國 立 清 華 大 學 通 訊 工 程 研 究 所

博 士 論 文

多型樣式匹配演算法於網路系統之研究

Multi-Pattern Matching Algorithms for Networks

學生姓名：許慈芳 (899606)

Tzu-Fang Sheu

指導教授：黃能富 教授

Prof. Nen-Fu Huang

中 華 民 國 九 十 七 年 十 一 月

ACKNOWLEDGEMENTS

It is a pleasure to thank the many people who made this dissertation possible.

My foremost thank goes to my thesis adviser Dr. Nen-fu Huang, who has advised me in various aspects of my research, and has assisted me in numerous ways.

I thank the rest of my thesis committee members: Dr. Shiuhpyng Shieh, Dr. Chi-Sung Laih, Dr. Chin-Laung Lei, Dr. Wei-Kuan Shih, Dr. Han-Chieh Chao, and Dr. Yin-Te Tsai. Their valuable feedback helped me to improve the dissertation in many ways. I gratefully thank to my master thesis adviser Dr. Shiann-Tsong Sheu who introduced me to the field of network studies.

I gratefully acknowledge the MediaTek for the award of the MediaTek Fellowship, which has supported me during my three years of research; thank the National Science Council of the Republic of China for the award of field research in the Northwestern University. I would like to express my sincere thanks to the host professor Dr. Chung-Chieh Lee of Northwestern University, for his valuable advice and friendly help.

I cannot end without thanking my family, on whose constant encouragement and love I have relied throughout my time at the Academy. For my parents who raised me with love and supported me in all my pursuits. For my younger brother and sister who always listened to my heart and being my side. And most of all for my loving, supportive, encouraging, and patient boyfriend Ping who is my best research partner.

This dissertation is in memory of my grandfather whom I adore; in memory of my beloved grandmother who was the mentor of my life. I miss you dearly.

It is to them that I dedicate this work.

Tzu-Fang Sheu
Natioanl Tsing-Hua University
Nov. 2008.

Contents

List of Figures

List of Tables

Multi-Pattern Matching Algorithms for Networks

ABSTRACT

In-depth packet inspection engines, which search the whole packet payload to identify packets of interest that contain certain patterns, are urgently required. The searching results from the inspection engines can be utilized in the network equipment for varied application-oriented management. The most important technology for fast packet inspection is an efficient multi-pattern matching algorithm, which performs exact string matching between packets and a large set of patterns. This study discusses state-of-the-art pattern matching algorithms and proposes three efficient multi-pattern matching algorithms for networks: a *hierarchical multi-pattern matching algorithm* (HMA), an *enhanced hierarchical multi-pattern matching algorithm* (EHMA), and an *Aho-Corasick with Magic Structures* (ACM) algorithm.

HMA and EHMA are built based on hierarchical and cluster-wise matching strategies. The hierarchical matching strategy of HMA and EHMA can efficiently reduce the number of external memory (L2) accesses and the amount of memory space. EHMA contributes modifications to HMA and includes the ideas of *Sampling Windows* and a *Safety Shift Strategy*. The *Safety Shift Strategy* can significantly speed up the scanning process of packet inspection. HMA and EHMA improve the average-case performance of

multi-pattern matching, and are useful for the network equipment that locates at the general network environment.

Moreover, the proposed ACM presents a novel *Magic Structure* based on the Chinese Remainder Theorem. ACM needs only a small amount of memory space and does not increase computational time complexity. ACM has better worst-case performance than state-of-the-art algorithms, and is suitable for the network equipment that usually suffers heavy attacks or requires guaranteed performance.

In this study, the analyses and simulation results show that the proposed algorithms in this study outperform others. HMA and EHMA successfully reduce the average number of L2 memory accesses to about only 0.06–0.37 per code, and improve the performance to about 0.89–1161 times better than the state-of-the-art algorithms. The overall cost of ACM is about 1.1–459 times better than the existing implementations. In particular, HMA, EHMA, and ACM use only simple and easy instructions, and no special hardware is required. Therefore, the proposed multi-pattern matching algorithms are easy to be implemented in both hardware and software. Consequently, the proposed multi-pattern matching algorithms can be efficiently applied to packet inspection engines for network equipment.

1 INTRODUCTION

Many applications run over the Internet today create a high demand on in-depth network management. Low-layer network equipment checks specified fields of the packet *headers*, such as layer 2/3 switches and layer-4 firewalls. Checking only packet headers is insufficient for application-oriented management, owing to the increasing amount of information stored in packet *payloads*. Network management systems urgently need efficient and in-depth packet inspection engines for high-layer network equipment. The packet inspection engine is used to find packets of interest over the network.

A packet inspection engine in the high-layer network equipment, such as an intrusion detection system (IDS), anti-virus appliance, application firewall or layer-7 switch, typically contains a policy or rule database. In the database, every rule consists of several patterns (or signatures) and a matching action (or a series of actions). These patterns describe the fingerprints of traffic flows. A packet inspection engine applies the pre-defined patterns to identify or manage packets of interest over the network. The pattern form depends on the application of the network equipment. However, the patterns have similar features: (1) a database generally contains a few thousand patterns, of various lengths, and (2) the patterns may appear *anywhere* in any packet *payload*.

For instance, Snort is an open-source *network-based intrusion detection system* (NIDS), which is adopted to listen in packets on a network link, identify anomalous intruder behavior with a set of patterns, and generate logs and alerts through predefined actions [1]. Snort describes one pattern of the Nimda worm as “GET /scripts/root.exe?/c+dir” [2], [3]. If the Snort inspection engine detects a packet with this

pattern in its payload, then it generates appropriate alerts to warn network administrators. Pattern matching is known to be the most resource-intensive task in the Snort [4], [5], [6]. It has been shown that the pattern matching routine of Snort needs 31% of the total execution time, which is the most expensive routine [7]. Therefore, the emerging high-layer network equipment needs an efficient packet inspection engine to search the entire packet headers and payloads for pattern matching. This study focuses on the nascent issues of payload inspection, and proposes three fast multi-pattern matching algorithm.

The most important component of an inspection engine is a powerful *multi-pattern matching algorithm*, which can efficiently perform exact string matching to keep up with the growing data volume in the network. However, conventional string-matching algorithms are impractical for packet inspection [1], [8]. Because of the large pattern database, an effective inspection engine must be able to *simultaneously* search for a set of patterns, rather than iteratively performing the single-pattern matching. The performance of processing packets is not only affected by the computation time, but also strongly affected by the number of external memory accesses.

It is well known that the rate of improvement in processor speed exceeds the improvement in memory speed. The gap has been the largest problem for system builders. For example, the latency of one external memory access is about 150–250 times more than the time of one instruction cycle in the Intel IXP2x00 network processor systems [9]. Therefore, a high-speed multi-pattern matching algorithm should aim to minimize the number of external memory accesses.

This study proposes three efficient multi-pattern matching algorithms for in-depth packet inspection: a *hierarchical multi-pattern matching algorithm* (HMA), an *enhanced hierarchical multi-pattern matching algorithm* (EHMA), and an *Aho-Corasick with magic structures* (ACM) algorithm. These three algorithms can simultaneously search the packet payload for all patterns in a set. HMA, EHMA and ACM are proposed for different network situations. HMA and EHMA have better average-case performance, while ACM has better worst-case performance than the state-of-the-art algorithms. Usually, algorithms of good average-case performance work well in the general network systems. However, algorithms of good worst-case performance are very important especially for the equipment in the core and edge network requiring guaranteed services. Consequently, HMA and EHMA are useful for general network applications, and ACM is suitable for reliable network applications.

The increasing problem of network security threats means that NIDSs have become essential network applications [20], [23]. NIDSs protect network infrastructure from attacks and intrusions without modifying end-user software. To ensure effective protection, NIDSs must be capable of real-time packet inspection, and be fast enough to keep up with the ever-increasing data volume over the network. Hence, this study illustrates HMA, EHMA and ACM with the promising NIDS that makes use of a set of patterns describing known intrusions.

The rest of this study is organized as follows. Section 2 presents the background of pattern matching algorithms and the motivation of the proposed HMA, EHMA and ACM algorithms. From Section 3 to Section 5, the details of the proposed algorithms: HMA, EHMA and ACM, are described respectively, and the analyses and experimental results

are also shown and discussed. Finally, Section 6 gives the conclusions and the future works of this study.

2 BACKGROUND

This section describes the background of the *exact string matching* algorithms. The fundamental definitions and notations used in this study are firstly presented. Then the related works are discussed in this section.

2.1 General Definitions and Notations

An array is adopted to represent a *string* of characters from an alphabet set Λ. Namely, an element of string T at the position i is $T[t]$, where $T[t] \in \Lambda$. The absolute value of an object signifies the size of the object. For instance, $|T|$ represents the length of the string T, and $|\Lambda|$ is the number of elements in the set Λ. Define a function sub(T, t, B), which is the substring of T that starting from $T[t]$ to $T[t+B-1]$. A string can also be given as a set of B-grams, where a *gram* is defined as a group of characters, and B is the number of grouped characters in a gram. For exampe, the string “green” can be translated into a set of grams {‘gr’, ‘re’, ‘ee’, ‘en’} when B=2.

Let $\boldsymbol{P} = \{p_i\}$ denote a set of *distinct* patterns, where p_i is a pattern with an identification number (ID) i. Note that in the set $\boldsymbol{P}$, $p_i \neq p_j$ when $i \neq j$. Assume that the payload of an input packet T and each pattern $p_i \in \boldsymbol{P}$ are both strings drawn over Λ.

A search request ($|\boldsymbol{P}|$=1) in a conventional exact string matching algorithm generally only contains one pattern. A single-pattern matching algorithm is used to search a string (or text) T for the *first* occurrence or *all* occurrences of *one* given pattern. A multi-pattern matching algorithm is adopted to search the input T for *all* occurrences of *any* pattern $p_i \in \boldsymbol{P}$ where $|\boldsymbol{P}| \neq 1$, or to confirm that no pattern of $\boldsymbol{P}$ is in T. That is, the goal of the

multi-pattern matching is to find *all* the matched patterns in *T*, say $\boldsymbol{P_M} \subset \boldsymbol{P}$, such that $\boldsymbol{P_M}$ = $\{p_i \mid \forall p_i \subset T$ and $p_i \in \boldsymbol{P}\}$. $\boldsymbol{P_M}$ can be applied to any high-level decision policy, such as the high-priority-win, first-matched-win or other state-concerned rules.

The notation *e.f* denotes the value of the field (or offset) *f* at the entry (or address) *e*. If *e* is a table, then *e.f* means all fields named *f* of the table *e*.

2.2 Previous Works

Single-pattern matching algorithms were originally proposed to perform text searching in computer systems. In single-pattern matching, Boyer-Moore-based algorithms provide the best average-case performance in terms of computation complexity, which is sublinear to the input string [8], [13], [21]; while the Aho-Corasick algorithm has the best worst-case performance, which is linear to the input string [1], [31]. Since algorithms with better average-case performance typically work better in the real world, Boyer-Moore-based algorithms are widely used in the practical implementations. Some multi-pattern matching algorithms that modify the Boyer-Moore-based algorithms have been proposed for the IDSs in [21], [27], [29], [36]. The details are as follows.

2.2.1 The Boyer-Moore-Like Algorithms

For single-pattern matching, the Boyer-Moore algorithm (BM) [13] employs a *bad character* heuristic and a *good suffix* heuristic to build a *skip* table and a *shift* table respectively. The Boyer-Moore-Horspool algorithm (BMH), which is a variant of BM, slightly modifies the bad character heuristic to build a single *skip* table [21]. The tables of the Boyer-Moore-based (BM-based) algorithms are precomputed, and are used to obtain the number of safety shifts of *every character* during the searching process [13], [21].

Table 1. Comparisons of single-pattern matching for cmputers and multi-pattern matching for network packets.

	Single-pattern Searching	Multi-pattern Matching
Pattern Length	Long	Many patterns are very short.
Pattern Database	1. One pattern 2. $\sum_{i=1}^{\|P\|} \| p_i \| < \|T\|$.	1. Hundreds of patterns 2. Usually, $\sum_{i=1}^{\|P\|} \| p_i \| > \|T\|$
Memory Requirement	Small	Large

Therefore, some characters of the input text T can be skipped during the matching process. In other words, the safety shift (jump) of each alphabet $a \in \Lambda$ when searching a single given pattern p, say $J(a, p)$, is precomputed, and $J(a, p) \leq |p|$. The BM-based algorithm, while scanning T to verify the existence of p, checks $J(a, p)$ to locate the next character of T to scan after the input character $T[t] = a$ is scanned. This shift method speeds up the searching process.

Some algorithms apply the BM-based algorithms *iteratively* for each pattern to solve the multi-pattern matching problem. However, these algorithms were originally designed for single-pattern matching. BM-based approaches are not applicable for packet inspection, because of the different pattern length, scale of the pattern database and memory capacity. Table 1 shows these differences.

Although BMH is the best average-case algorithm for general pattern lengths in the single-pattern matching, several studies have concluded that the Brute Force method outperforms the BM-based approaches in the extreme cases of pattern length less than three characters or close to the length of the input string [8], [22], [31]. Generally, the patterns in many network systems are very short. For example, 13.7% of the patterns in the Snort pattern set have pattern lengths of less than three characters, and the range of pattern lengths is 1–122 bytes. Conventional single-pattern searching algorithms are designed for text file searching in computers, where the length of an input string is

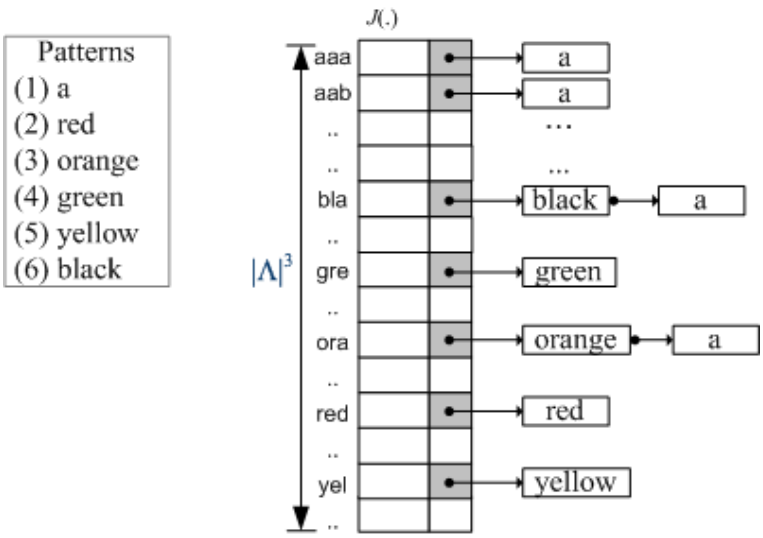


Figure 1. The memory architecture of WM-PH, where the prefix size D = 3.

typically larger than that of a pattern string. However, the input string for multi-pattern matching across a network is a packet, whose length is much smaller than the sum of the length of all patterns. Moreover, the pattern set ($|\boldsymbol{P}|$) is generally very large in a network system. Notably, the required amount of memory are very important, especially in a hardware-based design. For single-pattern searching, the table size of BM-based algorithms is $O(|\Lambda|)$. However, for multi-pattern matching, the table size of BM-based algorithms rises significantly to $O(|\boldsymbol{P}|\times|\Lambda|)$.

To search for a set of patterns, Snort runs a BM-based algorithm iteratively for each pattern. In this case, the time complexity is $O(\sum_{i=1}^{|P|}|p_i|+|\boldsymbol{P}|\times|T|)$ for one input string T [5], [31]. BM-based algorithms obviously have poor packet inspection performance due to the large pattern set in the network. The complexity of implementing the conventional matching algorithm has been cited as a reason why it has not been adopted extensively in

Table 2. Comparing the shifts of BM-based, FV, WM, and WM-PH algorithms.

	Shift Value	Maximum Shift
BM-based	$J(a, p)$	$\|p\|$
FV	$\min\{J(a, p_i) \mid \forall\ p_i \in \boldsymbol{P}\}$	$\min\{\|p_i\| \mid \forall\ p_i \in \boldsymbol{P}\}$
WM	$\min\{J(g, p_i) \mid g \subset p_i,\ \forall\ p_i \in \boldsymbol{P}\}$, where $\|g\| = D$	$\min\{\|p_i\| \mid \forall\ p_i \in \boldsymbol{P}\} - D + 1$
WM-PH	$\min\{J(x, p_i) \mid \text{sub}(p_i, 1, D) = x,\ \forall\ p_i \in \boldsymbol{P}\}$	D

multiple-pattern matching [7], [21]. Markatos's approach promoted Snort by using a bitmap filter before BMH, but still searching for only one pattern in each iteration [29].

Several modifications to BM-based algorithms have been developed to solve the multiple-pattern matching problem. Fisk and Varghese's method (FV) *groups* all patterns to precompute the safety shifts [7]; Wu and Manber's algorithm (WM) groups *D*-grams of the prefixes of all patterns to build a shift table based on the *bad gram* heuristic, where each entry contains the safety shift of each *D*-gram [36]; Liu *et al.* presented an algorithm (WM-PH) that groups the prefixes of all patterns to build a large hash table, where the length of the prefix is *D* [27]. Figure 1 displays the memory architecture of WM-PH. Notably, WM-PH has to duplicate the patterns of length smaller than *D* in the hash table to avoid a miss. Table 2 presents the shift values of BM-based, FV, WM and WM-PH algorithms.

Obviously, grouping a large number of patterns leads to a small average shift. The valid shift decreases as the size of pattern set grows. Additionally, the maximum shift value of the FV and WM must be less than the minimum pattern length in $\boldsymbol{P}$ in order to avoid missing any pattern. Hence, FV and WM are unfeasible for the inspection engines when the pattern set includes single-symbol patterns. The required memory space of the table for WM and WM-PH is $O(|\Lambda|^D)$. Generally, $D = 3$, and the table requires 16M entries when the alphabet size is 256. These large tables must be held in the external memory, which leads to long access delay during the matching process. Furthermore, because the

safety shift of a *D*-gram *g* in the BM-like algorithms relates to all patterns containing *g*, it is a very complicated process to derive the shifts and updating the tables when the pattern set is changed. The BM-like inspection engines must be suspended for table update, even when only one pattern is added or removed.

2.2.2 The Aho-Corasick-Based Algorithms

The Aho-Corasick (AC) algorithm is a well-known algorithm that provides the best worst-case computational time complexity [1], [31]. AC is an automaton-based algorithm. By using a simple data structure, the memory space required to store the *transition matrixes* of the states is in the order of $O(|\Lambda| \times S)$, where S is the number of states of the automaton. Using a compressed structure, Tuck *et al.* modified AC (named AC-C), and lowered the required memory to about 2% of the original AC [34]. However, the data structure of AC-C is still too large to be cached in the on-chip cache of general chips. Although the AC-based algorithms have the best worst-case computational time complexity, the latency of external memory access dominates the processing performance rather than the computational time. Even in the best-case scenario, AC still needs at least two memory references per character. Additionally, even when only one pattern is removed, AC must rebuild the failure table since AC's failure table is built by correlating the entire pattern set. AC-C also needs to rebuild the entire state machine when it adds or deletes a pattern, because the structures of AC-C are compressed. Consequently, the AC-based inspection engine has to be suspended for pattern update, and the suspended time is proportional to the total length of *all* patterns in $\boldsymbol{P}$ [1].

Coit *et al.* proposed a matching algorithm for Snort by combining AC and BM [15]. However, their algorithm requires three times the memory of the standard version, and may yield inconsistent results.

2.2.3 Other Approaches

In the case of hardware solutions, Li *et al.* developed an FPGA-based inspection engine for NIDSs, using the internal content addressable memory (CAM) to speed up multi-pattern matching [28]. Because the size of an internal CAM of FPGA is not large enough to store all patterns, Li *et al.*'s engine dynamically reloads a block of patterns into the CAM, resulting in long latency. Moreover, Li *et al.*'s approach does not solve this problem while the patterns of varied lengths complicate the formulation of a CAM for exact matching.

Additionally, Dharmapurikar *et al.* adopted Bloom Filters (BFs), and Kim *et al.* employed mask filters in the FPGA-based packet inspection [17], [24]. However, these two methods only act as filters and have to cooperate with another string matching algorithm to verify a match. Furthermore, this Bloom-Filter-based algorithm can be used only in the case that all patterns are longer than a certain length.

Lu *et al.* used several binary CAMs and BFs to implement parallel compressed deterministic finite automata (DFAs), and Dharmapurikar *et al.* combined AC with BFs for packet inspection [18], [22]. Both approaches utilize parallel BFs, and assume that a BF can execute one query every clock cycle. However, these architectures and assumptions are only valid in specific hardware implementations. BFs are inefficient in software implementations, because one BF is composed of several hash functions, which

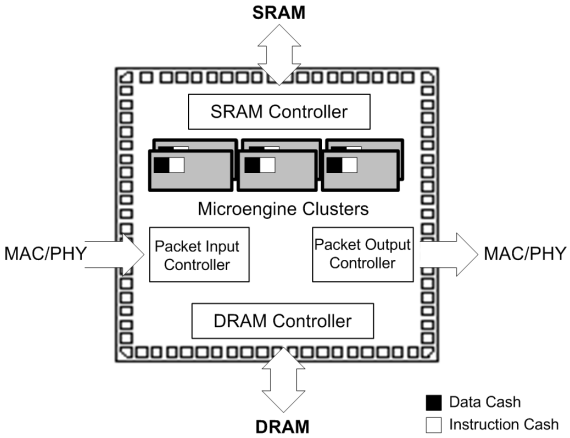


Figure 2. The architecture of a network processor.

generally have long computation times in software [19], unless hash functions are carefully selected for different CPUs.

A Piranha algorithm, based on the idea that a pattern can be identified from its *least popular D*-gram of a pattern, has been presented [11]. A least popular gram of a pattern is selected as an index key of a pattern. However, the Piranha algorithm cannot handle patterns with lengths smaller than *D*, and require a large memory space ($O(|\Lambda|^D)$).

2.3 Motivations

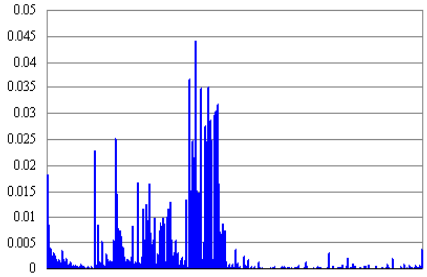
Generally, there are three ways to improve the performance of a real-life appliance: (1) reduce the number of required instructions for a task (computation complexity); (2) reduce the memory requirement (space complexity); (3) reduce the number of memory references, especially the external memory references (access latency).

2.3.1 Network Processors and Micro-processors

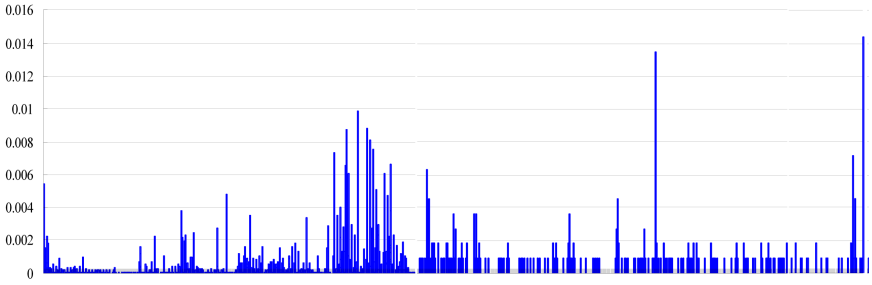
Programmable chips, such as network processors, FPGAs, networking on chips (NOCs) or system-on-a-programmable-chips (SOPCs), are increasingly used in

implementations in order to have the performance and flexibility at the same time [6], [12]. Although microprocessors are slower than general CPUs, microengine clusters using pipeline or parallel technologies have been proposed to overcome this shortage. A network processor system generally consists of microengines, on-chip memory (L1 cache), external memory (L2 memory), and packet control modules (Figure 2). Due to the cost and power consumption issues, programmable chips generally have small on-chip cache. For example, the Intel IXP2x00 network processor has only a 4 KB instruction cache and a 2 KB data cache in each microengine, while the Vitesse IQ2000 network processor has a 4 KB data cache [22], [35]. Nevertheless, the required memory capacity of the existing multi-pattern matching algorithms for Snort's database is usually larger than 300 KB. Because the number of patterns is still growing, the on-chip cache of general programmable chips is typically too small to store the tables and patterns for the existing algorithms. Therefore, the pattern content and lookup tables built by matching algorithms have to be stored in the external memory.

However, frequently accessing the external memory (to read patterns or tables) significantly decreases the matching efficiency due to the long and indeterminable access latency of the external memory. It has been pointed out that processor speed doubles every 18 months, while the memory latency improves by only 7% per year. For example, Intel IXP2x00 needs about one cycle for one basic instruction, but about 150 cycles for one access from SRAM (or 250–300 cycles from DRAM) [9]. While considering implementation issues, the system performance is strongly affected by memory latency. Therefore, reducing the number of required external memory accesses is more important than reducing the amount of computational time [34].



(a) Spectrum of 1-gram.



(b) Spectrum of 2-gram.

Figure 3. The pattern spectrum when $|\boldsymbol{P}| = 1200$ from Snort's rule set.

The proposed three multi-pattern matching algorithms: HMA, EHMA and ACM, all try to lower the number of external memory accesses, and reduce the amount of required memory space at the same time.

2.3.2 Hierarchical Architectures

As shown in Section 2.2 (and later shown in Table 7 and Table 13), every existing algorithm uses a large index table for multi-pattern matching. To reduce the number of external memory accesses, the idea of HMA and EHMA is to use hierarchical architectures: hierarchical memories and hierarchical matching strategies.

The hierarchical architecture is a common idea and has been used to solve many problems. However, how to obtain a small first-tier table is the major point for a fast and

efficient multi-pattern matching algorithm. This study will propose novel methods to obtain smaller index tables.

2.3.3 Pattern Spectrum

Firstly, Snort's patterns are analyzed, because index tables of matching algorithms are constructed by the patterns. Figure 3 plots the pattern spectrum of the Snort patterns. The pattern spectrum indicates the occurrence frequency of grams of patterns. Figure 3 (a) shows the distribution of 2-grams of patterns, and Figure 3 (b) is the distribution of characters of patterns.

As shown in the figures, they are not normally or uniformly distributed, and have several peaks, which mean that some grams obviously occur more frequent than others. Hence, the idea of finding a small first-tier table is possible.

2.4 The Sketches of the Proposed Algorithms

Generally, an algorithm of better average-case performance performs better in real-life applications. However, some applications working in special situations require guaranteed performance and reliable systems, such as core routers. In this case, an algorithm of better worst-case performance is demanded. Consequently, this study proposes three algorithms for different requirements. The hierarchical-based algorithms, called HMA and EHMA, have better average-case performance; while the automaton-based algorithm, called ACM, has better worst-case performance than the state-of-the-art algorithms.

2.4.1 HMA

The *hierarchical multi-pattern matching algorithm* (*HMA*) for in-depth packet inspection simultaneously searches the packet payload for all patterns in a set. A small first-tier table from the *most frequent common-codes* of patterns is used to narrow the searching scope. HMA significantly reduces external memory accesses and pattern comparisons by two-tier and cluster-wise matching strategies. HMA requires much less memory space than current state-of-the-art multi-pattern matching algorithms [12], [21], [27], [34], [36]. For instance, HMA requires less than 350 KB to import the Snort database of 1200 patterns and it reveals small-scale and cost-effective implementations. The average number of external memory accesses in HMA is about only 0.1–0.37 per byte, which efficaciously improves the performance of the inspection engine. Simulation results demonstrate that HMA performs about 0.9–410 times better than the state-of-the-art algorithms [21], [27], [34]. HMA has better best-case and average-case performance, and also manageable worst-case performance. HMA furthermore has an incremental pattern update mechanism to make it reliable and appropriate for on-line network equipment. Consequently, HMA is a very cost-effective and efficient mechanism that can be employed in fast network content inspection.

