algorithms

MDPI

Article

A Flexible Pattern-Matching Algorithm for Network Intrusion Detection Systems Using Multi-Core Processors

Chun-Liang Lee * and Tzu-Hao Yang

Department of Computer Science and Information Engineering, School of Electrical and Computer Engineering, College of Engineering, Chang Gung University, Taoyuan 33302, Taiwan; m9929012@gmail.com

* Correspondence: cllee@mail.cgu.edu.tw; Tel.: +886-3-211-8800 (ext. 5196)

Academic Editor: Andras Farago

Received: 15 March 2017; Accepted: 20 May 2017; Published: 24 May 2017

Abstract: As part of network security processes, network intrusion detection systems (NIDSs) determine whether incoming packets contain malicious patterns. Pattern matching, the key NIDS component, consumes large amounts of execution time. One of several trends involving general-purpose processors (GPPs) is their use in software-based NIDSs. In this paper, we describe our proposal for an efficient and flexible pattern-matching algorithm for inspecting packet payloads using a head-body finite automaton (HBFA). The proposed algorithm takes advantage of multi-core GPP parallelism and single-instruction multiple-data operations to achieve higher throughput compared to that resulting from traditional deterministic finite automata (DFA) using the Aho-Corasick algorithm. Whereas the head-body matching (HBM) algorithm is based on pre-defined DFA depth value, our HBFA algorithm is based on head size. Experimental results using Snort and ClamAV pattern sets indicate that the proposed algorithm achieves up to 58% higher throughput compared to its HBM counterpart.

Keywords: network security; pattern matching algorithm; deep packet inspection; intrusion detection system

1. Introduction

Toward the goal of improving Internet network security, firewalls are widely deployed to provide protection by inspecting source and destination IP addresses, port numbers, protocols, and other packet header fields. However, since firewalls can only provide limited protection again attacks, network intrusion detection systems (NIDSs) have been proposed as an alternative for providing greater security [1–3]. There are two NIDS categories: anomaly-based, which monitor and analyze network activities in search of abnormal behaviors [4–6]; and signature-based, which execute deep packet inspection tasks to determine whether incoming packet payloads contain attack patterns known as "signatures". Compared with anomaly-based NIDSs, signature-based NIDSs generally provide better detection against known attacks, and thus they have been the focus of a large number of studies. The focus of this study is also on signature-based NIDSs.

Pattern matching, which can consume up to 70% of system execution time [7,8], is the most important factor in overall signature-based NIDS system performance. There are two types of pattern matching algorithms: software-based and hardware-based, with the second achieving high matching speed via special-purpose devices such as field programmable gate arrays (FPGAs) [9–13], content addressable memory (CAM) [14,15], and application-specific integrated circuits (ASICs) [16]. However, special-purpose devices are susceptible to scalability issues in terms of pattern set size and/or speed. Further, special-purpose device adaptation is generally costly, inflexible, and slow to develop and

market [17]. In contrast, software-based algorithms utilize central processing units (CPUs) or graphical processing units (GPUs) characterized by high flexibility and programmability [18–27]. We therefore focused our efforts on designing a pattern-matching algorithm for software-based NIDSs.

Software-based NIDS throughput is highly dependent on processor computing power. More efficient pattern-matching algorithms take advantage of parallel computation associated with multi-core processors. Although GPUs have superior processing power compared to CPUs, a significant amount of extra energy and cost are required for GPU-based pattern-matching algorithms. In addition, the single-instruction multiple-data (SIMD) operations supported by most CPUs can be used to accelerate pattern matching. Similar to the head-body matching (HBM) algorithm proposed in [27], our proposed flexible head-body matching (FHBM) algorithm uses the Aho-Corasick (AC) algorithm to construct a deterministic finite automaton (hereafter referred to as AC-DFA). The AC-DFA is partitioned into a head and a body. In the HBM algorithm, the AC-DFA is partitioned according to a pre-defined depth value that exerts a significant impact on throughput [27]. However, we have found that even in cases where a good depth value is selected, the HBM algorithm may still fail to achieve good throughput due to the way it partitions the AC-DFA. In comparison, our proposed FHBM algorithm is more flexible in terms of AC-DFA partitioning, resulting in higher throughput.

