. Then it defines a block of B characters used for matching window shifts. The block size is recommended to be either two or three. Then the algorithm builds three tables during pattern preprocessing: shift, hash, and prefix

The shift table is constructed by computing the shift value for each substring of size B taken from the first m characters of the pattern. The shift table is a hash table where the key is the signature substring and its value is computed using the equation *shift [key]=m-q*, where q denotes the right most location in the pattern substring. The default value of this table is defined by the equation *shift [key] = m-B+1*. The shift value represents the number of maximum characters to skip forward when a mismatch occurs. The character blocks that have a shift value of zero indicate a probable match. All patterns that share those zero shifts probable matches are programmed into the hash table. The main purpose of the prefix table is to make finding probable matches in the hash table faster by hashing the prefixes of those patterns. Additionally, it is used to distinguish between the patterns that have the same suffix but differ in the prefix.

The search starts by dividing the network traffic traces into sliding window of size B. Each time the search string of size B returns a zero shift value when traversing the shift table, the algorithm accesses the hash table and searches the list of patterns associated with the key to find the match.

An example with actual signatures is presented to better understand the algorithm. Table 1 shows the shift table for the following patterns extracted from the Snort FTP rule set {RMD, XMKD, MDTM, MKD} for block size of two characters, $B=2$. The minimum pattern length is three characters, $m=3$ and the default shift value is $m–B +1$, which equals $3–2+1 = 2$. Take the block "DT", which exists in pattern "MDTM". The shift value is $m - q$, where q is the rightmost occurrence of DT in any pattern, hence, shift [DT] is $3–2=1$. On the other hand, take block "MD", which can be found in two patterns "RMD" and "MDTM". The shift [MD] is $3–3=0$ taken from pattern "RMD" and not $3–2=1$ as in pattern "MDTM". Figure 2 (a) shows the hash table, which holds pointers to the patterns that contain the probable matching signatures.



Step	R	T	D	T	M	X	M	K	D	D	T	S	shift	output
1		↑											2	
2				↑									0	false alarm, shift =2
3						↑							2	
4								↑					0	{XMKD}, {MKD}
5										↑			1	
6											↑		2	

Figure 2 (a). Wu-Manber hash table; *(b)* Wu-Manber search phase

Figure 2 (b) shows the step-by-step search stage over the following hypothetical packet "RTDTMXMKDDTS" with a matching window of three characters. In step one, WM examines the first search window: "RTD". The two-character suffix for the search windows "TD" is hashed to find the index to access the shift table. The shift value for {TD} is two; WM then shifts the search window by two to become "DTM". In step two, the hash value for the block "TM" is zero, which means a probable match. Therefore, WM searches the hash table with the same index from hashing "TM". No match is found and that was a false alarm. The default shift of two is applied which makes the next search window "MXM". In step three the shift for "XM" is two making the next search window "MKD". In step four, the shift value is zero for "KD", the hash table is searched and two matches are found {XMKD} and {MKD}. A shift of two is applied and the shift for "DT" is one. Step six, ends the search with the window reaching the end of packet.

WM search time does not surge significantly as the number and size of signatures increase. On the contrary, the average case performance beats all competing algorithms for longer signatures. However, while the overhead introduced by preprocessing scales linearly with size and numbers of signatures, it is still negligible compared to the search time.

Table 1.
WM-Manber Shift Table

BC	MD	KD	MK	TM	DT	Others
Shift	0	0	1	0	1	2

2.4. Theory of Bloom filters

Bloom filters rely on a long binary vector where a set of patterns can be programmed and reprogrammed efficiently. The filter runs a few hash functions on a set of patterns and sets the corresponding bits to the resulting hash values. This vector represents membership information on the programmed patterns and consumes a lot less space as opposed to the original dataset. The Bloom vector can easily be probed to verify membership. Simply run the same hash functions on the new pattern and check the corresponding bits. If they are not set then the Bloom filters provides a 100% assurance that the pattern is not a member of the original patterns dataset programmed into the vector. However, if the bits are set this means that there is a chance the pattern is a member of the original set (Bloom, 1970). That is, false negatives are zero, which is exactly what we need to verify a packet is clean without performing expensive hash table search. On the other hand, Bloom filters false positives rate, *f*, is given by Eq. (1).

$$f = \left(1 - e^{-\frac{nk}{s}}\right)^{k} \qquad (1)$$

Where, *n* is the number of strings programmed into the Bloom filter, *s* represents the vector size and *k* is the hash functions number. The false positives rate can be reduced by increasing the values of *s* and *k* to be appropriate for the strings number *n*. In addition, the value of *s* has to be larger than the given size of the string set, *n*.

It is possible to have multiple strings result in setting overlapping bits. Therefore, deleting a string would be an issue, because it resets the corresponding bits, which might happen to be set by another string. Counting Bloom filters (Fan, Cao, Almeida, & Broder, 2000) maintain a counter for each bit in the bitmap corresponding to the number of patterns that cumulatively set that bit. Consequently, when a new pattern is inserted or an old pattern is deleted the counter corresponding to its hash values is incremented or decremented. When the counter reaches zero, the bit is cleared.

