This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE GLOBECOM 2005 proceedings.

A Novel Hierarchical Matching Algorithm for Intrusion Detection Systems

Tzu-Fang Sheu[+], Nen-Fu Huang[*,+] and Hsiao-Ping Lee[*]

[+]Institute of Communication Engineering, National Tsing-Hua University, Taiwan
[*]Department of Computer Science, National Tsing-Hua Universuty, Taiwan
{sunnie@ieee.org, nfhuang@cs.nthu.edu.tw }

***Abstract* – As more and more network security threats are emerging today, the network-based intrusion detection system (NIDS) is one of the most important systems to protect the network from attacks and intrusions without modifying end-user software. Searching through entire packet headers and payloads, NIDSs can identify and classify the packets that contain malicious patterns. The most essential technology to the NIDS is an efficient multiple-pattern matching algorithm, which performs exact string matching between packets and a large set of patterns. This paper proposes a novel *hierarchical multiple-pattern matching algorithm* (*HMA*) for intrusion detection, which is a two-tier and cluster-wise matching algorithm. HMA drastically reduces the amount of external memory access as well as required memory space, enabling an efficient and cost-effective real-time IDS. The simulations show that HMA significantly improves the matching performance in both the average and the worst cases (about 1.7~63 times better than the state-of-the-art algorithms).**

***Keywords* — Network Security, Intrusion Detection, Content Inspection, Matching Algorithm.**

I. INTRODUCTION

With each passing day, there are more critical threats to the data and systems on the network emerging. Different from firewalls which only checks specified fields of the packet *headers*, intrusion detection systems (IDSs) are proposed to detect the malicious information in the *payloads*. The network-based IDSs (NIDSs) can protect the network systems from attacks and intrusions without modifying end-user software. The NIDS must be capable of real-time packet analyzing and fast enough to keep up with the ever-increasing data volume over the network; otherwise the protectorate will not be defended strictly. An IDS typically contains a pattern database applied to finding harmful packets over the network. In the database, each rule compromises several patterns (also called signatures) and a matching action (or a series of actions). These patterns describe the fingerprints of user behavior. The number of patterns is generally a few thousands and the lengths of the patterns are varied. The patterns may appear *anywhere* in any packet *payload*. Therefore, the NIDS requires a *pattern detection engine* capable of in-depth packet inspection, which searches the entire packet *headers* and *payloads* for pattern matching. For example, Snort is an open-source NIDS, which is used for eavesdropping the packets on a network link, detecting anomalous intruder behavior with a set of patterns, and generating logs and alerts by the predefined actions [1]. One of the patterns of Nimda worm is described as "GET /scripts/root.exe?/c+dir" [2], [3]. When the detection engine of Snort finds this pattern existing in a packet, the corresponding alert will be generated to warn network administrators. It has been pointed out that the pattern matching is the most resource intensive tasks in the Snort detection engine [4], [5]. Thus in this paper, we focus our efforts on the nascent issues of payload inspection for multiple-pattern matching.

Without exception, the most essential technology of a detection engine is a powerful multiple-pattern matching algorithm, which can efficiently execute exact pattern matching to keep up with the growing data volume in the network. However, the state-of-the-art matching algorithms are impracticable for packet inspection in realistic implementations, though on computation complexity the Boyer-Moore-based algorithms provided the best average-case performance [6], [7]; while the Aho-Corasick-based algorithms had the best worst-case performance [9], [10], [14]. This is because the performance of processing packets is not only affected by the required computation time, but also strongly affected by the number of required memory reference. Nevertheless, the previous proposed algorithms only addressed on reducing the computation time. For example, the latency of one external memory access is about 150~250 times more than one instruction cycle in the Intel IXP2x00 network processor systems [17]. Therefore, the critical issue of designing a high-speed detection engine is to reduce the number of external memory access.