2.4.2 EHMA

The *Enhanced Hierarchical Multi-pattern Matching Algorithm* (*EHMA*) for fast in-depth packet inspection can simultaneously searches the packet payload for a set of patterns. EHMA contributes modifications to HMA [38], and introduces the idea of a *sampling window* and a *Safety Shift Strategy* in addition. EHMA is a two-tier and cluster-wise matching algorithm, and can perform fast skippable payload scan. Based on

the occurrence frequency of *grams*, this study discovers a small set of signatures from the *patterns* themselves to narrow the searching domain. A Min-Max strategy is used in the EHMA. The hit rate of the first-tier table in the EHMA is minimized, while the spread of patterns in the second-tier table is maximized. Accordingly, EHAM significantly reduces the number of memory accesses and pattern comparisons. EHMA can skip unnecessary payload scans by applying the proposed *Safety Shift Strategy*, which is based on a *frequency-based bad gram heuristic*. The frequency-based bad gram heuristic is a modification of the *bad grouped character heuristic* of Wu-Manber algorithm (WM) [36]. Therefore, EHMA has the advantages of both HMA and WM.

The memory space and the number of external memory accesses required by the proposed EHMA are much smaller than those required by the state-of-the-art multi-pattern matching algorithms. EHMA needs less than 40KB memory space to construct required tables for the Snort of 1200 patterns, and therefore enables small-scale and cost-effective hardware implementations. Using only 768 bytes on-chip memory, EHMA reduces the average number of external memory accesses to 0.06–0.19, and thus significantly improves the matching time of the detection engine. Simulation results reveal that the matching performance of EHMA is about 0.89–1161 times better than other matching algorithms [12], [21], [27], [34], [36], [38]. Even under real-life intense attack, EHMA still outperforms others. Because employing only basic instructions and two small index tables, EHMA is very simple for hardware and software implementations. Consequently, the proposed EHMA is a very cost-effective and efficient mechanism for real-life network detection systems.

2.4.3 ACM

Guaranteed performance is very important especially for the equipment in the core and edge network. The AC algorithm has the best worst-case computational time complexity for multi-pattern matching, where the number of state transitions for each input symbol is at most two [1], [19]. However, as for realistic implementations, the performance of an algorithm is not only affected by the computation time, but also strongly affected by the number of required memory references. Because using the conventional simple structures to implement the AC algorithm requires a large amount of memory, the performance of AC in a realistic implementation is not good as the theoretical value. Therefore, this study proposes a *Magic Structure* based on the property of Chinese Remainder Theorem and contributes modifications to the AC algorithm (named ACM) for fast in-depth packet inspection.

The Magic Structure needs only a small amount of memory and features fast traversing schemes. This study uses NIDSs to illustrate the performance of ACM. The results show that ACM has better worst-case performance than others. The overall cost of ACM is about 1.1–459 times better than the existing implementations. The performance of the Magic Structure is analyzed, which shows that the Magic Structure performs very well especially for sparse graphs.

3 THE HIERARCHICAL MULTI-PATTERN MATCHING ALGORITHM (HMA)

Based on the concept of hierarchical and cluster-wise matching, the proposed HMA can effectively reduce the number of external memory accesses and string comparisons without sacrificing the memory space. HMA comprises two stages: the off-line preprocessing stage and the on-line matching stage. The off-line stage constructs two small tables for the on-line stage.

A *frequent common-code searching* algorithm (*FCS*) and a *cluster balancing strategy* (*CBS*) are proposed for the table construction. To obtain smaller index tables and narrow the searching scope, an idea of *frequent common-codes* of patterns is used. FCS is proposed to find out the frequent common-code set ***F***, which is used to build the *first-tier table*: H^1; the ***F*** and CBS are used to build the balanced *second-tier table*: H^2. H^1 and H^2 act as two filters to avoid unnecessary external memory accesses and pattern comparisons, and thereby pass the innocuous packets quickly in the on-line matching stage. The second-tier matching activates only after the first-tier gets a match, and H^2 indicates a small cluster of patterns that are similar to the input packet for real comparisons. HMA compares only a few *selected* patterns of ***P*** with the *suspected* substrings of a packet, rather than comparing *all* patterns with *all* substrings of a packet. Consequently, HMA significantly improves the matching performance. The FCS and CBS algorithms and the on-line hierarchical matching stage of HMA are described in the following subsections.

FCS Algorithm
Input: A set of patterns $\boldsymbol{P}$.
Output: A set of frequent common-codes $\boldsymbol{F}$.
1 **Initialize**: $\boldsymbol{F} \leftarrow \varnothing$;
2 **For** each pattern p_i of $\boldsymbol{P}$, $0 \le i < |\boldsymbol{P}|$ **do**
3 Transfer the first $|p_i|$-1 codes of p_i into a vector M by setting $m_j = 1$ if $j \in p_i$; otherwise $m_j = 0$, for all j, $0 \le j < |\Lambda|$;
If p_i is a single-code pattern, set $m_j = 1$ if $j = p_i$.
4 Read M. For each $m_j = 1$, set the elements of matrix R: $r_{jk} = r_{jk} + m_k$, for all k, $0 \le k < |\Lambda|$;
5 **While** $r_{ii} \neq 0$, $0 \le i < |\Lambda|$ **do**
6 Find a frequent common-code f, where $r_{ff} = \max\{r_{ii} \mid \forall\, i, 0 \le i < |\Lambda|\}$;
7 Add this code into $\boldsymbol{F}$: $\boldsymbol{F} = \boldsymbol{F} \cup \{f\}$;
8 **For** $0 \le i < |\Lambda|$ **do** /* refresh R*/
9 $r_{ii} = r_{ii} - r_{fi}$, if $r_{ii} > r_{fi}$; otherwise, $r_{ii} = 0$;
10 **Return**;

Figure 4. The FCS algorithm.

3.1 The FCS Algorithm

Since the packet payload T and the patterns in $\boldsymbol{P}$ are strings drawn over the same alphabet set Λ, and in addition the patterns may appear anywhere in the packet payload, to recognize the packets that have the patterns is difficult. HMA assumes that a small set of *signatures* can be found from the patterns themselves, and then by using the signatures, distinguishing the suspicious substrings of T will become easier. A set of *significant codes* is defined as representatives of a pattern set $\boldsymbol{P}$, given by $\Im \subset \Lambda$. A pattern of $\boldsymbol{P}$ may exist in the payload only when at least a significant code exists. In other words, for each pattern $p_i \in \boldsymbol{P}$, at least one character of p_i occurs in $\Im$. Many innocent characters of T that do not belong to $\Im$ can be skipped without further processing when scanning the input T. Obviously, smaller $\Im$ leads to fewer pattern comparisons, and thus faster pattern matching. The FCS is proposed to find the smallest $\Im$ from $\boldsymbol{P}$.

Define $\boldsymbol{P_c}$ as a subset of $\boldsymbol{P}$, and all patterns in $\boldsymbol{P_c}$ contain a *common-code c*, which means $\boldsymbol{P_c} = \{p_i \mid c \in p_i \text{ and } p_i \in \boldsymbol{P}\}$. Obviously, if there is a common-code that appears in

distinct patterns more frequently than other codes, and it is selected as one of $\Im$, then a smaller $\Im$ is found. Based on this inference, FCS is designed to find the *frequent common-code set* of a given ***P***, denoted $\boldsymbol{F} = \{f_i \mid f_i \in \Lambda\}$, such that ***F*** is the minimum set of significant codes to represent the pattern set ***P***, where f_i is a *frequent common-code*.

The FCS algorithm is presented in Figure 4, using a $|\Lambda|$ vector $M = (m_i)$ and a $|\Lambda| \times |\Lambda|$ matrix $R = (r_{ij})$ as temporary memory, where $0 \le i, j < |\Lambda|$. M is a bit-map recording the occurrence of each character in a pattern. R is used to record the occurrence frequency, where r_{ij}, $i \neq j$, indicates the relations of concurrent occurrence between two alphabets a_i and a_j in ***P***, and r_{ii} records the frequency of an alphabet $a_i \in \Lambda$ occurring in different patterns. For example, r_{ij}=2 means that currently two patterns in ***P*** contain both a_i and a_j. Firstly, the FCS algorithm records the character occurrence of each pattern in the bit-map M, and then accumulates the elements of M into the corresponding elements of R respectively (lines 2-4). Secondly, FCS finds the largest occurrence frequency r_{ff}, and consequently the corresponding alphabet a_f is selected to be one of ***F***. Then the elements of R relating to a_f are subtracted accordingly to renew R (lines 6-9). FCS repeats until all elements on the diagonal of R become zero.

After FCS finds out ***F*** from ***P***, ***F*** is used to construct a small index table, called the *first-tier table* (H^1). To speed up the process, H^1 uses a direct index table of $|\Lambda|$ entries. The *a*th entry of H^1 is denoted $H^1(a)$, where each entry has two fields: the frequent code ID, say $H^1(a).fid$; and the single-symbol pattern ID, $H^1(a).pid$. That is $H^1(a).fid = \{i \mid a = f_i \in \boldsymbol{F}\}$, and $H^1(a).pid = \{i \mid |p_i| = 1, p_i = \text{‘}a\text{’}$ and $p_i \in \boldsymbol{P}\}$. The unused fields of H^1 are set as NULL. Since H^1 is a small table, e.g. only 256 entries in the case of one-byte coding, it can be stored in the on-chip cache. Later, H^1 acts as a first-tier filter in the on-line stage to

quickly discover whether a packet contains a pattern. Namely, HMA makes use of $\boldsymbol{F}$ to narrow the searching scope to the most likely subset of patterns (clusters).

3.2 The Cluster Balancing Strategy (CBS)

Generally, most packets are innocent and a harmful packet may contain only few patterns. Hence, comparing all of the patterns in the large $\boldsymbol{P}$ with each input packet is time consuming. If the patterns in $\boldsymbol{P}$ can be distributed into different small *clusters* based on their similarity, then only the patterns in few clusters that are most similar to the input need to be compared. Therefore, the efficiency of the matching process is improved. This subsection presents strategies to attain this goal. First, the method of clustering a set $\boldsymbol{P}$ based on the similarity of patterns is described. Then a cluster balancing strategy (CBS) is used to balance the cluster size, and finally a *second-tier table* (H^2) for on-line matching based on the clustering results is built.

Define the *clustering pivots* as the keys used to distribute patterns, where each clustering pivot is a common-code of patterns defined previously. Two common-codes are employed as a pair of clustering pivots, called a *pivot pair* and noted as (a, b), where the first pivot is a frequent common-code of $\boldsymbol{F}$, and the second pivot is the code following the frequent common-code. Let $\boldsymbol{P}_{a,b}$ represent a cluster of selected patterns (a subset of patterns) with the pivot pair (a, b), which means that $\boldsymbol{P}_{a,b} = \{p_i \mid \text{`}ab\text{'} \subset p_i, a \in \boldsymbol{F} \text{ and } b \in \Lambda\}$, where '$ab$' is the combination of two strings a and b, and is a substring of p_i. Notably, a pattern is assigned to only one cluster in the clustering strategy, although a pattern may have more than one pivot pair. That is, the clusters have the following properties: any cluster $\boldsymbol{P}_{a,b} \subset \boldsymbol{P}$, $\bigcup_{\text{all } a,\, b} \boldsymbol{P}_{a,b} = \boldsymbol{P}$ and $\bigcap_{\text{all } a,\, b} \boldsymbol{P}_{a,b} = \varnothing$. Since a pattern may have several

opportunities to select a cluster, a better assignment can lower the maximum cluster size, and thereby improving the worst-case performance of HMA.

In order to lower the worst matching time, CBS is employed to balance the size of clusters. In CBS, an $|\boldsymbol{F}|\times|\Lambda|$ matrix N = ($n_{a,b}$) is used to record the current size of a cluster $\boldsymbol{P}_{a,b}$. The algorithm is as the followings. Firstly, CBS reads one pattern at a time from $\boldsymbol{P}$ and scans the pattern. According to FCS, for any given p_i, there exists a character such that $p_i[k]\in \boldsymbol{F}$, where $1\le k<|p_i|$. To balance the cluster size, CBS finds the smallest $n_{a,b}$ among all available pivot pairs of p_i, say (a, b), where $a\in \boldsymbol{F}$ and 'ab'$\in p_i$. After group p_i into the smallest cluster $\boldsymbol{P}_{a,b}$, the corresponding $n_{a,b}$ is then incremented. All patterns are distributed sequentially into the designate clusters in the same way.

The second-tier table H^2 is constructed based on the cluster assignments. H^2 contains the pattern contents and the patterns' formatted information for fast on-line matching. Let $H^2(a, b)$ denote an entry of H^2, storing the head pattern of a cluster $\boldsymbol{P}_{a,b}$, and defined as

$$H^2(a, b) = h(a)\times|\Lambda| + b,$$

where $h(a) = H^1(a).fid$. Each entry $H^2(a, b)$ consists of five fields: the pattern size $H^2(a, b).size$, the pattern content $H^2(a, b).data$, the position of the frequent common-code in the pattern $H^2(a, b).offset$, the pattern ID $H^2(a, b).pid$, and a pointer $H^2(a, b).next$ to the entry of the next pattern in the same cluster or the fragmented content of the current pattern. Transferring the information of patterns into a predefined format can accelerate the matching procedure. The patterns in the same cluster are linked by the linked-list structure to optimize the memory utilization.

For example, if p_i is clustered to $\boldsymbol{P}_{a,b}$ and $H^2(a, b)$ is empty, then the information of p_i is saved into $H^2(a, b)$, where $H^2(a, b).size = |p_i|$, $H^2(a, b).data = p_i$, $H^2(a, b).offset = k$ if

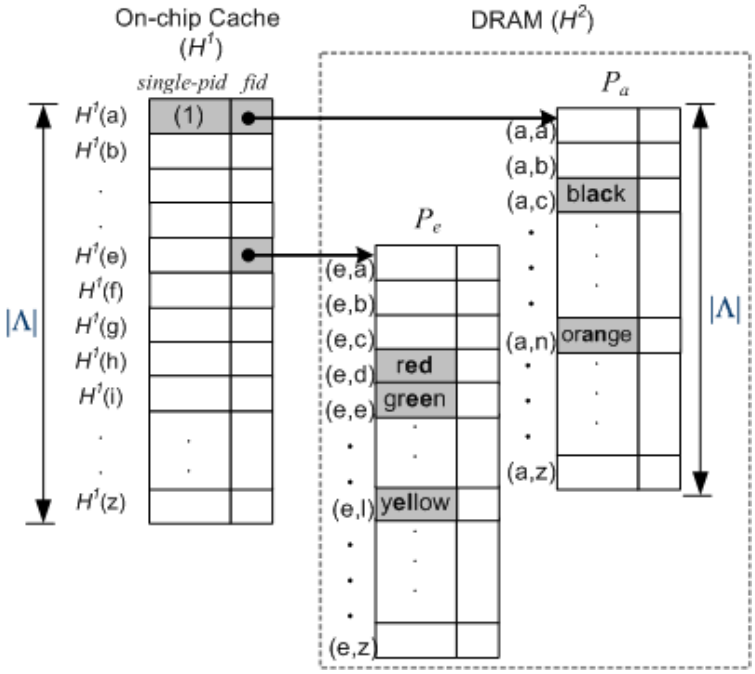


Figure 5. The hierarchical index table of HMA.

$p_i[k] = a$, $H^2(a, b).pid = i$, and $H^2(a, b).next$ is NULL. If the pattern size of p_i is larger than the width of data field, p_i is fragmented and the remaining part is saved in a free entry $H^2(a', b')$ in the shared memory pool. $H^2(a', b').size$ is $H^2(a, b).size$ minus the width of the data field, and $H^2(a, b).next$ is pointed to $H^2(a', b')$. Similarly, if another p_j is also clustered to $\boldsymbol{P}_{a,b}$, then a free entry is also assigned to p_j, and p_i and p_j are linked by a pointer.

Figure 5 illustrates the logical architecture of the index tables of HMA, assuming the alphabets are 26 English letters. This example has six patterns as shown in Figure 1. Since 'e' and 'a' are the most frequent common-codes that both occur in three different patterns, FCS discovers $\boldsymbol{F}$ = {e, a} as the signatures of these six patterns. Since H^1 has only |Λ| (= 26) entries, it can be stored in the on-chip cache. The *fid* fields of H^1 are pointing to the corresponding offsets of H^2. Since the first pattern 'a' is a single-pattern, its *pid* (= 1) is

stored in the H^1 table. As the pattern 'r**ed**' has 'e'$\in \boldsymbol{F}$ and the pivot pairs (e, d), 'r**ed**' is grouped to the cluster $\boldsymbol{P}_{e,d}$ according to CBS. The remainders of the patterns follow the same clustering strategy.

3.3 The On-line Hierarchical and Cluster-wise Matching

The off-line stage of HMA constructs two tables H^1 and H^2, holding the index and pattern information in the cache memory and external memory respectively. These two tables are regarded as the two-tier filters and as indices for the on-line matching. In this subsection, the on-line stage of HMA are presented in detail. Notably, HMA is designed for multi-pattern matching, where the pattern lengths are varied from one character to hundreds of characters. HMA has no constratnt on the minimum length of patterns.

In a packet inspection engine, an input packet is a given object and forwarded to the engine for multi-pattern matching. Then the inspection engine returns the searching results of matched patterns $\boldsymbol{P}_{\boldsymbol{M}}$. This study focuses on the payload inspection and assumes that every input is a packet payload T. To reduce the times of external memory accesses, HMA uses a hierarchical matching scheme. The matching process of HMA is divided into two tiers: the first-tier matching and the second-tier matching.

3.3.1 The First-tier Matching

In the matching stage, T is scanned from left to right, and each character $T[t]$ is used as the index key to fetch the entry $H^1(T[t])$ in H^1. H^1 acts as the first-tier filter of HMA, using to check weather T contains any pattern of $\boldsymbol{P}$. Since H^1 is small enough to be kept in the embedded memory of microengines, the latency of accessing H^1 is much less than that of accessing external memory.

In the first-tier matching, if $H^1(T[t]).pid$ is not NULL, then $T[t]$ is a single-symbol pattern, and this matched pattern will be added into $\boldsymbol{P_M}$. Whether $H^1(T[t]).pid$ is NULL or not, then the first-tier matching procedure checks the *fid* field.

If $H^1(T[t]).fid$ is NULL, i.e., $T[t] \notin \boldsymbol{F}$, $T[t]$ will be skipped with no pattern comparison, and thereby no external memory is necessary. Then the on-line matching stays in the first-tier matching, proceeding to the next character $T[t+1]$ and checking the $H^1(T[t+1]).pid$ as previous steps. Since $|\boldsymbol{F}|$ is much smaller than $|\Lambda|$, most characters of T can gain the skips and avoid the second-tier matching. Consequently, both the number of character comparisons and costly memory accesses can be reduced.

If $T[t] \in \boldsymbol{F}$, T may contain a pattern $p_i \in \boldsymbol{P}$, where $T[t] \in p_i$. That is, as $H^1(T[t]).fid$ is not NULL, T may have a pattern (or more than one) belonging to the cluster $\boldsymbol{P}_{T[t],T[t+1]}$. Then the second-tier matching is activated to identify the pattern.

3.3.2 The Second-tier Matching

After the first-tier matching, as long as $H^1(T[t]).fid$ is not NULL, the matching procedure proceeds to the second-tier matching. $H^2(T[t], T[t+1])$ indicates the location of the corresponding cluster $\boldsymbol{P}_{T[t],T[t+1]}$ according to the input T. As a cluster-wise matching, HMA checks only the patterns in the small cluster $\boldsymbol{P}_{T[t],T[t+1]}$, which are most similar to T.

In the second-tier matching, firstly the *pid* field of H^2 is checked. If $H^2(T[t], T[t+1]).pid$ is NULL, it means the cluster $\boldsymbol{P}_{T[t],T[t+1]}$ has no pattern. Afterward, the next character $T[t+1]$ is scanned, and the matching procedure returns to the first-tier matching. Otherwise, if $H^2(T[t], T[t+1]).pid$ is valid, it means the cluster $\boldsymbol{P}_{T[t],T[t+1]}$ has patterns similar to T. Then, HMA compares the pattern content in $H^2(T[t], T[t+1])$ with the suspected part of T, sub(T, $T[t\text{-}H^2(T[t], T[t+1]).offset]$, $H^2(T[t], T[t+1]).size$). If the

```
   Procedure OnlineMatching(T, H^1, H^2)
   Input: Packet payload T, two preprocessed indexing tables: H^1 and H^2
   Output: The matched pattern set of T: P_M, and its corresponding pid PID_M
1  Load the input payload into buffer T;
2  Initialize: P_M ← ∅;
3  For each T[t] do
4     If (k ← H^1[T[t]].pid) ≠ NULL then P_M ← P_M ∪ {p_k} and PID_M ← PID_M ∪ {k};   /* First-tier matching*/
5     If (k ← H^1[T[t]].fid) ≠ NULL && t < |T| then
6        Load data from the external RAM at entry H^2(T[t], T[t+1]) to a local buffer LB;
7        While (k ← LB.pid) ≠ NULL do                                  /* Second-tier matching*/
8           Compare the substring start at T[(t-LB.offset)] with the pattern LB.data of length LB.size;  /*
   Assume no fragmentation here*/
9           If the comparison is matched then P_M ← P_M ∪ {p_k} and PID_M ← PID_M ∪ {k};
10          If LB.next ≠ NULL then
11             Load data from the external RAM at entry LB.next to the local buffer LB;
12          Else
13             Break;
14 Return;
```

Figure 6. The on-line matching procedure of HMA.

pattern size $H^2(T[t], T[t+1]).size$ is larger than the width of a *data* field, the next fragment of the pattern at $H^2(T[t], T[t+1]).next$ is fetched and compared only when the current fragment gets a match. If the next field of the last pattern fragment points to a valid next pattern, say at $H^2(a, b)$, similarly the pattern in $H^2(a, b).data$ is compared with the substring of T starting at $T[t\text{-}H^2(a, b).offset]$. All matched patterns are added to $\boldsymbol{P_M}$.

Notably, if a pattern p_i exists in T, then all characters of p_i will appear in T. Definitely, the clustering pivot pair of pattern p_i, say $p_i[k]$ and $p_i[k+1]$, will be found in T, say at $T[t]$ and $T[t+1]$, where $T[t] = p_i[k] \in \boldsymbol{F}$. When T compares with the patterns in the cluster $\boldsymbol{P}_{T[t],T[t+1]}$ during the matching procedure, p_i will be recognized. Consequently, no patterns in the payload T will be missed.

The on-line matching procedure of HMA is presented in Figure 6. Obviously, only few suspected patterns are loaded from external memory, and the number of string comparisons is decreased. HMA can scan the packets rapidly by using H^1 and H^2, since

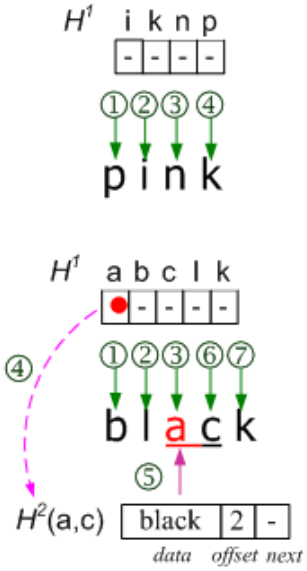


Figure 7. Examples of HMA on-line matching, where the input strings are ‘pink’ and ‘black’.

most packets in the network are generally innocent and the obtained ***F*** narrows the searching scope.

Figure 7 demonstrates the on-line matching of HMA. Assume the H^1 and H^2 tables have been constructed as Figure 5, where ***F*** = {e, a}. HMA scans the input *T* from left to right. If *T* = ‘pink’, after checking *T* with the on-cache H^1 for four times and finding that all characters of *T* do not belong to ***F***, HMA knows that *T* contains no pattern and no external memory access is required. If *T* = ‘black’, HMA stays in the first-tier matching until ‘a’ is scanned, and finds that ‘a’$\in$***F*** (H^1(a).*fid* is valid) and ‘a’ is a single-symbol pattern (H^1(a).*pid* = 1). Then, ‘a’ and its following ‘c’ are used as the index keys (pivot pair), and the second-tier matching loads an entry from H^2(a, c) for further checks. Because H^2(a, c).*pid* (= 6) is not NULL, HMA compares the substring(s) of *T* with the pattern(s) in $\boldsymbol{P}_{a,c}$, where H^2(a, c).*data* = ‘black’, and a match is got. As H^2(a, c).*next* is NULL, the on-line matching process returns to the first-tier matching as the previous steps.

Since ‘c’ and ‘k’$\notin \boldsymbol{F}$, the scanning process of this input is finished. For the input ‘black’, only one external memory access is required. The result of this case is $\boldsymbol{P_M}$={a, black}.

```
   Procedure AddPattern(p_i, |F|, H^1, H^2)
   Input: A new pattern p_i with pattern ID i, the current |F|, H^1 and H^2
   Output: The new H^1 and H^2
1  Initialize: flag ← FALSE;
2  If |p_i|==1 && H^1(p_i[1]).pid ≠ NULL then
3     H^1(p_i[1]).pid ← i;
4  Else
5     For each p_i[j] && j<|p_i| do
6        If H^1(p_i[j]).fid ≠ NULL then
7           flag ← TRUE;
8           If H^2[p_i[j]][p_i[j+1]].pid==NULL then
9              Save the information of p_i to the entry H^2(p_i[j], p_i[j+1]); /* Assume no fragmentation here*/
10             Return;
11    If flag==TRUE then                                   /* No empty cluster*/
12       Choose a random number r, 1 ≤ r<|p_i|, such that p_i[r]∈ F;
13       Add the pattern p_i to the cluster P_{pi[r], pi[r+1]}: Find the last entry of the cluster, say H^2(a', b'), and save
   the information of p_i to an empty entry of H^2, say H^2(a'', b'');
14       H^2(a', b').next←H^2(a'', b'');
15    Else
16       Choose a random number r, 1 ≤ r<|p_i|, set H^1(p_i[r]).fid←|F|++ and save the information of p_i to the
   entry H^2(p_i[r], p_i[r+1]);
17 Return;
```

```
   Procedure DelPattern(p, H^1, H^2)
   Input: A pattern p to delete, the current H^1 and H^2
   Output: The new H^1 and H^2
1  Initialize: Set temporary registers prev←NULL and this←NULL;
2  If |p|==1 && H^1[p[1]].pid ≠ NULL then
3     H^1[p[1]].pid←NULL;
4  Else
5     For each p[j] do
6        If (k←H^1[p[j]].fid) ≠ NULL && j < |p| then
7           Load data from the external RAM at entry this←H^2(p[j], p[j+1]) to a local buffer B, and f←p[j];
8           While (k← B.pid) ≠ NULL do
9              Compare the substring start at p[(j-B.offset)] with the pattern B.data of length B.size;
10             If the comparison is matched then              /* Assume no fragmentation here*/
11                If prev ≠ NULL then prev.next ←B.next;
12                Clear the data of the entry at this and Goto Line 16;
13             Else If (prev←B.next) ≠ NULL then
14                Load data from the external RAM at entry B.next to a local buffer B;
15             Else Break;
16 If for all a∈ Λ that H^2(f, a) = NULL then H^1(f).fid←NULL;      /* Check the frequent common-code
   after delete p*/
17 Return;
```

Figure 8. The incremental update of HMA.

3.4 The Incremental Update

A packet inspection engine, just like most network equipment, must work persistently to avoid missing any packet. When an inspection engine suspends, even for only 30 seconds, millions of packets will cut through it without any inspection. Nevertheless, the pattern database of network equipment has to be updated frequently, because contents over networks change with each passing day. For example, when a new attacking scenario is discovered, the new patterns must be added to the databases of IDSs as soon as possible. However, the BM-like and AC-based algorithms have to suspend for a long time for pattern update. For the BM-based algorithms, they calculate the valid shift by correlating one pattern with others in $\boldsymbol{P}$ based on bad character and good suffix heuristics. For the AC-based algorithms, they build the fail tables by correlating the substrings of all patterns in $\boldsymbol{P}$. Accordingly, BM-based and AC-based algorithms have to modify many entries of the lookup tables even when only a pattern is added or deleted. The number of the modified entries relates to the length of the updated patterns. Contrarily, the proposed HMA has the advantage of incremental update, with which at most three entries of the tables have to be updated for one changed pattern.