The rest of this paper is organized as follows. Section 2 briefly reviews the literature related to this work. Our proposed algorithm is described in detail in Section 3. Experimental results are presented and discussed in Section 4, and the conclusion is given in Section 5.

2. Related Work

Pattern matching is used for tasks such as intrusion detection, virus scanning, and information retrieval. The well-known Knuth-Morris-Pratt (KMP) [28] and Boyer–Moore (BM) algorithms [29] were created to search for single patterns, while the Aho-Corasick (AC) [30] and Wu-Manber (WM) [31] multi-pattern matching algorithms are capable of inspecting multiple pattern sets simultaneously. The WM algorithm has a major advantage in terms of memory requirement, but it is less effective with very small and large minimum pattern sizes. Characterized by deterministic worst-case performance, the AC algorithm is insensitive to pattern sets as well as the content being inspected. For these reasons, the AC algorithm has attracted much greater attention, with a large number of researchers searching for ways to mitigate its significant memory requirement. Based on the observation that only a small number of entries in a state transition table generated by the AC algorithm stores valid transitions, Tuck et al. [32] used a bitmap and variable-length list of transitions to successfully reduce required memory and to produce better throughput.

Bremler-Barr et al. [33] observed that the name used by common AC-DFA encoding is meaningless and proposed a CompactDFA scheme which compress AC-DFAs by encoding state in such a way that all transitions to a specific state are represented by a single prefix that defines a set of current states. They reduced the pattern matching problem to the longest prefix matching (LPM) problem, which has been studied extensively. With a TCAM, CompactDFA can reach a throughput of 10 Gbps. Although the authors mentioned that CompactDFA can be implemented in software, only experimental results with TCAM were provided in [33].

Liu et al. [34] focused on reducing the number of states and proposed a general DFA model called DFA with extended character-set (DFA/EC), in which part of each state is removed and incorporated with the next input character. However, their proposed model reduces the number of states at the cost of increasing the size of the transition table. To address this problem, they proposed a method to encode the complementary state into a single bit. As a result, the number of memory access required to inspect each byte in packet payloads is only one.

Yang and Prasanna [27] tried to improve AC-DFA throughput from a different perspective. They found that the match ratio of an input stream with respect to a given pattern set exerts a significant impact on AC-DFA throughput. For large pattern sets and input streams with high match ratios, AC-DFA throughput can significantly degrade for reasons associated with memory access overhead.

To address this problem, they have proposed both a new architecture called the head-body finite automaton (HBFA) and a HBM algorithm. The HBFA consists of a head DFA (H-DFA) and body NFA (B-NFA). The H-DFA has the same structure as the AC-DFA, but with much fewer states and higher average access probability. The B-NFA was designed so that it can be accelerated by the SIMD operations that are commonly found in commodity processors. Their test results indicate that, compared to the AC algorithm, HBM algorithm performance in terms of matching throughput improved by a factor ranging from 2 to 7.

