Macrothink Institute™

Network Protocols and Algorithms
ISSN 1943-3581
2012, Vol. 4, No. 3

Function and Data Parallelization of Wu-Manber Pattern Matching for Intrusion Detection Systems

Mazen Kharbutli, Monther Aldwairi, and Abdullah Mughrabi

Faculty of Computer and Information Technology

Jordan University of Science and Technology, Irbid, Jordan

E-mail:{kharbutli,munzer,atmughrabi07}@just.edu.jo

Received: July 7, 2012 Accepted: August 10, 2012 Published: September 29, 2012

DOI: 10.5296/npa.v4i3.2069 URL: http://dx.doi.org/10.5296/ npa.v4i3.2069

Abstract

The safeguarding of networks from malicious activities and intrusions continues to be one of the most important aspects in network security. Intrusion Detection Systems (IDSs) play a fundamental role in network protection. Unfortunately, the speeds of existing IDSs are unable to keep up with the rapid increases in network speeds and attack complexities. Fortunately, parallel computing on multi-core systems can lend a helping hand mitigating this performance gap. In this paper, novel and effective parallel implementations of the Wu-Manber (WM) algorithm for signature-based detection systems are proposed, implemented, and evaluated. The proposed function and data parallel algorithms prove to be effective in terms of execution time reduction and load balancing, thus providing swift intrusion detection at increased network bandwidths. The algorithms achieve an optimal load balance and an average speedup of 2x for four cores.

Keywords: Intrusion detection, pattern matching, parallel programming, Snort, Wu-Manber.

1. Introduction

Intrusion Detection Systems (IDSs) play a significant role in protecting and maintaining the security of the network. Since the early insights of James Anderson in his seminal paper [1], IDSs have grown significantly in terms of importance and complexity. IDSs complement firewalls that generally have a limited protection scope. Firewalls are primarily used to prevent attacks by limiting inter-network accesses, and are limited to checking packet headers only. Consequently, they cannot detect intra-network attacks or attacks with valid packet headers and corrupt payloads. IDSs, on the other hand, can detect inter- and intra-network attacks by examining all network communications. In addition, IDSs inspect packet headers and payloads which significantly improve their detection capabilities. This comes at the expense of more intensive inspection requirements.

Two significant factors drive the development of new IDSs. The first factor is the rapid increase in the complexity and speed of networks protected by IDSs, placing pressure on IDS designers to develop fast and efficient algorithms. The second factor is the fast growth of network-based attacks in terms of quantity, variety, and complexity. This requires IDSs to comprehensively and efficiently check for an increasing number of signatures at increasing line speeds. Unfortunately, IDSs are infamous nowadays for their lacking capabilities in switched, encrypted, and high-speed networks. One example is the popular Snort open-source IDS [2], which was demonstrated to have a lagging performance at network speeds that were greater than 100 Mbps [3]. These performance issues will continue to steer the direction of IDS research in the upcoming years.

IDSs can be implemented in both hardware and software. While hardware implementations are generally faster, they suffer from a couple of shortcomings that limit their usability. First, since they require custom hardware, they are more expensive to implement, maintain, and modify. Second, since hardware modification is difficult, they are less resilient to algorithm improvement. Software implementations, on the other hand, can run on existing hardware (CPUs) and are more resilient.

Fortunately, multiprocessing has become the norm nowadays and can lend a helping hand to the speed limitation of software based IDSs. Nowadays, most general-purpose CPUs are shipped with multiple cores allowing the parallel execution of multiple threads. Multiprocessors overcome the performance limitations of uniprocessors imposed by instruction-level parallelism (ILP) and clock rate-power limitations. By developing parallel IDS implementations that run on multiple existing processors, IDSs' performance, efficiency, and comprehensiveness can be greatly improved.

Pattern matching lies at the center of IDSs, comprising about 70% of the workload in Snort [4]. Pattern matching algorithms are computation-intensive making them excellent candidates for parallel implementation. In this paper, we propose, implement, and evaluate novel parallel implementations of Wu-Manber [5], a pattern matching algorithm that is widely-used in IDSs. Unlike previously proposed parallel implementations [2][6-12] which focused on data parallelism only, our proposed implementation is based on function level or task parallelism. Implementations that are based on data parallelism require the availability of

the full checked trace beforehand and may suffer from load-imbalance limiting their performance improvement potential.

The proposed function parallel algorithm utilizes the concept of multiple sliding scanning windows running concurrently. It does not require prior knowledge of the trace size and can provide better load balance compared to data parallelism techniques. The proposed algorithms are evaluated on a multicore machine running Linux and using a set of standard traces. Compared to a serial approach, the algorithms provide near-optimal load balancing and scalable speedups proportional to the number of threads.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 explains IDSs, pattern matching, and parallel computing approaches, thus laying the foundations for our work. Section 4 describes the different proposed parallel implementations. Section 5 provides an experimental evaluation of the proposed techniques. Finally, Section 6 concludes the paper.

2. Related Work

There have been several proposals for parallel IDS pattern matching algorithms in the literature recently. Unfortunately, most proposed algorithms focus on data parallelism only. Data parallelism can lead to load-imbalance problems between the threads limiting performance improvement opportunities.