3. METHODOLOGY

We propose Exhaust: EXclude HAsh table Unnecessary Search Time. We use the counting Bloom filters 100% exclusion property to eliminate unnecessary hash table searches. Remember that most of network traffic is benign and naturally it does not contain any malicious signatures. Therefore, if we program the Bloom filter with substrings, of size *B*, from the pattern prefixes from the hash table entries, then we can query the filter before we search the hash table. Querying the Bloom filter is a lot faster than searching the table. This means we can save the time to search the hash table for all clean traffic and we incur the cost of running two hash functions. Remember that a zero shift value from the shift table does not necessarily mean a definite match. On the contrary, quite often zero shifts are false alarms caused by the small WM block size of two or three. This small block size makes it more coincidental that the search window and the signature end up with the same suffix. Those false alarms can be handled faster if a Bloom filter is used to exclude those blocks that are not in the hash table, cutting the time to perform unnecessary searches for the large hash table.

Therefore, the Bloom filter provides a more accurate mechanism to determine probable matches and help skip the majority of zero shifts caused by benign traffic. We will prove later that this significantly improves the WM algorithm's performance while adding a very small memory overhead for the Bloom filter and a negligible preprocessing time.

Algorithm 1 presents Exhaust preprocessing pseudo code, which is similar to WM algorithm except for the additional Bloom filter programming steps. First, the algorithm starts by determining the minimum pattern length, *m*. Then Exhaust populates the shift table with the default shift value, of *m* – *B* +

1. Both the shift and hash tables are accessed by the same hash function index calculated on the character block. Next, it computes the shift values for all block substrings (x) of size B used to fill the shift table. If the shift value is zero, the corresponding entries in the hash and prefix tables are entered. Additionally, Exhaust programs the last B characters of pattern into the Bloom filter. The programming is simply running the selected hash functions on substring of size B and setting the corresponding bits in the Bloom vector.

Algorithm 2 present Exhaust's search stage, where a sliding a window of size (w) is passed over the packet. For each sliding window the index (i) for shift table is calculated by running a hash function on the suffix of B characters. If the *shift[i]* value is not zero then slide the window by the shift amount.

On the other hand, if the *shift [i]* value is zero then we must search the hash and prefix tables to verify and find the match. The Bloom filter reduces the search time, by computing two hash functions on the B character suffix and examining the corresponding bits in the Bloom vector. If the Bloom vector membership is negative, then we skip the hash table search and move to the next sliding window. If the Bloom filter membership is positive then we must search the hash and prefix tables to verify the match.

The Bloom filters do not have false negatives, which make them perfect to exclude strings from the hash table. However, they have false positives, which need to be reduced to maximize the number of times Exhaust skips the hash table. Therefore, we use two distinct and pairwise independent hash functions: SDBM and SAX. SDBM (Partow, 2015) hash is an algorithm used in the open source SDBM project. It has a good distribution for different datasets and when there is a high variance in the dataset members. For a character *c,* the hash value is calculated as shown by Eq. (2). SAX, on the other hand, is simple hash function proposed by Ramakrishna and Zobel (Ramakrishna, & Zobel, 1997). It is very fast because of the use of the common operations of shift, ADD and XOR as shown by Eq. (3).

$$HAR = c + (hash \ll 6) + (hash \ll 16) - hash \qquad (2)$$

$$hash = c + (hash \ll 5) + (hash \gg 2) \qquad (3)$$

***Algorithm 1** Exhaust Initialization*

```
procedure Initialize
for each pattern (P) in signatures set
if B < len(P) < m
m ← len(P)
end for
fill SHIFT [i] ← m – B + 1
    for every substring (x) of size B
        for each pattern (P)
      if x ϵ any P with last occurrence of q
SHIFT[i] ← m – q
      if SHIFT[i] = 0
fill(HASH)
fill(PREFIX)
    Bloom vector ← hashFcns(x)
     end for
end for
end procedure
procedure Initialize
for each pattern (P) in signatures set
```

***Algorithm 2** Exhaust Search*

```
procedure Search
for each sliding w until the end of packet
if HASH(hashFcns(last block of w)) != 0
      shift w by HASH(hashFcns(last block of w))
     else if w !ϵ Bloom vector
       shift w by 1
     else
       search HASH and PREFIX tables for exact match
end for
end procedure
```

4. RESULTS AND ANALYSIS

We evaluate Exhaust's performance through simulations using actual Snort rules and extremely malicious traffic traces representing worst-case scenario. Subsection 1 presents the details of the testing process and environment. Subsection 2 lays out the metrics to be measured. Subsection 3 explains how Snort attack signatures are extracted and cleansed, while Section 4 analyzes the traffic traces. Subsection 5 measures the number of times the hash table search is skipped and compares the Exhaust runtime to WM. Subsection 6 measures the overhead in terms of preprocessing time and memory usage. Subsection 7 suggests solutions to reduce the Bloom filters false positives to further improve Exhaust's performance. Finally, Subsection 8 analyzes the algorithm complexity and provides formal proof.