3. Flexible Head-Body Matching Algorithm

As shown in Algorithms 1, our proposed FHBM algorithm uses an AC-DFA plus a pre-defined maximum head size as the input, and then partitions the AC-DFA into a head and a body. After all head part states are returned, both head and body parts are processed using the HBM algorithm. More specifically, the head part remains the same structure as an AC-DFA, while the body part is converted to a compact NFA for parallel processing. The head part is initially set to empty (Line 1). An AC-DFA can be analyzed as having a tree structure (Figure 1). If there is a valid transition from state A to state B, then state A is considered a parent of state B and state B a child of state A. State depth is defined as the number of edges that separate it from a root state that has a depth of 0. All states are processed starting from the root state. For each depth h, the first task is to determine whether all states at that depth can be included in the head part. If the sum of the number of states in the head part (*HEAD*.*size*()) and the number of states at that depth (*AC_DFA*.*depth*[h].*size*()) is less than or equal to the maximum head size (*HSIZE*), then all states at that depth can be included in the head part (Line 4). Otherwise, the depth contains too many states to be entirely included in the head part. Note that the HBM algorithm partitions the AC-DFA based on a pre-defined depth value. Accordingly, states at the same depth are entirely included in either the head part or body part. However, since the number of states at any specific depth can be very large, including all states at the same depth in either the head part or body part cannot achieve good throughput. Our proposed FHBM algorithm lacks this restriction, and can therefore select appropriate states for inclusion in the head part.

In the example presented in Figure 1, assume that the number of states at depths lower than h-1 is less than *HSIZE*, and that the number of states at depth h is too large for inclusion in the head part. According to both the HBM and our proposed algorithms, states at depths lower than h-1 are partitioned in the head part. One straightforward method for fully utilizing *HSIZE* is to include states at depth h one-by-one until there is no available room for additional states. However, according to the head and body part structures, all states with the same parent state must be 100% in the head part or 100% in the body part. As shown in Figure 1, state s has four child states. If only a partial number of child states are partitioned in the head part, the HBM algorithm cannot build the body part. Thus, the problem is to include as many states at depth h as possible in the head part according the *HSIZE* constraint, and to guarantee that states with the same parent state are either entirely included or excluded from the head part.

The optimal solution for this problem makes use of the greedy algorithm described below. First, a temporary set named T (in which each element is an ordered pair denoted by (s, n)) is used to store the information of a state s at depth h-1 and a number of its child states (n) (Lines 7–9). Next, all elements in the set are sorted by the second coordinate (i.e., the number of child states) in descending order (Line 10). All elements in T are processed one-by-one starting with the largest number of states (Lines 11–20). Given that the currently processed element is (s, n), if the number of states in the head part (after adding all child states of state s) does not exceed *HSIZE*, all child states of state s will be added to the head part (Lines 13–15). Otherwise, the next element in the temporary set will be processed. If all elements are processed, or if the size of *HEAD* is equal to *HSIZE*, the outermost for-loop (Lines 2–23) will be terminated, after which the head set is returned (Line 24).

Algorithm 1 Partitioning Algorithm

```
   Input: AC_DFA (a DFA constructed with AC algorithm)
          HSIZE (maximum level of head size)
   Output: HEAD (set of head states)
1  HEAD ← ∅;
2  for h ← 0 to AC_DFA.MAX_DEPTH do
3    if HEAD.size() + AC_DFA.depth[h].size() ≤ HSIZE then
4      HEAD = HEAD ∪ AC_DFA.depth[h];
5    else
6      T ← ∅;
7      foreach state p in AC_DFA.level[h-1] do
         // p.childStates() returns the set of child states of state p.
8        T ← T ∪ (p, p.childStates().size());
9      end
10     Sort T by the second coordinate in descending order;
11     foreach (s,n) in T do
12       if HEAD.size() + n ≤ HSIZE then
13         foreach state t in s.childStates() do
14           HEAD ← HEAD ∪ t;
15         end
16         if HEAD.size() = HSIZE then
17           return HEAD;
18           end
19         end
20       end
21       break;
22     end
23   end
24 return HEAD;
```