Wheeler [6] used two approaches for parallelization which utilized data parallelism over a number of pattern matching algorithms. The first approach was applied on a rule dataset by separating signatures into two groups based on their size, with one group containing one-byte rules and the other containing the larger than one-byte rules. This approach mitigated the fact that the Wu-Manber (WM) algorithm executed poorly on rules that were one-byte in length. By diverting the one-byte rule detection mandate to another algorithm and letting WM check for rules that are larger than one byte in size, Wheeler was able to grasp the performance offered by both algorithms on average cases, consequently, achieving a dual algorithm scheme. The second approach was based on the concept of spreading hash groups; this was done by distributing non-empty hash table entries evenly across all processing elements. Kopek [7] also followed in Wheeler's footsteps and provided some improvements in terms of performance. However, both approaches do not provide a parallel implementation of the Wu-Manber algorithm itself, but are based on a dual-algorithm approach. Furthermore, their solutions are dependent on the network traffic size which could lead to load-imbalance and performance improvement limitations. This could happen if checked pattern sizes were not uniformly distributed, thus putting pressure on the processing element implementing the algorithm corresponding to the dominant size.

Kouzinopoulos and Margaritis [9] presented parallel implementations for Naïve, Knuth-Morris-Pratt [10], Boyer-Moore-Horspool [11] and the Quick-Search [12] online exact string matching algorithms using the CUDA toolkit on GPU systems. It was shown that the parallel implementation of the algorithms was up to 24 times faster than the serial

implementation, particularly when greater text and smaller pattern sizes were used. The study did not include the Wu-Manber algorithm and was limited to data parallelism. Our proposed algorithm is implemented and evaluated on a general-purpose multi-core CPU.

Zhang et al. [8] probed the WM algorithm in more depth. The main goal of their study was overcoming the limitations of WM when dealing with short patterns resulting in the proposal of a High Concurrence WM algorithm (HCWM). This was done through splitting all patterns into different sets according to their length, and then processing those sets respectively to that separation. The result demeaned the effect on performance caused by short patterns. In addition, when independent data structures were used for different pattern sets, the high concurrence was gained through which it significantly enhanced the matching speed of HCWM. Moreover, experimental results showed that HCWM presents higher performance than WM. Although this approach improved on Wu-Manber, it sacrificed the simplicity of the Wu-Manber algorithm and did not provide a parallel implementation of the algorithm.

More recently, Aldwairi and Ekailan proposed a hybrid Aho-Corasick and Wu-Manber pattern matching algorithm to exploit the fact that Wu-Manber is more efficient for longer patterns and Aho-Corasick has a superior performance for short patterns [2]. They proposed a new pattern partitioning algorithm to solve the load balancing issues. The best case runtime reduction for four threads was 33% over serial Wu-Manber implementation.

3. Background

This section lays the foundations of our study by discussing the bases of Network Based Intrusion-Detection Systems (NIDSs), pattern matching algorithms, and parallelization techniques and approaches.

3.1 Network Based Intrusion Detection Systems

NIDSs are built with the objective of detecting attacks over the entire segment of the network. They are designed to run silently in the background capturing and checking all network traffic. A captured packet first goes through various pre-processing steps leading to the signature pattern matcher. The signature matching stage manifests itself as the bottleneck of NIDSs comprising most of the processing time. In order to illustrate the operation steps of IDSs, let us consider Snort [13], a popular IDS available on Sourcefire. Snort provides real-time traffic analysis and is available as an open source software package, making it an attractive choice for researchers as well as for the industry. Snort primarily uses the misuse (signature-based) detection model. The packet handling work-flow in Snort [14] starts with capturing the packet when it first enters the network, which is usually done via the system's *libpcap* facility or through a user trace file containing network activities that are saved for processing at later times.

The captured packet then gets decoded and its content gets normalized. This is achieved through the transformation of data to raw binary format. In addition, it handles the

examination of cumbersome attacks that rule matching cannot detect such as packet reassembly. The matching detection engine handles the discovery of attacks through matching the pre-processed packets to a dataset of rules or signatures combined in a database which is generated beforehand. When a match occurs between a packet and a signature, the action linked to that rule is executed with possible actions including alerting the admin or dropping the packet altogether. The detection engine harbors the heart and core of any IDS which is the pattern-matching algorithm. Pattern matching algorithms usually are categorized to single- and multiple-pattern matching. Snort uses the Boyer-Moore (BM), Wu-Manber (WM), and Aho-Corasik (AC) pattern matching algorithms [15]. BM is considered a single pattern matching algorithm, while both AC and WM algorithms are considered multi-pattern matching algorithms. A single pattern algorithm searches for the occurrence of one pattern in a searchable string at a time, whereas a multi-pattern matching algorithm searches for multiple patterns at the same time. This is made possible by the usage of hash tables (WM) or the usage of trees that are usually referred to as "trie" (AC). In both cases, the signatures consist of a finite set of patterns given beforehand.