4.1. Test methodology and environment

We perform the experiments on a PC workstation with Intel Core 2 duo processor, running at 1.83 GHz, with a L1 cache of 32 KB, L2 cache of 2 MB, and 1 GB of main memory. We use Microsoft Visual Studio 2008 running on 32-bit Microsoft Windows 7 Professional.

To evaluate the algorithm's performance, we use actual network traffic traces and Snort rules. The signatures and packets are stored and read offline from files. Each experiment is repeated five times and the average is reported. Certain experiments require varying the number of signatures or characters. To be able to achieve that, signatures from different Snort rules classes are combined together to form eight sets of patterns. The first set contains 500 patterns from Specific-Threats class. The second set includes 1000 signatures composed of the previous 500 in addition to another 500 from Backdoor class and other classes. We incrementally pile signatures to end up with eight sets containing signatures ranging between 500 and 4000.

4.2. Evaluation metrics

The best metrics to evaluate the performance enhancement is the run time and speedup over WM algorithm. We exert every effort to accurately measure time by averaging five readings. However, since time measurements are not bullet proof we believe that counting the number of times we skip the hash table is a better metric. Therefore, we define the HAC and HSC metrics to measure the number of times Exhaust skips the hash table search. Where, HAC is the hash table access count and HSC is the hash table skips count. An access means that the Bloom filter gives a probable match, that is, it fails to avoid hash table search. A skip happens when the Bloom filter successfully skips the hash table search. Naturally, the higher the HSC the better because of the savings from skipping the hash table search as opposed to just computing two hash functions.

In addition, to better understand the performance improvements we calculate the hash table access ratio (HAR), and the hash table skip ratio (HSR). The normalized ratios are calculated according to Eq. (4) and Eq. (5). Moreover, to measure the Bloom filter overhead we report the added preprocessing time and memory. Finally, we analyze the false positives resulting from adding the Bloom filter.

$$HAR = \frac{HAC}{HAC+HSC} \qquad (4)$$

$$HSR = \frac{HSR}{HAC+HSC} \qquad (5)$$

4.3. Signatures extraction

We develop a script to extract the values of content keywords from Snort 2.8.4.1 rules database released in July 2009 (Snort rules, n.d.). We elected to use this version because it contains more attack signatures (9,945 rules) as opposed to the 2017 Snort v2.9 community rules. The latter includes only 3518 rules, because of the cleansing performed after Cisco's Talos participated in authoring Snort rules. We believe release 2.8.4.1 serves as a worst-case test dataset for Exhaust.

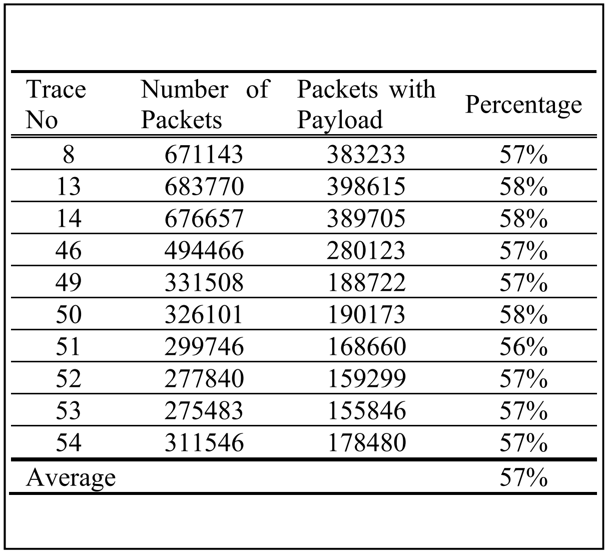
We only extract signatures from *content* and *uricontent* keywords as the *pcre* keyword contains regular expressions and not an exact match. If a rule contains more than one *content* keyword, the script merges those patterns with space character as a delimiter. All signatures are subsequently converted into hexadecimal equivalent to the ASCII codes. This way Exhaust is able to handle all 256 possibilities including nonprintable characters.

4.4. Traffic analysis

We use DEFCON17 Capture the Flag (CTF) game packet traces from 2009 (DEFCON Organization, n.d.). Capture the Flag is a hacker game where teams compete to capture computers of other teams while defending their own computers. The traces from the game are collected and made available to the public. We use those traces to gauge the worst-case performance of the new algorithm.

Our analysis shows that 51.62% of all packets in the 78 CTF traces have payload. Of those traces, we pick the ten that contains the highest percentage of packets with payload to represent the worst case. Table 2 shows the most malicious traces with total number of packets, number of packets with payload, and the percentage. The percentage of malicious content for the picked traces averages 57% which will result in a lot of signature matches. The numbers in Table 2 exclude fragmented packets.