This paper proposes a novel *hierarchical multiple-pattern matching algorithm* (*HMA*) for real-time packet inspection, which searches the packet payload for a set of patterns simultaneously. HMA is a two-tier and cluster-wise matching algorithm that drastically reduces the number of required external memory access and pattern comparisons. The memory requirement for HMA is very small (less than 350 KB for the current Snort pattern set). The average number of external memory access of HMA is about only 0.35 per input character, which efficaciously improves the performance of the detection engine. The simulation results show that the performance of HMA is better than that of the state-of the-art algorithms [7], [12], [14]. HMA provides better best-case and average-case performance as well as controllable worst-case performance. Consequently, the proposed HMA is a very cost-effective and efficient mechanism that can be employed into the real-time NIDSs.

II. THE HIERARCHICAL MULTIPLE-PATTERN MATCHING ALGORITHM

To improve the performance as well as reduce the size and cost of the network equipment, there is a tendency towards hardware implementations [5], [15], and generally they have both the embedded memory and the external memory elements. For example, the Vitesse IQ2000 network processor [16] has 4

FCS Algorithm;
Input: A set of patterns P.
Output: A set of frequent-common codes F.
1 Initialize: $F \leftarrow \varnothing$;
2 **For** each pattern p_i from P, $0 \le i<|P|$ **do**
3 Transfer the pattern p_i into a vector M by setting $m_j = 1$ if $j \in p_i$; otherwise $m_j = 0$, for all j, $0 \le j<|\Lambda|$;
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 F : $F = F \cup \{f\}$;
8 **For** $0 \le i<|\Lambda|$ **do**
9 $r_{ii} = r_{ii} - r_{fi}$, if $r_{ii} > r_{fi}$; otherwise, $r_{ii} = 0$;
10 **Return;**

Fig. 1. The FCS algorithm.

KB data cache, 2 KB for local storage and 2 KB for reserved header buffers. As the size of the pattern database and the lookup table for the state-of-the-art matching algorithms [6]-[14] are usually larger than 300 KB, which is still growing, the patterns of the IDS must be stored in the external memory. However, frequently accessing the external memory to read patterns or tables will extremely reduce the matching efficiency, as the latency of the external memory access is very long and indeterminist. Therefore, decreasing the amount of computational time is not the only way to improve the throughput of detection engines. The most important is reducing the required number of external memory access.

As a hierarchical and cluster-wise matching algorithm, the proposed HMA effectively reduces the number of external memory access and string comparisons, without sacrificing the memory space. HMA comprises two stages: the off-line stage and the on-line matching stage. The off-line stage constructs two small tables (H^1 and H^2) for hierarchical multiple-pattern searching, where a *frequent common-code searching* algorithm (*FCS*) and a clustering procedure are proposed for the table construction. H^1 and H^2 act as two filters to avoid unnecessary external memory access and pattern comparisons respectively, and thereby pass the innocuous packets quickly in the on-line matching stage. The second-tier matching activates after the first-tier is matched, where H^2 indicates a small subset of patterns that are similar to the input packet. HMA compares only a few *selected* patterns in P with the *suspected* substrings of the packet, instead of comparing all patterns with all substrings of the packet. Thus, HMA significantly improves the matching performance.

Let $P = \{p_i\}$ be a set of distinct patterns, where p_i is a pattern with an identification number (ID) i, and $|P|$ means the number of patterns in P. We consider the payload of an input packet T and the pattern $p_i \in P$ are both strings drawn over Λ, with finite-length $|T|$ and $|p_i|$ respectively. A multiple-pattern matching algorithm is used to search the input T for *all* occurrences of *any* pattern $p_i \in P$ where $|P| >> 1$, or to corroborate that no pattern of P is in T. The matched patterns will be added into the set P^M. We note that *all* matched patterns will be found and P^M can be used for high-level decisions, such as the first-matched-win or the high-priority-win.