3.2 The Boyer-Moore (BM) Algorithm

BM is regarded as one of the most practical and widely-used methods for exact single pattern matching. It is especially considered a good choice for large sets of alphabets and long patterns such as bio-applications. The Wu-Manber algorithm itself builds upon BM as will be explained later. BM relies on the concept of a sliding window for pattern matching. To reduce the number of required comparisons in a text of length T, BM relies on shifts with steps longer than one. This is done by shifting the pattern P of length n to the right in longer steps, typically less than m characters. In more details, the BM algorithm constructs upon two heuristics which reduce the shifting occurrences to obtain a match. The first is called the bad character heuristic which avoids bad or non-useful comparisons. In other words, if a mismatch occurs in the pattern that we are searching for, we check if that character appears in the searching pattern or not, if it does not appear, then we shift the searching pattern so that it is one position past the mismatching character. However, if the character that is mismatched actually occurs in the searching pattern, we shift so that the rightmost occurrence of that character in the searching pattern is aligned with the mismatched character. The second rule is called the good suffix heuristic. It considers the matching of patterns and signatures that are already successfully matched, and this can occur when a mismatch is triggered while in the middle of the search pattern, meaning that there is a suffix that matches with the search pattern so the following shift will be to the next occurrence of the suffix in the pattern. It should be noted that an improved version of BM appeared later by Horspool [11], which enhanced upon it by using only the bad character heuristic property.

To elaborate more let us consider the example in Fig.1, which represents a signature to be checked using Snort rule-set for a tcp overflow attempt. At step (a), we start the iterative process of matching the pattern "STOR" from right to left while traversing or shifting the searched string from left to right. As explained before the characters R and U create a mismatch and the character U also does not occur in the pattern "STOR", therefore, it is safe to shift the pattern one position past the U character in "STOU". At step (b), after three

matches, a mismatch occurs between character S and the space character, so a shift to the right one position past "TOR" is safe. At step (c), we can see that there is a mismatch between the "R" character and the "O", but also notice that "O" exists in the pattern.

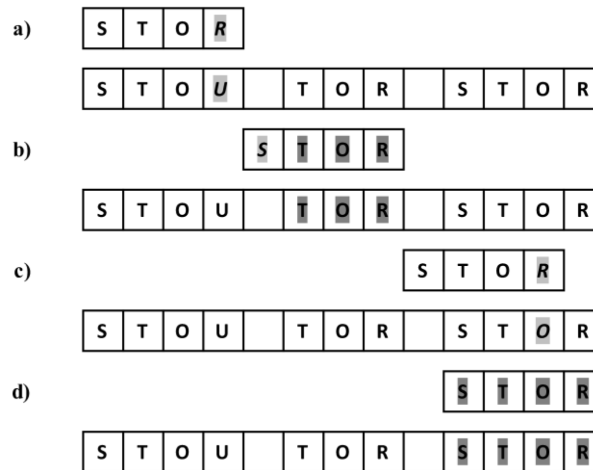

Figure 1. The Boyer-Moore algorithm illustrated while checking a signature in Snort rule-set for a tcp overflow attempt.

Therefore, we shift the pattern by one position to align it with the "O" character in the searched string. Finally at step (d) we find a match. Notice that the string length is 13 characters whereas the number of comparison attempts is only 4. Had we searched using a conventional shift by one method, it would have taken 10 steps instead. This illustrates the strength of using such an algorithm as it reduces the searching steps drastically.

3.3 The Wu-Manber (WM) Algorithm

Wu-Manber builds on the Boyer-Moore algorithm by using block skips "bad character heuristics" from BM in combination with the usage of hash tables, consequently providing the multi-pattern searching capability. In fact, Snort uses a modified simpler version of WM (MWM) [16] as the default pattern matcher. This is attributed to its better performance in average cases [17] compared to its counterpart pattern matching algorithms such as Aho-Corasick and Boyer-Moore.

Before the MWM algorithm can be used, its hash and shift tables must be generated using a finite set of patterns provided beforehand. This is a one-time step that needs to be redone only when there are new sets of patterns. It should be pointed out here that the original WM algorithm adds a prefix table used to differentiate between patterns when their suffixes are the same but their prefixes are different. The first step in generating the tables is to find the minimum pattern length *m* from all signatures. This length *m* represents the number of characters that will be taken from the first letters of all signatures to form the shift table as will be explained later.

The shift table is built based on two variables: The first, *B*, is usually predetermined to a value of *2* or *3*, albeit there are some other suggested methods for obtaining its value in accordance to the minimum length of the patterns and the number of rules provided.

Nonetheless, our study and implementation of WM use a value of $B = 3$ due to the fact that we had a large set of rules to implement. The second variable, m, represents the minimal length from among all patterns. As a result, each pattern is mapped to the shift table by taking all subset strings of size B from the first m characters of each signature, where each subset string is considered as a hash key to the shift table. Those shift keys are assigned a shift value according to their location q in the pattern using the equation $shift[key] = m - q$. Consequently, this accomplishes the construction of the bad character shift table. The building of the hash table starts with mapping the last B sub-strings of all m sub-patterns which can be obtained by getting the sub-strings hash keys with a zero value and putting them in the hash table. As a result, each hash key in the hash table is a pointer to a list of patterns that share the rightmost B sub-string in them.