The incremental pattern update mechanism is shown in Figure 8. The pattern insertion is similar to the table construction of HMA. To add a new pattern p_i into the pattern set $\boldsymbol{P}$, HMA has to modify at most one field: H^2.next, $H^1.pid$, or $H^1.fid$; and add one entry. The new p_i is scanned from left to right. (1) If there is a character $p_i[j]$ such that $p_i[j] \in \boldsymbol{F}$ (i.e., $H^1(p_i[j]).fid$ is valid), and cluster $P_{p_i[j],p_i[j+1]}$ is empty (i.e., $H^2(p_i[j], p_i[j+1]).pid$ is NULL), then p_i is added to the entry $H^2(p_i[j], p_i[j+1])$. If there is no empty cluster for p_i, then a random number r is chosen, such that $p_i[r] \in \boldsymbol{F}$. Thereupon, p_i is added

to the cluster $P_{p_i[r],p_i[r+1]}$ and saved in a free entry of H^2, say $H^2(a'', b'')$. Then the *next* field of the last pattern of $P_{p_i[r],p_i[r+1]}$, say $H^2(a', b').next$, is modified from NULL to $H^2(a'', b'')$. (2) If there is no character of p_i belongs to $\boldsymbol{F}$, then a random character of p_i, say $p_i[r]$, is chosen as a new frequent common-code and added to $\boldsymbol{F}$. A new ID of frequent common-code is assigned and saved in $H^1(p_i[r]).fid$. If p_i is a single-symbol pattern, i.e., $|p_i| = 1$, then modify $H^1(p_i[r]).pid$ to i; otherwise, p_i is saved in the entry $H^2(p_i[r], p_i[r+1])$. The size of the cluster is balanced by randomly choosing a cluster if the information of all cluster size (the matrix N) is not kept in the system. If the matrix N is kept in the system, the pattern insertion procedure is the same as the table construction.

To delete a pattern p_i, only one entry has to be cleared and at most two of $H^2.next$, $H^1.fid$, and $H^1.pid$ have to be modified. The first process is to find out the pattern p_i in H^2 by using a matching process similar to the on-line matching. If $|p_i| = 1$, then change $H^1(p_i[j]).pid$ to NULL. If $|p_i| \neq 1$ and p_i is in the cluster $P_{p_i[j],p_i[j+1]}$, then (1) when p_i is not the only pattern in the cluster $P_{p_i[j],p_i[j+1]}$, link p_i's previous entry and its next one in H^2 before clearing the entry of p_i ($H^2.next$ is modified); (2) when p_i is the only one in the cluster $P_{p_i[j],p_i[j+1]}$, check whether any pattern exists in the subset $P_{p_i[j]\in F}$ after clearing the entry of p_i. If no, it means the frequent common-code $p_i[j] = f$ is not used any more. Then, the code f can be removed from $\boldsymbol{F}$ and the field $H^1(f).fid$ is set to NULL.

Obviously, the number of modified fields due to HMA's pattern updates are constant (at most three), and thus the updating time is deterministic and negligible. Therefore, HMA provides more reliable inspection engines for real-time network equipment.

3.5 An Example: Network Intrusion Detection System

HMA can be used in many novel network applications to inspect packets, such as NIDSs, anti-virus appliances, and layer-7 switches, which search for a set of patterns in packets. The only difference between these applications is the pattern format. Most patterns of virus codes are binary codes; while most patterns of layer-7 switches are formed by English letters. The patterns in NIDSs are written in mixed plain text and hex formatted bytecodes. In this section, we illustrate an application of HMA with NIDSs.

Two complementary techniques are used to cope with the intrusion detection problem: anomaly detection and misuse detection [25]. Anomaly detection techniques attempt to model normal behavior; while misuse detection techniques attempt to model abnormal behavior. Anomaly-based IDSs are deployed based on machine learning, data mining or statistical algorithms, which are more sensitive to new attacks than signature-based IDSs. However, anomaly-based IDSs usually trigger up to 99% false positive alarms, and their complex normal models result in poor performance. Several researchers have proposed new schemes to improve the anomaly detection [25], [32], [37]. Misuse detection is assumed to be more accurate and efficient than anomaly detection, and therefore signature-based IDSs are commonly used today. Some effort has focused on automatic signature generation to improve the robustness of the signatures [32]. An example of signature-based NIDSs using HMA is shown in this section.

As network processors have been widely used to develop novel network equipment [33], a network processor platform is used to illustrate the HMA-base NIDS. A network processor development system usually consists of several on-chip multi-context processing engines, each with a small on-chip cache, one host CPU, external memory,

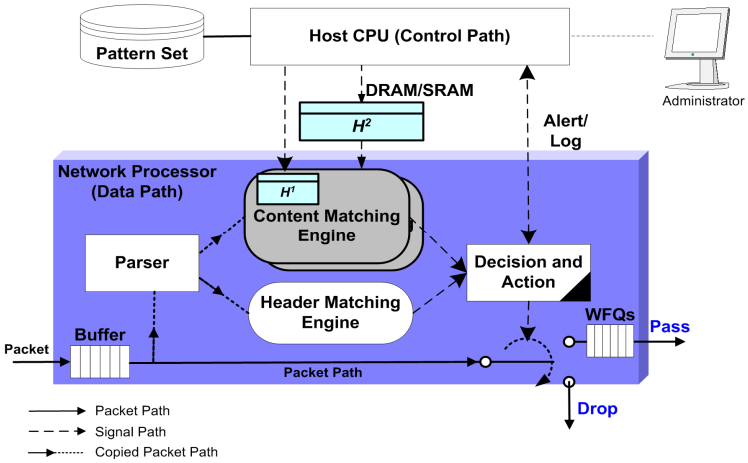


Figure 9. The architecture of a network-processor-based NIDS.

built-in Ethernet MAC modules and queue modules, such as weighted fair queues (WFQs) [35]. A NIDS can be a sniffer for intrusion analyses, or can combine with an embedded queue module (as in Figure 9) to intercept malicious packets. To accelerate the performance of the network equipment, the network-processor-based systems generally divide the tasks into two paths: the *control path* and the *data path*. The host CPU processes the non-real-time tasks in the control path, including the table construction, the pattern set management, the log analyses, and the user interface control. The on-chip microengines handle the real-time tasks in the data path, including the packet parsing, header matching, content matching, decision control and queue management. The content matching engine utilizes the proposed HMA, which is usually the most resource-intensive element; while the header matching engine uses a hardware-supported classification module. The H^2 table of HMA is stored in the external memory and H^1 is in the on-chip cache of the content matching engines.

When a packet comes in, the packet input module makes some standard checks and put it into a packet buffer. Because the intrusion detection rules have both the header and

payload patterns, one engine for header matching and two engines for content matching are employed to accelerate the processes. The header matching engine checks the packet headers; and in the meantime the content matching engines search for the patterns in the packet payloads. The matching results both forward to a decision procedure, which decides to drop the packet, to generate alerts/logs, or to send the packet to an assigned WFQ. A management procedure in the host CPU analyzes the logs and feedbacks to the decision procedure that controls the bandwidth of the suspicious flows by using the WFQs. Thereby, the HMA-based NIDS can efficaciously avoid attacks. Furthermore, the HMA-based NIDS can react fast to the new attacks with low false negative alarms when it is cooperating with proactive bandwidth management and analysis feedbacks.

3.6 Performance Analyses

The performance analyses of HMA are presented in this section. The *input* and *patterns* in worst, best and average formatted type are used to analyze the worst, best, and average performance of HMA respectively.

3.6.1 Average Case

Since the pattern database is usually predefined and static, assume the given patterns are uniformly distributed. First of all, the cluster organization is modeled. Assume that the occurrence probability of any alphabet in a pattern p_i is an uniform distribution:

$$\Pr\{ p_i[j] = a \mid a \in \Lambda \} = \frac{1}{|\Lambda|}. \qquad (3\text{-}1)$$

Based on the table construction procedure of HMA, the probability that any two-character substring, say '*ab*', is a pivot pair, and also exits in a pattern p_i is denoted C_{pivots}, where

$$C_{pivots} = \Pr\{\text{`}ab\text{'} \subset p_i \text{ and } a \in F,\ b \in \Lambda \mid \text{`}ab\text{'} \in \Lambda^2\} = \left(\bar{p} - 1\right)\frac{|F|}{|\Lambda|^3}, \quad (3\text{-}2)$$

and $\bar{p}$ is the average pattern size. In other words, C_{pivots} is the probability that a cluster $\boldsymbol{P}_{a,b}$ is one of the available clusters for a pattern p_i. Thereby, the average number of pivot pairs in a pattern, denoted N_{pivots}, can be derived by the following equation:

$$N_{pivots} = \sum_{k=1}^{\bar{p}-1} k \binom{|\Lambda||F|}{k} C_{pivots}{}^{k} (1 - C_{pivots})^{|\Lambda||F|-k}, \quad (3\text{-}3)$$

where $\binom{m}{n} \equiv \frac{m!}{n!(m-n)!}$. According to CBS, when a cluster, say $\boldsymbol{P}_{a,b}$, is one of the available clusters of p_i, and also is the smallest one, then p_i is grouped to $\boldsymbol{P}_{a,b}$. Since the patterns are uniformly distributed and CBS classifies the patterns as balance as possible, the probability that an available cluster is the minimum one is $1/N_{pivots}$. Consequently, the probability that a pattern is grouped to a designate cluster is

$$C_{p \to cluster} = \frac{C_{pivots}}{N_{pivots}}. \quad (3\text{-}4)$$

As k patterns are grouped into the same cluster, the cluster size is k. Thus, the probability that the size of a cluster is k for a given pattern set $\boldsymbol{P}$ is

$$C_k = \binom{|P|}{k} C_{p \to cluster}{}^{k} (1 - C_{p \to cluster})^{|P|-k}. \quad (3\text{-}5)$$

Let $N_{cluster}$ represent the average cluster size, which can be derived by the following equation:

$$N_{cluster} = \sum_{k=1}^{|P|} k C_k. \quad (3\text{-}6)$$

Note that the structures of H^1 and H^2 depend on the predefined $\boldsymbol{P}$, and additionally are controllable and balanced by HMA. Therefore, generally the tables will not be

constructed badly, whether the input string is the best, average or worst-case string for the algorithm. In fact, $N_{cluster}$ can be reduced by increasing the size of H^2, using B characters for the second pivot instead of one character. Then $C_{pivots\text{-}B}$ is smaller than C_{pivots}, which is

$$C_{pivots\text{-}B} = \left(\bar{p} - 1\right)\frac{|F|}{|\Lambda|^{2+B}}. \tag{3-7}$$

Reducing $N_{cluster}$ by using the smaller $C_{pivots\text{-}B}$ can improve the matching performance, but however will increase the required memory space. This is a trade-off between the matching performance and the memory cost.

In the average case, assume that an input string T is drawn randomly from the alphabet set Λ. As defined previously, H^1 is a direct indexing table for each character. The *fid* field of the entry $H^1(a)$ is assigned a valid ID, say i, for all $a = f_i \in \boldsymbol{F}$. Thus, the probability that an entry of H^1 has a valid *fid* is:

$$\Pr\{H^1.fid \neq \text{NULL}\} = \frac{|F|}{|\Lambda|}, \tag{3-8}$$

where $|\boldsymbol{F}|$ is the number of frequent common-codes. In the matching process, if $H^1.fid \neq$ NULL, the next step is to check the *pid* field of the indexed H^2 in the external memory, and proceed to the second-tier matching. Accordingly, the probability that the on-line matching goes to the second-tier matching for any input character $T[t] \in \Lambda$ is defined as C_{tier2}^{AVG}, and

$$C_{tier2}^{AVG} = \sum_{0}^{|\Lambda|-1} \frac{1}{|\Lambda|} \times \frac{|F|}{|\Lambda|} = \frac{|F|}{|\Lambda|}. \tag{3-9}$$

The first step of the second-tier matching is to fetch the entry and check the *pid* field, and thus one external memory access is required. If there is more than one pattern in the cluster, additional external memory access will be needed to fetch those patterns. We

assume every pattern can be loaded into microprocessors within one external memory access. Let N_{RAM}^{AVG} represent the average number of external memory accesses per one input character, and it is

$$N_{RAM}^{AVG} = C_{tier2}^{AVG} \times \left(1 + \sum_{k=2}^{|P|} (k-1) \binom{|P|}{k} C_{p \to cluster}{}^{k} (1 - C_{p \to cluster})^{|P|-k} \right)$$

$$= \frac{|F|}{|\Lambda|} \left(N_{cluster} + (1 - C_{p \to cluster})^{|P|} \right). \quad (3\text{-}10)$$

If the indexed $H^2(T[t], T[t+1]).pid$ is valid, it means T may have a pattern $p_i \in \boldsymbol{P}_{T[t],T[t+1]}$. Then T has to be compared with the patterns in the cluster $\boldsymbol{P}_{T[t],T[t+1]}$, where the average number of patterns in $\boldsymbol{P}_{T[t],T[t+1]}$ is $N_{cluster}$. Let $N_{fetch}^{AVG\_T}$ be the average number of patterns fetching from an external memory for a given average-case input T. $N_{fetch}^{AVG\_T}$ can be derived from the previous equations:

$$N_{fetch}^{AVG\_T} = (|T| - 1) \times C_{tier2}^{AVG} \times N_{cluster}. \quad (3\text{-}11)$$

Thereby, the number of XOR instructions used in string comparisons between T and the patterns for a given average-case input T, denoted $N_{XOR}^{AVG\_T}$, can be obtained by

$$N_{XOR}^{AVG\_T} = N_{fetch}^{AVG\_T} \left\lceil \frac{\overline{p}}{\omega} \right\rceil, \quad (3\text{-}12)$$

where ω is the computer wordsize. In the average case of HMA, let N_{XOR}^{AVG} represent the average number of XOR comparisons between T and the patterns in the cluster for one input character, which can be derived by

$$N_{XOR}^{AVG} = \frac{N_{XOR}^{AVG-T}}{|T|} < \frac{|F|}{|\Lambda|} \left\lceil \frac{\overline{p}}{\omega} \right\rceil N_{cluster}. \quad (3\text{-}13)$$

3.6.2 Worst Case

If a given string T is formed badly that has to do the exact string comparisons the most times, the performance of HMA for the bad-formed T is the worst case. Assume the largest cluster size is L_c. When every character of T ($T[t]$) belongs to $\boldsymbol{F}$, and every corresponding indexed cluster is the largest ($|\boldsymbol{P}_{T[t],T[t+1]}| = L_c$), this is the worst scenario of HMA. As every character $T[t] \in \boldsymbol{F}$, the probability to fetch the table H^2 for the worst case is one. Thus, the number of external memory accesses per character in the worst case is

$$N_{RAM}^{WST} = \frac{(|T|-1)\times L_c}{|T|} < L_c. \tag{3-14}$$

Assume the largest pattern size in $\boldsymbol{P}$ is L_p. When every input character points to the largest cluster, in which every pattern has the longest size, the worst case requires the largest number of comparisons. Hence, the number of XOR character comparisons for one input character is

$$N_{XOR}^{WST} < L_c \left\lceil \frac{L_p}{\omega} \right\rceil. \tag{3-15}$$

Obviously, the worst-case performance depends on L_c. To derive L_c, assume there is a largest cluster, say $\boldsymbol{P}_{x,y}$. Since $\boldsymbol{P}_{x,y}$ is the largest cluster, assume that the cluster size is always larger than one, and initially the probability that its cluster size increases from 0 to 1 is one. That is

$$\Pr\{|P_{x,y}| = 0 \rightarrow 1\} = 1. \tag{3-16}$$

In the worst case, the patterns are assumed formed badly and have a bias on the pivot pair (x, y). Since $\boldsymbol{P}_{x,y}$ is the largest cluster, based on CBS, a given pattern p will not be

clustered into $\boldsymbol{P}_{x,y}$, unless all available pivot pairs of p are not in the set $\boldsymbol{F}\times\Lambda$ except (x, y). Therefore, the probability that $|\boldsymbol{P}_{x,y}|$ increases from i to i+1 is

$$\Pr\{\left|P_{x,y}\right| = i \rightarrow i+1\} = \left(\frac{\left|\Lambda\right|^2 - \left|F\right|\times\left|\Lambda\right|+1}{\left|\Lambda\right|^2}\right)^{|p|-2}, \quad (3\text{-}17)$$

where $|p|$ is the given pattern size. As in the worst-case scenario, every pattern has the longest size L_p, the equation is rewritten

$$\Pr\{\left|P_{x,y}\right| = i \rightarrow i+1\} = \left(\frac{\left|\Lambda\right|^2 - \left|F\right|\times\left|\Lambda\right|+1}{\left|\Lambda\right|^2}\right)^{|L_p|-2}. \quad (3\text{-}18)$$

Thereby, the probability that the cluster size of $\boldsymbol{P}_{x,y}$ is L_c is derived

$$\Pr\{\left|P_{x,y}\right| = L_c\} = \left(\frac{\left|\Lambda\right|^2 - \left|F\right|\times\left|\Lambda\right|+1}{\left|\Lambda\right|^2}\right)^{(|L_p|-2)(L_c-1)}. \quad (3\text{-}19)$$

When $|\boldsymbol{P}|$ is 1200 with $|\boldsymbol{F}|$ = 77, $|\Lambda|$ = 256 and L_p = 128, the probability that L_c = 4 is about 7×10^{-79}, which is very small. When replacing L_p with the average pattern size of Snort ($|p|$ = 11), then the probability that L_c = 4 is about 3.6×10^{-6}, and is still very small. Thus N_{RAM}^{WST} and N_{XOR}^{WST} are very small. Consequently, we can say that $L_c << |\boldsymbol{P}|$, and the worst-case performance of HMA is moderate and acceptable.

3.6.3 Best Case

If a given string T is a good string, where every character $T[t] \notin \boldsymbol{F}$ for all t, $1 \le t < |T|$, this good string will gain the best-case performance of HMA. In this case, no external memory access and no pattern comparison are necessary. The good string can be processed quickly, and only one embedded memory lookup (checking H^1 to see whether $T[t] \notin \boldsymbol{F}$ or not) is needed per input character.

Table 3. The pattern size distribution of Snort.

Pattern Length	=1	≤ 4	≤ 8	≤ 12	≤ 16	>16
Ratio	0.028	0.245	0.482	0.653	0.813	0.187

3.7 Results

This section shows the simulation results of HMA, compared with the state-of-the-art multi-pattern matching algorithms: BMH [21], WM-PH [27], and bitmap compressed AC (AC-C) [34], which have been used in the IDSs. BMH and AC-C have been deployed in a famous open-source NIDS – Snort, and WM-PH has been proposed for a network-processor-based NIDS. In the simulations, a network processor development system is used as a simulation platform [33]. HMA, BMH, WM-PH and AC-C are emulated by assembly-like microprograms respectively, and the number of instructions and that of memory accesses are calculated. One microprocessor is used in the simulations to simplify the evaluation, though a network processor may have several microengines.

Snort is the most famous open-source NIDS today and the patterns (rules) used in Snort are provided and tested by the Sourcefire Vulnerability Research Team (VRT), which is the largest group dedicated to advances in network security industry [1]. The free and real pattern set released by VRT is used in the simulations (the statistics of the pattern set are listed in Table 3), although the pattern set can be any self-defined or commercial pattern set. The number of *distinct* patterns used in the simulations is 200–1200, where each pattern is about 11.2 bytes on average. Since the patterns of Snort are written in mixed plain text and hex formatted bytecodes, the alphabet size ($|\Lambda|$) is 256 in the simulations.

Table 4. The measurements.

Notation	Meaning
N_I	The average number of RISC instructions per input character (including comparisons and calculations)
N_L	The average number of local memory accesses (including reading data from cache to registers)
N_E	The average number of external memory accesses for loading the packet, querying the entries of tables in the external memory, and fetching the patterns
w_I	The time of one instruction or one local memory/register access
w_E	The time of one external memory access
ψ_I	The average computation cycles: $\psi_I = N_I \times w_I$
ψ_M	The average memory latency: $\psi_M = N_E \times w_E + N_L \times w_I$
Ψ	Total average matching time: $\Psi = \psi_I + \psi_M$

3.7.1 Measurements

Table 4 shows the measurements used in the simulations. Notably, we assume that the skip table of BMH was small enough to be loaded into the cache memory in the simulations, and thus only one external memory access was counted for each pattern during the matching process of BMH. We also assume AC-C needed one external memory access per input code, although it generally requires two memory references (one for reading the next pointer and one for traversing failure pointers or reading the patterns).

3.7.2 Input Traffic Models

3.7.2.1 Models I and II

In the Models I and II, the malicious packets are generated by randomly choosing patterns from ***P*** and spreading over the packet payloads. Attack load λ is defined as the expected number of malicious patterns in one packet. For example, if λ is 0.5, it means every two packets have one harmful pattern on average.

The characters in a payload besides the patterns are called background characters. Two forms of background characters are respectively used in the Model I and Model II. In the Model I, the payloads of *random background* are formed by characters randomly

Table 5. The traffic models.

	Packet Format		Packet Length	Number of Packets
	Background	**Number of Patterns**		
Model I	Random	λ	640 bytes	10 million
Model II	Pure	λ	640 bytes	10 million
Model III	All permutations		4 bytes	2^{32}
Model VI	Real Traces from Defcon			

drawn from Λ to imitate the normal packet contents. However, the random background may unconsciously contain some patterns of ***P***. To evaluate the impact of λ on the performance of algorithms, *pure background* is used in the Model II. The pure background is formed by the characters that never appeared in ***P***.

3.7.2.2 Model III

Since different multi-pattern matching algorithms have different string forms that cause their best-case or worst-case performance, all permutations of four-character input strings (2^{32} strings) are used in the Model III to examine the extreme performance of every algorithm. We choose the length of four characters because 24.5% of Snort's patterns are less than or equal to four characters (see Table 3), and the test pool of 2^{32} input strings is large enough for simulations. Because it is very difficult to obtain the best-case and worst-case traces for every algorithm, it is quite feasible by using this model to evaluate the extreme cases of every algorithm.

3.7.2.3 Model VI

To evaluate the performance of algorithms in an intense attack, a real trace from the Capture-the-Flag contest held at Defcon9 was adopted as the input traffic in the Model VI. The Defcon Capture-the-Flag contest is the largest security hacking game. In this contest, competitors try to break into the servers of others while protecting their own servers,

Table 6. The simulation parameters.

Items	**Value**
Time for one RISC instructions or one local memory access (w_I)	1 cycle
Latency for each external memory access (w_E)	100 or 250 cycles
Number of patterns in $\boldsymbol{P}$ ($\|\boldsymbol{P}\|$)	200, 400, ..,1200
Range of pattern length	1–122 bytes

Table 7. The extra memory requirements.

	HMA	**WM-PH**	**BMH**	**AC-C**
Cache memory space (M_I)	$O(\|\Lambda\|)$	$O(1)$	$O(\|\Lambda\|)$	$O(1)$
External memory space (M_E)*	$O(\|\boldsymbol{F}\|\times\|\Lambda\|+\|\boldsymbol{P}\|)$	$O(\|\Lambda\|^3+\|\boldsymbol{P}\|)$	$O(\|\boldsymbol{P}\|\times\|\Lambda\|+\|\boldsymbol{P}\|)$	$O(S+\|\boldsymbol{P}\|)$

*$|\boldsymbol{F}| < |\Lambda| << |\boldsymbol{P}| < S$

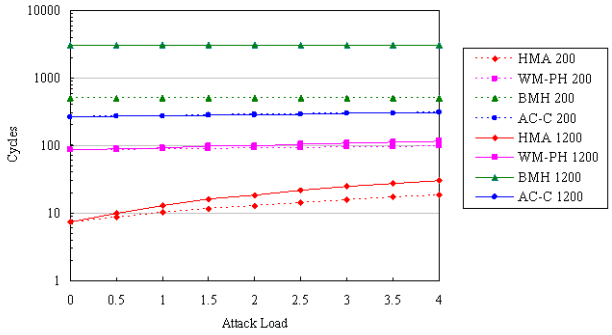
where each server hides several security holes [14]. The summary of the traffic models are shown in the Table 5 and the simulation parameters are listed in Table 6.

3.7.3 Memory Requirements

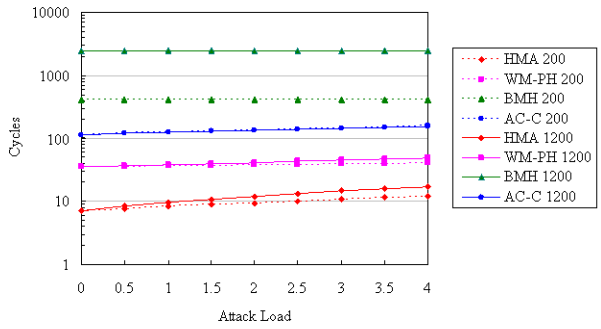
The lookup information and patterns are generally saved in the memory using a tabular structure for fast lookup and matching. Therefore, the memory requirements are shown in terms of the number of entries. Since the H^1 of HMA is a direct lookup table, the cache memory space (M_I) of HMA is $|\Lambda|$ entries. Based on the proposed schemes, FCS and CBS, the number of entries in H^2 is the total number of possible clusters. As all possible pivot pairs are in the space $\Lambda\times\boldsymbol{F}$, the maximum size of H^2 is $|\boldsymbol{F}|\times|\Lambda|$ entries along with a shared space of no larger than $|\boldsymbol{P}|$ entries for collisions. Thereby, the external memory space (M_E) of HMA is $O(|\boldsymbol{F}|\times|\Lambda|+|\boldsymbol{P}|)$. The lookup table of WM-PH is based on a direct prefix hash table with prefix length of D, where D = 3 in the simulations. Accordingly, M_E of WM-PH is $|\Lambda|^D+|\boldsymbol{P}|$ entries for the index table and pattern contents. In the BMH, every pattern has its own skip table of $|\Lambda|$ entries, so that M_E of BMH is $O(|\boldsymbol{P}|\times|\Lambda|+|\boldsymbol{P}|)$. Since each skip table of BMH is small enough to be loaded to the local

Table 8. The number of frequent common-codes versus the pattern set size.

$\|\boldsymbol{P}\|$	100	200	300	400	500	600	700	800	900	1000	1100	1200
$\|\boldsymbol{F}\|$	11	28	32	37	45	49	52	58	66	74	75	77



(a)w_E = 250



(b) w_E = 100

Figure 10. The average matching time (Ψ) versus the attack load (λ) for HMA, WM-PH, BMH and AC-C with different pattern set sizes ($|\boldsymbol{P}|$=200 and 1200), using Model II, and (a) w_E = 250, (b) w_E = 100.

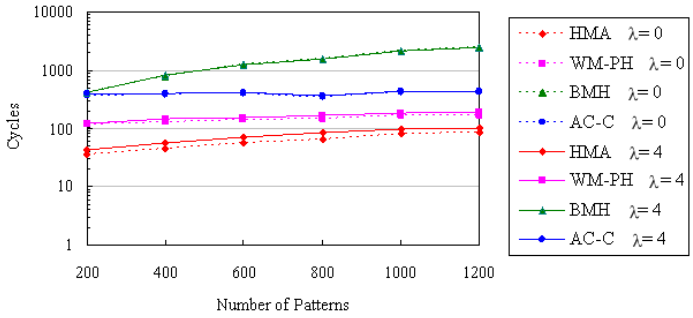
memory, we allocate a cache memory space for BMH in the simulations for fair comparisons. WM-PH and AC-C also need cache memory for loading one skip value or one state during matching process. The required memory used in HMA, WM-PH, BMH and AC-C is summarized in Table 7 including lookup tables and pattern contents.