Table 2. *Most Malicious Traffic*



Trace No	Number of Packets	Packets with Payload	Percentage
8	671143	383233	57%
13	683770	398615	58%
14	676657	389705	58%
46	494466	280123	57%
49	331508	188722	57%
50	326101	190173	58%
51	299746	168660	56%
52	277840	159299	57%
53	275483	155846	57%
54	311546	178480	57%
Average			57%

4.5. Speedup

First, we measure the HAC and HSC. That is the number of times the hash table is accessed and skipped. Figure 3 (a) shows the hash table access and skip counts for increasing number of signatures for trace number 8. Obviously, as the as the number of attack signatures increases, there will be more matches within the trace. Therefore, the number of hash tables accesses and skips increases. There is noticeable increase in savings as the number of number of signatures increases.

A more accurate picture is provided by Figure 3 (b), which presents the HAR and HSR for the same traffic trace. That is, the normalized hash table access and skip ratios. On average the hash table is skipped between 2.6% and 13.7% of the time with an average savings of 10.6%. The most important conclusion to draw from the figure is that the skip percentage is not correlated to the number of signatures. In other words, Exhaust performance remains fixed regardless of the number of attacks it searches for.

Next, we fix the number of signatures at 3500 and plot the HAR and HSR in Figure 4 (a). It is evident that the skip ratio is highly dependent on the traces' contents. HSR varies significantly from 0.6% for trace 46 to 39.1% for trace 52. This is completely dependent on the nature of packets and attacks within each trace. To better understand this, we focus on traces 46 and 52, with the lowest and highest savings, respectively. Figure 4 (b) shows the access and skip counts for trace 52 for varying number of characters. It can be clearly seen that the HAC and HSC counts increase as the number of characters increases, which confirms the earlier finding reported by Figure 3 (a). Furthermore, we zoom in to one packet and plot the HAC and HSC versus varying number of characters in Figure 5 (a). The skip count increases exponentially as the number of characters in the packet payload increases.

Before shifting our attention to measuring runtime and speedup, which are easier for readers to comprehend, it is worth mentioning that the preprocessing overhead time is incurred only once during Bloom vector programming. The overhead resulting from the query of Bloom filter search is too insignificant to measure. We will discuss the preprocessing overhead in the next subsection. The search time might be affected by: the processor speed, memory size and cache hierarchy as well the number of signatures used. To assess the worst-case improvements, Figure 5 (b) compares Exhaust search time to WM. We use all Snort signatures and present columns for each of the ten most malicious traces. On average for all traces, Exhaust is 33% faster than WM, with reported 5.972s runtime compared to 8.912s for WM.

Finally, Figure 6 confirms the earlier findings that Exhaust is not affected by varying number of signatures as opposed to AC where runtime increases linearly. The figure plots runtime for traces 14 and 52 for different number of signatures. The savings are consistent with earlier findings.

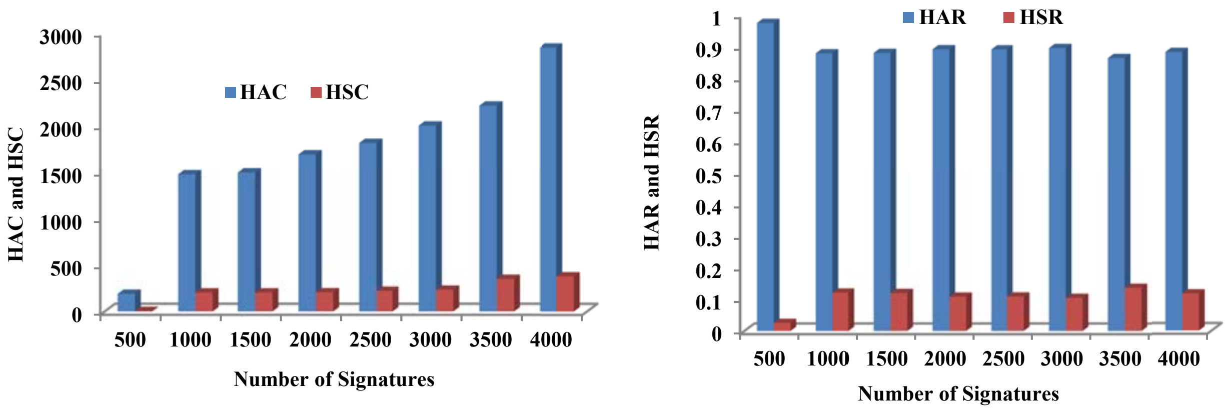


Figure 3 (a). HAC and HSC for varying number of signatures for trace 08; *(b)* HAR and HSR for varying number of signatures for trace 08