During the scanning process, a sliding window of size B scans through the searched string each time obtaining a sub-string of that size and getting its shift value from the shift table and shifting accordingly. Nonetheless, if the sub-string key does not exist in the shift table, a maximum safe skip/shift of length $(m - B + 1)$ is used. On the other hand, a shift value equal to zero requires access to the hash table and the traversal/search of all patterns that are linked to the key until a match is found or the list ends with no match. This iterative process is repeated until the whole string is completed.

To illustrate the operations described earlier, let us reflect on the example in Fig.2. Consider the patterns STOU, STOR, and CMD, which represent signatures to be checked in Snort rule-set. The first step is to set the values of B and m. We will assume a value $B = 2$ and then find the shortest pattern "CMD" which sets the length $m = 3$. Next, and taking the pattern "STOU" as an example, we start by extracting the first three characters in accordance to the value of m thus obtaining "STO", then with compliance to the value of B the hash keys ("ST","TO") for the shift table can be acquired. After that, the shift value can be found by attaining the value of q and using the shift equation $shift[key] = m - q$. Hence for our sample $q = 2$ for ST and therefore $shift[ST] = 1$. Likewise, $q = 3$ for TO and thus $shift[TO] = 0$. Moreover, hash keys that have a shifting value equal to zero are correspondingly assigned to the hash table. This iterative process is applied for all the patterns.

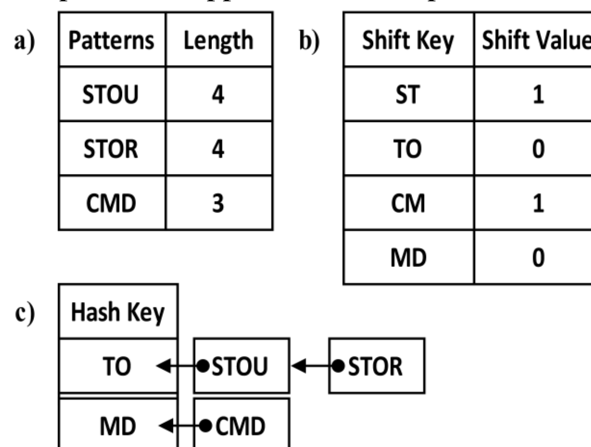

Figure 2. Wu-Manber algorithm, (a) patterns and their length, (b) shift table, used for obtaining shift values, (c) hash table, used for comparing patterns in the searched text.

3.4 Parallel Programming Techniques

Parallel computing is a programming methodology where the different tasks that comprise a large problem are divided on processes/threads executing concurrently with the main goal of lowering the execution time. There are two main approaches to parallel computing.

3.4.1 Data Parallelism (SIMD)

Data parallelism is the simplest form of parallel computing. It divides the data being operated on among the different threads. The different threads usually perform the same set of operations but on different parts of the dataset. This is why it is also referred to as loop-level parallelism or single-instruction-multiple-data (SIMD) parallelism. As an example, consider the task of adding two arrays each of size N. Assuming P threads running in parallel, the first thread would add the first N/P elements, the second thread would add the second N/P elements, and so on. This approach is simple to implement but suffers from load-balancing and performance problems when there are dependencies between the subtasks. That happens when a task is dependent on the outcome of another task. Because of its simplicity, this form of parallelism can be implemented easily, sometimes automatically with the help of the compiler.

3.4.2 Functional (Task) Parallelism

This method of parallelism usually utilizes the division of the system into functional processing blocks. The basic principle can be simply explained as a pipeline: the output of one process serves as an input to another. Despite the simplicity behind the concept, intricacies can arise due to the fact that partitioning an algorithm to a set of different tasks requires a deep understanding of the problem. Furthermore, some algorithms cannot be partitioned to distinct tasks as a result of their nature. Another herald is achieving load balancing especially when one task could be bottlenecked more than others. Moreover, there are restrictions in some cases on the number of cores that could be used due to the limitation on the number of distributed tasks that could be achieved through the usage of this model. Despite all that, the performance boost that can be obtained using this approach is often more rewarding than data parallelism.

Finally, it should be noted that a hybrid parallelism approach utilizing both data and task parallelism is sometimes possible.

3.5 OpenMP

OpenMP [18] is an application programming interface (API) that provides the means to write parallel programs facilitating the shared-memory parallel programming method for symmetric multiprocessors (SMPs). OpenMP offers a set of compiler directives and runtime callable routines that deliver a separate extension to the C/C++ libraries.

4. Parallel Implementations for Wu-Manber

In this section, we propose three novel parallel programming approaches to the Wu-Manber algorithm. First, we propose the shared position algorithm which is the main contribution of this paper. It utilizes multiple racing sliding windows for optimal load balancing. Next, we combine the shared position algorithm with data parallelism to maximize speedups.

4.1 The Shared Position Algorithm

As described earlier, the scanning process in the WM algorithm can be abstracted as a sliding "scanning window" of size *B* that scans through the searched string, each time obtaining a sub-string of that size. Each substring is then used as a hash key to the shift table from which the appropriate shift value can be found. In the case of a zero shift amount, the hash table is then traversed in search of a string match.