A. The Off-line Stage

Since the patterns may appear anywhere in the packet payload, and the packet payload T and the pattern p_i are both strings drawn over Λ, it is hard to recognize the patterns within the payload. We assume that if there is a smaller code set ($< \Lambda$), denoted F, to represent the patterns, and F can help to distinguish the suspected substrings of T from the innocent parts, then the pattern matching will go faster. The FCS algorithm is proposed to find $F=\{f_i \mid f_i \in \Lambda\}$, called the *frequent common-code set*, where f_i is a *frequent common-code*. In the FCS, we gather the occurrence frequency of each character in P, and select the most frequent character into F until for each p_i there is at least one character of p_i belongs to F. The FCS algorithm is presented in Fig. 1.

Thereafter F is used to construct a small hash table, called the *first-tier hash table* (H^1). To achieve fast hashing, a direct hash table of $|\Lambda|$ entries is used for H^1. The ath entry of H^1 is denoted $H^1(a)$, where each entry has two fields: the frequent common-code ID, say $H^1(a).fid$; and the single-symbol pattern ID, $H^1(a).pid$. The notation $e.f$ means the value of the field (or offset) f at the entry (or address) e. The unused fields of H^1 are set as null. Since H^1 is a small hash table (256 entries in the case of one byte coding for example), it can be stored in the data cache of microengines. Hereafter H^1 acts as a filter in the on-line matching to quickly find out whether the packet is a suspect that contains a pattern. Namely, HMA makes use of H^1 to narrow the searching field and to focus on the most suspected packets.

It has been pointed out that in the general situations, most packets (more than 98%) are innocent. Thus it is time consuming to compare all of the patterns in the large P with each input packet. We consider that if the patterns in P can be distributed into different small clusters based on their *similarity*, and only the patterns in the clusters that are similar to the current packet T have to be compared with the current one, then the matching process will perform more efficiently.

Let $P_{a,b}$ represent a cluster of selected patterns that have the same two-character substring '*ab*', called the *clustering pivot*, namely $P_{a,b} = \{p_i \mid$ 'ab'$\in p_i, p_i \in P\}$, where the clustering pivots are the similarity of the patterns in the same cluster. An $|F| \times |\Lambda|$ matrix N = $(n_{a,b})$ is used to record the current size of the cluster $P_{a,b}$ during the pattern clustering procedure. We distribute the patterns based on the clustering pivot in each pattern p_i, say 'ab', where $a \in F$, $b \in \Lambda$ and 'ab'$\in p_i$. In the pattern clustering procedure: (**1**) Fetch a pattern p_i from P one at a time. (**2a**) Scan the pattern. If we can find the clustering pivot of p_i, say 'ab', that $a \in F$, $b \in \Lambda$, and the current size of the cluster $P_{a,b}$ is zero ($n_{a,b} = 0$), then the pattern p_i is grouped to the cluster $P_{a,b}$. (**2b**) If there is no such a clustering pivot 'ab' in p_i with $n_{a,b} = 0$, then we select the substring of p_i, say 'cd', such that $c \in F$ and $n_{c,d}$ is the smallest among all possible

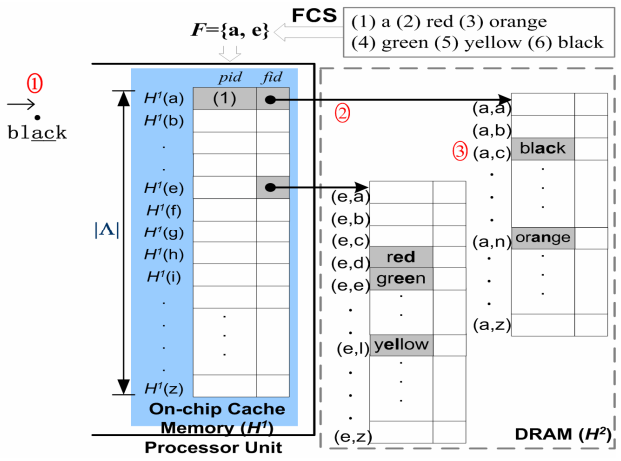

Fig. 2 The architecture of hierarchical hash tables.

clustering pivots of p_i. Then the pattern is grouped to the cluster $P_{c,d}$, which is the smallest cluster that p_i can choose. (**3**) After grouping p_i into a certain cluster, the size of the corresponding cluster is also incremented. Sequentially all patterns are distributed into the designate clusters. We note that a pattern will be assigned to only one cluster.