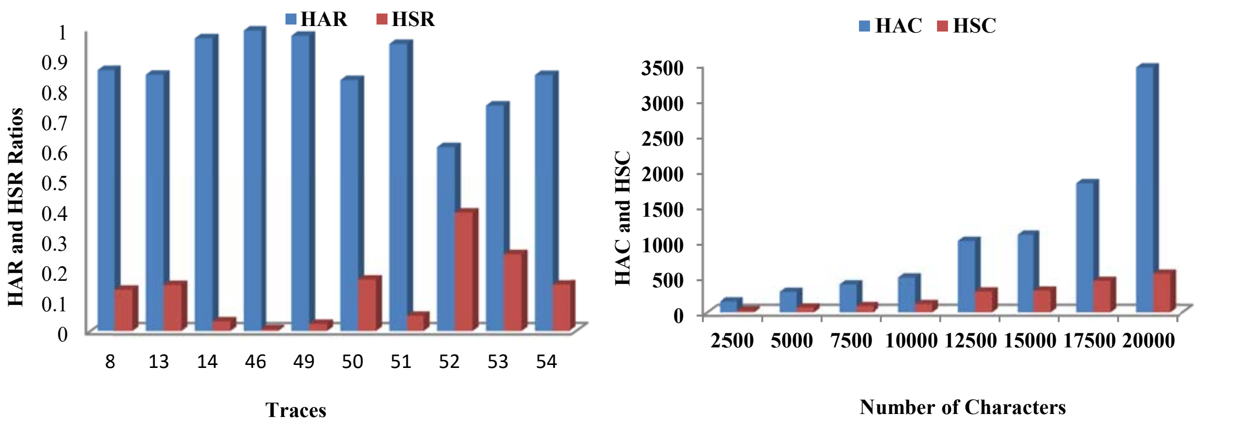


Figure 4 (a). HAR and HSR for different traces using 3500 signatures; *(b)* HAC and HSC vs. number of characters for trace 52

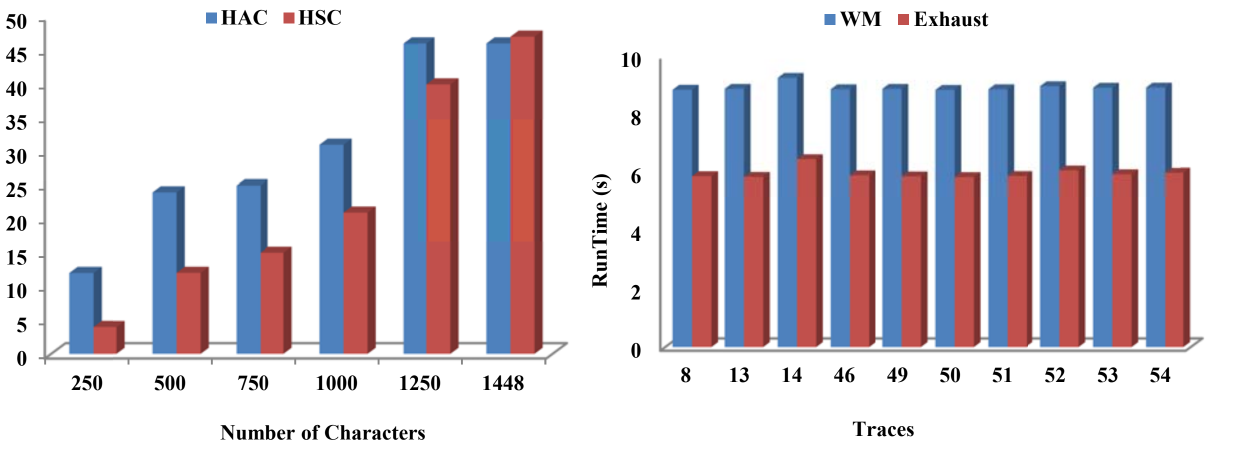


Figure 5 (a). HAC and HSC vs. number of characters for a single packet; *(b)* Exhaust runtime and WM for different traces