Table 8 lists the relations between the pattern set size $|\boldsymbol{P}|$ and the number of frequent common-codes $|\boldsymbol{F}|$ in the HMA. It shows that the growing rate of $|\boldsymbol{F}|$ is much slower than that of $|\boldsymbol{P}|$. In the simulations with $|\boldsymbol{P}|$ = 1200 for example, the maximum M_E of HMA is

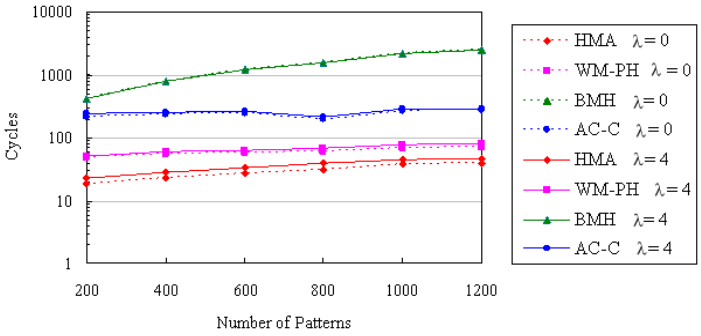
20192 entries (326.75KB external memory when the size of an entry is 16 bytes, including pattern contents and formatted information); WM-PH needs more than 16M entries (16MB for shift values, excluding pattern contents); BMH needs more than 300K entries (300KB for shift values, excluding pattern contents); and AC-C requires 10731 states (461KB when the size of a node is 44 bytes, excluding $|\boldsymbol{P}|$ entries for pattern IDs). Consequently, the required memory space of HMA is very small.

3.7.4 Results and Discussions

Figure 10 shows the attack load λ on the average matching time Ψ using Model II with different attack loads $|\boldsymbol{P}|$ = 200 and $|\boldsymbol{P}|$ = 1200 respectively. Since the ratio of instruction cost (w_I) and external memory cost (w_E) are varied in different deployed systems, Figure 10 (a) and (b) also show the performance of each algorithm with different weights w_E = 100 and w_E = 250 respectively. Simulation results reveal that HMA outperforms WM-PH, AC-C and BMH even when $|\boldsymbol{P}|$ and λ increase. The curves of HMA and WM-PH are slightly increased with λ rising because HMA and WM-PH need more external memory accesses and string comparisons when more malicious patterns exist in a packet. With the larger pattern set, the matching time increases a little faster in both HMA and WM-PH. This is because the probability that the input strings hit the lookup tables (H^1 and H^2 for HMA and the prefix table for WM-PH) increases. HMA has higher growth rate than WM-PH because the table size of HMA is much smaller than that of WM-PH. WM-PH gains performance by having a large direct index table. The curves of BMH seem flat with λ rising, since the tiny increment of BMH is caused by the increasing number of comparisons with relatively low w_I when compared to w_E. Because AC-C needs one external memory access in addition to time intensive *popsum* to count the



(a) $w_E = 250$

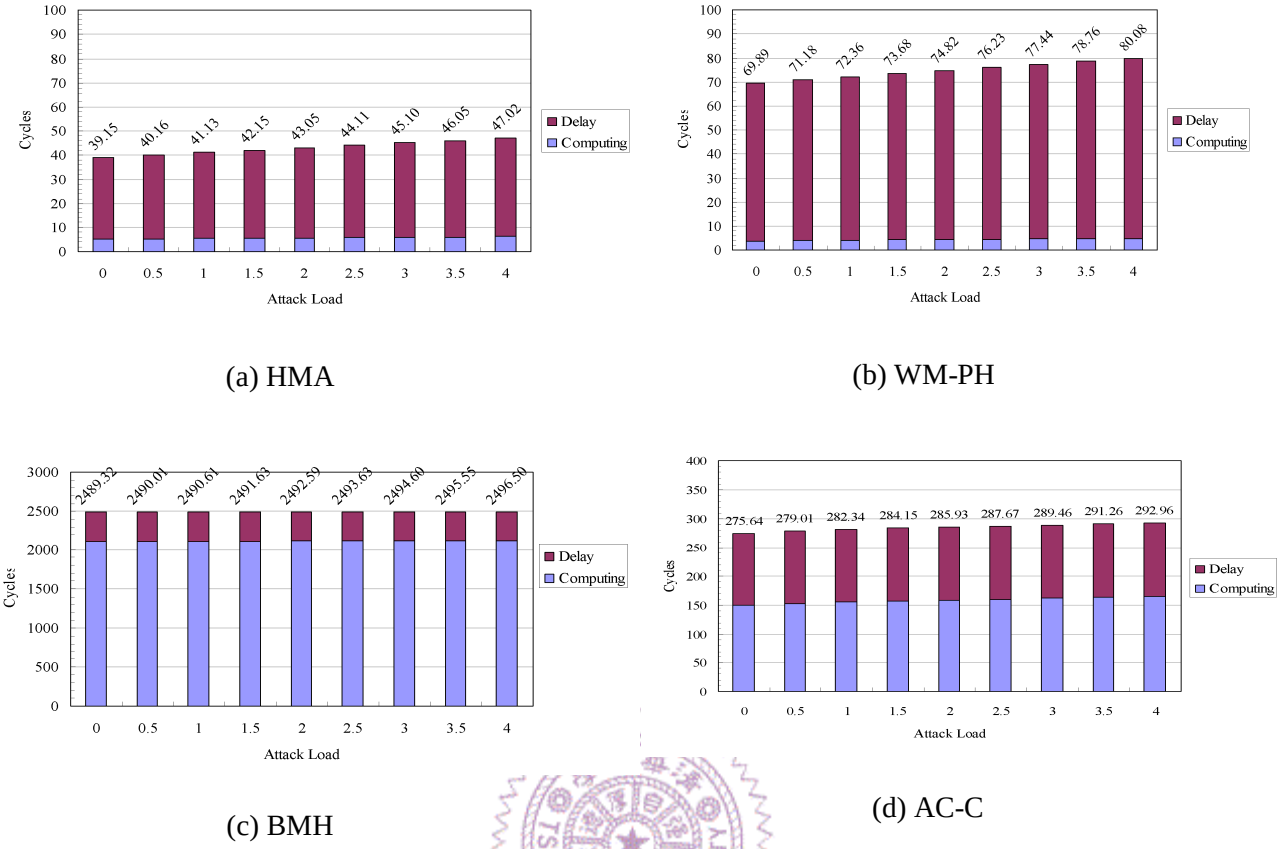


(b) $w_E = 100$

Figure 11. The average matching cost (Ψ) versus pattern set size ($|\boldsymbol{P}|$) for HMA, WM-PH, BMH and AC-C with different attack loads (λ), using Model I, and (a) w_E = 250, (b) w_E = 100.

next state for every character, its Ψ is high. In the case of w_E = 250 and $|\boldsymbol{P}|$ = 200 ($|\boldsymbol{P}|$ = 1200), the matching time of HMA is about 26.5–68 (99.7–409.5) times less than that of BMH, 4.2–10.6 (2.8–10.6) times less than that of WM-PH, and 15.5–34.8 (9.1–34.7) times less than that of AC-C under different attack loads. In Figure 10, when λ is low, HMA significantly outperforms WM-PH, BMH, and AC-C. Consequently, HMA is very suitable for IDSs in a general network environment, because most packets are innocent ($\lambda \approx 0$).

The simulation results shown in Figure 11– Figure 13 use Model I as input traffic. Figure 11 compares Ψ of HMA, WM-PH, AC-C and BMH with different attack loads



(a) HMA

(b) WM-PH

(c) BMH

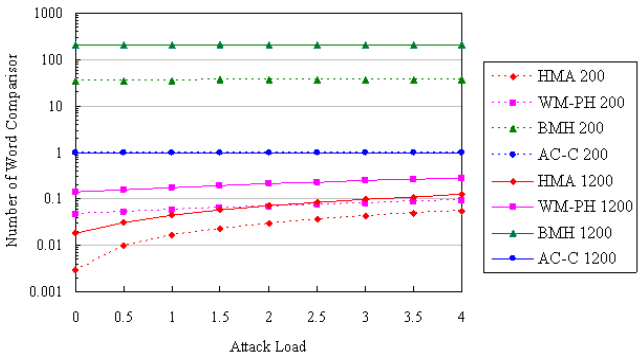
(d) AC-C

Figure 12. ψ_I and ψ_M versus attack load (λ), where $|\boldsymbol{P}|$=1200 and w_E = 100, using Model I. The labeled value above each bar is Ψ. (a) HMA, (b) WM-PH, (c) BMH and (d) AC-C.

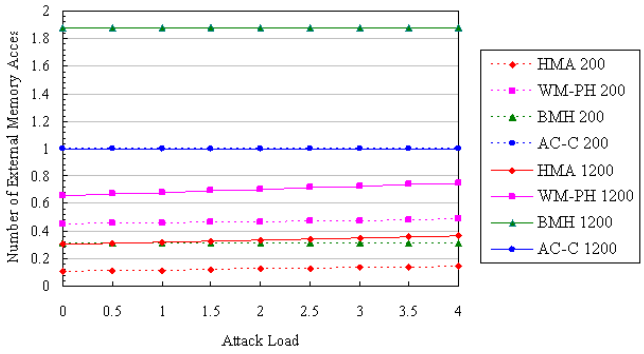
$\lambda = 0$ and $\lambda = 4$ respectively. It also shows the impact of $|\boldsymbol{P}|$ on Ψ. Simulation results reveal that HMA outperforms WM-PH, BMH and AC-C even when $|\boldsymbol{P}|$ and λ are increasing. For both $\lambda = 0$ and $\lambda = 4$, the matching costs of HMA and WM-PH both rise with $|\boldsymbol{P}|$. This is because while $|\boldsymbol{P}|$ rises, the number of patterns in $\boldsymbol{P}$ that have similar substrings also rises. This leads to the increasing number of marked entries that request for comparisons in HMA and WM-PH. Hence, HMA and WM-PH require more string comparisons and memory accesses with increasing $|\boldsymbol{P}|$. HMA has slightly higher growth rate than WM-PH, because the table size of HMA (H^1 and H^2) is about 830 times smaller than that of WM-PH. The increasing $|\boldsymbol{P}|$ makes the matching time of BMH rise steeply,

because the BMH is originally a single-pattern matching algorithm that simply executes iteratively for every pattern. In the case of w_E = 250 and $\lambda = 0$ ($\lambda = 4$), the matching time of HMA is 14.5–35.8 (11.7–29.8) times less than that of BMH, 2–3.3 (1.9–2.8) times less than that of WM-PH, and 11.9–22.2 (9.5–24.3) times less than that of AC-C under different pattern set sizes. Figure 11 reveals that HMA is quite stable due to slight increment of its Ψ while $|\boldsymbol{P}|$ increases.

The processing time Ψ includes the computation time (ψ_I) and memory access delay (ψ_M). Figure 12 (a)–(d) illustrate the proportion of ψ_I to Ψ and ψ_M to Ψ respectively for all approaches with $|\boldsymbol{P}|$ = 1200 and various λ. In these figures, the upper and lower part of the bar are represented as ψ_M and ψ_I respectively. The results show that HMA's ψ_I is close to WM-PH's, but HMA's ψ_M is much less than others. Therefore, the hierarchical matching strategy of HMA is highly effective in reducing the memory latency, only tiny overhead of the computation time is needed. The proportion of ψ_M to Ψ of BMH seems smaller than others. The reason is that the whole skip table of a pattern is idealistically assumed to be loaded within one external memory access, and kept in the cache during the matching process. Because AC-C compresses the size of each node, it requires more time to calculate the next state pointer. Thereby, AC-C does not have the smallest ψ_I. Simulation results show that the ψ_I does not significantly rise with λ in any of the experiments, because each algorithm has already tried to reduce the computation load (ψ_I). However, ψ_M dominates the overall matching cost. This reveals that the number of external memory accesses is the bottleneck of almost all algorithms. The result also reflects our opinion mentioned previously that the essential issue in



(a) Comparison



(b) Memory access

Figure 13. The average number of XOR comparisons and that of external memory access versus the attack load (λ) for HMA, WM-PH and BMH with different pattern set sizes (|***P***|), using Model I: (a) Comparison, (b) Memory access.

designing a high-speed detection engine is to reduce the number of required external memory accesses.

Since different systems have different implementation overheads, Figure 13 extract two basic measurements from overall costs to compare the algorithms themselves. The results in Figure 13 (a) plot the average word comparisons (N_{XOR}^{AVG}) versus λ for every approach, with |***P***| = 200 and 1200 respectively. Figure 13 (a) shows that N_{XOR}^{AVG} of HMA grows moderately with λ and |***P***|, and is more efficient than others, especially when λ is low. Figure 13 (b) shows the average number of external memory access (N_{RAM}^{AVG}). It

Table 9. Analysis and simulation results of HMA with Model I and $\lambda = 0$.

|$|P|$	N_{RAM}^{AVG}		N_{XOR}^{AVG}	
	Analysis	**Simulation**	**Analysis**	**Simulation**
200	0.109417	0.109965	0.0122	0.00282
400	0.144658	0.146164	0.0244	0.005762
600	0.191621	0.193972	0.0366	0.008785
800	0.226885	0.229967	0.0488	0.011705
1000	0.289458	0.293	0.0610	0.014587
1200	0.301327	0.305335	0.0732	0.017624

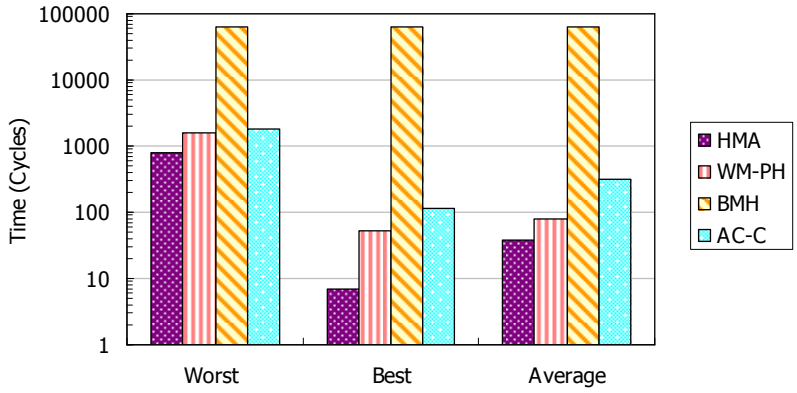
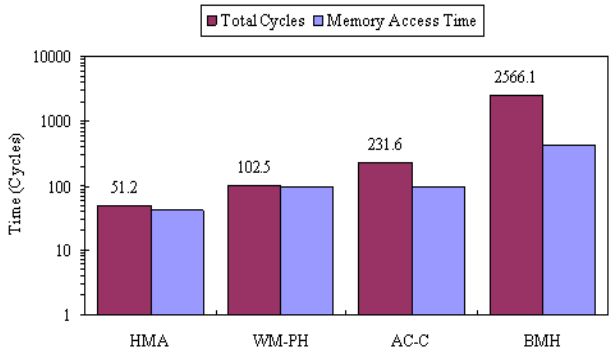
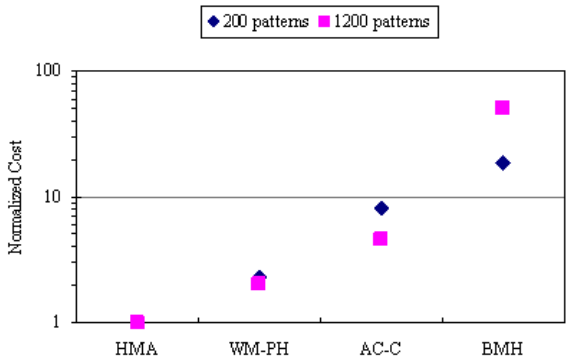


Figure 14. The pure costs of the matching algorithms in the worst-case and best-case situations using Model III .

demonstrates that HMA effectively reduces the number of required external memory accesses. N_{RAM}^{AVG} of HMA is only 0.109–0.369 when $|\boldsymbol{P}| = 200–1200$ and $\lambda = 0–4$. In other words, HMA can successfully filter out about 90%–63% payloads without any external memory accesses and string comparisons.Table 9 lists both the analysis and simulation results of N_{XOR}^{AVG} and N_{RAM}^{AVG} respectively, using Model I and $\lambda = 0$. The simulation and analysis results of N_{RAM}^{AVG} are very close. The simulation results of N_{XOR}^{AVG} are a little smaller than the analysis results. The reason is that the comparison between the input string and patterns will stop in the



(a) Processing Time



(b) Normalized Costs

Figure 15. The processing time and the norrmalized costs using Model VI with w_E = 100: (a) Ψ and ψ_M where $|\boldsymbol{P}|$ = 1200 (b) The matching costs normalized to HMA where $|\boldsymbol{P}|$ = 200 and 1200.

simulations if there is one unmatched word; while we assume that the whole string has to be compared in the analysis.

Figure 14 plots the best-case, the worst-case and the average performance of HMA, WM-PH, BMH and AC-C, using Model III with w_E = 100. The matching time shown in Figure 14 excludes the cost for loading packets from input modules into the processor, because every algorithm has the same cost. Recall that different algorithms may have different extreme scenarios. This simulation uses Model III and records the extreme and

average results for each algorithm respectively. Figure 14 shows that HMA outperforms WM-PH and BMH in all cases. In the best case, HMA requires only seven instruction cycles to process an input character. In the worst case, the performance of HMA is still better than others. Therefore, HMA significantly improves the best-case and average-case performance and has moderately worst-case performance for the multi-pattern matching, enabling practical implementations.

The simulation results using a real trace (Model VI) are shown in Figure 15, where w_E = 100. Figure 15 (a) draws the overall cost (Ψ) and the memory access time (ψ_M) respectively, where $|\boldsymbol{P}| = 1200$. To compare the performance of the state-of-the-art algorithms, the matching time Ψ of WM-PH, AC-C and BMH are normalized to HMA and shown in Figure 15 (b). Although the Defcon trace (Model VI) contains a lot of malicious packets, Figure 15 shows that HMA performs well and much better than others. It also demonstrates that the memory access time of HMA is much smaller than others (note that figures are in logarithmic scale), which means HMA successfully reduces the number of memory accesses. In other words, the small first-tier filter of HMA can still work well even under heavy attacking loads.

4 THE ENHANCED HIERARCHICAL MULTI-PATTERN MATCHING ALGORITHM (EHMA)

Enhanced Hierarchical Multi-Pattern Matching Algorithm (EHMA) contributes modifications to HMA [38], and introduces the idea of a *sampling window* and a *Safety Shift Strategy* in addition. EHMA is a two-tier and cluster-wise matching algorithm, and can perform fast skippable payload scan. Based on the occurrence frequency of *grams*, this study discovers a small set of signatures from the *patterns* themselves to narrow the searching domain. A Min-Max strategy is used in the EHMA. The hit rate of the first-tier table in the EHMA is minimized, while the spread of patterns in the second-tier table is maximized. Accordingly, EHAM significantly reduces the number of memory accesses and pattern comparisons. EHMA can skip unnecessary payload scans by applying the proposed *Safety Shift Strategy*, which is based on a *frequency-based bad gram heuristic*. The frequency-based bad gram heuristic is a modification of the *bad grouped character heuristic* of Wu-Manber algorithm (WM) [36]. Therefore, EHMA has the advantages of both HMA and WM.

4.1 The Basic Idea of EHMA

Based on a hierarchical and cluster-wise architecture, EHMA comprises two small index tables, namely the *first-tier table* (H^1) and the *second-tier table* (H^2). These two tables act as filters to avoid unnecessary external memory accesses and pattern comparisons, and thereby pass the innocuous packets quickly in the on-line matching process. The second-tier procedure (*Tier-2 Matching*) activates only after the first-tier

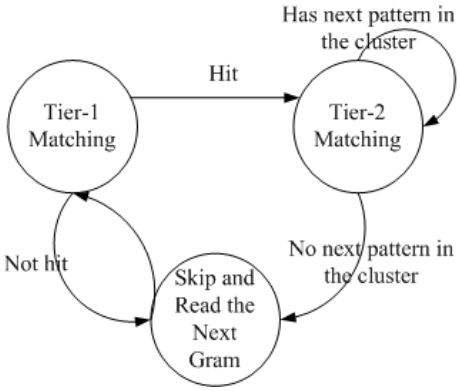


Figure 16. A simple state machine of the *EHMA* matching process.

procedure (*Tier-1 Matching*) gets a match. Using H^2, which indicates a small subset of patterns that are similar to the input packet, EHMA compares only a few *selected* patterns of ***P*** with the *suspected* substrings of the packet, rather than comparing all patterns with all substrings of the packet. Furthermore, a *frequency-based bad gram heuristic* is proposed in the EHMA to determine the safety *shifts* on the input strings during the on-line matching process. In other words, some characters of the input packets can be safely skipped without any process.

Figure 16 displays the simple state machine of the EHMA, which illustrates the hierarchical and skippable matching flows. External memory accesses are needed only in the Tier-2 matching state. Consequently, EHMA significantly enhances the matching performance, and effectively reduces the number of external memory accesses, string comparisons and character scans, by utilizing two small index tables.

This study proposes a *general frequent-common gram searching* algorithm (*GFGS*) and a *cluster balancing strategy* (*CBS*) to lower the size of the tables H^1 and H^2. The following subsections describe the GFGS, CBS and the Safety Shift Strategy in detail.

The hierarchical on-line matching using these two index tables, namely Tier-1 and Tier-2 Matching, are then shown.

4.2 The GFGS Algorithm

In the high-layer intrusion detection, patterns may appear *anywhere* in the packet payload, making the attacking packets difficult to recognize. GFGS assumes that a small set of *signatures* can be found from the patterns themselves, then the suspicious substrings of T may be easier to distinguish from the innocent parts, and the pattern matching is therefore faster. A set of *significant grams* is defined as representatives of a pattern set $\boldsymbol{P}$, given by $\Im \subset \Lambda^{B_1}$, where the size of a gram is B_1 characters. The set $\Im$ is much smaller than Λ^{B_1}. Only when at least a significant gram occurs in the payload, a pattern may exist. That is, when at least one B_1-gram of p_i belonging to $\Im$ occurs in the payload T, the pattern $p_i \in \boldsymbol{P}$ may be found in T. Many innocent B_1-grams of T, that do not belong to $\Im$, can be filtered in the Tier-1 Matching when scanning the packet payload. Obviously, smaller $\Im$ leads to fewer pattern comparisons, and thus faster pattern matching. The GFGS is proposed to find the smallest $\Im$ from $\boldsymbol{P}$.

Define $\boldsymbol{P}_g$ as a subset of $\boldsymbol{P}$, that $\boldsymbol{P}_g = \{p_i \mid p_i$ has the gram $g, \ \forall p_i \in \boldsymbol{P}\}$, where g is called the *common gram* of those patterns in the set $\boldsymbol{P}_g$. Notably, if a common gram appears in the distinct patterns more frequently than other grams, and it is selected as one of the significant grams, then a smaller $\Im$ is found. Based on this inference, the GFGS algorithm is designed to find the *frequent-common gram set* $\boldsymbol{F}$, such that $\boldsymbol{F}$ is the minimum set of significant grams to represent a pattern set $\boldsymbol{P}$. In the GFGS, the common grams are searched only from the *sampling window*, which is defined as the last W

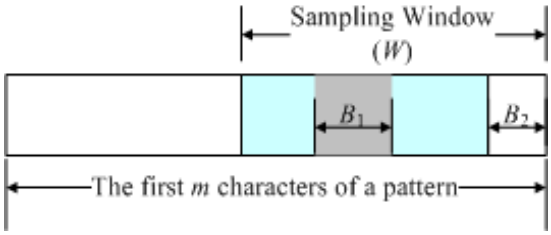


Figure 17. The sampling window.

GFGS Algorithm;
Input: Given a set of patterns ***P***, the parameters: W, B_2, B_1, and m.
Output: A set of frequent-common grams ***F***.

1 Initialize: $\boldsymbol{F} \leftarrow \varnothing$, V and R are set to zero;
2 **For** each pattern p_i of ***P***, $0 \le i < |\boldsymbol{P}|$ **do** /*build a matrix R */
3 Transfer the first W- B_2 bytes of the sampling window of the pattern p_i into B_1-grams, and set the element of a vector V: $v_j \leftarrow 1$ if B_1-gram = j; otherwise $v_j \leftarrow 0$;
4 Read V. For each $v_j = 1$, set the elements of matrix R: $r_{jk} \leftarrow r_{jk} + v_k$, $\forall\ k, 0 \le k < |\Lambda|^{B1}$;
5 **While** ($r_{ii} \neq 0$, $\forall\ 0 \le i < |\Lambda|^{B1}$) **do**
6 Find a frequent-common gram g_f, where $r_{ff} = \max\{r_{ii} \mid \forall\ i, 0 \le i < |\Lambda|^{B1}\}$;
7 Add this gram into $\boldsymbol{F}$: $\boldsymbol{F} \leftarrow \boldsymbol{F} \cup \{ g_f \}$;
8 **For** $0 \le i < |\Lambda|^{B1}$ **do** /* refresh the diagonal of R */
9 $r_{ii} \leftarrow r_{ii} - r_{fi}$, if $r_{ii} > r_{fi}$; otherwise, $r_{ii} \leftarrow 0$;
10 **Return;**

Figure 18. The general frequent-common gram searching algorithm (GFGS).

characters of the first m characters of a pattern. The range of m is $M \le m \le |p_i|$, where M denotes the minimum pattern length of all patterns, and $|p_i|$ is the current pattern length. Figure 17 illustrates the sampling window, where B_1 is the size of a frequent-common gram, $B_1 \le W$, and B_2 is the size of the second pivot in the H^2 table, which is explained later.

The GFGS algorithm is presented in Figure 18. A bit-map vector $V = (v_i)$ and a matrix $R = (r_{ij})$ are temporary memory, where $0 \le i, j < |\Lambda|^{B_1}$. Vector V records the occurrence of each B_1-gram in a pattern; R is used for recording frequency, where r_{ij}, $i \neq j$, indicates the number of concurrent occurrences of two B_1-grams g_i and g_j in ***P***; and r_{ii} records the frequency of the B_1-gram g_i occurring in distinct patterns. For instance, $r_{ij} = 2$

means there are two patterns, each containing both g_i and g_j. In the GFGS algorithm, each pattern is first transferred into a set of B_1-grams, and the occurrence of each B_1-gram is recorded in the bit-map V, where B_1 is pre-defined and depends on the available on-chip memory space. Matrix R is then derived from V (as shown in line 4 of Figure 18). Second, the largest occurrence frequency r_{ff} is found, and its corresponding gram g_f is selected as one of $\boldsymbol{F}$. The elements of R relating to g_f are subtracted accordingly to renew R. GFGS is repeated until all elements on the diagonal of R become zero. GFGS uses only a matrix and a vector to discover $\boldsymbol{F}$ from $\boldsymbol{P}$.

4.3 Cluster Balancing Strategy (CBS)

Most packets are innocent in general situations. Even a harmful packet may contain only few patterns. Therefore, comparing all of the patterns in the large $\boldsymbol{P}$ with each input packet is time consuming. If the patterns in $\boldsymbol{P}$ can be distributed into different small *clusters* based on their similarity, then only the pattern in each cluster that is most similar to the suspected packet needs to be compared, thus improving the efficiency of the matching process. This subsection presents strategies to attain this goal. First, the method of clustering a set $\boldsymbol{P}$ based on the similarity of patterns is described. Then a cluster balancing strategy (CBS) is adopted to balance the cluster size. A *second-tier table* (H^2) for on-line matching can be constructed based on the clusters.

The *clustering pivots* are the keys used to distribute patterns, where each clustering pivot is a common gram of patterns defined previously. Two common grams are employed as a pair of clustering pivots, called a *pivot pair*, say (a, b), where the first pivot is a frequent-common gram, and the second pivot is the substring following the frequent-common gram. Let $\boldsymbol{P}_{a,b}$ represent a cluster of selected patterns (a subset of

patterns) with the pivot pair (a, b), which means that $\boldsymbol{P}_{a,b} = \{p_i \mid$ 'ab' $\subset p_i$, $a \in \boldsymbol{F}$ and $b \in \Lambda^{B_2}\}$, where 'ab' is the combination of two strings a and b and is a substring of p_i; $\boldsymbol{F}$ is the result of GFGS, and B_2 is the length of the second pivot. Notably, a pattern is assigned to only one cluster in the clustering strategy, although a pattern may have more than one pivot pair. That is, the clusters have the following properties: for any cluster $\boldsymbol{P}_{a,b} \subset \boldsymbol{P}$, that $\cup_{\text{all } a,b} \boldsymbol{P}_{a,b} = \boldsymbol{P}$, and $\cap_{\text{all } a,b} \boldsymbol{P}_{a,b} = \varnothing$. Since a pattern may have several opportunities to select a cluster, a better assignment can lower the maximum cluster size, and thereby improve the worst-case performance of EHMA.