The H^2 table is constructed based on the cluster assignments. H^2 contains the pattern contents and formatted information of patterns for fast on-line matching. Consider $H^2(a, b)$ is a function indicating an entry of H^2, which stores the head pattern of the cluster $P_{a,b}$. Define $H^2(a, b) = H^1(a).fid \times |\Lambda| + b$. For fast lookup, the patterns in the same cluster $P_{a,b}$ will hash to the same head entry $H^2(a, b)$, and be connected by the linked-list structure to optimize the memory utilization. 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 $H^2(a, b).pid$ of the saved pattern, and a pointer $H^2(a, b).next$ that points to the next entry containing the pattern that also belongs to the same cluster $P_{a,b}$ or the fragmented content of the current pattern. Transferring the information of patterns into a predefined format can speed up the matching procedure.

Fig. 2 illustrates the logical architecture of the hash tables of HMA, where assuming the alphabets are 26 English letters. Since the H^1 table is only $|\Lambda|$ (= 26) entries, it can be stored in the cache memory. Considering we have six patterns and according to FCS, we obtain F={a, e}. Since the first pattern 'a' is a single-pattern, its *pid* (= 1) is stored in the entry of H^1 table. As the pattern '**red**' has 'e'$\in F$ and the clustering pivots 'ed' with $n_{e,d}$ = 0, it is grouped to the cluster $P_{e,d}$. Then $n_{e,d}$ is incremented. The remainders of the patterns follow the same clustering strategy.

B. The On-line Hierarchical and Cluster-wise Matching

Since the pattern set P may contain single-symbol patterns ($|p_i| = 1$), each character of T must be checked without any jump over T. As a hierarchical matching, the on-line matching procedure of HMA is divided into two tiers: the *first-tier matching* and the *second-tier matching*. The on-line matching algorithm is shown in Fig. 3.

```
Procedure OnlineMatching(T, H1, H2)
Input: Packet payload T, two preprocessed indexing tables: H1 and H2
Output: The matched pattern set of T: PM, and its corresponding pid PIDM
1  Load the input payload into buffer T;
2  Initialize: PM ← ∅ ;
3  For each T[t] do
4    If (k←H1[T[t]].pid) ≠ NULL then PM←PM ∪ {pk} and
       PIDM←PIDM ∪ {k}; /* First-tier matching*/
5    If (k←H1[T[t]].fid) ≠ NULL && t < |T| then
6      Load data from the external RAM at entry H2(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 PM←PM ∪ {pk} and
           PIDM←PIDM ∪ {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;
```

Fig. 3. The on-line matching algorithm.

The given T is scanned from left to right, and each character $T[t]$ is directly used as the hash key to fetch the entry $H^1(T[t])$. In the first-tier matching, (**1**) if $H^1(T[t]).pid$ is not null, we know that $T[t]$ is a single-symbol pattern, and this matched pattern will be added into P^M. Whether $H^1(T[t]).pid$ is null or not, then matching procedure checks the *fid* field. (**2a**) If $H^1(T[t]).fid$ is null ($T[t] \notin F$, $T[t]$) will be skipped without any pattern comparison, and thereby fetching the pattern from external memory is unnecessary. Then the next character of payload string $T[t+1]$ is processed to check the $H^1(T[t+1]).pid$ as previous steps, and the matching procedure stays in the first-tier matching. Since the size of F is much smaller than that of Λ, most characters of T will gain the skips and avoid the second-tier matching. Consequently, both the number of character comparisons and costly memory access can be drastically reduced. (**2b**) Otherwise, as $H^1(T[t]).fid$ is not null, T may have a pattern that belongs to the cluster $P_{T[t],T[t+1]}$. In this case, the matching procedure activates the second-tier matching to identify the pattern.