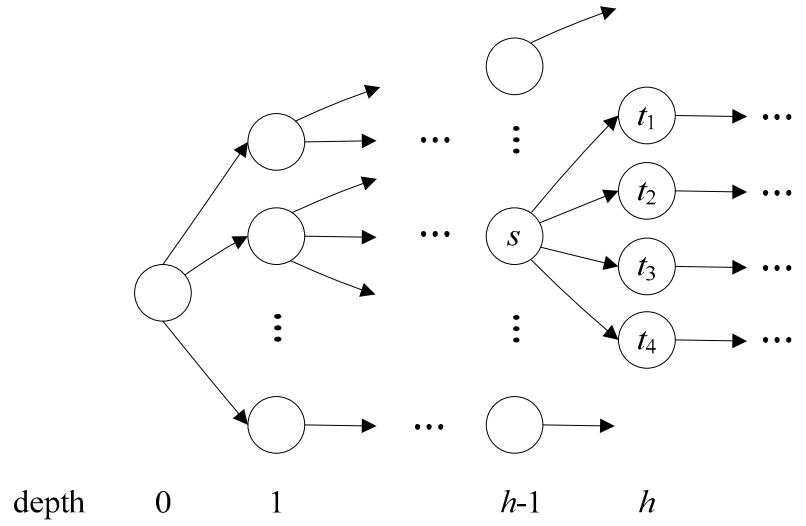


Figure 1. An illustrative example of AC-DFA.

Given pattern set S = {account, advance, in, inner, insert, invert, stand, stood}, the AC-DFA can be constructed as shown in Figure 2. Numbered circles represent states, with state 1 designated as the start state. A double circle represents a matching state. If a matching state is achieved, it indicates that at least one pattern has been found. For the sake of clarity, Figure 2 omits all backward transitions.

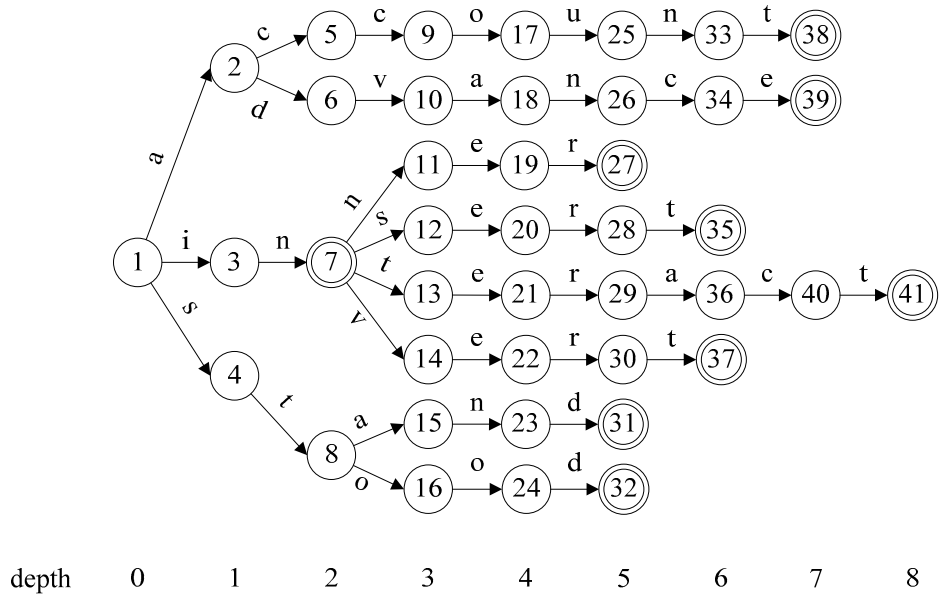


Figure 2. AC-DFA for the *S* pattern set.