From the illustration provided above we can expand the concept of the "scanning window" by implementing several "scanning windows" on several processing units or threads. The fundamental idea behind this approach is that all scanning windows share the highest current position where one of the scanning windows is currently at in the searched string. Furthermore, any scanning window that obtains a shift value that moves it beyond the highest current position variable is granted a shift to that location while updating the value of the highest current position. Nevertheless, if the shift value obtained from the shift table puts the scanning window in a position to the left of the highest current position variable, then the window is shifted to one position past the highest current position. This can be explained as a result of the fact that the skipped section is being checked by other scanning windows and is safe to traverse past it. We should note that each scanning window has a private position variable so that its present location would not be affected in case it was active while a change in the highest position variable occurred.

This approach has several advantages. First and most importantly, it can be applied on traces with no prior knowledge of their size or length, as it only depends on the relative starting position in the trace. In the case of a data parallelism approach, it would be difficult to divide the dataset uniformly on the processing elements if the dataset size is not known beforehand. Second, while data parallelism approaches would suffer from load imbalance if the locations of the positive matching substrings are clustered in a small part of the string, the proposed approach has all processing elements working on the same string section at any given time, thus providing better load balance between the threads.

To better understand the idea, let us consider the example in Fig. 3. This builds upon the previous example provided for the Wu-Manber algorithm in Fig. 2. Two scanning windows are provided; both are running simultaneously on independent threads. Let $B=2$ and the safest shift is 2 in conformity to our previous calculations in Fig. 2. Starting at step (a) "window one" appears as a solid square and starts at position ($i = 3$), while "window two" appears as a dashed square and starts at position ($i = 4$). Notice that at "window one," the value obtained

from the shift table is zero; hence a traverse and comparison via the hash table must be performed. Meanwhile, "window two" proceeds with a shift of two blocks because “OU” does not exist in the shift table as illustrated in step (b). In step (c), “window two” safely skips two positions because “ T” does not exist in the shift table. “Window two” now points to “OR” and updates the current highest location variable. As "window one" finishes traversing the hash table and finds no match, it shifts to one position past the highest current position. Window one now points to “R ” and updates the highest current position. Finally, in step (d) "window one" finishes this time before window two and shifts to “ST”, while the suggested shift for “window two” is 2 because “OR” is not found in the hash table. Because of the shift of 2, the window is positioned before the highest current position variable. Therefore, it shifts past the highest current position by one reaching “TO” and it updates the highest current position. “TO” has a shift of zero which invokes the hash table for a comparison. The upswing of different flows for the algorithm is caused by race conditions and timing variations that have no predictable behavior and depend on the state of the system at that instance. Therefore, the “scanning windows” would appear as if jumping over each other. It should also be noted that the usage of the shared position technique can produce duplicate intrusion detections; as a result of accessing the same hash key at the same time due to race conditions. However, this does not happen often and the benefits the algorithm offers in regards to performance outweigh such a predicament.

From this example we can see that all the detections were made with a minimum number of shifts. For instance, windows one and two only required four shifts to catch the intrusion’s signature, whereas a serial WM implementation requires seven shifts to perform the aforementioned task, about 42% increase in the number of shifts. Those observations if applied to larger sets of traces would lead to many benefits in terms of swifter detections and faster scans, especially that the approach can be scaled to a larger number of processors/scanning windows thus achieving higher speedups.

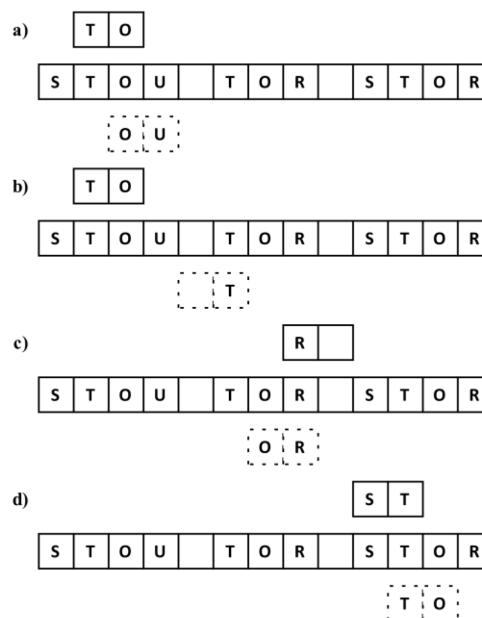

Figure 3. Shared Position approach, the solid block is window one, while the dashed block is window two.

4.2 The Trace Distribution Algorithm

This algorithm is a simple implementation of data parallelism approach. It uses the multiprocessor system to run a code that divides the dataset on several nodes, each running the same segment of processing logic on its data segment. In our case this dataset is represented by the trace file, where it is divided and distributed to equal segments in accordance to the number of processors provided by the system. Additionally, each segment is independently accessed by a serial implementation of the WM algorithm, thus performing detections in n segments of the trace file concurrently. This Algorithm is fairly easy, has no race conditions as each segment is processed separately, and provides good performance. Nonetheless, it has no regards to the signature bursts that occur in some segments. This is due to the uneven distribution of intrusions signatures in the trace files, which results in longer times to finish processing some segments because they form bottlenecks. This hugely affects the overall performance of the algorithm because the signature-congested segments increase the processing time of the critical path of the algorithm. Moreover, such an approach is only applicable if the trace file is available beforehand making this parallelization approach inapplicable to online systems checking network traffic live at line speed.