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. According to the input T, the hash function $H^2(T[t], T[t+1])$ indicates the location of the corresponding cluster $P_{T[t],T[t+1]}$. As a cluster-wise matching, only the patterns in the small set $P_{T[t],T[t+1]}$, which are most similar to T, will be loaded to the microengine for further checks. In the second-tier matching, (**1**) first the *pid* field of H^2 is checked. (**2a**) If the $H^2(T[t], T[t+1]).pid$ is null, it means there is no pattern in the cluster $P_{T[t],T[t+1]}$ and no pattern comparison is necessary. Afterward the next character $T[t+1]$ will be processed and the on-line matching procedure returns to the first-tier matching. (**2b**) Otherwise, if the $H^2(T[t], T[t+1]).pid$ is valid, it means there are patterns in $P_{T[t],T[t+1]}$ similar to T. Then HMA will

compare the pattern content in $H^2(T[t], T[t+1])$ with the corresponding substring of T, starting at $T[t\text{-}H^2(T[t], T[t+1]).offset]$ of length $H^2(T[t], T[t+1]).size$. (**3**) If the next field of the last pattern fragment is valid and pointing to the next entry, say $H^2(a, b)$, similarly the pattern in $H^2(a, b).data$ will be compared with the substring of T starting at $T[t\text{-}H^2(a, b).offset]$. The process continues until all patterns in the cluster $P_{T[t],T[t+1]}$ are compared. All matched pattern will be added to the matched pattern set P^M.

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

For example, assume the H^1 and H^2 tables have been constructed as Fig. 2. When the input T is 'pink', since all characters of T do not belong to F, by only checking T with the embedded table H^1 HMA can know that T contains no pattern. If T = 'black', scanning from left to right, HMA will stay in the first-tier matching until it matches 'a', where 'a' $\in F$ (H^1(a).*fid* is valid) and it is a single-symbol pattern (H^1(a).*pid* = 1). Then the second-tier matching activates, and 'ac' will be the hash keys (clustering pivot). HMA will load the entry from H^2(a,c) to check H^2(a,c).*pid*. As H^2(a,c).*pid* (= 6) is not null, HMA compares the input T with the pattern(s) in $P_{a,c}$ (where H^2(a,c).*data* = 'black') and get a match. Afterwards, the first-tier matching will continue. Due to 'c' and 'k' $\notin F$, the on-line matching of this input T is finished and no second-tier matching is necessary. For the input 'black', only one pattern is loaded from DRAM for exact string comparisons. The matching results of T are P^M={a, black}.

III. Results and Discussions

This section will show the simulation results of HMA, compared with the Boyer-Moore-Horspool algorithm (BMH) which is deployed in a famous NIDS – Snort [7], the Boyer-Moore-Horspool algorithm with a grouped prefix table (BM-PH) which is employed in a network-processor-based NIDS [12], and the Aho-Corasick algorithm with memory compression (AC-C) [14]. We emulate the assembly-like microprograms for HMA, BMH, BM-PH and AC-C respectively by the RISC instructions of general network processors, and calculate the number of instructions and memory access needed to process a packet. We assume one microprocessor is used in the simulations to simplify the evaluation though there are several microengines in the network processor systems.

In the simulations, with detachment we use the free and real pattern set released by Snort in Aug. 2004 [1], although the pattern set can be self-defined or any commercial pattern set. 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.

TABLE 1. THE MEMORY REQUIREMENTS.