Assume a maximum head size of 12 states. According to the FHBM algorithm shown in Algorithms 1, the start state (state 1) is the first to be added to the head part, after which the three states in depth 1 are also added, since *HEAD.size*() + *AC_DATA.depth*[1] = 1 + 3 < = *HSIZE*. Similarly, states at depth 2 are added to the head part, increasing the number of states in the head part to 8. Since the number of states at depth 3 is also 8, the addition of all states to that depth will exceed the maximum head size, thus triggering the execution of the else part in Lines 6–21. Recall that the value of variable *h* is 3. After executing the foreach loop in Lines 7–9, T = {(5, 1), (6, 1), (7, 4), (8, 2)}. Set T is sorted by the second coordinate in descending order, resulting in T = {(7, 4), (8, 2), (5, 1), (6, 1)}. The foreach loop in Lines 11–20 initially selects (7, 4) and checks to see if all child states of state 7 can be added to the head part. Since *HEAD.Size*() + n = 8 + 4 <= *HSIZE*, all child states of state 7 are added to the head part. Since the head part is filled to maximum, the partitioning algorithm is terminated by returning the head part. Figure 3 shows the AC-DFA head and body parts after partitioning.

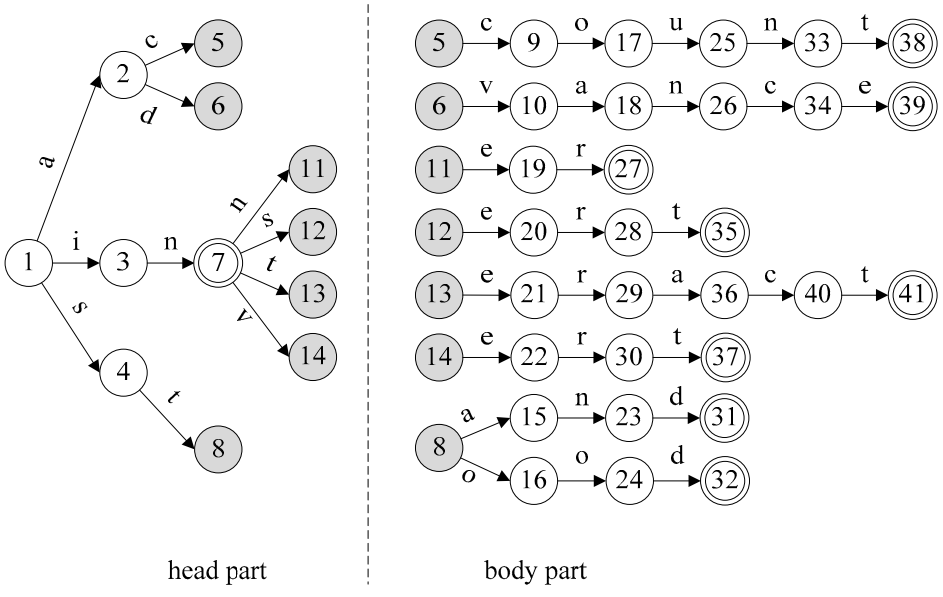


Figure 3. Head and body parts for the example AC-DFA.

4. Experiment Evaluation

4.1. Setup

We used an Intel platform to evaluate our proposed FHBM algorithm; a summary of the hardware configuration used in our experiments is shown in Table 1. The HBM algorithm source code, obtained from the authors of [35], was used to create the FHBM algorithm. Source codes were compiled using GCC 4.8.4. The operating system was 64-bit Ubuntu 14.04 (kernel version 3.13).

Table 1. Hardware configuration for the experiments.

Component		Specification
Main board	Brand/Model	Asus/P8H77-V
CPU	Brand/Model	Intel/i7-3770
	Number of cores	4
	Frequency	3.4 Ghz
	L1 cache size	256 KB
	L2 cache size	1024 KB
	L3 cache size	8 MB
Main memory	Type	DDR3
	Size	8 GB
	Frequency	1600 Mhz

Three pattern sets (one from Snort [36] and two from ClamAV [37]) were used for performance evaluation. Snort is a free, open-source NIDS; ClamAV is free, open-source antivirus software. According to the pattern set statistics shown in Table 2, the Snort set contained the largest number of patterns, but the number of characters was the smallest among the three, since the maximum depth of the Snort pattern set was smaller than those of the other two. As shown in Table 2, the longest patterns were 232 bytes for Snort, 362 for ClamAV type 1, and 382 for ClamAV type 3. According to the pattern length distributions of all sets, most ClamAV type 1 and ClamAV type 3 patterns were longer than 16 bytes, but only 43.9% of the Snort patterns exceeded 16 bytes (Table 3). This clarifies the relationship between the pattern and character numbers shown in Table 2.