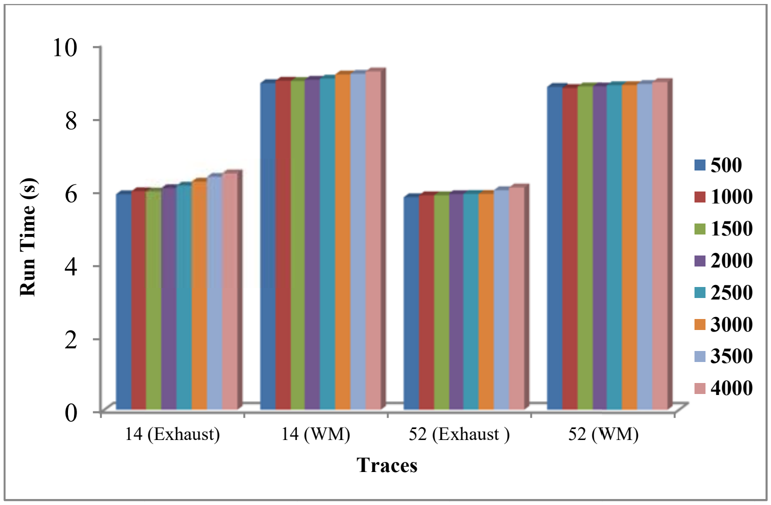


Figure 6. Runtime of Exhaust and WM for traces 14 and 52

4.6. Preprocessing overhead

The Bloom filter introduces minimal overhead in both the preprocessing and search stages. During preprocessing, there is the time to run the aforementioned hash functions and setting the bits in the Bloom vector. Figure 7 (a) compares the preprocessing time of Exhaust and WM for increasing number of signatures. The largest measured overhead is 62ms, which is equivalent to only 1.08% increase. On average the overhead is 50ms, which is equivalent to 0.8% increase. The gap between the two curves slightly increases as the number of signatures increases. We can safely conclude that Exhaust overhead time is minimally affected by increasing number of signatures.

To measure the memory usage overhead introduced by the Bloom filter we use MS Windows Task Manager to estimate the memory consumed by Exhaust. Although the numbers are bloated due to Windows Task Manager's own overhead but they still give a clear idea about the memory increase. Figure 7 (b) compares Exhaust and WM memory usage in MB for increasing number of signatures. The memory scaling is the same for Exhaust and WM with linear increase with more signatures. The worst-case memory overhead is 1,308KB equivalent to 0.33%. The average overhead is 1,285KB equivalent to 0.32%.

To further prove that linear increase in memory usage with increasing signatures is a good trend, Figure 8 (a) plots the memory usage in MB for both Exhaust and Aho-Corasick against increasing number of signatures. The state explosion in Aho-Corasick as the number of signatures increase, results in a sharper exponential memory increase as can be clearly seen. Exhaust is superior to AC when it comes to memory scaling.

4.7. Reducing false positives

The filter provides 100% certainty (i.e. true negatives), which saves execution time. True positives (TP) are not important because that simply means we had to search the hash table. It is important to stress that introducing the Bloom filter does not affect the accuracy of WM. False positives simply mean we do not save on execution time. Figure 8 (b) shows the FPs probability given by Eq. (1) for increasing number of signatures. For such a large vector the FPs probability is insignificant with a maximum of 1^{-5}.

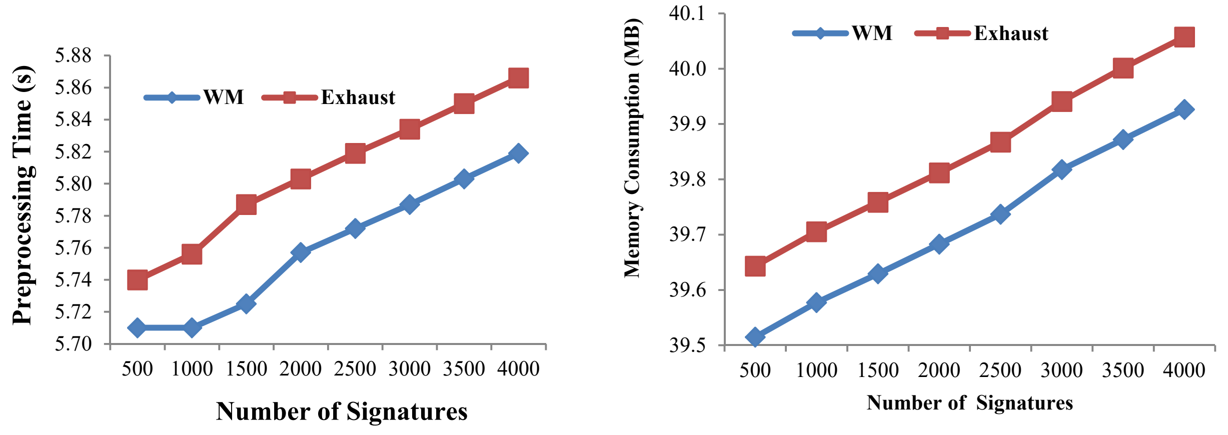


Figure 7 (a). Preprocessing time of Exhaust and WM for varying number of signatures; *(b)* Memory usage for Exhaust and WM for varying number of signatures