	HMA	BM-PH	BMH	AC-C
Cache memory	$O(\|\Lambda\|)$	$O(1)$	$O(\|\Lambda\|)$	$O(1)$
External memory*	$O(\|F\|\times\|\Lambda\|+\|P\|)$	$O(\|\Lambda\|^3+\|P\|)$	$O(\|P\|\times\|\Lambda\|+\|P\|)$	$O(S+\|P\|)$

*$|F| < |\Lambda| << |P| < S$

Define N_I as the average number of RISC instructions; N_L as the average number of local memory access for each input character required in the pattern matching. N_E represents the average number of external memory access per input character. Respectively w_I represents the time required by one instruction or one local memory/register access, and w_E is the time for one external memory access. Thereby, we have the measurements: the average computation cycles $\psi_I = N_I \times w_I$; the average memory latency $\psi_M = N_E \times w_E + N_L \times w_I$; and the total average matching cost $\Psi = \psi_I + \psi_M$. In the simulations, note that we assume the skip table of BMH is small enough to be loaded into the cache memory, and thus only one external memory access is counted during the matching process of BMH for each pattern; one external memory access is assumed for AC-C although it generally requires two memory references for fetching the transition matrixes and the matched patterns. In the simulations, the payload length is 640 bytes, $|P| = 1200$, $w_I = 1$, and $w_E = 100$.

The memory requirements of HMA, BM-PH, BMH and AC-C are summarized in TABLE 1, including the requirements of lookup tables (nodes) and pattern contents (where S is the number of states). In the simulations, with $|P|$=1200, the external memory size of HMA is 20192 entries (326.75 KB with each entry of 16 bytes, including pattern contents and formatted information), where $|F|$=77; BM-PH needs more than 16M entries (16 MB for the skip table, excluding $|P|$ entries for pattern contents); BMH needs more than 300K entries (300 KB for skip table, excluding $|P|$ entries for pattern contents); and AC-C requires 10213 states (439 KB with each node of 44 bytes, excluding $|P|$ entries for pattern IDs). Consequently, the required memory space of HMA is very small.

In the simulations, the malicious packets are generated by randomly choosing patterns from the pattern set P and spreading over the packet payloads. An attack load λ is defined to represent the expected number of malicious patterns existing in one packet. Except for the patterns in the payload, other characters of the payload are randomly drawn from Λ. Fig. 4 shows the comparisons of Ψ, ψ_M and N_E for HMA, BM-PH, BMH and AC-C with different λ. Since different systems introduce different implementation overheads, we extract N_E from overall matching cost to examine the performance of algorithm itself. This figure demonstrates that HMA effectively reduces the number of required external memory access (only around 0.35 for each input character). We can find that the memory latency predominates the matching cost of every approach. This result reflects our

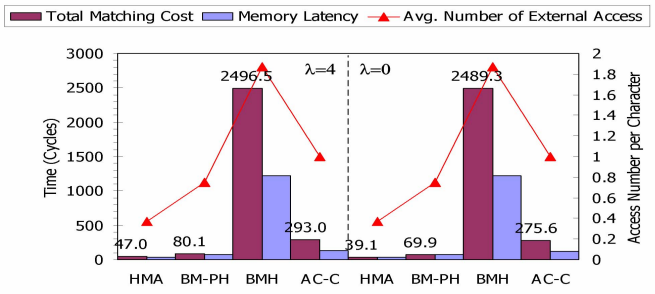

Fig. 4. The total processing time (Ψ), the memory latency (ψ_M), and the average number of external memory access (N_E) for each input character, with $\lambda = 4$ and $\lambda = 0$ respectively.

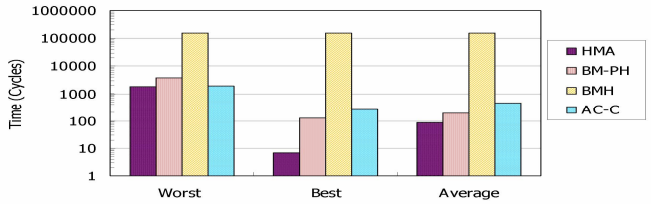

Fig. 5. The costs of the matching algorithms in the worst-case, the best-case and the average situations.

opinion mentioned previously that the essential issue to design a high-speed detection engine is to reduce the number of required external memory access. As HMA significantly reduces ψ_M without sacrificing ψ_I, HMA outperforms BM-PH, BMH and AC-C. When λ=0, the performance of HMA is 1.7, 63.5, and 7 times better than that of BM-PH, BMH and AC-C respectively. Consequently, HMA is very appropriate for network environment because generally most packets are innocent. The faster to process the innocent packets, the more time the detection engine will gain to process the malicious packets.