Table 2. Pattern set statistics.

Pattern Set	Number of Patterns	Number of Characters	Number of Depths
Snort	8673	196,967	232
ClamAV type 1	5248	498,014	362
ClamAV type 3	2899	262,256	382

Table 3. Pattern length distribution.

Pattern Length	Snort		ClamAV Type 1		ClamAV Type 3	
≤ 4	977	(11.3%)	56	(1.1%)	7	(0.2%)
5–8	1559	(18.0%)	81	(1.5%)	29	(1.0%)
9–12	1284	(14.8%)	103	(2.0%)	53	(1.8%)
13–16	1045	(12.0%)	92	(1.8%)	62	(2.1%)
> 16	3808	(43.9%)	4916	(93.7%)	2748	(94.8%)
Total count	8673		5248		2899	

For each pattern set, three input data streams with different match ratios (1, 8, 32%) were generated to simulate different levels of attacks. For a given pattern set, the match ratio of an input data stream refers to the proportion of the length of malicious content to the length of the input data stream.

A substring in the input data stream is considered malicious if it is a significant prefix string of any pattern in the pattern set. A prefix string is significant if it covers a significant part (e.g., >80%) of the full string. For each input data stream, according to the match ratio, proper prefixes were randomly chosen from the pattern set and embedded into a clean data stream. For the Snort pattern set, the plain text from an HTML-formatted King James Bible was used as the clean data stream. For the ClamAV type 1 and ClamAV type 2 pattern sets, all files under /usr/bin in a typical Linux server installation were concatenated to be the clean data stream. Since both the HBM and FHBM algorithms were designed for multi-core platforms, and since the experimental platform has four physical cores, we created four threads, with each performing its own pattern matching task using the shared HBFA. All data represent averages from 100 simulations.