The pattern grouping is based on $\boldsymbol{F}$. To lower the worst matching time, CBS is adopted to balance the size of all clusters. In CBS, an $|F| \times |\Lambda|^{B_2}$ matrix $N = (n_{a,b})$ is used to record the current size of every cluster $\boldsymbol{P}_{a,b}$ during the pattern grouping procedure. The CBS is as follows.

(1) First, read one pattern at a time from $\boldsymbol{P}$ and scan the pattern.

(2) According to GFGS, for any given p_i, there exists a B_1-gram $g \in \boldsymbol{F}$, where B_1 is the length of a frequent-common gram. To balance the cluster size, CBS finds the smallest $n_{a,b}$, given by $n_{x,y}$, among all available pivot pairs (a, b)s of p_i, for all $a \in \boldsymbol{F}$ and 'ab' $\subset p_i$.

(3) After grouping p_i into the smallest cluster $\boldsymbol{P}_{x,y}$, the corresponding $n_{x,y}$ is also incremented.

All patterns are distributed sequentially into the designate clusters. Accordingly, GFGS and CBS divide the large $\boldsymbol{P}$ into smaller subsets. Figure 19 illustrates the pattern clustering architecture.

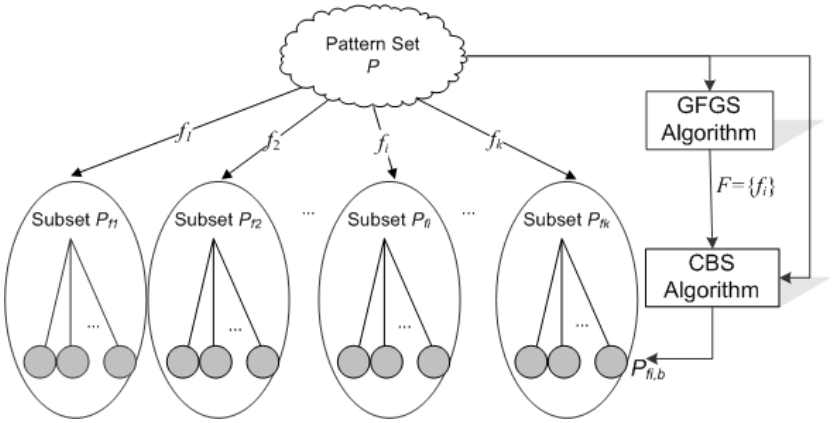


Figure 19. The pattern clustering architecture.

4.4 Safety Shift Strategy

This section presents a safety shift strategy to derive the values of the *shift* fields of H^1 and H^2. H^1 and H^2 can use the same strategy to derive their safety shifts respectively. As mentioned previously, as long as no frequent-common gram is matched in input strings, then no pattern exists. Therefore, if no frequent-common gram is missed, then no pattern will be missed. The safety shift strategy is based on a modified *bad grouped character heuristic* [9], named *frequency-based bad gram heuristic* in this study. The safety shift strategy ensures that no frequent-common gram is missed during a skippable scanning process. The proposed strategy helps EHMA to speed up the on-line matching process, since certain characters can be skipped unhesitatingly.

Assume that x identifies all possible index keys, and that the length of x is B. Because the index keys of H^1 and H^2 are different, the parameters used to determine the *shift* fields of these two tables are different. For H^1, as the length of a frequent-common gram is B_1, thus $x \in \Lambda^{B_1}$ and $B = B_1$. For H^2, since x is all the possible of the pivot pairs (a, b), $x \in \boldsymbol{F} \times \Lambda^{B_2}$ and $B = B_1+B_2$. The basic concept of the safety shift strategy is that: if x is not

a gram of any pattern, and any suffix of x is not any prefix of any pattern in $\boldsymbol{P}$, then it is safe to shift m when x is scanned; otherwise, the number of safety shifts is the offset between the rightmost occurrence position of x and the position of the frequent-common gram nearest to x. Two parameters are needed to derive the safety shifts, namely W, and m, as shown in Figure 17. Assume that $B \le W \le m$, and define the safety shifts of each entry ($H(x).shift$) as follows:

(1) Initially, all *shift* fields of the table H are set as

If $m > W$, then

$$H(x).shift = m - W + q, \tag{4-1}$$

where $q = \min\{q \mid \exists\ \text{sub}(x, q+1, B-q) = \text{sub}(p, 1, B-q),\ \forall\ p \in \boldsymbol{P} \text{ and } 1 \le q < B\}$ when $B > 1$ and q exists; otherwise $q = B$.

Else

$$H(x).shift = r, \tag{4-2}$$

where $r = \min\{r \mid \exists\ \text{sub}(x, r+1, B-r) = \text{sub}(f, 1, B-r),\ \forall\ f \in \boldsymbol{F},\ 1 \le r < B, \text{ and } r + B < W\}$ when $B > 1$ and r exists; otherwise $r = B$.

(2) Scanning every pattern p, for each i-th B-gram of each pattern $p^B[i]$, where $1 \le i \le m\text{-}W$, set $x \leftarrow p^B[i]$ if the entry $H(x)$ exists:

If the current $H(x).shift > m\text{-}W\text{-}i+1$, then update the entry, so that

$$H(x).shift = m\text{-}W\text{-}i+1. \tag{4-3}$$

(3) For each i-th B-gram of each pattern $p^B[i]$, where $m\text{-}W < i \le m\text{-}B+1$, set $x \leftarrow p^B[i]$ if the entry $H(x)$ exists:

If $x \in \boldsymbol{F}$, then

$$H(x).shift = 0; \tag{4-4}$$

Else If the current $H(x).shift > r$, then update the entry:

$$H(x).shift = r, \tag{4-5}$$

where $r = \min\{r \mid \exists\ \text{sub}(x, r+1, B-r) = \text{sub}(f, 1, B-r),\ \forall f \in \boldsymbol{F},\ 1 \le r < B, \text{ and } r+B < W\}$ when $B > 1$ and r exists; otherwise $r = B$.

Notably, the maximum *shift* of EHMA is m while $W = B$. The frequent-common grams and the sampling window are introduced in the proposed frequency-based bad gram heuristic to improve the flexibility and the efficiency. Additionally, comparing EHMA with WM, the maximum safety shift is raised from $m-B+1$ to m. The *shift* value of the proposed EHMA is similar to but larger than the *shift* value of WM, when the given parameters are $m = M$ and $W = B$.

4.5 Table Construction

The result of GFGS, $\boldsymbol{F}$, is used to construct the small table H^1, which is stored in the on-chip memory. A direct index table of $|\Lambda|^{B_1}$ entries is used for H^1 to achieve fast lookup. B_1 is usually very small ($B_1 = 1$ or 2), and is pre-defined according to the available size of on-chip memory. An entry of H^1 is denoted as $H^1(a)$, where a is a B_1-gram, and each entry has three fields: the frequent-common gram ID, $H^1(a).fid$; the pattern ID when a itself is a pattern, $H^1(a).pid$, and the safety shift number in the Tier-1 Matching, $H^1(a).shift$. Namely, $H^1(a).fid = \{i \mid a = f_i \in \boldsymbol{F}\}$, and $H^1(a).pid = \{i \mid |p_i| = |f_i| = B_1, p_i = \text{‘}a\text{’}$ and $p_i \in \boldsymbol{P}\}$. The unused fields of H^1 are set to NULL. Since H^1 is a small table (for instance, 256 entries in the case of one-byte coding and $B_1 = 1$), it can be stored in the on-chip cache. Later, H^1 acts as a filter in the on-line matching to quickly discover whether the packet contains a pattern. Namely, EHMA employs H^1 to quickly scan and

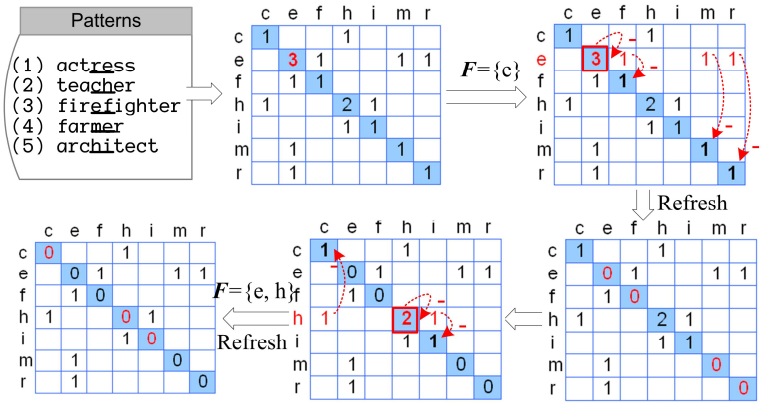
jump over the innocent substrings of the input packets, and to narrow the searching field to the most likely clusters.

The H^2 table is built based on the cluster assignments. H^2 contains the pattern contents and formatted information of patterns for fast on-line matching. Let $H^2(a, b)$ denote an entry of H^2, indicating the head pattern of the cluster $\boldsymbol{P}_{a,b}$, and defined as

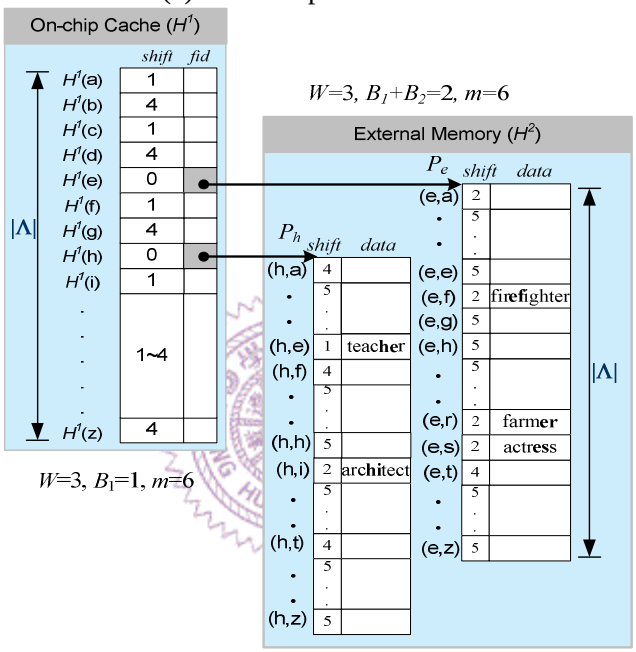
$$H^2(a, b) = H^1(a).fid \times |\Lambda|^{B_2} + b, \quad (4\text{-}6)$$

where B_2 is the length of the second pivot b, and is pre-defined according to the available size of the external memory. Each entry $H^2(a, b)$ consists of six fields[1]: the safety shift number in the Tier-2 Matching $H^2(a, b).shift$, the position of the frequent-common gram in the pattern $H^2(a, b).offset$, the pattern size $H^2(a, b).size$, the pattern content $H^2(a, b).data$, the pattern ID $H^2(a, b).pid$, and a pointer $H^2(a, b).next$ to the entry of the next pattern in the same cluster $\boldsymbol{P}_{a, b}$ or the fragmented content of the current pattern. Transferring the information of patterns into a predefined format can accelerate the matching procedure. The patterns in the same cluster $\boldsymbol{P}_{a, b}$ point to the same head entry

[1] Only the first two fields are specialized for EHMA. The other four fields are used for structured patterns as other algorithms.



(a) An example of GFGS.



(b) The architecture of the hierarchical hash tables.

Figure 20. An example of EHMA, where $B_1 = 1$, $B_2 = 1$, $m = M = 6$, $W = 3$ and $\boldsymbol{F}=\{e, h\}$.

$H^2(a, b)$, and are linked by the linked-list structure to optimize the memory usage. The required memory size of H^2 is $|F|\times|\Lambda|^{B_2}$ entries plus the share memory pool.

For example, if p_i is clustered to $\boldsymbol{P}_{a, b}$ by CBS and $H^2(a, b)$ is empty, then the information of pattern p_i is saved into $H^2(a, b)$, where $H^2(a, b).size = |p_i|$, $H^2(a, b).data = p_i$, $H^2(a, b).offset = k$ if the k-th B_1-gram of p_i is a, $H^2(a, b).pid = i$, and $H^2(a, b).next$ is NULL. If another p_j is also clustered to $\boldsymbol{P}_{a, b}$, then a free entry is also assigned to p_j and

linked with the previous pattern p_i. Similarly, if the pattern size of p_i is larger than the width of data field, then p_i is fragmented, and the remaining part is saved in a free entry of the share memory pool, and the address is saved in $H^2(a, b).next$.

Figure 20 shows an example of EHMA, which has five patterns: 'actress', 'teacher', 'firefighter', 'farmer', 'architect', where the alphabet set comprises the 26 English letters. The parameters for EHMA are assumed $B_1 = 1$, $B_2 = 1$, $m = 6$ and $W = 3$. Figure 20 (a) demonstrates the GFGS. According to the GFGS (lines 2 – 4 of Figure 18), after scanning the first W-B_2 characters of the sampling window of every pattern (the underlined characters of the patterns in Figure 20 (a)), the matrix R is obtained and shown in the figure. In the first run, the maximum value on the diagonal of R is three, and thus the corresponding gram '*e*' is added into $\boldsymbol{F}$. After refreshing the elements on the diagonal of R (lines 8 – 9 of Figure 18), GFGS finds that the maximum value on the diagonal of R is two in the second run, and the corresponding gram is '*h*'. GFGS stops while all elements on the diagonal of R are zero, and gets $\boldsymbol{F}=\{e, h\}$. Figure 20 (b) displays the logical architecture of the two-tier tables of EHMA. Because $B_1= 1$, and the H^1 table has only 26 entries, the H^1 table can be stored in the cache memory. The *fid* fields of H^1 point to the corresponding offsets of H^2. As the pattern 'actress' has '*e*'$\in \boldsymbol{F}$ and the pivot pair '*es*', according to CBS it is grouped to the cluster $\boldsymbol{P}_{e, s}$. The *shift* fields of H^1 and H^2 are obtained from the proposed safety shift strategy. Initially, since $B_1 \leq 1$, $H^1.shift = 4$. While $B_1+B_2 > 1$, $H^2.shift$ is set to 5 for those entries whose second pivot is not the prefix of any pattern (that is, $b \notin$ {'a', 'f', 't'}); otherwise, $H^2.shift$ is set to 4. When scanning the pattern 'actress', the *shift* fields of H^1('a'), H^1('c') and H^1('t') are updated to 3, 2 and 1 respectively (the 2nd safety shift strategy); the *shift* fields of H^1('r') and H^1('s') are both

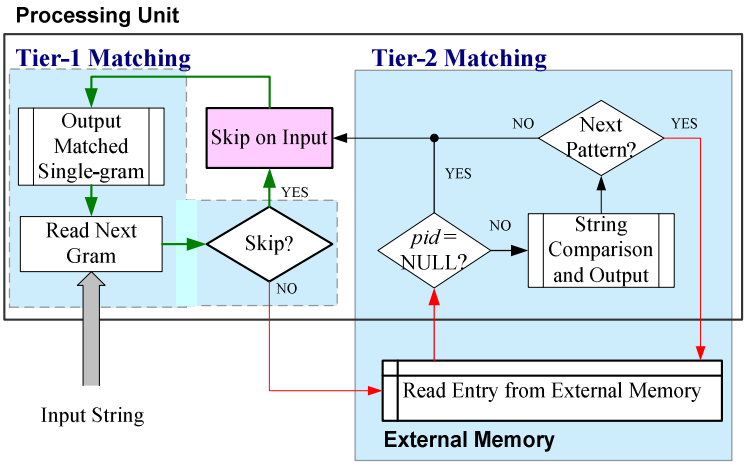


Figure 21. The processing flows of the on-line matching.

updated to 1, while the H^1('e').*shift* is updated to zero, because 'e'$\in \boldsymbol{F}$ (the 3rd strategy). As for the table H^2, only the existing entry H^2('e', 's') has to be updated to two, because $B = B_1+B_2 = 2$, and no prefix of $\boldsymbol{F}$ is the suffix of 'es' (the 3rd strategy). The remainders of the patterns follow the same clustering and safety shift strategy. The *shift* fields of H^1 and H^2 tables are updated when the new *shift* is less than the previous one. Let us see H^1('a') for example. When scanning the pattern 'actress', H^1('a').*shift* = 3 (as $p^1[i]$ = 'a', i = 1 and m-W-i+1 = 3); while scanning the pattern 'teacher', H^1('a').*shift* is updated to 1 (as 'a' is the third character of 'teacher': i = 3, then m-W-i+1 = 1), because the new value is smaller than the previous one (the 2nd strategy). Finally, H^1('a').*shift* = 1 is saved in the table because the remaining patterns do not have H^1('a').*shift* smaller than one. Notably, the maximum shift of H^1 and H^2 is large (4 and 5 respectively). Consequently, the number of scans and comparisons can be significantly reduced.

4.6 The On-line Hierarchical and Cluster-wise Matching

The previous subsections presented the off-line stage of EHMA, which builds two index tables H^1 and H^2, holding the indexing and pattern information in the cache memory

and external memory respectively. These two tables are regarded as the two-tier filters and indices for the on-line matching. This subsection presents the on-line matching procedure in detail.

In network intrusion detection systems, an input packet is forwarded to a detection engine. The detection engine then returns the search results of matched patterns $\boldsymbol{P}_M$. This study focuses on the payload inspection, and assumes that each input is a packet payload T. As a hierarchical matching, the on-line matching procedure of EHMA is divided into two tiers: Tier-1 Matching and Tier-2 Matching. The hierarchical architecture is applied to decrease the number of external memory accesses. The small H^1 is stored in the cache of the processing unit for Tier-1 Matching, while the H^2 with pattern content is in the external memory for Tier-2 Matching. Figure 21 illustrates the processing flows of EHMA, and shows that the on-cached Tier-1 Matching does not access the external memory, but does act as a pre-filter. The external memory access is necessary only when the Tier-2 Matching is invoked. This process is described in detail in the following subsections.

4.6.1 Tier-1 Matching

In on-line matching, the payload T is scanned from left to right, and each B_1-gram of T is the key to fetch the entry $H^1(t_1)$, where $t_1 = T^{B_1}[i]$. The H^1 acts as the first-tier filter of EHMA, by checking whether T may likely contain patterns belonging the pattern set $\boldsymbol{P}$. Because H^1 is small enough to be stored in the on-chip memory during the on-line matching procedure, the latency of accessing H^1 is very small.

In the Tier-1 Matching, first the *shift* field is checked. If $H^1(t_1).shift \neq 0$, i.e., $t_1 \notin \boldsymbol{F}$, then no external memory is necessary. The obtained $H^1(t_1).shift$ also determines the

number of grams that can be skipped without further process. The next gram to check is then $T^{B_1}[i + H^1(t_1).shift]$. After read the next gram, the matching process repeats as in the previous steps, and remains in the Tier-1 Matching. Because $|\boldsymbol{F}| \ll |\Lambda|^{B_1}$, the probability of $t_1 \in \boldsymbol{F}$ is small and most grams of T gain the shifts, thus avoiding the Tier-2 Matching. Consequently, both the number of string comparisons and the costly memory accesses can be significantly reduced.

Otherwise, if $t_1 \in \boldsymbol{F}$, then T may contain a malicious pattern $p_k \in \boldsymbol{P}$, where $t_1 \subset p_k$. Simply stated, if $H^1(t_1).shift = 0$, then T may have a pattern that belongs to the cluster of pivot pair (t_1, t_2), where $t_2 = T^{B_2}[i + B_1]$. Therefore, the matching procedure activates Tier-2 Matching to identify the pattern. If $H^1(t_1).pid$ is not NULL, then the current gram t_1 itself is a pattern, and this matched pattern is also added into $\boldsymbol{P}_M$.

4.6.2 Tier-2 Matching

After the Tier-1 Matching, if $H^1(t_1).shift = 0$, then the matching procedure proceeds to the Tier-2 Matching. The function $H^2(t_1, t_2)$ indicates the location of the corresponding cluster according to input T. Since EHMA is a cluster-wise matching algorithm, only the patterns in the small cluster of pivot pair (t_1, t_2), which are similar to T, are loaded to the processing unit for further checks.

Tier-2 Matching first checks the *pid* field of H^2. If $H^2(t_1, t_2).pid$ is NULL, then the cluster (t_1, t_2) contains no pattern, and no pattern comparison is necessary. Otherwise, if $H^2(t_1, t_2).pid$ is not NULL, then this cluster contains patterns. The pattern content in the $H^2(t_1, t_2).data$ is then compared with the corresponding substring of T: sub(T, i-$H^2(t_1, t_2).offset$, $H^2(t_1, t_2).size$). If $H^2(t_1, t_2).next$ is valid, and points to the next entry, here

```
     Procedure Tier-1Matching(T, H^1, M, W, B_1)
     Input: Packet payload T, a first-tier hash table: H^1, the minimum pattern length M, the length of the
     frequent-common gram B and the length of the sampling window W.
     Output: The output of Tier-2Matching.
1    i←M-W+1;
2    While i <= |T|- B_1 do
3      Read the i-th B_1-gram of T: gram←T^B1[i];
4      If H^1(gram).shift > 0, then shift←H^1(gram).shift;
5      Else
6        If H^1(gram).fid ≠ NULL, then shif←Tier-2Matching(T, H^2, B_2, i);
7        If H^1(gram).pid ≠ NULL, then
           P_M←P_M ∪ {gram};
           If shift == 0, then shift←1;
8      Jump over the string: i←i+shif;        /*shift and read the next*/
9    End While
10   Return;

     Procedure Tier-2Matching(T, H^2, B_2, i)
     Input: Packet payload T, a preprocessed indexing table: H^2, the length of the second pivot B_2, and the
     current pointer i
     Output: A safety shift number for Tier-1 Matching: shift, the matched pattern set of T: P_M, and its
     corresponding pid PID_M
1    Load data from the external RAM at entry H^2(T^B1[i], T^B2[i+B_1]) to a local buffer LB;
2      shift←LB.shift;
3    While (k←LB.pid) ≠ NULL do
4      Compare the substring of T: sub(T, i-LB.offset, LB.size) with the pattern LB.data;  /*Assume no
     fragmentation here*/
5      If it is matched then P_M←P_M ∪ {p_k} and PID_M←PID_M ∪ {k};
6      If LB.next ≠ NULL then
7        Load data from the external RAM at entry LB.next to the local buffer LB;
8      Else
9        Jump to Line 10;
10   End While
11   Return shift;
```

Figure 22. The on-line matching procedure, including Tier-1 Matching and Tier-2 Matching.

given by $H^2(a, b)$, then the cluster contains other patterns. Similarly, the pattern in $H^2(a, b).data$ is also fetched and compared with the substring of T starting at $T[i-H^2(a, b).offset]$ of length $H^2(a, b).size$. Every matched pattern is added to the matched pattern set $\boldsymbol{P}_M$ and its corresponding matched *pid* set $\boldsymbol{PID}_M$ in order. Until all patterns in this cluster are checked, the next gram $T^{B_1}[i+H^2(t_1,t_2).shift]$ is then read, and the on-line matching procedure returns to the Tier-1 Matching. $H^2(t_1, t_2).shift$ also indicates

the number of characters of T that can be skipped, since the next possible frequent-common gram may only appear far than $H^2(t_1, t_2).shift$ away.

Notably, if a pattern p_k exists in T, then all grams of p_k appear in T. The clustering pivot pair of pattern p_k, ($p_k^{B_1}[j]$, $p_k^{B_2}[j+B_1]$) is certainly scanned, say at t_1 and t_2, so that $t_1 = p_k^{B_1}[j] \in \boldsymbol{F}$ and $t_2 = p_k^{B_2}[j+B_1]$. Pattern p_k is then recognized when T is compared with the patterns in the cluster (t_1, t_2) during the on-line matching procedure. Based on the Safety Shift Strategy, EHMA never skips any frequent-common gram. Consequently, no patterns in the payload T are missed.

The on-line matching procedure of EHMA is described in Figure 22, including Tier-1 Matching and Tier-2 Matching. Since EHMA introduces H^1 and H^2 as filters, and CBS is employed, only a few suspected patterns are loaded from external memory and compared with T. Because generally most of the packets are innocent over the network, and the frequent-common grams ($\boldsymbol{F}$) narrow the searching field, EHMA performs a fast scan over the packets. The returned result $\boldsymbol{P}_M$ includes all matched patterns for a given T, and is applied to make the final decision and to analyze the impending attacks. The final decision depends on decision-making rules.

An example is provided to demonstrate the online matching of EHMA. Assume that the H^1 and H^2 tables have been built as Figure 20 where $W = 3$ and $M = 6$. Assume that the input T is ‘kangaroo’ as given in Figure 23. The scan runs from left to right. The scan starts at ‘g’ (($M-W+1$)-th gram), obtaining H^1(‘g’).$shift$ = 4. Therefore, Tier-1 Matching shifts four characters. Because the pointer goes beyond $|T|-B_1$ after the shift, EHMA completes scanning the input T. This example only requires one on-cache table lookup,

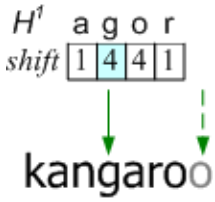


Figure 23. An example of matching process with input ‘kangaroo’.

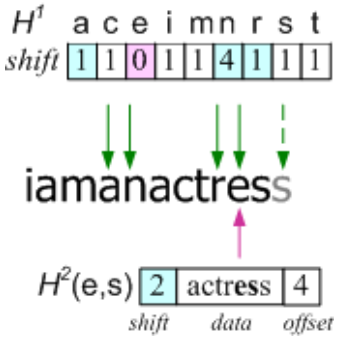


Figure 24. An example of matching process with input ‘iamanactress’.

and no external memory access. By only checking T with the embedded table H^1, EHMA can know that T contains no pattern.

Considering another example where T = ‘iamanactress’ as shown in Figure 24, the first scanned B_1-gram is ‘a’, yielding H^1(‘a’).*shift* = 1. Thus the matching process stays in the Tier-1 Matching, and the next B_1-gram ‘n’ is read after shifting one character, yielding H^1(‘n’).*shift* = 4. Similarly, staying in the Tier-1 Matching, and the next B_1-gram ‘n’ is read after shifting one character, yielding H^1(‘n’).*shift* = 4. Similarly, staying in the Tier-1 Matching, the matching process obtains H^1(‘r’).*shift* = 1 and H^1(‘e’).*shift* = 0 in order after shifting. While H^1(‘e’).*shift* = 0, the Tier-2 Matching is activated. After checking the field H^2(‘e’, ‘s’).*pid* and finding that it is not NULL, EHMA knows a suspected pattern may exist. The Tier-2 Matching then compares input T with the pattern in the cluster $\boldsymbol{P}_{e,s}$, where H^2(‘e’, ‘s’).*data* = ‘actress’, and gets a match. Because this cluster contains no other patterns, the matching process returns to Tier-1 Matching with H^2(‘e’, ‘s’).*shift* = 2. Since the pointer goes beyond $|T|-B_1$ after shifting two characters, the matching process

for the input T is finished. In this case, H^1 is checked four times, and H^2 is fetched only once for the string T of twelve characters. EHMA thus significantly reduces the latency caused by memory accesses.

4.7 Incremental Update

EHMA can achieve incremental update by adding a *count* field in the H^2, which records the current size of every cluster. The *count* field has the same function as the matrix N of CBS. When a pattern p is added into $\boldsymbol{P}$, after checking the *count* fields of the possible entries according to the pivot pairs of p, the smallest cluster, say $\boldsymbol{P}_{x,y}$, can be found. Then, p is added into the cluster $\boldsymbol{P}_{x,y}$ by following the steps of the table construction mentioned previously. If no B_1-gram of p belongs to $\boldsymbol{F}$ and p finds no existing entry in the H^2, then a random B_1-gram of p, say g, is chosen and added into $\boldsymbol{F}$ ($H^1(g)$ is modified accordingly), and a memory space is allocated for cluster set $\boldsymbol{P}_g$ in the H^2. A random pivot pair of p, say (g, h), is chosen and then p is added into the cluster $\boldsymbol{P}_{g,h}$. The *shift* fields of H^1 and H^2 may be modified because of the added p. Since the safety shift strategy scans the patterns *one by one* to calculate the *shift* values, no modification to the safety shift strategy is required for pattern addition. The added p can be recognized as the last scanned pattern of the safety shift strategy. At most $|p|$-B_1+1 fields of H^1 and $|p|$-B_2+1 fields of H^2 are modified for a pattern addition.