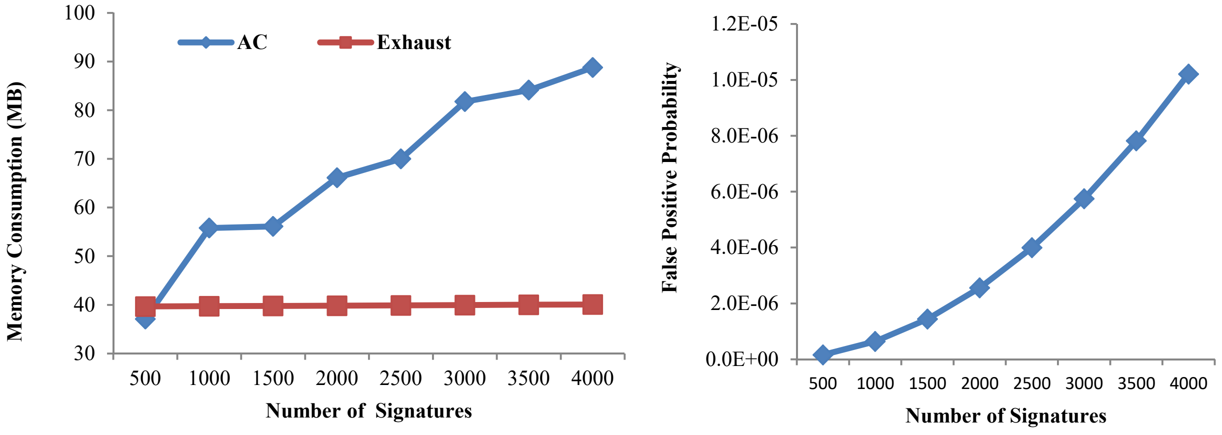


Figure 8 (a). Memory usage for Exhaust compared to AC for varying number of signatures; *(b)* False positives probability versus the number of signatures

4.8. Complexity Analysis

To illustrate the added complexity of Exhaust algorithm, we must consider both the original WM algorithm and the extra cost for adding the Bloom filter. Let N be the size of the text, P the number of patterns, m the size of one pattern, k the number of hash functions used in the Bloom filter, and assume that $M=mP$ is the total size of all patterns.

The size of a substring block B that is used to address the shift table is defined as $B=logc2M$, where c is the size of the alphabet. In the preprocessing phase the shift table construction time is $O(M)$, since that each B block of any pattern is considered once and it consumes constant time on average. On the other hand, the Bloom filter programming time is $O(k)$ because each hash function is used to address every programmed pattern. The search phase time for WM in either the case of nonzero shift value or the case of zero shift value is $O(BN/m)$, due to the suggested lemma proofed by Wu and Manber which says, "The probability that a random string of size B leads to a shift value of i, $0\leq i \leq m\text{-}B\text{+}1$, is $\leq \frac{1}{2} m$", and the benefit from prefix table extra filtering that makes the probability of false positives extremely small. The complexity incurred from the Bloom filter querying is *O(k)* (Wu, & Manber, 1994).

5. CONCLUSIONS

There exists a need to speed up intrusion detection systems. The main bottleneck is the pattern matching part of the problem. There has been a lot of research into new pattern matching algorithms and architectures for speeding up intrusion detection. Hardware architectures are fast, but they suffer from high cost and power requirements as well as configurability issues. Software based IDSs remain more popular and dominate the IDSs market, but increasing signatures requires faster pattern matching. Wu-Manber is one of the fastest multiple pattern matching algorithms used for intrusion detection but falls short of achieving the required speed.

We proposed Exhaust, a new modified Wu-Manber based pattern matching algorithm for intrusion detection systems. The new algorithm benefits from Bloom filters exclusion property to reduce the number of expensive hash table searches. The hash table can grow extremely large as the number of patterns grows.

The metrics we use to evaluate the speedup are the hash table skips ratio and execution time. We evaluate the algorithm with worst case traffic and find that Exhaust greatly improves the speed of WM at minimal cost. At best the hash table is skipped 39.1% of the time and 10.6 % on average. Exhaust reduces the running time by 33% on average for worst case traffic. The worst-case preprocessing time overhead is 1.1% and the memory overhead is 0.33%. We also show that the new algorithm has insignificant false positives probability and minor added complexity.

ACKNOWLEDGMENTS

This work was supported in part by a grant from Jordan University of Science and Technology School of Graduate Studies and in part by Zayed University Research Office, Research Incentive Fund grant # R17060.

REFERENCES

Roberts, L. (2000). Internet growth trends. *IEEE Computer Magazine Internet watch* column 2000.

Zheng, K., Cai, Z., Zhang, X., Wang, Z., Yang, B. (2015). Algorithms to speedup pattern matching for network intrusion detection systems, *Computer Communications*, 62, 47-58.

Aldwairi, M. (2006). Hardware-efficient pattern matching algorithm and architectures for fast intrusion detection. Available from NCSU Theses and Dissertations Institutional Repository (id 1840.16/3558).

Jirachan, T., & Piromsopa, K. (2015). Applying KSE-test and K-means clustering towards scalable unsupervised intrusion detection. *Proceedings of the 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, (82–87). IEEE.

Aldwairi, M., Khamayseh, Y., & Al-Masri, M. (2015). Application of artificial bee colony for intrusion detection systems. *Security and Communication Networks*, 8(16), 2730-2740. doi:10.1002/sec.588.

Roesch, M. (1999). Snort – lightweight intrusion detection for networks. *Proceedings of the 13th USENIX Systems Administration Conference (LISA '99).* Seattle, WA.

Snort. (2016). *Network Intrusion Detection & Prevention System.* Retrieved from https://www.snort.org/.