Since different multiple-pattern matching algorithms have different string forms that cause their extreme (best-case or worst-case) performance, we examine the performance with all permutations of four-character input strings (2^{32} strings). We choose the length of four characters because 24.5% of all patterns are less than or equal to four characters, and the test pool of 4G input strings is large enough for simulations. We use this model to approach extreme and average evaluations in the practical network. Fig. 5 plots the best-case, the worst-case and the average performance for each approach. The matching costs shown in this figure exclude the cost for loading the input packets from external memory into the processor, as in this case, we are interested in the pure costs of the matching algorithms required for each input character. Fig. 5 shows that in the best case, HMA requires only seven instruction cycles to process an input character, which successfully reduces the processing delay of innocent packets. In the worst case, HMA offers the same performance as AC-C. As improving the worst-case performance, HMA can provide stronger defense against the attacks under the same hardware expenses. Therefore, HMA significantly improve the performance of the pattern matching.

IV. CONCLUSIONS

A novel hierarchical multiple-pattern matching algorithm (HMA) has been proposed in this paper for the real-time IDSs. HMA uses the proposed FCS to narrow the searching scope, and thereby speeds up the pattern matching processes. Furthermore, as a hierarchical and cluster-wise matching mechanism, HMA not only drastically reduces the required number of memory access as well as string comparisons, but also reduces the requirements on memory space. Simulation results show that HMA outperforms the state-of-the-art pattern matching algorithms. Therefore, HMA enables efficient, practical and cost-effective IDSs.

ACKNOWLEDGMENT

This work was supported by the MediaTek Fellowship and the MOE Program for Promoting Academic Excellent of Universities (II) under the grant number NSC-94-2752-E-007-002-PAE.

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] Martin Roesch, "Snort – Lightweight Intrusion Detection for Networks," *Proceedings of the 13th Systems Administration Conference*, 1999.
[5] 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.
[6] R.S. Boyer and J.S. Moor, "A Fast String Searching Algorithm," *Communications of the ACM*, Vol. 20, No. 10, pp. 762-772, October 1977.
[7] R. Nigel Horspool, "Practical Fast Searching in Strings," *Sofetware Practice and Experience*, Col. 10, No. 6, pp. 501-506, 1980.
[8] R.A. Baeza-Yates, "Improved String Search," *Software – Proctice and Experience*, Vol. 19, No. 3, pp. 257-271, Martch 1989.
[9] A.V. Aho and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, Vol. 18, Np. 6, pp. 330-340, June 1975.
[10] Graham A. Stephen, "String Matching Algorithms," *World Scientific* (ISBN 981-02-1829-X), 1994.
[11] 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.
[12] 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., Aug. 2004.
[13] E. Markatos, S. Antonatos, M. Polychronakis and K. Anagnostakis, "Exclusion-based Signature Matching for Intrusion Detection," *Proceedings of IASTED International Conference on Communications and Computer Networks (CCN 2002)*, October 2002.
[14] Nathan Tuck, Timothy Sherwood, Brad Calder, George Varghese, "Deterministic Memory –Efficient String Matching Algorithms for Intrusion Detection," *Proceedings of the IEEE Infocom Conference*, Hong Kong, March 2004.
[15] 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.
[16] Vitesse Network Processors, http://www.vitesse.com.
[17] Sridhar Lakshmanamurthy, Kin-Yip Liu, Yim Pun, Larry Huston, and Uday Naik, " Network Processor Perfromance Analysis Methodology," *Intel Technology Journal*, Vol. 6, Aug. 2002.