To delete a pattern p from $\boldsymbol{P}$, first step is to find the pattern. When p is found, just link p's previous entry to p's next entry by modified its *next* field in the H^2, and delete p from tables. Then, subtract the *count* field of the cluster that p belongs to. The shift fields are not modified for pattern deletion. Because the *shift* values are universal minimum in the safety shift strategy, they may not be optimum after pattern deletion. However, no error

will occur after pattern deletion, even while the *shift* fields are not modified. Consequently, EHMA needs not recalculate the whole index tables as long as the pattern database is changed. EHMA can refresh the index tables when the system is not busy.

4.8 Worst Case

If a given string T is formed badly that has to do the exact string comparisons the most times, and no character of T can be skipped during the on-line matching process, processing this bad-formed T is the worst case of EHMA. Assume the largest cluster size is L_c. When every character $T[t] \in \boldsymbol{F}$, $H^1(T[t]).shift = 0$, and each corresponding indexed cluster is the largest ($|\boldsymbol{P}_{T[t],T[t+1]}| = L_c$), T is a bad-formed string and this is the worst scenario of EHMA. As for all $T[t]$, $T[t] \in F$ and $H^1(T[t]).shift = 0$, the probability to fetch the table H^2 for the bad-formed T is one. Thus, the number of external memory accesses per character in the worst case is

$$N_{RAM}^{WST} = \frac{(|T| - B_2) \times L_c}{|T|} < L_c, \tag{4-7}$$

where assume that fetching one pattern needs one memory access. Define the largest pattern size in $\boldsymbol{P}$ as L_p. When every input character points to the largest cluster, in which every pattern has the longest size, this bad-formed T requires the largest number of comparisons. Hence, the number of character comparisons per input character is

$$N_{CMP}^{WST} = N_{RAM}^{WST} \times L_p < L_c \times L_p . \tag{4-8}$$

Obviously, the worst-case performance depends on L_c. To derive L_c, assume there is a largest cluster, say $\boldsymbol{P}_{x,y}$. Since $\boldsymbol{P}_{x,y}$ is the largest cluster, assume that the cluster size is always larger than one, and initially the probability that its cluster size increases from 0 to 1 is one.

As $\boldsymbol{P}_{x,y}$ is the largest cluster, based on CBS, a given pattern p will not be clustered into $\boldsymbol{P}_{x,y}$, unless all available pivot pairs of p are not in the set $F\times\Lambda$ except (x, y). Since the pattern database is usually predefined and static, assume the given patterns are uniformly distributed. Therefore, the probability that $|\boldsymbol{P}_{x,y}|$ increases from i to i+1 is

$$\Pr\{|P_{x,y}| = i \to i+1\} = \left(\frac{|\Lambda|^2 - |F|\times|\Lambda| + 1}{|\Lambda|^2}\right)^{|p|-B_2-1}. \quad (4\text{-}9)$$

As in the worst-case scenario, every pattern has the longest size L_p, the equation is rewritten

$$\Pr\{|P_{x,y}| = i \to i+1\} = \left(\frac{|\Lambda|^2 - |F|\times|\Lambda| + 1}{|\Lambda|^2}\right)^{|L_p|-B_2-1}. \quad (4\text{-}10)$$

Thereby, the probability that the cluster size of $\boldsymbol{P}_{x,y}$ is maximum (L_c) is derived

$$\Pr\{|P_{x,y}| = L_c\} = \left(\frac{|\Lambda|^2 - |F|\times|\Lambda| + 1}{|\Lambda|^2}\right)^{(|L_p|-B_2-1)(L_c-1)}. \quad (4\text{-}11)$$

When $|\boldsymbol{P}|$ is 1200 with $|\boldsymbol{F}|$ = 77, $|\Lambda|$ = 256 and L_p = 128, the probability that L_c = 4 is only 7×10^{-79}. When replacing L_p with the average pattern size, which is about eleven in the Snort, then the probability that L_c = 4 is about 3.6×10^{-6}. The probability that L_c = 4 is tiny, which infers that EHMA has a small L_c, and thus N_{RAM}^{WST} and N_{CMP}^{WST} are small. Consequently, the worst-case performance of EHMA is moderate and acceptable because L_c is much smaller than $|\boldsymbol{P}|$.

4.9 Results

As the number of network security threats rises, the NIDS has become one of the most important applications of packet inspection [20], [23]. Therefore, this study demonstrates the feasibility of integrating the proposed EHMA with the promising NIDS. This section presents the simulation results of EHMA deployed in the NIDS, compared with the original hierarchical matching algorithm (HMA) [38], the Boyer-Moore-Horspool algorithm (BMH) [21], the Wu-Manber algorithm (WM) [36], a variant of the Wu-Manber algorithm using a grouped prefix hash (WM-PH) [27], and the Aho-Corasick algorithm with memory compression (AC-C) [34]. In the simulations, the assembly-like microprograms were emulated for EHMA, BMH, WM, WM-PH and AC-C using RISC instructions of general network processors (such as ADD, XOR, MOV), and the number of instructions and the number of memory accesses needed to process a packet were calculated. To simplify the evaluation, the simulation assumed that one microprocessor was employed.

4.9.1 Measurements

Define I as the average number of RISC instructions (including comparisons and calculations), and L as the average number of local memory accesses (including reading data from the cache to the registers for further processes), for each payload character in the pattern matching. E represents the average number of external memory accesses per input character, which includes loading the input packets, querying the entries of tables in the external memory, and fetching the patterns. w_I indicates the time needed by one instruction or one local memory/register access, and w_E indicates the time for one external memory access. The following measurements are given: the average computation cycles

Table 10. The simulation parameters.

Items	Value
Time of one RISC instruction or one local memory access (w_I)	1 cycle
Latency for each external memory access (w_E)	10, 100 cycles
Packet payload length for Model I	512 bytes
Number of patterns in P ($\|\boldsymbol{P}\|$)	200, 400,…,5000
Simulation time for Model I	10 million packets

Table 11. The pattern size distribution of Snort rule set R_1.

Pattern Size	=1	≤ 4	≤ 8	≤ 12	≤ 16	>16
Ratio	0.028	0.245	0.482	0.653	0.813	0.187

$\psi_I = I \times w_I$; the average memory latency $\psi_M = E \times w_E + L \times w_I$; and the total average matching time $\Psi = \psi_I + \psi_M$, which is regarded as the overall performance.

In the simulations, the skip table of BMH was assumed to be small enough to be loaded into the cache memory, and therefore only one external memory access was counted during the matching process of BMH for each pattern. One external memory access was assumed for AC-C, although it typically needs two memory references to fetch the transition matrices, and the fail table or the matched patterns. Table 10 lists the simulation parameters.

4.9.2 Traffic Models

The simulations used two free and real pattern sets, R_1 and R_2, from Snort in Aug. 2004 and May 2008 respectively [1], although the pattern set can be self-defined or any commercial pattern set. The number of *distinct* patterns is about 1250 in the R_1, where the average length of a pattern is about 11.2 bytes (the statistics of the pattern set listed in Table 11); while the number of distinct patterns becomes up to about 5000 in the R_2. Since Snort patterns are written in mixed plain text and hex formatted bytecodes, the alphabet

size ($|\Lambda|$) was set to 256 in the simulations. In the simulation traffic models, Model I and II use R_1, and Model III uses R_2 as the matching pattern sets.

Table 8 shows the relationships between the number of patterns $|\boldsymbol{P}|$ and the number of frequent-common grams $|\boldsymbol{F}|$ of the EHMA, where the lengths of patterns are in the range from 1–122, $m = |p_i|$, and the patterns are randomly selected from R_1. The results in Table 8 reveal that the growth rate of $|\boldsymbol{F}|$ is much slower than that of $|\boldsymbol{P}|$.

4.9.2.1 Model I

In the Model I, the synthetic malicious packets are generated by randomly choosing patterns from the pattern set $\boldsymbol{P}$ and spreading over the packet payloads. The attack load λ is defined to represent the expected number of malicious patterns existing in one packet. For instance, if $\lambda = 2$, then each packet contains two harmful patterns on average. Except for the injected patterns parameterized by λ, the background characters of a packet were randomly drawn from Λ to imitate the normal packet content. Hence, the random background may unconsciously contain patterns.

4.9.2.2 Model II

To evaluate the performance of algorithms in a real intense attack, a trace from the Capture-the-Flag contest held at Defcon9 was adopted as the input traffic in the Model II. The Defcon Capture-the-Flag contest is the largest security hacking game, in which competitors try to break into the servers of others while protecting their own servers, each hiding several security holes [14].

Table 12. The statistics of the traffic traces.

Statistics	Model II	Model III
Average Packet Size (Byte)	467.71	896.1
The Standard Deviation of the Size of each Packet (Byte)	651.06	690.99
Data Transmission Rate (Kbps)	254.13	280.03
Number of Packets per second	69.55	40
Packet Type: TCP (%)	48.48%	97.18%
UDP (%)	0.65%	2.56%
Others (%)	50.87%	0.26%

4.9.2.3 Model III

Model III uses a real 2-hour trace as the input traffic, and the more recent Snort rules R_2 as the pattern set $|\boldsymbol{P}|$. This real trace recorded all IP packets in a laboratory of Providence University for 2 hours. The laboratory has an FTP server, a web server, and three PCs running several network application clients.

Table 12 lists the statistics of the traffic traces used in Model II and Model III, where the values are measured by traffic analysis tools: *tcpstat* and *tcptrace*.

4.10 Memory Requirements

For fast lookup and matching, the lookup information and patterns are usually saved in the memory using a tabular structure. Therefore, the memory requirements are estimated according to the number of entries. Since all algorithms need to keep the pattern content in the (external) memory, this section only discusses the extra memory requirement for the tables of each algorithm. In the simulations, the numbers of characters in the clustering pivots (B_1 and B_2) were both assumed to be 1. Because the H^1 of EHMA is a direct index table, the cache memory space (M_1) of EHMA comprises $|\Lambda|$ entries. Based on GFGS and CBS, the number of entries in H^2 is the total number of possible clusters (plus a small memory pool). Since the domain of possible pivot pairs is $\boldsymbol{F} \times \Lambda$, the

Table 13. The memory requirements.

	EHMA	HMA	WM	WM-PH	AC-C	BMH	BMH-O
Cache Memory	O(|Λ|)	O(|Λ|)	O(1)	O(1)	O(1)	O(|Λ|)	O(1)
External Memory	O(|***F***|×|Λ|)	O(|***F***|×|Λ|)	O(|Λ|^3)	O(|Λ|^3)	O(*S*)	O(|***P***|×|Λ|)	O(|***P***|×|Λ|)

Table 14. A list of symbols.

Notation	Meaning
p_i	A pattern with an identification number (ID) *i*
P	Pattern set. ***P*** = {p_i}
|***P***|	The size of pattern set ***P***
Λ	Alphabet set
T	Input string
ℑ	Significant gram set
F	Frequent-common gram set
B-gram	A *gram* is defined as a group of characters, and *B* is the number of characters in a gram.
B_1	The size of a frequent-common gram
B_2	The size of the second pivot in the H^2 table
M	The minimum pattern length of all patterns
W	The size of *sampling window*
I	The average number of RISC instructions per input character (including comparisons and calculations)
L	The average number of local memory accesses (including reading data from cache to registers)
E	The average number of external memory accesses for loading the packet, querying the entries of tables in the external memory, and fetching the patterns
w_I	The time of one instruction or one local memory/register access
w_E	The time of one external memory access
ψ_I	The average computation cycles: $\psi_I = I \times w_I$
ψ_M	The average memory latency: $\psi_M = E \times w_E + L \times w_I$
Ψ	Total average matching time: $\Psi = \psi_I + \psi_M$

external memory space for H^2 (M_E) of EHMA is O(|***F***|×|Λ|). HMA has the same memory requirement as EHMA. The *shift* table of WM is also a direct hash table. The gram size of WM (block size *B*) was 3 in the simulations, so the *shift* table of WM had |Λ|^3 entries. The grouped *skip* table of WM-PH used in the simulations was a direct prefix hash table with a prefix length of three characters. Therefore, the *skip* table of WM-PH comprises |Λ|^3 entries. Every pattern in the BMH has its own *skip* table of |Λ| entries, so that the table of BMH has |***P***|×|Λ| entries. Because each *skip* table of BMH (for one pattern) is small

enough to be loaded into the local memory, for fairness, a cache memory space was allocated to lower the number of external memory accesses. The BMH-O is the original BMH with no local cache and assesses the latency penalty. Notably, WM-PH, AC-C and BMH-O also require cache memory to store the skip value or one state during the matching process. Table 13 lists the memory requirements of EHMA, HMA, WM, WM-PH, BMH and AC-C. The scale relation of the parameters is $|\boldsymbol{F}| < |\Lambda| \ll |\boldsymbol{P}| < S \ll |\Lambda|^3$.

In the simulations using Model I, when $|\boldsymbol{P}|$ is 1200, the H^1 and H^2 of EHMA needs 256 and 19712 entries respectively (about 768 bytes on-chip memory and 38.5KB external memory, including the shared memory pool); HMA has the same number of entries as EHMA, but needs smaller entry size as HMA has no *shift* field; the table of WM needs more than 16M entries (16MB external memory, in the case without using an additional prefix table); the table size of WM-PH is the same as that of WM; BMH and BMH-O need more than 300K entries (300KB external memory); and AC-C needs 10731 states (461KB with each node of 44 bytes). The memory size of all algorithms listed previously excludes pattern content. Obviously, the required memory space of EHMA is quite small. Table 14 lists the symbols used in the Section 4.

4.11 Results and Discussion

The minimum pattern length of the feeding patterns in Figures 24–27 is only one character, i.e., $M = 1$. Because the minimum pattern length of WM is restricted to be larger than the gram size, in this case three characters, WM is not compared in these figures. In Figures 24–27, the results labeling EHMA in the following simulations use the

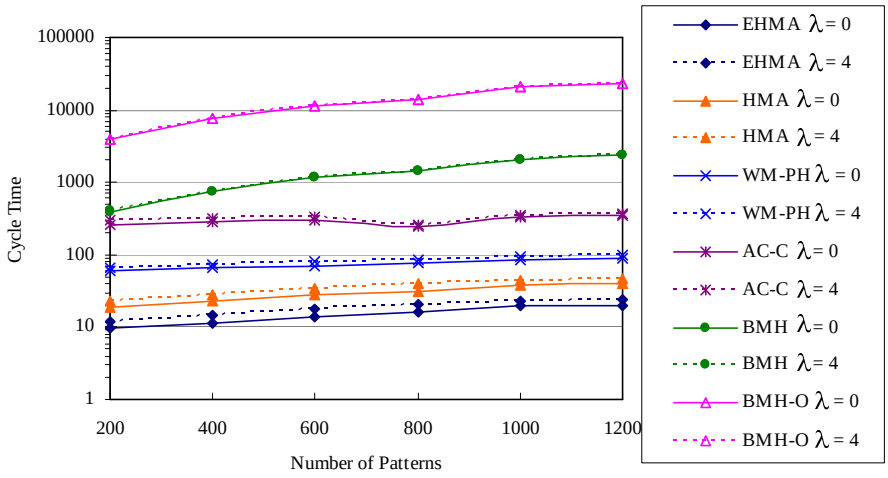
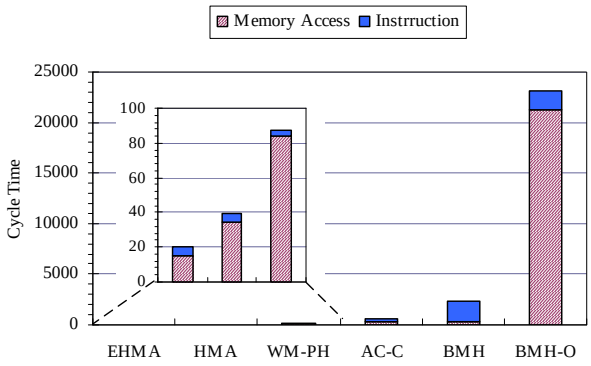


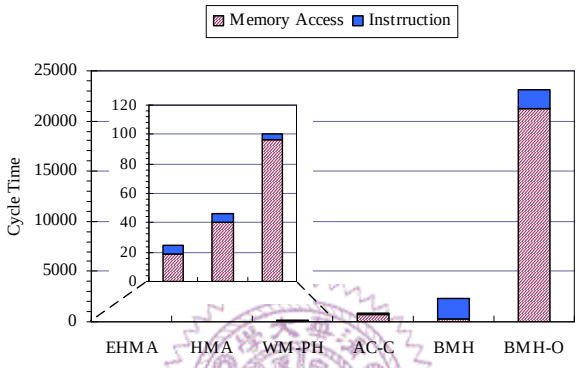
Figure 25. The average matching time (Ψ) versus the number of patterns (|***P***|), using Model I with $\lambda = 0$ and $\lambda = 4$, where $w_E = 100$.

sampling window with parameters $W = m = |p_i|$, which means that each pattern is sampled in its entirety.

Figure 25 compares the average matching time (Ψ) of EHMA, HMA, WM-PH, AC-C, BMH and BMH-O using Model I with different attack loads $\lambda = 0$ and $\lambda = 4$ respectively. It also shows the impact of the number of patterns (|***P***|) on the matching time. Simulation results reveal that EHMA outperforms HMA, WM-PH, AC-C, BMH and BMH-O even when |***P***| and λ increase. EHMA has slightly higher growth rate than WM-PH, because it has a much smaller table size. WM-PH gains performance by having a large direct index table. Notably, the matching time of the original AC using basic structure is independent from |***P***| and λ. The curves of AC-C increase with |***P***| and λ owing to the *popsum* used in the AC-C algorithm. The increasing |***P***| makes the matching time of BMH (BMH-O) rise steeply, because the BMH is originally a single-pattern matching algorithm that simply executes iteratively for multi-pattern matching.



(a) λ = 0 and $|\boldsymbol{P}|$ = 1200.



(b) λ = 4.0 and $|\boldsymbol{P}|$ = 1200.

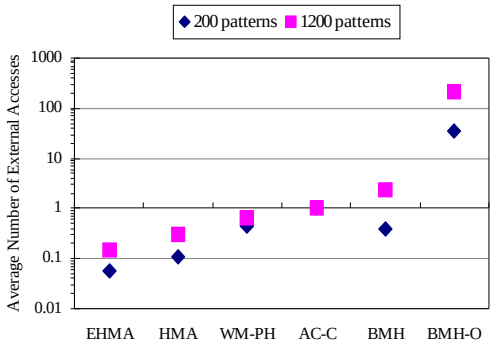
Figure 26. The proportion of ψ_I to Ψ and ψ_M to Ψ using Model I with $|\boldsymbol{P}|$ = 1200 and w_E = 100: (a) λ = 0 and (b) λ = 4.

The case λ = 0 means that the traffic has no malicious packets. In this case, the proposed EHMA needs only 9.5–19.9 cycles per character on average, which is about 0.9, 3.3–5.3, 16.3–26.8, 40–117 and 408–1161 times less than the matching time of HMA, WM-PH, AC-C, BMH and BMH-O, respectively, under various pattern set sizes. We can say that EHMA is very appropriate for network equipment, because generally most packets are innocent ($\lambda \approx 0$). The time available for the detection engine to process the malicious packets rises as the innocent packets are processed more quickly.

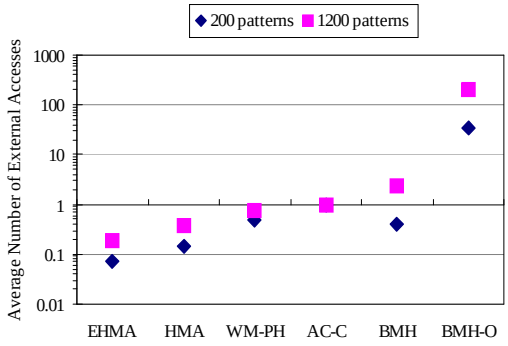
When λ = 4, then the systems are under heavy attack, and the traffic contains many monitored patterns. In this situation, the matching time of EHMA is about 0.89–0.94,

3.1–4.5, 14.1–24.9, 33.2–96.4 and 335–957 times less than that of HMA, WM-PH, AC-C, BMH and BMH-O respectively. Additionally, the performance of EHMA is quite stable, since Ψ rises only slightly as λ or $|\boldsymbol{P}|$ rises.

The processing time of the pattern matching includes the time necessary for instructions (ψ_I) and the time for memory accesses (ψ_M). To investigate their impacts on the algorithms, these two measurements are separated from overall matching costs since different systems introduce different implementation overheads. Figure 26 displays the proportion of ψ_I to Ψ and ψ_M to Ψ respectively, for all approaches using Model I with $|\boldsymbol{P}|$ = 1200, where Figure 26 (a) shows the results under λ = 0, and Figure 26 (b) shows the results under λ = 4. In Figure 26, the upper part of the bar is ψ_I and the lower part of the bar is ψ_M. The results show that the ψ_I of EHMA is close to HMA's and WM-PH's, but ψ_M of EHMA is much less than others. The proportion of ψ_M to Ψ of BMH seems smaller than others, because the whole skip table of a pattern is idealistically assumed to be loaded within one external memory access and kept in the cache during the matching process for each pattern. Because AC-C compresses the data structure of the state machine, it requires more time to derive the next state pointer. Therefore, AC-C does not have the smallest ψ_I. Simulation results show that the ψ_I does not significantly rise with λ in any of the experiments, because each algorithm has already tried to reduce the computation load (ψ_I). However, ψ_M dominates the overall matching cost. This reveals that the number of external memory accesses is the bottleneck of almost all algorithms. This result also reflects our opinion mentioned previously that the essential issue in designing a high-speed detection engine is to reduce the number of required external memory accesses.



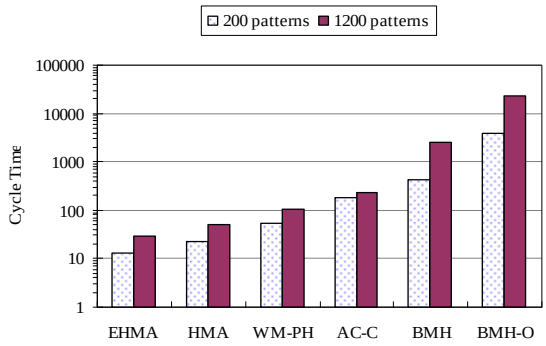
(a) $\lambda = 0$.



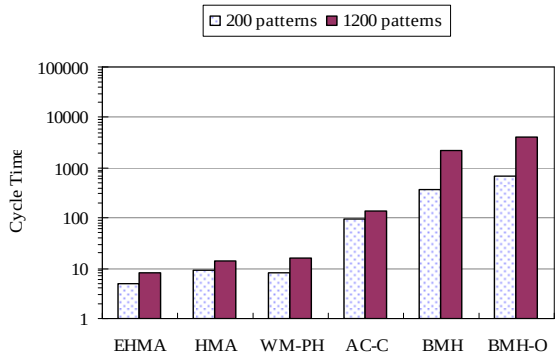
(b) $\lambda = 4$.

Figure 27. The comparisons of average number of external memory accesses (E) using Model I with $w_E = 100$: (a) $\lambda = 0$ and (b) $\lambda = 4$.

Figure 27 compares the average number of external memory accesses per character (E) of the state-of-the-art pattern matching algorithms. The figure shows that the E of EHMA is only 0.06–0.19, which is much smaller than others. In other words, EHMA can successfully filter out about 94% payloads when $|\boldsymbol{P}| = 200$, and 81% when $|\boldsymbol{P}| = 1200$, requiring no external memory accesses and string comparisons. The E of EHMA rises only slightly with rising λ. The increasing rate of E is slightly higher in EHMA than in WM-PH when $|\boldsymbol{P}|$ rises, because EMHA has much smaller table size than WM-PH. Since BMH is based on the single-pattern-matching algorithm, its E is proportional to $|\boldsymbol{P}|$. Consequently, the hierarchical matching along with the safety shift strategy is highly effective in reducing the memory latency.



(a) w_E = 100.



(b) w_E = 10.

Figure 28. The average matching time (Ψ) versus the number of patterns (|***P***|), using Model II: (a) w_E = 100 and (b) w_E = 10.

Figure 28 and Figure 29 adopted the Model II as a real-life network environment under intense attack to evaluate the performance of the state-of-the-art algorithms. Since different implementation systems may have different external memory costs (w_E), Figure 28 illustrates two results with w_E = 100 and w_E = 10 respectively. To lower the impact of w_E on an algorithm, a very small value of w_E is adopted in Figure 28 (b). The results in Figure 28 indicate that EHMA significantly outperforms others in both cases of small and large pattern set size even in the intense attack. EHMA still performs better than others even when the penalty on the external memory access (w_E) is reduced (as shown in Figure 28 (b)). Comparing EHMA with HMA in the Figure 25 - Figure 28 reveals that the

proposed safety shift strategy significantly reduces the number of external memory accesses and thus improves the matching performance.

The minimum length of Snort patterns is one character. However, some detection systems, such as virus detection systems, have larger minimum pattern lengths. The performance of matching algorithms with long minimum pattern lengths was examined using Model II, including only the patterns with lengths greater than 10 (M = 10) from Snort patterns, as drawn in Figure 29. Since the number of patterns whose length larger than ten characters in R_1 is around 500, Figure 29 shows the cases of $|\boldsymbol{P}|$ = 200 and $|\boldsymbol{P}|$ = 500, respectively. Figure 29 (a) shows the average processing time (Ψ);Figure 29 (b) shows the memory requirement of the fast index/hash tables, excluding the memory for pattern contents, and Figure 29 (c) compares the average number of external memory accesses (E) of all algorithms. Since here M is larger than the gram size of WM, which is three as mentioned before, the performance of WM is compared here. The result labeling EHMA(W=5) is the case using EHMA algorithm with $m = M$ = 10 and W = 5. Recall that the sampling window of EHMA is entire pattern content, that is, $m = M = |p_i|$. To observe the performance of WM and WM-PH with smaller hash tables, Figure 29 also displays two additional cases with block size of two characters, WM(B = 2) and WM-PH(B = 2).

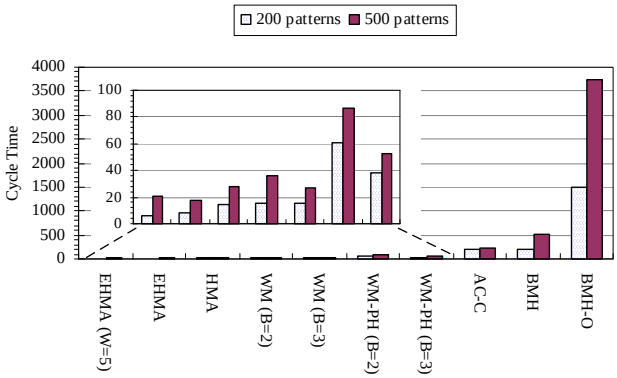
Before discussing the simulation results of Figure 29, Table 15 presents the effect of the size of sampling window (W) on the performance of EHMA in terms of the average shift values of H^1 and H^2, the size of the set of frequent-common grams ($|\boldsymbol{F}|$) derived from GFGS, the average number of actual shifts and the average number of external memory accesses, using the same traffic model as in the Figure 29.

Table 15. The impact of the size of sampling window (*W*) on the shift values of tables (H^1.*shift* and H^2.*shift*), |***F***|, actual average shifts and *E*, using Model II.