4.3 Combined Shared Position and Trace Distribution

The following technique stems from the shared position and the trace distribution algorithms; by partitioning the trace file into various segments, we can run the shared position approach on each segment instead of the serial WM algorithm. This technique provides on par performance to the distributed algorithm while overcoming the segment performance problems due to intrusion bursts, where the slowdown usually occurs.

5. Experimental Results

The aforementioned parallel algorithms of Wu-Manber are implemented using the C++ programming language and OpenMP API under GCC. Moreover, the Intel threading building blocks package [19] was used because it provides thread-safe containers (e.g. concurrent hash tables and vectors) with automatic locking mechanisms thus eradicating some of the problems that arise through using shared data between processes such as deadlocks and race conditions. Experiments are performed on a quad-core Intel i5 M480 CPU with 4 GB of RAM running Linux.

Experiments are run on a set of trace files collected using Wireshark network protocol analyzer [20]. A thorough description and analysis of the traces is presented by Aldwairi and Alansari [21]. The number of intrusions and the distribution of the intrusions across the traces vary from trace to trace. The set of traces are carefully selected to test the best and worst case performance of the algorithms. Table 1 lists the 8 traces used in the evaluation along with related statistics. The ugly traces “1” and “22” are chosen to test the worst case performance. They suffer from pathological performance due to their large sizes, the high percentage of

intrusion signatures in them, and the small skips which slow down the traversing window. The bad traces "58" and "51" lie in the middle range with regards to the number of intrusion signatures and pathological performance. Finally, the good traces "av", "gd", "hm", and "lc" are normal everyday traffic traces representing audio/video streaming, good download, Hotmail and live chat. They are considered good because of the small percentage of signatures in the traces. The signatures are extracted from Snort 2.9.0.4 rule.

Table 1. Packet Traces Statistics

Bad Trace	Serial Detections	Length (char)	Size (MB)	Intrusions (%)
58	846971	96512779	91.78	0.88
51	1250575	96101726	91.36	1.30
Ugly Trace	**Serial Detections**	**Length (char)**	**Size (MB)**	**Intrusions (%)**
1	10676527	91053842	86.18	11.73
22	7598082	91425792	86.56	8.31
Good Trace	**Serial Detections**	**Length (char)**	**Size (MB)**	**Intrusions (%)**
av	863	1033606	0.98	0.08
gd	5299	7085480	6.75	0.07
hm	21111	1917710	1.83	1.10
lc	5986	359006	0.34	1.67

Fig. 4 shows the execution times for each of the eight traces when run using the Trace Distribution algorithm (TD), the Shared Position algorithm (SP), the Combined Shared Position and Trace Distribution algorithm (DSP), as well as serial implementation. The figure shows that all three parallel implementations perform very well reducing the processing time of each trace to about half that of a serial implementation.

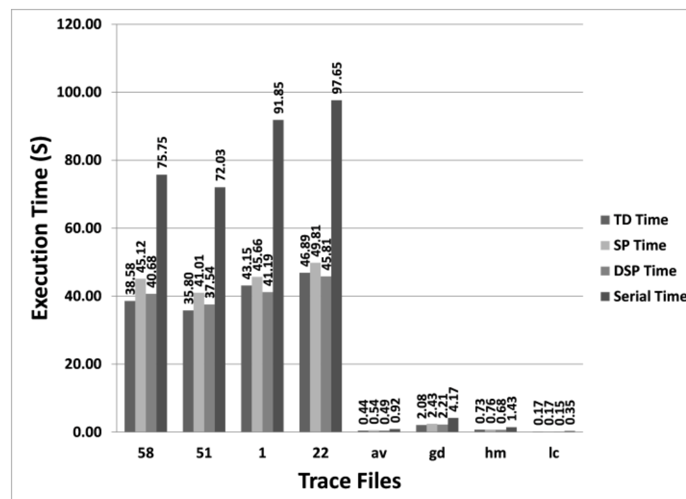

Figure 4. Execution times for the Trace Distribution method (TD), the Shared Position method (SP), the Combined Shared Position and Trace Distribution method (DSP), and a serial implementation (Serial).

To better assess the performance boost, Fig. 5 shows the speedups obtained from running each of the three parallel implementations relative to the serial implementation. Speedups average around 2x and range from 1.68x to 2.33x. It is noticeable that the speedups are the highest for the ugly traces number 1 and 22. This is expected because they contain the largest number of intrusions which slows down the serial implementation and gives the parallel algorithms more room for time saving. The normal traffic represented by the good traces of audio/video, good download and Hotmail exhibit the smallest speedups because those traces have a small number of intrusion signatures and therefore the serial algorithm is not slowed down. Live chat on the other hand has a higher than expected speedup which can be explained by the short length of the traces and evenly distributed attacks (see Fig. 6-8) allowing parallel algorithms to skip through the traces faster. Finally, all three parallel implementations perform comparably well with no clear best algorithm in terms of speedups. Although SP speedups are slightly lower than the data parallel TD, it has an important advantage by evenly balancing the load between all threads. The balanced loads will translate into swifter and faster detections at greater bandwidths.