Lam, V.T., Mitzenmacher, M., & Varghese, G. (2010). Carousel: scalable logging for intrusion prevention systems. *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10)* (pp. 24–39). Berkeley, CA, USA: USENIX Association.

Antonatos, S., Anagnostakis, K. & Markatos, E. (2004). Generating realistic workloads for network intrusion detection systems. *SIGSOFT Software Engineering Notes.* 29(1), 207–215.

Aldwairi, M., & Alansari, D. (2011). Exscind: fast pattern matching for intrusion detection using exclusion and inclusion filters. *Proceedings of the Next Generation Web Services Practices (NWeSP)* (24-30). Salamnca, Spain: IEEE. doi:10.1109/NWeSP.2011.6088148

Aldwairi, M., Conte, T., & Franzon P. (2004). Configurable String Matching Hardware for Speeding up Intrusion Detection. *Proceedings of the Workshop on architectural support for security and anti-virus (WASSA), in conjunction with ASPLOS XI.* Boston, MA.

Gharaee, H., Seifi, S. & Monsefan, N. (2014). A survey of pattern matching algorithm in intrusion detection system. *Proceedings of the 7th International Symposium on Telecommunications (IST)* (946-953), Iran.

Dharmaprikar, S., Krishnamurthy, P., Sproull, T. S., & Lockwood, J. W. (2004). Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1), 52–61.

Yang, D., Xu, K. & Cui, Y. (2006). An improved Wu-Manber multiple patterns matching algorithm. *Proceedings of the 25th IEEE International Performance, Computing, and Communications Conference (IPCCC)*, (680–686).

Sunday, D. (1990). A very fast substring search algorithm. *Communications of the ACM,* 33(8), 132–142.

Xunxun, C., Binxing, F., Lei, L., & Yu, J. (2005). WM+: An optimal multi-pattern string matching algorithm based on the WM algorithm. *Proceedings of the 6th International Workshop on Advanced Parallel Processing Technologies (APPT)* (515–523). Hong Kong, China.

Liu, C., Chen, A., Wu, D., & Wu, J. (2011). A DFA with extended character-set for fast deep packet inspection. *Proceedings of the 2011 International Conference on Parallel Processing (ICPP)*(1-10).

Beale, J., Baker, A., Esler, J. & Northcutt, S. (2007). Snort: IDS and IPS toolkit. Burlington, MA: Syngress Publishing, Elsevier.

Peng, Z., Wang, Y. & Xue, J. (2014). An Improved Multi-pattern Matching Algorithm for Large-Scale Pattern Sets. *Proceedings of the Tenth International Conference on Computational Intelligence and Security (CIS)* (197–200).

Zhang, W. (2016). An Improved Wu-Manber Multiple Patterns Matching Algorithm. *Proceedings of the* 2016 IEEE International Conference on Electronic Information and Communication Technology (ICEICT 2016)(286-289).

Lee, J. K. Woo, J., & An, J. H. (2016). Improved Pattern Matching Method for Intrusion Detection Systems under DDoS Attack. *Indian Journal of Science and Technology*, 8(25), 1-4.

Aldwairi, M., & Al-Khamaiseh, K. (2015). Exhaust: Optimizing Wu-Manber Pattern Matching for Intrusion Detection using Bloom Filters. *Proceedings of the 2nd World Symposium on Web Applications and Networking (WSWAN'2015)*(1-6). Sousse, Tunisia: IEEE. doi:10.1109/WSWAN.2015.7209081

Dittrich, D. (2015, May 15). The DoS Project's trinoo distributed denial of service attack tool analysis. University of Washington. Retrieved from http://staff.washington.edu/dittrich/misc/trinoo.analysis.

Kharbutli, M., Aldwairi, M., & Mughrabi, A. (2012). Function and Data Parallelization of Wu-Manber Pattern Matching for Intrusion Detection Systems. *Network Protocols and Algorithms*, 4(3), 46–61.

Boyer, R. S. & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10), 762–772.

Aho, A. & Corasick, M. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18, 333–340.

Wu, S., & Manber, U. (1994). Fast algorithm for multi-pattern searching. Technical Report TR94-17. University of Arizona at Tuscon. Retrieved from http://webglimpse.net/pubs/TR94-17.pdf.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), 422–426.

Fan, L., Cao, P., Almeida, J. & Broder, A. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), 281–293.

Partow, A. (2015, May 15). General purpose hash function algorithms. Retrieved from http://www.partow.net/programming/hashfunctions/.

Ramakrishna, M., & Zobel, J. (1997). Performance in practice of string hashing functions. *Proceedings of the 5th*

International Conference on Database Systems for Advanced Applications (215—223).

Snort rules. (n.d.). Retrieved, May 15, 2015, from Snort website, http://www.snort.org/.

DEFCON Organization. (n.d.). Retrieved, May 15, 2015, from DEFCON website, http://www.defcon.org.