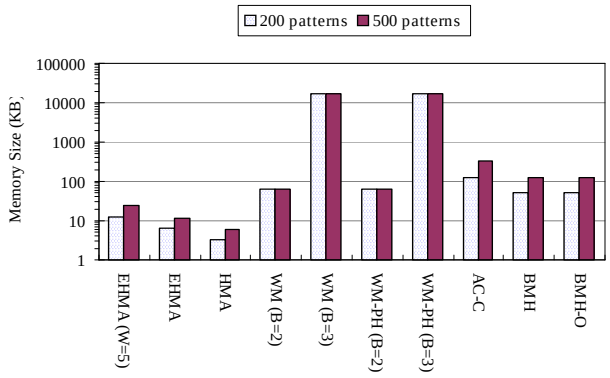
|***P***|	**200**				**500**			
	EHMA	EHMA (*W*=7)	EHMA (*W*=5)	EHMA (*W*=3)	EHMA	EHMA (*W*=7)	EHMA (*W*=5)	EHMA (*W*=3)
H*^{1}*.shift	0.94	2.71	3.66	4.74	0.91	1.86	2.02	2.49
H*^{2}*.shift	1.99	4.89	6.79	8.71	1.99	4.84	6.72	8.65
|*F*|	13	20	25	39	23	33	47	65
Average Shift	1.5	1.74	1.79	1.84	1.49	1.68	1.74	1.8
E	0.0377	0.0441	0.0431	0.0434	0.1243	0.16	0.1635	0.2512

Table 15 shows that the number of candidate common grams increases with increasing *W*, resulting in smaller |***F***|. The average number of H^1.*shift* and H^2.*shift* increases when *W* decreases. Since the traffic spectrum is not normally distributed, the actual average number of shifts during matching process is not the same as the average of H^1.*shift* and H^2.*shift*. However, the trend is the same. *E* is effected by both |***F***| and the actual average shift.

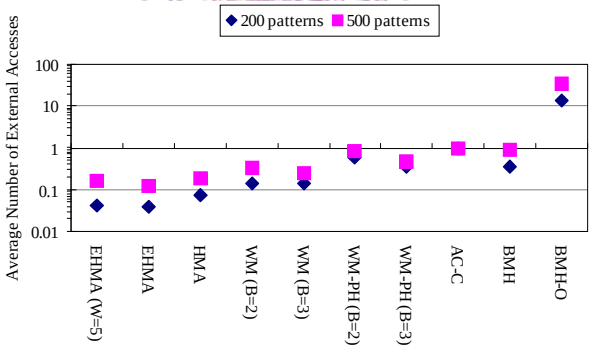
Figure 29 (a) shows EHMA(*W* = 5) outperforms EHMA and others when |***P***| = 200; while EHMA performs better than EHMA(*W* = 5) and others when |***P***| = 500. Therefore, reducing |***F***| becomes more important than increasing the average number of shift values when |***P***| is large. Since all algorithms need a copy of the pattern contents, Figure 29 (b) only displays the extra memory requirement of every algorithm for the index/hash tables. Figure 29 (b) shows that the required memory of EHMA is only slightly larger than that of HMA but much smaller than that of others. The required memory of EHMA grows moderately with |***P***|. The memory of EHMA(*W* = 5) is greater than that of EHMA due to the larger |***F***|. As shown in Figure 29, EHMA is highly effective in reducing the required external memory, providing efficient performance even in the virus-detection-like model.



(a) Average matching time.



(b) Memory Requirement.



(c) The average number of external memory accesses.

Figure 29. The costs versus the number of patterns (|***P***|), using Model II, w_E = 100 and M = 10: (a) Average matching time, (b) Extra memory requirement, and (c) The average number of external memory accesses.

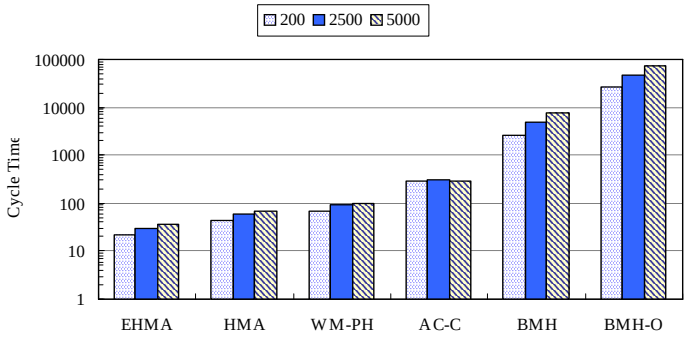


Figure 30. The average matching time (Ψ) versus the number of patterns (|***P***|), using Model III, w_E = 100.

Figure 30 uses Model III as real-life normal traffic to show the performance of the algorithms. Meanwhile, to demonstrate the effect of the rising number of patterns on the matching performance, a more recent Snort ruleset R_2 of about 5000 patterns are used in Model III. Figure 30 shows that EHMA performs better than others even when the patter set is very large. The matching time of EHMA only moderately increases with the rising |***P***|.

5 AC WITH MAGIC STRUCTURES (ACM)

To deal with the ever-increasing data volume over the network, many algorithms have been proposed to improve the performance of variant network equipment. Usually, the network equipment has to inspect all incoming packets and compares the packets with pre-defined data to find a match or multiple matches. Tri- and automaton-based lookup algorithms have been proposed and widely used in these network applications.

For example, many IP-lookup algorithms use multi-bit tries on longest prefix matches to speed up searching time, such as Lulea algorithm [40] and Etherton algorithm [41] which is now reaching wide deployment in routers. Aho-Corasick (AC) algorithm, an automaton-based algorithm, is a fast multi-pattern matching algorithm [1]. AC algorithm has the best worst-case computational time complexity, and thus it has been modified for intrusion detection systems (IDSs) and network content searching engines [18], [34]. Additionally, deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs) are often employed in regular expression matching and deep packet inspection [26], [42]. While tri- and automaton-based schemes are utilized in different applications, they are very much analogs of one another in that both need similar data structures. To implement these algorithms on real-life appliances, an efficient structure is the most essential part to the performance of appliances. However, the existing algorithms usually did not consider the implementation issues.

Furthermore, as many critical and personal data are accessible on the Internet, people demand more secure networks and systems. Intrusion Detection Systems (IDSs) are one

of the most useful tools to identifying the malicious attempts and protecting the systems without modifying the end-user software. Different from firewalls that only checks specified fields of the packet *headers*, IDSs detect the malicious information in the *payloads*. IDSs must be capable of real-time packet analyzing even when suffering serious attacks; otherwise the protectorate will not be defended strictly. Many studies recently have aimed for upgrading the performance and accuracy of IDSs.

As mentioned previously, the required memory capacity of the existing multi-pattern matching algorithms for Snort's database is usually larger than 300 KB. The number of patterns is still growing. Therefore, an IDS requires a *pattern detection engine* capable of in-depth packet inspection. Without exception, the most essential technology of a detection engine is a powerful multi-pattern matching algorithm.

Many multi-pattern matching algorithms have been proposed, and most of them are filter-based searching algorithms, such as BM-based algorithms, WM, WM-PH, and BF-based algorithms. However, in these filter-based algorithms, if there is a match in the pre-filters, the exact string matching (usually using *sequential search*) in the second stage is also required. Furthermore, the performance of these algorithms decreases while the number of patterns increases. Consequently, these algorithms have bad worst-case performance.

Guaranteed performance is very important especially for the equipment in the core and edge network. The Aho-Corasick algorithm (AC) had the best worst-case computational time complexity, where the number of state transitions for each input symbol is at most two [1], [19]. However, as for realistic implementations, the performance of an algorithm is not only affected by the computation time, but also

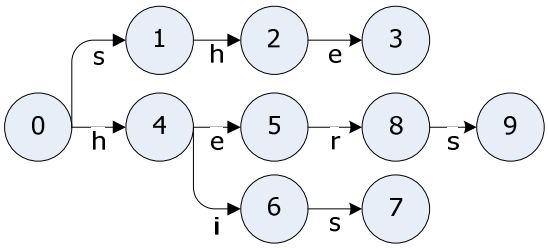
strongly affected by the number of required memory references. The off-chip memory reference costs about 80 ~ 200 clock cycles and the gap may keep increasing [19]. Because of requiring large memory space, the AC needs frequent off-chip memory references and then results in poor performance. Tuck et al modified the AC with a compressed data structure, which reduced the memory size, but also increased the processing time [34].

Therefore, this study proposes a practical multiple-pattern matching algorithm that has better worst-case performance as well as smaller required memory, called ACM. The proposed ACM uses Magic Structures based on the property of Chinese Remainder Theorem, and contributes modifications to the AC algorithm for fast in-depth packet inspection.

5.1 Previous Works

5.1.1 The Aho-Corasick Algorithm (AC)

The Aho-Corasick algorithm (AC) provided the best worst-case computational time complexity. AC is an automaton-based algorithm. There are three functions in the AC: Goto(*st*, *code*), Fail(*st*) and Output(*st*), where *st* is the state identification and the *code* is a scanned character. In other words, the Goto function is a state transition function, which is constructed by a set of patterns (or keywords): ***P***. The Goto function maps a pair (*st*, *code*) into a state or a *fail* message. In the state machine, every prefix of the patterns is only represented by one state. The Fail function points to a next state that is the longest suffix of the current state. The Output function stores the matched patterns belonged to ***P*** corresponding to the current state. These three functions are constructed off-line and will be used in the in-line matching stage.



(a) Goto function.

st	0	1	2	3	4	5	6	7	8	9
Fail	0	0	4	5	0	0	0	1	0	1

st	Output
3	she, he
5	he
7	his
9	hers

(b) Fail and Output Function

Figure 31. The Aho-Corasick algorithm.

Figure 31 shows the Goto, Fail, and Output functions of the AC algorithm with a pre-defined pattern set P = {*she, he, his, hers*}. In the matching stage, given an input T = '*sihe*' for example, the matching procedure scans one character at a time and starts from the rooted state of the automaton, say state 0. Since, Goto(0, *s*) = 1, the machine goes to state 1 and read the next character '*i*'. Because '*i*' is not an expected character in the state 1 (Goto(1, *i*) = *fail*), the Fail function is called and get Fail(1) = 0. Then the machine goes to the state 0, and read the next character '*h*'. As Goto(0, *h*) = 4, and with the same steps as before we get Goto(4, *e*) = 5, the state machine goes to state 5 and have a valid output value: Output(5) = {*he*}. As a result, we can know that the input T contains one pattern '*he*'. This example illustrates how the AC matching algorithm works.

5.1.2 The Basic Implementation of the Aho-Corasick Algorithm

Tries and automata have the same architecture that a parent node has several paths to its child nodes. In both tries and automata, since the next node only depends on the current node and the current input while traversing the graph, we can simply consider only one set

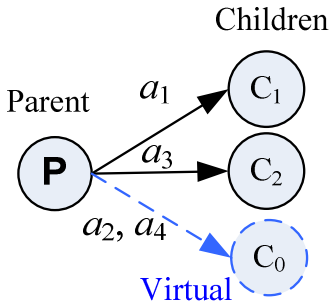


Figure 32. A parent-child set.

at a time: one parent and all of its child nodes (Figure 32). Assume the input is drawn from a set $\Lambda = \{a_i \mid i = 1,2,..,2^n\}$, where 2^n is the number of paths and n is the bit-width of the input code (the stride size of tries and automata).

A simple data structure to implement tries and automata is to save *all* pointers of child nodes in the parent node. When the branch size is four, it means every node in tries and automata needs four pointers to indicate its four child nodes while using the simple structure. The space complexity of the simple structure is $O(2^n)$. Since the paper [1] mentioned that the Goto and Fail functions could be combined into one next function: $\delta(st, code)$, the basic original data structure ACO is shown as follows.

```
struct ACO{
struct ACO *next_state[|Λ|];
struct Result *pattern_list;
};
```

The pointer *next_state* will indicate the address of the next state directly and the pointer *pattern_list* is the memory block storing corresponding patterns of this state. Since every next pointer is saved in a state, the address of the next state can be obtained directly and only one memory reference is required per character.

However, in a system with 32-bit pointers and $|\Lambda| = 256$, the ACO structure requires 1028 bytes per state. For example, retrieving 1200 distinct patterns from the Snort rule database, 10213 states are built in the AC algorithm, which means about 10 MB is required for the state machine. Moreover, the size of the commercial on-chip memory capacity to date is only several kilobytes in the microprocessors and Application Specific Integrated Circuits (ASICs), and 128–512 KB in the general CPUs. Although a few high-end ASICs providing large embedded memory are available, linking many memory blocks degrades the chip performance and poses challenges to power consumption issues. To build a full graph for a detection rule database usually requires large amounts of memory. Hence external memory is required in this case. As mentioned previously, the time to read data from external memory is very long and indeterminist. A compact and efficient data structure is essential to tries and automata.

5.1.3 The AC Algorithm with Bitmap (ACB)

Tuck et al proposed a data structure with a bitmap for the AC algorithm, named ACB in short, to compress the nodes in the state machine [34].

```
struct ACB {
bitmap next_flag[|Λ|];
struct ACB *fail_ptr;
struct ACB *next_start;
struct Result *pattern_list;
};
```

The state machine is still constructed based on the Goto and Fail functions of the AC algorithm. The bitmap *next_flag*[*a*] indicates whether there is a valid forwarding path for the given character *a* (in other words, Goto(*st*, *a*)≠*fail*). Thus, if Goto(*st*, *a*)≠*fail*, the *next_flag*[*a*] will be set as one; otherwise, it means the next state will traverse along the

```
    Procedure ACB_Matching
    Input: A string: T, the starting pointer of the ACB state machine: State.
    Output: The matched pattern set T: P_M.
1   Initialize: P_M ← ∅ .
2   For each input character: InCode ← T[i] do
3     If State->next_flag[InCode] is set then
4       pop_count ← 0 and j ← 0;
5       While j<InCode do
6         pop_count ← pop_count+State->next_flag[j];
7       End
8       State ← State->next_start + pop_count*Sizeof(ACB);
9       P_M ← P_M ∪ Out(State->pattern_list);
10    Else
11      State ← State->fail_ptr;
12  End
13  Return;
```

Figure 33. The ACB_matching Procedure

Fail function, and the bitmap *next_flag*[*a*] will be set as zero. This structure can successfully reduce the memory requirement to only 44 Bytes for each state (on a 32-bit pointer system and $|\Lambda| = 256$).

However, there is only one pointer, *next_state*, to indicate the address of the first valid next state. To obtain the address of a valid next state with a given character *a*, the matching process has to scan ***every*** bits in the bitmap *next_flag* prior to *a* and accumulates the number of valid prior bits. The accumulation routine is called “popcount”. The matching procedure using the ACB structure is shown in Figure 33.

The accumulation routine in lines 5-8 of the ACB_matching procedure is very time consuming. In the worst case, it costs $|\Lambda|$ bit-access and $|\Lambda|$ adds for each input character. Tuck et al admitted that the *popcount* is very expensive for software implementation. Although in the hardware implementation the *popcount* may have the opportunity to be optimized, the complexity and cost are still high.

Therefore, an efficient function to calculate the address of the next node is required. Consequently, this stidy will focus on providing an efficient data structure which has compact memory without sacrificing processing time.

5.2 The ACM Algorithm

Taking the commercial hardware/software constraints into account, this study proposes an efficient data structure, based on Chinese Remainder Theorem, named *Magic Structure*. The Magic Structure is suitable for both tri- and automaton-based lookup schemes. The Magic Structure needs only a small amount of memory and also reduces the number of external memory accesses, when compared with conventional data structures. Therefore, the performance of network equipment using the tri- and automaton-based algorithms can be efficiently improved.

Generally most nodes of tries and automata have only a few valid child nodes. Hence, allocating continuous memory only for the existing child nodes is much more efficient than for all child nodes. Additionally, assume that we can find a simple *magic function*, say $\Re$, so that the corresponding child nodes can be found very fast according to the inputs. As for the invalid input that does not have a valid path, a *virtual node* is assigned. That is, the function $\Re$ of Figure 32 is

$$\{a_1, a_2, a_3, a_4\} \xrightarrow{\Re} \{1,0,2,0\}, \qquad (5\text{-}1)$$

where a_i are the input codes and the identification of the virtual node is zero. Assume there is a *magic number* χ and define the magic function $\Re$ as

$$\Re_i : \chi \% f(a_i) = k, \qquad (5\text{-}2)$$

where f is a function that maps the code set Λ into a numerical set, n % m returns the remainder when n is divided by m, and k is the index number of a child node (C_k). In other words, $\Re$ acts as a path decoder that returns the correct next node for each input. Thus, if we can prove that the magic number χ exists, we can obtain the $\Re$.

5.2.1 Chinese Remainder Theorem

Because $\Re$ needs only one simple modulo operation, the path traversing process will go fast. It is interesting that the famous ***Chinese Remainder Theorem*** **(CRT)** can be applied here for this purpose [43]. The theorem is as follows:

Chinese Remainder Theorem (CRT). Let $M = \prod_{i=1}^{k} m_i$, where m_i are integers and relatively prime; that is, $gcd(m_i, m_j) = 1$ for $1 \le i, j \le k$, and $i \ne j$.[2] Let $x_1, x_2, ..., x_k$ be integers. Consider the system of congruences:

$$X \equiv x_1 (\mathrm{mod}\, m_1)$$
$$X \equiv x_2 (\mathrm{mod}\, m_2) \qquad (5\text{-}3)$$
$$...$$

[2]The $gcd(a, b)$ means the greatest common divisor of a and b.

$$X \equiv x_k \,(\mathrm{mod}\, m_k)\,,$$

where X and x_i are said to be congruent modulo m_i, $1 \le i \le k$. Then there exists exactly one X and $X \in \{0,1,...,M-1\}$. ■

Therefore, if let the function f number the symbols by prime numbers, that means $\{a_1, a_2, ..., a_k\} \xrightarrow{f} \{m_1, m_2, ..., m_k\}$, then by CRT we know the magic number χ exists. χ is now the X in CRT. Since f is one-to-one mapping, a *Prime* table can be used to store the prime number for each possible input symbol. The *Prime* table has at most $|\Lambda|$ entries, and so that it is very small and can be kept in the on-chip cache. Thus the prime number of an input symbol can be obtained by a fast lookup. To obtain the magic number χ, the following algorithm is applied.

Chinese Remainder Theorem Algorithm. Let $z_i = M/m_i$ and $y_i = z_i^{-1} (\mathrm{mod}\, m_i)$ for each $i = 1, 2,..., k$, where z_i^{-1} means the multiplicative inverse of z_i. (Note that z_i^{-1} exists if gcd(z_i, m_i) = 1.) Then the solution to the congruence system of the Chinese Remainder Theorem is

$$X = (\sum_{i=1}^{k} x_i y_i z_i) \bmod M \,. \qquad (5\text{-}4)$$

■

For example, assume the inputs {a_1, a_2, a_3, a_4} in Figure 32 maps to the relatively prime set {2, 3, 5, 7}. We want to find a magic number χ that satisfies χ% 2 = 1, χ% 3 = 0, χ % 5 = 2, and χ% 7 = 0. Then, we get χ= 147.

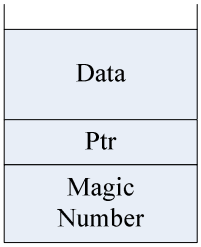


Figure 34. Magic structure.

5.2.2 The Magic Structure

Based on CRT, a *Magic Structure* (MS) is defined as shown in Figure 34, including a pointer to the first child node of this set (*Ptr*) and a magic number χ in addition to the data in a node. The address of the next node (*next_ptr*) for the input a_i can be fast obtained by

$$next\_ptr = \begin{cases} \text{Null} & ,\text{if } \chi \% f(a_i) = 0; \\ Ptr + (\chi \% f(a_i) - 1) \times \text{sizeof\_MS} & ,\text{if } \chi \% f(a_i) \neq 0, \end{cases} \quad (5\text{-}5)$$

where sizeof_MS is the size of the Magic Structure, which is the size of *one* pointer plus $\log_2 \chi$ in addition to the size of data in a node. The size of MS is much smaller than that of the simple structure of 2^n pointers.

Furthermore, MS has a special property: if a node has only one child, then the magic number χ will be one. Observing most tri- and automaton-based algorithms, we find that while approaching the leaf nodes, more and more nodes have only one child. Therefore, to improve the performance, a simple check on χ is done before operating *next_ptr*. If χ=1, *next_ptr* is *Ptr*. The next node can be obtained directly without computing.

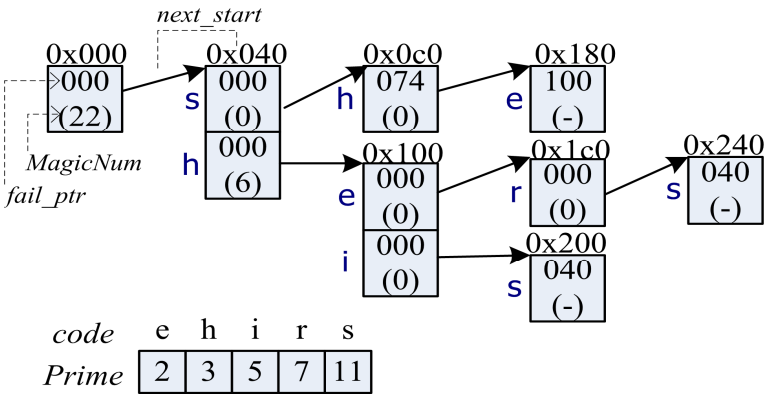


Figure 35. The architecture of ACM state machine, where the number in the parentheses is the magic number.

5.2.3 AC with Magic Structures

In this section, we will show a case of using AC algorithm with Magic Structures for multi-pattern matching. Modifying the original Magic Structure and adding some required fields, the data structure for AC, named ACM, is proposed as follows:

```
struct ACM{
bitmap next_flag[|∧|];
struct ACM *fail_ptr;
struct ACM *next_start;
struct Result *pattern_list;
long_int MagicNum;
};
```

In the structure ACM, a bitmap *next_flag* is used for fast checking whether there is a valid child. To reduce the size, only one pointer *next_start* pointing to the first valid child state is stored in the data structure. The *MagicNum* stores the magic number χ. The ACM state machine is organized based on the Goto and Fail functions of the AC algorithm. Figure 35 illustrates the memory organization of the ACM state machine when using the same example shown in Figure 31(a), and the prime numbers for possible input codes are

```
    Procedure ACM_Matching
    Input: A string: T, the starting pointer of the ACM state machine: State, and an array with
    prime numbers: Prime.
    Output: The matched pattern set T: P_M.
1   Initialize: P_M ← ∅.
2   For each input character: InCode ← T[i] do
3     If nextState->next_flag[InCode] is set then
4       If nextState->MagicNum = 0 then
5         nextState ←nextState->next_start;
6       Else
7         nextState ←nextState->next_start +
    ((nextState->MagicNum)%Prime[InCode])*Sizeof(ACM);
8       P_M ← P_M ∪ Out(nextState->pattern_list);
9     Else
10      nextState ←nextState->fail_ptr;
11  End
12  Return;
```

Figure 36. The matching procedure using the ACM structure.

also listed. Note that when there is no valid child for the leaf nodes, the magic numbers of the leaf nodes are labeled NULL.

Note that since a bitmap *next_flag* is used for fast checking in ACM, the mapping organization is slightly modified, and virtual node is not used. Thus the first valid child node will have $\Re = 0$. Using Figure 35 as an example, the *MagicNum* of the root state (state 0) has to satisfy that {'*s*', '*h*'}→{0, 1}, where the prime numbers for '*s*' and '*h*' are 11 and 3, respectively. This means *MagicNum* % 11 = 0, and *MagicNum* % 3 = 1. Then the *MagicNum* is 22.

The matching procedure using the ACM structure is illustrated in Figure 36. In the ACM matching, given an input symbol *a* for example, if *next_flag*[*a*] is not flagged, then the machine traverses the pointer *fail_ptr* until a state has a flagged *next_flag*[*a*] or the machine returns to the root state. If the machine traverses to the root state and the *next_flag*[*a*] is not flagged, the machine will stop in the root state and read the next

symbol. Otherwise, while *next_flag*[*a*] is flagged, slightly modifies the Equ.(5-5), the pointer to the next state is

$$nextState = next\_start + \Re \times \text{sizeof\_ACM} \quad (5\text{-}6)$$

$$= next\_start + (MagicNum \ \% \ Prime[a]) \times \text{sizeof\_ACM},$$

where sizeof_ACM is the structure size of ACM, $\Re$ is the offset to the first valid next state (*next_start*), and *Prime* is a small on-cache table keeping the prime number for each possible input character. Obviously, only three reads (*next_start*, *MagicNum* and *Prime*[*a*]) and three operations are required for indicating the next state. According to lines 7-10 of the ACM matching procedure in Figure 36, the worst case cost is three read sand three operations. As the number of fail transitions is never more than the depth of a state, the number of state transitions for each input symbol will be at most two. Consequently, the cost of fail transition is small.

Due to the definitions of $\Re$ and CRT, ACM matching has a special property as mentioned before: if there is only one child, *MagicNum* will be zero. Observing the ACM state machine, we can find that approaching the leaf nodes, more and more states have only one child state. Therefore, this heuristic can be used in the ACM matching to reduce the computation. That is, if the *next_flag*[*a*] is set and the *MagicNum* in the current state is zero, then there is only one child state and the pointer to the next state for the symbol *a* is *next_start*. The forwarding path can be obtained directly without computing the $\Re$.

The ACM structure needs only 52 bytes for each state when the size of magic number is 8 bytes, which is much smaller than the ACO structure of 1028 bytes, and so that it successfully reduces the memory requirement. Additionally, the state transition time will be fast because of the simple path decoder $\Re$ and the magic number heuristic.

Figure 35 illustrates the ACM matching, and assumes that the input string is '*ish*'. Scan the string from left to right, and start from the root state at 0x000. As the bitmap *next_flag*['*i*'] is not flagged, the machine stays in the state at 0x000. Reading the next symbol '*s*', the process finds that it is flagged, and then gets *MagicNum* = 22 in the state 0x000. As *Prime*['*s*'] = 11 and the *next_start* of the state 0x000 is 0x040, the address of the next state for '*s*' can be calculated by 0x040+(22%11)×0x34 = 0x040, where the size of ACM is 52 bytes (0x34). Then the machine goes to the state at 0x040 and checks the bitmap for the next symbol '*h*'. Since the *next_flag*['*h*'] is flagged and the *MagicNum* is zero, ACM matching knows that it is the only child and the address of the next state is 0x0c0, which is read directly from the *next_start* of the state 0x040.

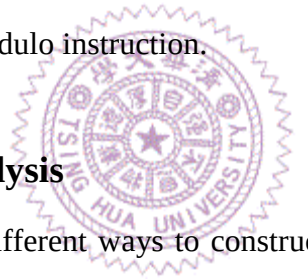
5.2.4 Implementation Issues

According to the magic number definition and the CRT theorem, it is noted that if there are too many child states and the alphabet set is large, the magic number will be a large number. In the hardware implementations, this is not a problem. Many papers have proposed optimized hardware designs for high performance modular arithmetic with long operands, which can archive one operation per clock cycle [44]. Therefore, the ACM matching algorithm can be easily implemented in the hardware and gain high performance.

However, in the software implementations, software has its limitation on the length of an operand. Two mechanisms can be employed to overcome this: (a) use a bitmap check before $\Re$; (b) if the magic number is still too large, then use *running sums* in partial nodes. In the first method, the bit of a valid input is set; otherwise, it is not. Then only the valid inputs have to find their next nodes. This is the method that used in the case

study: ACM matching. The invalid inputs pointing to the virtual node are not involved in calculating the magic number. For example, in the Figure 32 only $\{a_1, a_3\} \rightarrow \{1, 2\}$ are used in calculating χ and get $\chi = 7$, which becomes much smaller.

In the second method, the running sum scheme is employed instead of $\Re$ in partial nodes. A *union structure* is used here and then eight running sums and the 64-bit magic number share the same memory space. Fortunately, in the case of importing 1200 distinct patterns from the Snort database, only 0.078% states of the state machine has to use the running sum scheme. Another issue of implementing ACM on some general processors is sometimes the expensive cost of modulo operations. This study will show the simulation results later and illustrate that ACM outperforms ACB even when running on a general processor without optimized modulo instruction.