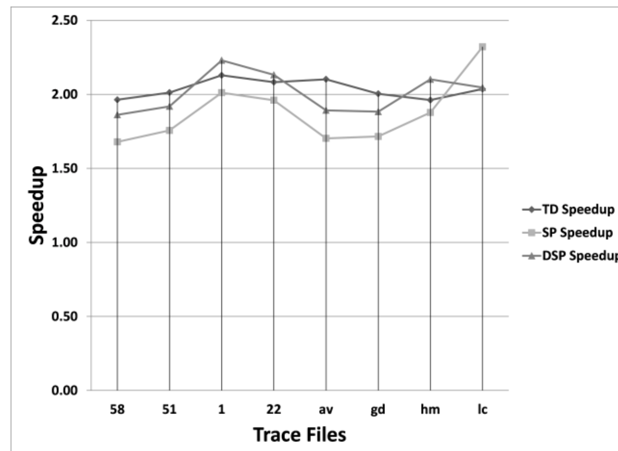

Figure 5. Speedups relative to a serial implementation for the Trace Distribution method (TD), the Shared Position method (SP), and the Combined Shared Position and Trace Distribution method (DSP).

Fig. 6, 7 and 8 show the distribution of detected intrusions over the four threads. On one hand, the distribution of the detected intrusions over all threads in the SP implementation is even, indicating a uniform load balance between the threads. The algorithm threads jump through the whole packet traces processing different and random segments resulting in optimal load balancing. On the other hand, the distribution of detected intrusions is far from even for both the TD and DSP methods indicating load imbalance. The reason is the serial nature of dividing the data trace into four distant and fixed segments resulting in uneven distribution of signatures. This is more obvious in the cases of highly clustered signatures in the good download traces. The use of a larger number of threads will worsen the imbalance. Load imbalance can lead to pathological behavior in terms of bottlenecks. This observation puts the SP algorithm as a forerunner in terms of overall performance.

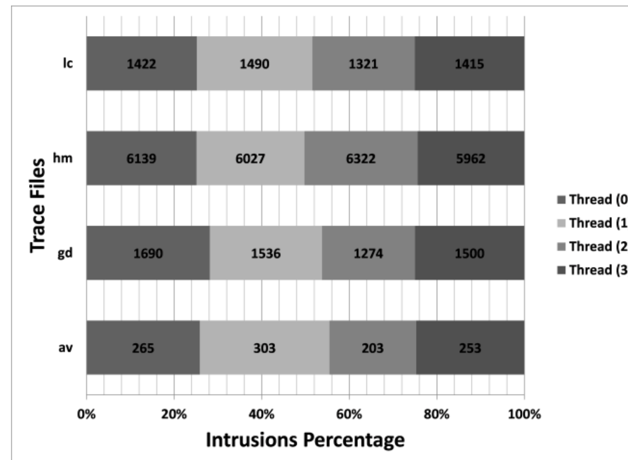

Figure 6. Intrusions detected by each of the four threads using the Shared Position (SP) method.

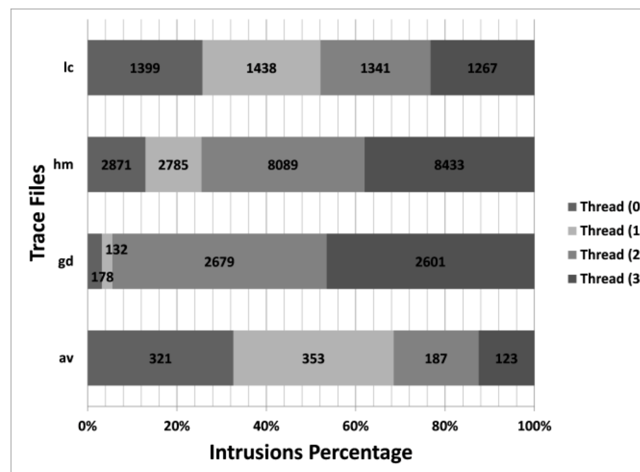

Figure 7. Intrusions detected by each of the four threads using the Combined Shared Position and Trace Distribution method (DSP).

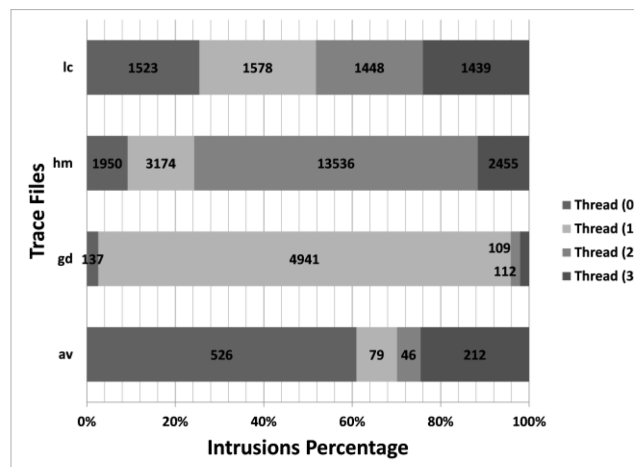

Figure 8. Intrusions detected by each of the four threads using the Trace Distribution (TD) method.