5.3 Performance Analysis

Different algorithms use different ways to construct and traverse their graphs, but while in searching processes, they all need to calculate the address of the next node, which is a required cost. Based on the heuristic of the magic number and the magic structure, the average time to calculate the address of the next node (average *addressing time*, T_{next}) is

$$T_{next} = \alpha T_{\chi=1} + (1-\alpha) T_{\chi\neq 1} + T_L , \qquad (5\text{-}7)$$

where α is the probability that a parent node has only one child node, $T_{\chi=1}$ and $T_{\chi\neq 1}$ are the time spent on getting the address of the next node when $\chi = 1$ and $\chi \neq 1$ respectively, and T_L is the time of one lookup to check the magic number. When $\chi = 1$, it implies that the current node has only one child node, and the address of the next node is *Ptr*. Thus,

$$T_{\chi=1} = T_{read}, \tag{5-8}$$

where T_{read} is the time of one read. If $\chi \neq 1$, the pointer of the next node is calculated by Equ. (5-5). Hence,

$$T_{\chi\neq1} = T_{\mathrm{mod}} + T_{add} + T_{sub} + T_{mul} + T_L + 3T_{read}\ , \tag{5-9}$$

where T_{mod}, T_{add} and T_{mul} are the time of one modulo, add and multiply operation respectively, and T_L is the time of one lookup on the direct mapping table (*f*). It requires three reads while reading operands from data structures or registers is considered. When the structures can be kept on cache, T_{read} is very small. Assume that T_{add}, T_{sub}, T_{read}, T_L, and T_{mul} each needs τ cycle time since the multiplication in Equ. (5-5) is a simple constant multiplication, and T_{mod} needs $b\tau$ cycles. b is very small because the optimizations for modulo operations in hardware and software have been proposed in many articles [44]. By substituting Equ. (5-8) and Equ. (5-9) into Equ. (5-7), the average addressing time is

$$T_{next} = (b+8)\tau - (b+6)\alpha\tau\ . \tag{5-10}$$

In the worst case, the addressing time is

$$\max\{T_{next}\} = T_{\chi\neq1} + T_L = (b+8)\tau\ . \tag{5-11}$$

As a bitmap is involved in MS, the average addressing time is

$$T_{next}^{B} = \beta T_{next} + T_L = (1+(b+8)\beta)\tau - (b+6)\alpha\beta\tau\ , \tag{5-12}$$

where β is the probability that the current code exists a path for a certain input. For a sparse graph, α and β are very small. The addressing time of Tuck and Lulea algorithms which used popcount to obtain the offset of a child node is

$$T_{next}^{pop}(a_k) = \sum_{i=1}^{k}(T_{add} + T_{read}) + T_{mul} + T_{add} + T_{read} + T_L\ , \tag{5-13}$$

Table 16. The memory size (in Bytes) of a node for path traversing using simple structure, Bitmap structure, and MS plus bitmap.

2^n	Valid Children	Simple	Bitmap	MS+bitmap
8	4	32	5	7
16	8	64	6	12.25
64	16	256	12	28.125
256	16	1024	36	56.875

where a_k is the current input code. The worst case of T_{next}^{pop} is

$$\max\{T_{next}^{pop}\} = 2^n \times 2\tau + 4\tau = (2^{n+1} + 4)\tau \,, \qquad (5\text{-}14)$$

when the input code is n-bit coding. Compared Equ. (5-11) with Equ. (5-14), we can see that MS outperforms BM structure, while n is usually larger than two and b is very small. Notably, T_{next}^{pop} increases exponentially. Consequently, MS performs much better than BM structure, especially for the algorithm that has a sparse graph or that uses larger strides to reduce the searching depth.

5.4 Results and Discussions

Firstly, Table 16 lists the required memory of simple structure, bitmap structure (BM) and MS plus bitmap for path traversing in a 32-bit addressing system. This shows only the memory for node addressing, excluding the data for algorithms themselves. Recall that the simple structure has to save all pointers to child nodes, and the bitmap structure uses an n-bit bitmap and one pointer that indicates the starting address of the first child node. Table 16 shows that MS needs only a small amount of memory. Although MS requires a little larger memory than BM structure, it is still small enough to be stored in the cache memory.

To show the performance of these structures on a real system, we implement an IDS of 1200 rules using the AC algorithm with different data structures. In the following simulations, with detachment we use the free and real pattern set released by Snort [1] Since the patterns of Snort are written in mixed plain text and hex formatted bytecodes, we assume that the alphabet size ($|\Lambda|$) is 256 in the simulations.

To evaluate the performance of algorithms in a real intense attack, we use a trace from the Capture-the-Flag contest held at the Defcon9 as the input streams of the programs. The Defcon Capture-the-Flag contest is the largest security hacking game, which tries to break into the servers of others while protecting your own server hiding several security holes [14]. In the simulations, we evaluate the performance by calculating the number of instructions used in the algorithms and then multiplying the cost of each instruction. The costs of the instructions refer to a real AMD processor [45], where the number of instructions per clock for add, mov, mul, cmp, bt, and mod is 3, 3, 1, 3, 3, 1/71 respectively, and the operation cost of mod is high.

Let C_M represent the total memory requirement and C_T be the average execution time. Figure 37 and Figure 38 show C_M and C_T for the ACM, ACB and ACO matching respectively in the case of 200 patterns and 1200 patterns. Note that the C_M of ACM includes the memory requirement of the *Prime* table. We can see that the total memory requirement of ACM is 519.2 KB in the case with a big pattern set $|\boldsymbol{P}| = 1200$, which is only 5.1% of the basic AC and a little (18%) more than that of ACB. Furthermore, the memory size of ACM is still in the scale of the on-chip cache that general chipsets support. Therefore, we can say that the ACM can be easily implemented in the hardware and software, and can gain high performance due to no off-chip memory access.

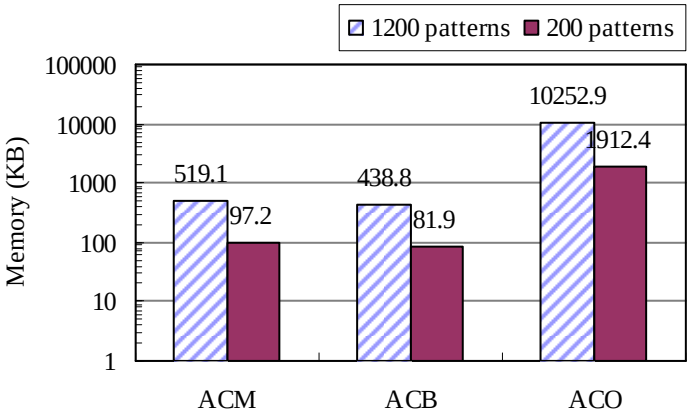


Figure 37. The total memory requirement for the ACM, ACB and ACO structures in the case of 1200 and 200 patterns respectively.

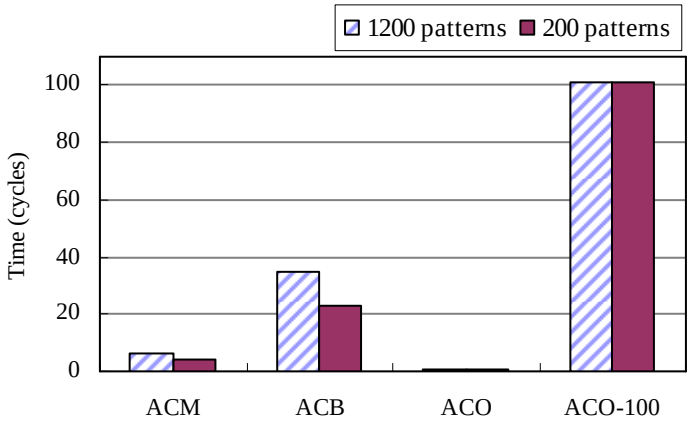


Figure 38. The average execution time per symbol of ACM, ACB, and ACO matching in the case of 1200 and 200 patterns respectively.

To date, the largest size of on-chip memory supported by the FPGAs is about 1 MB and the size of L1 cache and L2 cache of general processors is only 128 KB ~ 2 MB [22]. Therefore, according to the memory requirement shown in Figure 37, the full state machine of ACO can not be stored in the on-chip memory. The external memory references are required in the ACO matching draws the average execution time per byte of ACM, ACB and ACO matching respectively in the case of importing 200 and 1200 patterns. There are two cases for ACO matching: the result labeling ACO is not assessed

Table 17. The normalized cost of ACM, ACB and ACO in the case of 200 and 1200 patterns.

	C/C^{ACM} (Num. of Patterns = 200)	C/C^{ACM} (Num. of Patterns = 1200)
ACM	1	1
ACB	4.492	4.795
ACO	2.141	3.048
ACO-100	323.306	460.197

any latency penalty for the external memory references, and the other one labeling ACO-100 needs 100 cycles for each external fetch.

Figure 38 shows that ACM performs about 4.67 times better than ACB in the case of 1200 patterns, and 4.34 times over ACB in the case of 200 patterns. Comparing ACM with ACO and ACO-100, we can see that ACM outperforms ACO-100 and the external memory references drastically affect the performance of ACO matching. Note that the cost of modulo operation in the simulations is extremely higher than others. Even assessed the penalty of high operation cost, ACM still outperforms ACB and is moderately slower than ACO. If implemented in embedded systems or FPGAs, ACM will be more efficient.

As the required time and memory are usually trade-off, to compare the overall costs of these three algorithms, we define an evaluation function C: $C = C_M \times C_T$. The higher C means the more cost is required in the implementations. The total cost for ACM, ACB and ACO is labeled C^{ACM}, C^{ACB}, and C^{ACO} respectively.

For easy comparison, we show the normalized cost ($=C/C^{ACM}$) of each algorithm in Table 17.

Table 17 demonstrates that the cost of ACM is smaller than others. Even requiring a little more memory than ACB, ACM has better overall efficiency, which is about 3.4–3.7 times better than ACB. Although the theoretic execution time of ACO is shorter than that

of ACM, the overall cost of ACM is about 1.1–2 times smaller than ACO. For realistic implementations, we can see that the overall cost of ACM is about 322–459 times better than that of ACO-100. Therefore, we can say that ACM is a time- and memory-efficient algorithm for IDSs, and the Magic Structure is efficient for the automaton-based algorithms.

6 CONCLUSIONS AND FUTURE WORKS

The increasing variety of network applications and stakes held by various users are creating a strong demand for fast in-depth packet inspection. The most important component of in-depth packet inspection is an efficient multi-pattern matching algorithm. This study has proposed three novel multi-pattern matching algorithms for network content inspection: a *hierarchical multi-pattern matching algorithm* (HMA), an *enhanced hierarchical multi-pattern matching algorithm* (EHMA), and an *Aho-Corasick with Magic Structures* (ACM) algorithm. HMA and EHMA have better average-case performance, while ACM has better worst-case performance than the state-of-the-art algorithms. This study also has discussed and evaluated current multi-pattern matching algorithms for NIDSs.

HMA applies the most frequent-common *codes* to quickly filter out innocent packets, and to reduce memory accesses. The frequent-common codes are used to build small hierarchical index tables for simple and fast checks. The hierarchical scheme improves the matching performance significantly by reducing the average number of external memory accesses to only 10%–37%. The required memory of HMA is only about 350 KB including the 1200 patterns of Snort rules. Particularly, HMA use simple architecture and functions, and it can be easily implemented in both software and hardware. Simulation results have shown that HMA performs about 0.9–409 times better than others. HMA significantly improves the best-case and average-case performance, and also provides moderately worst-case performance of the multi-pattern matching. Moreover, an incremental pattern update mechanism has also proposed for HMA.

Improving HMA, EHMA applies the frequent-common *grams* obtained by the proposed GFGS to narrowing the searching scope and to quickly filtering out the innocent packets. The matching process then focuses only on the most suspected packets. EHMA concentrates the patterns into a small on-chip table, and performs simple and fast checks. Additionally, EHMA uses the frequency-based bad gram heuristic to speed up the scanning process. The hierarchical matching significantly reduces the average number of external memory accesses to only 6%–19%, thus improving the matching performance. The required memory of EHMA is only about 40KB in additional to the pattern contents of Snort rules. Particularly, EHMA is very simple and can be easily implemented in both software-based and hardware-based platforms. Simulation results have shown that EHMA performs about 0.89–1161 times better than others. Even under real-life intense attack, EHMA significantly outperforms others. EHMA also works well for the systems with larger minimum pattern size, such as virus detection systems. Consequently, HMA and EHMA facilitate the creation of efficient and cost-effective packet inspection engines.

In this study, an efficient Magic Structure (MS) for multi-pattern matching algorithms has been proposed in ACM, and the proposed algorithm ACM contributes better worst-case performance of pattern detection for NIDSs. The MS is based on an idea behind congruence systems, and uses a magic number derived from Chinese Remainder Theorem. The analysis and simulation results have shown that ACM can efficiently reduce the required amount of external memory access. ACM is an automaton-based algorithm, and it features fast traversing between the nodes in the state machine. Furthermore, ACM uses only simple instructions other than specific operations or

hardware. Therefore, ACM can be easily implemented in hardware and software. The results have shown that ACM outperforms others. The overall cost of ACM is about 1.1–459 times better than the existing implementations. Consequently, ACM enables an efficient IDS that can survive under heavy attacks.

As the proposed EHMA has nice average-case performance and the ACM has good worst-case performance, combining this two algorithms for a powerful and adaptive multi-pattern matching algorithm is worthy of further research in the future. Furthermore, the proposed Magic Structures may be able to apply to different network applications, such as tri-based algorithms and IP lookup algorithms. In this study, the proposed three algorithms are applied to in-depth packet inspection in wired intrusion detection systems. The multi-pattern matching algorithms can also be applied to other research areas, such as wireless network security, searching engines, etc. Therefore, extending the use of the proposed ideas is also a notable issue.

REFERENCES

[1] Snort, http://www.snort.org.

[2] Brian Caswell, Jay Beale, James C. Foster, and Jeremy Faircloth, "Snort 2.0 Intrusion Detection," *Syngress*, Feb, 2003.

[3] CERT/CC. "The Nimda worm has the potential to affect both user workstations (clients) running Windows 95, 98, ME, NT, or 2000 and servers running Windows NT and 2000." *CERT Advisory CA-2001-26*, Sep. 2001.

[4] Spyros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos, "Generating realistic workloads for network intrusion detection systems," *ACM Workshop on Software and Performance*, pp. 207–215, 2004.

[5] Martin Roesch, "Snort – Lightweight Intrusion Detection for Networks," *Proceedings of the 13th Systems Administration Conference*, pp. 229–238, 1999.

[6] Tomoaki Sato and Masa-aki Fukase, "Reconfigurable Hardware Implementation of Host-based IDS," *the 9th Asia-Pacific Conference on Communication*, Vol. 2, pp. 849–853, Penang, Malaysia, Sept. 2003.

[7] Mike Fisk and George Varghese, “Fast Content-Based Packet Handling for Intrusion Detection,” *UCSD Technical Report CS2001-0670*, May 2001.

[8] Alfred V. Aho and Margaret J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, Vol. 18, Np. 6, pp. 330–340, June 1975.

[9] Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun, Larry Huston, and Uday Naik, "Network Processor Perfromance Analysis Methodology," *Intel Technology Journal*, Vol. 6, Aug. 2002.

[10] Ricardo A. Baeza-Yates, "Improved String Search," *Software – Proctice and Experience*, Vol. 19, No. 3, pp. 257–271, March 1989.

[11] Spyros Antonatos, Michalis Polychronakis, P. Akritidis, Kostas G. Anagnostakis, Evangelos P. Markatos, "Piranha: Fast and memory-efficient Pattern Matching for Intrusion Detection," *Proceedings of the 20th IFIP International Information Security Conference* (SEC2005), pp. 393–408, May 2005.

[12] Gordon Brebner and Delon Levi, "Networking on Chip with Platform FPGAs," *Proceedings of 2003 IEEE International Conference on Field-Programmable Technology*, pp. 13–20, Dec. 2003.

[13] Robert S. Boyer and Strother J. Moor, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol. 20, No. 10, pp. 762–772, October 1977.

[14] Crispin Cowan, “Defcon Capture the Flag: Defending Vulnerable Code from Intense Attack,” *Proceedings of DARPA Information Survivability Conference and Exposition, Washington DC*, vol.2, pp. 71–72, April 2003.

[15] C. Jason Coit, Stuart Staniford, and Joseph McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort," *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition*, vol.1, pp. 367–371, 2001.

[16] Thomas H. Cormen, Dartmouth College, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, "Introduction to Algorithms - 2nd Edition," *MIT Press and McGraw-Hill*, Sep. 2001.

[17] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, John lockwood, "Deep Packet Inspection using Parallel Bloom Filters," *11th Symposium on High Performance Interconnects*, pp. 44–51, August 2003.

[18] Sarang Dharmapurikar and John Lockwood, "Fast and Scalable Pattern Matching for Network Intrusion Detection Systems," *IEEE Journal on Selected Area in Communications*, Vol. 24, No. 10, pp. 1781–1792, Oct. 2006.

[19] Ozgun Erdogan and Pei Cao, "Hash-AV: Fast Virus Signature Scanning by Cache-Resident Filters," *Proceedings of IEEE Global Telecommunications Conference*, Vol. 3, St. Louis, MO, Nov. 28, 2005.

[20] Mark Handley, Vern Paxson and Christian Kreibich, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics," *Proceedings of the 9th USENIX Security Symposium*, 2000.

[21] R. Nigel Horspool, "Practical Fast Searching in Strings," *Sofetware Practice and Experience*, Col. 10, No. 6, pp. 501–506, 1980.

[22] Intel Network Processors,

http://www.intel.com/design/network/products/npfamily/index.htm

[23] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," P*roceedings of IEEE Symposium on Security and Privacy*, pp. 285, May 2002.

[24] Sun Kim and Yanggon Kim, "A Fast Multiple String-Pattern Matching Algorithm," *17th AoM/IAoM Interantional Conference on Computer Science*, San Diego, CA, August, 1999.

[25] Vasilios Katos, "Network Intrusion Detection: Evaluating Clusterm Discriminant, and Logit Analysis," Information Sciences 177, pp.3060-3073, 2007.

[26] Hongbin Lu, Kai Zheng, Bin Liu, Xin Zhang, and Yunhao Liu, "A Memory-Efficient Parallel String Matching Architceture for High-Speed Intrusion Detection," *IEEE Journal on Selected Area in Communications*, Vol. 24, No. 10, pp. 1793–1804, Oct. 2006.

[27] Rong-Tai Liu, Nen-Fu Huang, Chih-Hao Chen and Chia-Nan Kao, "A Fast String Matching Algorithm for Network Processor-Based Intrusion Detection System," *ACM Transactions in Embedded Computing Systems*,Vol.3, Issue 3., pp. 614–633, Aug. 2004.

[28] Shaomeng Li, Jim Torresen, and Oddvar Soraasen, "Exploiting Reconfigurable Hardware for Network Security," *Proceedings of the 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 292, 2003.

[29] Evangelos P. Markatos, Spyros Antonatos, Michalis Polychronakis and Kostas Anagnostakis, "Exclusion-based Signature Matching for Intrusion Detection," *Proceedings of IASTED International Conference on Communications and Computer Networks (CCN 2002)*, pp. 146–152, October 2002.

[30] Vern Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, Vol. 31, No. 23-24, pp. 2435–2463, 1999.

[31] Graham A. Stephen, "String Matching Algorithms," *World Scientific* (ISBN 981-02-1829-X), 1994.

[32] Taeshik Shon and Jongsub Moon, "A Hybrid Machine Learning Approach to Network Anomaly Detection," Information Sciences 177, pp. 3799–3821, 2007.

[33] Tzu-Fang Sheu, Nen-Fu Huang, Hung-Shen Wu, Ming-Chang Shih, and Yuang-Fang Huang, "On the Design of Network-Processor-Based Gigabit Multiple-Service Switch," *Proceedings of IEEE ITRE 2005*, Hsinchu, Taiwan, 2005.

[34] Nathan Tuck, Timothy Sherwood, Brad Calder, George Varghese, "Deterministic Memory –Efficient String Matching Algorithms for Intrusion Detection," *Proceedings of the IEEE Infocom Conference*, Vol. 4, pp. 2628–2639, Hong Kong, March 2004.

[35] Vitesse Network Processors, http://www.vitesse.com

[36] Sun Wu and Udi Manber, "A Fast Algorithm for Multi-Pattern Searching," *Tech. Rep. TR94-17, Department of Computer Science, University of Arizona*, May 1994.

[37] Zhenwei Yu, Jeffrey J. P. Tsai and Thomass Weigert, "An Automatically Tuning Intrusion Detection System," IEEE Transactions on Systems, Man and Cybernetics – Part B: Cybernetics, Vol. 37, No. 2, pp. 373–384, April 2007.

[38] Tzu-Fang Sheu, Nen-Fu Huang and Hsiao-Ping Lee, " A Novel Hierarchical Matching Algorithm for Intrusion Detection Systems," *Proceedings of IEEE Global Telecommunications Conference* (*Globecom*), St. Louis, Nov. 2005.

[39] Yuke Wang, "New Chinese Remainder Theorems," *Conference Record of the Thirty-Second Asilomar Conference on Signals, Systems & Computers*. Vol. 1, pp. 165-171, Nov. 1998.

[40] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink, "Small forwarding tables for fast routing lookups," In Proceedings of *SIGCOMM*, pages 3–14, 1997.

[41] W. Eatherton, Z. Dittia, and G. Varghese, “Tree bitmap: Hardware/software IP lookups with incremental updates,” *ACM SIGCOMM Computer Communications Revie*w, 34(2), 2004.

[42] Reetinder Sidhu and Viktor K. Prasanna, "Fast Regular Expression Matching using FPGAs,” *IEEE Symposium on Field-Programmable Custom Computing Machines* (FCCM01), April 2001.

[43] Yuke Wang, “New Chinese Remainder Theorems,” *Thirty-Second Asilomar Conference on Signals, Systems & Computers*. Vol. 1, pp. 165-171, Nov. 1998.

[44] Saman Amarasinghe, Walter Lee, Ben Greenwald, ”Strength Reduction of Integer Division and Modulo Operations,” *M.I.T.*, 1999. http://www.cag.lcs.mit.edu/raw

[45] Torbjorn Granlund. Instruction Latencies and Throughput for AMD and Intel x86 processors. http://swox.com/doc/x86-timing.pdf. Sep. 2005.

TZU-FANG SHEU'S PUBLICATION LISTS

(A) Journal Papers

[1] **Tzu-Fang Sheu**, Nen-Fu Huang, and Hsiao-Ping Lee, "In-depth Packet Inspection Using a Hierarchical Pattern Matching Algorithm," *IEEE Transactions on Dependable and Secure Computing, 2008*. (to appear) (**EI**, **SCI** (2006 IF=1.762), **8/82** in subject categories COMPUTER SCIENCE, SOFTWARE ENGINEERING)

[2] Hsiao-Ping Lee, **Tzu-Fang Sheu**, Yin-Te Tsai and Chuan-Yi Tang, "Finding Homologous Sequences in Genomic Databases," *Journal of Computers- Special issue on Bioinformatics and Computational Biology*, Vol. 18, No. 3, October 2007.

[3] **Tzu-Fang Sheu**, Nen-Fu Huang, and Hsiao-Ping Lee, "A Hierarchical Multi-pattern Matching Algorithm for Network Content Inspection," *Information Sciences*, Mar. 2008. (**EI**, **SCI** (2007 IF=2.147), **10/92** in the subject categories COMPUTER SCIENCE, INFORMATION SYSTEMS)

[4] Shiann-Tsong Sheu and **Tzu-Fang Sheu**, "A bandwidth allocation/sharing/extension protocol for multimedia over IEEE 802.11 ad hoc wireless LANs," *IEEE Journal on Selected Areas in Communications (**JSAC**)*, Oct. 2001, pp. 2065–2080 vol.10. (NSC89-2218-E-032-012) (**EI, SCI**)

(B) Refereed Conference Papers

[5] **Tzu-Fang Sheu**, Nen-Fu Huang, and Hsiao-Ping Lee, "A Time and Memory Efficient String Matching Algorithm for Intrusion Detection Systems," *Proceedings of IEEE Global Telecommunications Conference **(GLOBECOM'06)***, San Francisco, USA, November 2006, pp. 1-5. (NSC-94-2752-E-007-002-PAE and NSC-94-2213-E007-021) **(EI)**

[6] **Tzu-Fang Sheu**, Nen-Fu Huang, and Hsiao-Ping Lee, "A Novel Hierarchical

Matching Algorithm for Intrusion Detection Systems," *Proceedings of IEEE Global Telecommunications Conference **(GLOBECOM'05)***, St. Louis, Missouri, USA, November 2005, Vol. 3, pp. 5-10. (NSC-94-2752-E-007-002-PAE) **(EI)**

[7] **Tzu-Fang Sheu**, Nen-Fu Huang, Hung-Shen Wu, Ming-Chang Shih, and Yuang-Fang Huang, "On the Design of Network-Processor-Based Gigabit Multiple-Service Switch," *Proceedings of 3rd International Conference on Information Technology (ITRE 2005)*, Hsin-Chu, Taiwan, June 2005. (NSC-94-2752-E-007-002-PAE) **(EI)**

[8] Hsiao-Ping Lee, **Tzu-Fang Sheu**, Yin-Te Tsai, Chin-Hua Shih and Chuan-Yi Tang. "Efficient Discovery of Unique Signatures on Whole-genome EST Databases," *Proceedings of the 20th ACM Symposium on Applied Computing **(SAC 2005)***, pp. 100-104. **(EI)**

[9] Hsiao-Ping Lee, Yin-Te Tsai, Chuan-Yi Tang, Ching-Hua Shih and **Tzu-Fang Sheu**, "A Seriate Coverage Filtration Approach for Homology Search," *Proceedings of the 19th ACM Symposium on Applied Computing **(SAC 2004)***, pp. 180-184. **(EI)**

[10] Hsiao-Ping Lee, **Tzu-Fang Sheu**, Yin-Te Tsai, Chin-Hua Shih and Chuan-Yi Tang, "An Efficient Algorithm for Unique Signature Discovery on Whole-Genome EST Databases," *Proceedings of the 3rd IEEE Computational Systems Bioinformatics Conference **(CSB2004)***, pp. 650-651. **(EI) (**ISBN13: 9780769521947**)**

[11] Hsiao-Ping Lee, Yin-Te Tsai, Ching-Hua Shih, **Tzu-Fang Sheu** and Chuan-Yi Tang, 'An IDC-based Algorithm for Efficient Homology Filtration with Guaranteed Seriate Coverage.' *Proceedings of the IEEE Fourth Symposium on Bioinformatics and Bioengineering (BIBE2004)*, page 395-402, 2004. **(EI) (**ISBN13:9780769521732**)**

[12] Hsiao-Ping Lee, Yin-Te Tsai, Ching-Hua Shih, **Tzu-Fang Sheu** and Chuan-Yi Tang,

'A Novel Approach for Efficient Query of Single Nucleotide Variation in DNA Databases.' *The Eighth Annual International Conference on Research in Computational Molecular Biology* ***(RECOMB 2004)***, Poster, San Diego, March, 2004. **(EI)**

[13] Nen-Fu Huang, Han-Chieh Chao, Reen-Cheng Wang, Whai-En Chen, and **Tzu-Fang Sheu**, "The IPv6 deployment and projects in Taiwan," 2003 *IEEE Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, Jan 2003, pp. 157–160. (NSC-90-2219-E-007-001 and MOE 89-E-FA04-1-4)

[14] Shiann-Tsong Sheu, **Tzu-Fang Sheu**, Chih-Chiang Wu, and Jiau-Yu Luo, "Design and implementation of a reservation-based MAC protocol for voice/data over IEEE 802.11 ad-hoc wireless networks," *Proceedings of the IEEE International Conference on Communications* ***(ICC)***, June 2001, pp. 1935–1939, vol.6. (ISSN 05361486) **(EI)**

[15] Shiann-Tsong Sheu and **Tzu-Fang Sheu**, "DBASE: a distributed bandwidth allocation/sharing/extension protocol for multimedia over IEEE 802.11 ad hoc wireless LAN," *IEEE Proceedings of Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies* ***(INFOCOM)***, April 2001, pp. 1558–1567 vol.3. (ISSN 0743166X) **(EI)**

[16] Shiann-Tsong Sheu and **Tzu-Fang Sheu**, "A hybrid data/header interleaving strategy for wireless ATM networks," *IEEE 1999 2nd International Conference on ATM (ICATM '99)*, June 1999, pp. 1-6.