6. Conclusions

This paper introduces three novel and effective parallel implementations of the Wu-Manber pattern matching algorithm which is a core part in many intrusion detection systems. The first implementation, the Shared Position (SP) algorithm, utilizes several scanning windows running in parallel and using a shared position variable. The second implementation, the Trace Distribution (TD) algorithm, divides the trace equally among the parallel threads. The third implementation (DSP) combines the first two algorithms. All three algorithms were thoroughly tested and found to provide excellent performance improvements halving the execution times of a serial approach when using four threads. These performance improvements translate to swifter detections at greater bandwidths thus mitigating the performance gap between modern network speeds and existing IDSs described earlier.

References

[1] Anderson, J. P., "Computer Security Threat Monitoring and Surveillance". Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[2] Aldwairi, M. and Ekailan, N., "Hybrid Multithreaded Pattern Matching Algorithm for Intrusion Detections Systems," Journal of Information Assurance and Security, vol. 6, no. 6 pp. 512-521, 2011.

[3] Schaelicke, L., Slabach, T., Moore, B., and Freeland, C., "Characterizing the Performance of Network Intrusion Detection Sensors," in Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID), Lecture Notes in Computer Science, September 2003. DOI: http://dx.doi.org/10.1007/978-3-540-45248-5_9

[4] Antonatos, S., Anagnostakis, K., and Markatos, E., "Generating Realistic Workloads for Network Intrusion Detection Systems," in Proceedings of the 4th ACM Workshop on Software and Performance, 2004. DOI: http://dx.doi.org/10.1145/974044.974078

[5] Wu, S., and Manber, U., "A Fast Algorithm for Multi-pattern Searching". Technical Report TR-94-17, Department of Computer Science, University of Arizona, 1994.

[6] Wheeler, P. S., "Techniques for Improving the Performance of Signature-Based Network Intrusion Detection Systems," M.S. thesis, University of California, Davis, 2006.

[7] Kopek, C.V., "Parallel Intrusion Detection Systems for High Speed Networks Using the Divided Data Parallel Method," M.S. thesis, Wake Forest University, North Carolina, May 2007.

[8] Zhang, B., Chen, X., Pan, X., and Wu, Z., "High Concurrence Wu-Manber Multiple Patterns Matching Algorithm," in Proceedings of the 2009 International Symposium on Information Processing, Huangshan, P. R. China, August 2009.

[9] Kouzinopoulos, C. and Margaritis, K., "String Matching on a Multicore GPU Using Cuda," in the 13th Panhellenic Conference on Informatics, 2009. DOI: http://dx.doi.org/10.1109/PCI.2009.47

[10] Knuth, D., Morris J., and Pratt, V., "Fast Pattern Matching in Strings," SIAM Journal of Computing, vol. 6, no. 2, pp. 323-350, 1977. DOI: http://dx.doi.org/10.1137/0206024

[11] Horspool, R., “Practical Fast Searching in Strings,” Software Practice and Experience, vol. 10, no. 6, pp. 501-506, 1980. DOI: http://dx.doi.org/10.1002/spe.4380100608

[12] Sunday, D., “A Very Fast Substring Search Algorithm,” Communications of the ACM, vol. 33, no. 8, pp. 132-142, 1990. DOI: http://dx.doi.org/10.1145/79173.79184

[13] Roesch, M., “Snort – Lightweight Intrusion Detection for Networks,” in Proceedings of USENIX LISA, November 1999.

[14] Koziol, J., “Intrusion Detection with Snort,” 2nd Edition, May 8, 2003. Pearson Education Inc.

[15] Aho, A., and Corasick, M., “Efficient String Matching: An Aid to Bibliographic Search,” in Communications of the ACM, vol. 18, no. 6, pp. 333-340, 1975. DOI: http://dx.doi.org/10.1145/360825.360855

[16] Sourcefire, “Snort 2.0 - Detection Revisited”. October 2002. http://www.forum-intrusion.com/ archive/Snort_20_v4.pdf (last access September 2012)

[17] Tuck, N., Sherwood, T., Calder, B., and Varghese, G., “Deterministic Memory-efficient String Matching Algorithms for Intrusion Detection,” in IEEE Infocom, Hong Kong, China, March 2004. DOI: http://dx.doi.org/10.1109/INFCOM.2004.1354682

[18] Chapman, B., Jost, G., and Van der Pas, R., “Using OpenMP: Portable Shared Memory Parallel Programming,” MIT Press, 2007.

[19] Pheatt, C., “Intel Threading Building Blocks,” Journal of Computing Sciences in Colleges, vol. 23, no. 4, pp. 298, 2008.

[20] Orebaugh, A., Ramirez, G., Beale, J., and Wright, J., “Wireshark and Ethereal Network Protocol Analyzer Toolkit,” Syngress Media Inc, 2007.

[21] Aldwairi, M. and Alansari, D., “Exscind: Fast Pattern Matching for Intrusion Detection Using Exclusion and Inclusion Filters,” in Proceedings of Next Generation Web Services Practices (NWeSP), October 2011. DOI: http://dx.doi.org/10.1109/NWeSP.2011.6088148

Copyright Disclaimer

Copyright reserved by the author(s).

This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/).