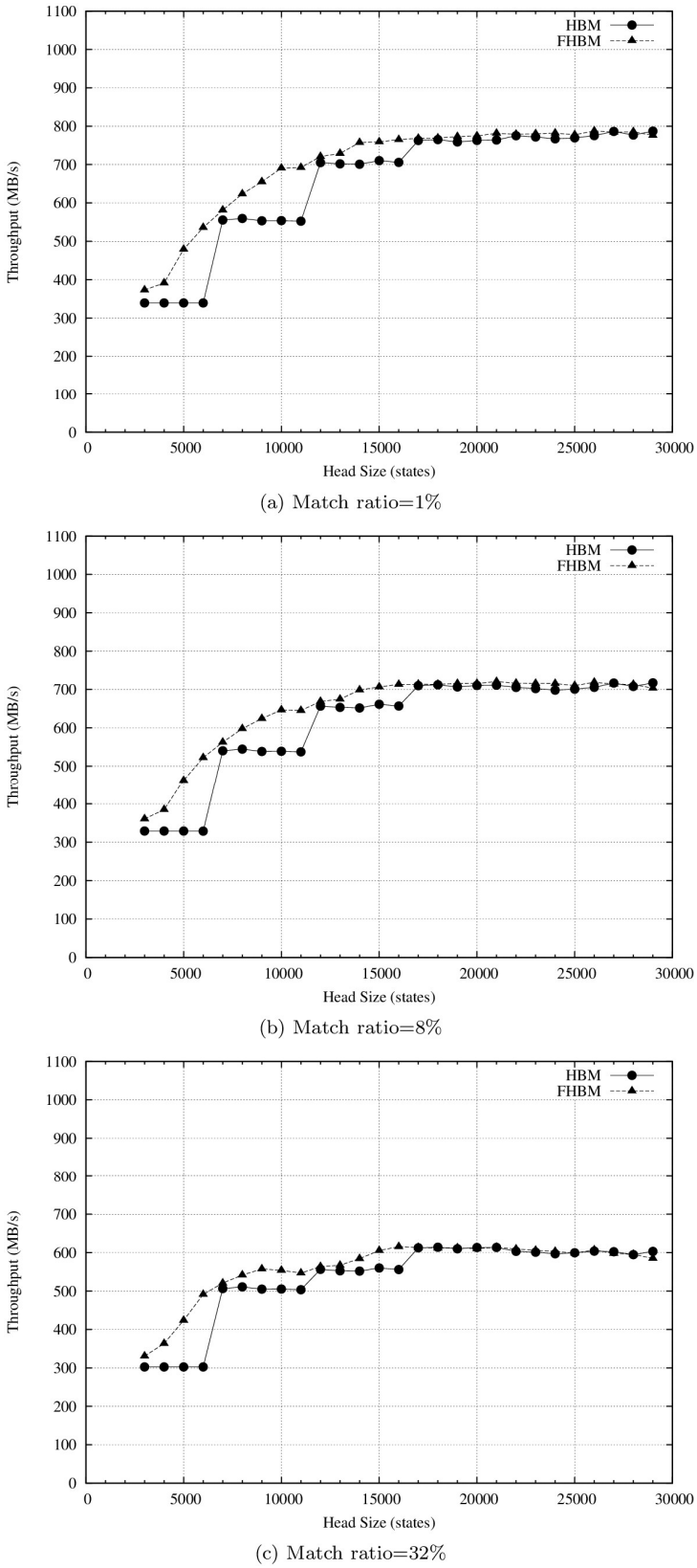
4.2. Results and Discussion

HBM and FHBM throughputs for the Snort, ClamAV type 1, and ClamAV type 3 pattern sets are respectively shown in Figures 4–6. Various head sizes were used to evaluate both algorithms. As shown in Figure 4a, HBM throughput values for head sizes between 3000 and 6000 states were similar. At a head size of 7000, the throughput value sharply increased from 339 MB/s to 555 MB/s. Since HBM partitions an AC-DFA into head and body parts according to a pre-defined head size to determine the maximum depth of states that can be put in the head part, a comparable situation was observed when the head size was increased from 11,000 to 12,000 states. However, as we discussed in an earlier section, the number of states at any specific depth can be very large, which stops HBM from fully utilizing its head size, resulting in poor throughputs. More specifically, the purpose of the head part of a HBFA is to provide fast transition between states that will be accessed more frequently. Since the head part utilizes fully-populated state transition tables (STTs), which can be accessed quickly but require more storage, it may lead to poor memory and throughput performance if the number of states in the head part is not controlled properly.

Table 4 lists the numbers of states at different depth ranges for the Snort, ClamAV type 1, and ClamAV type 3 pattern sets. Suppose that the maximum head size is 6000 states. For the Snort pattern set, since the combined number of states at depths 1, 2, and 3 was 6204, only the states at depths 1 and 2 could be moved to the head part. This explains why the head size did not exert any impact on HBM throughput at head sizes ranging from 3000 to 6000 states. In contrast, as long as the head size did not exceed 17,000 states, FHBM throughput increased as head size increased. The reason is that FHBM is capable of fully utilizing head size by intelligently partitioning the head and body states. Both HBM and FHBM throughputs were kept between 762 and 787 MB/s when head sizes were 17,000 states or higher. Since each state had to store 256 entries for child states, and with each entry consuming two bytes, each state consumed 2×256 bytes. The storage requirement for a head part with 17,000 states is 8.5 MB, which exceeds the L3 cache size, therefore larger head sizes did not enhance throughput. Furthermore, since most state access occurred at lower depths ($\leq$4) (Table 5), partitioning additional depths to the head part also did not enhance throughput.

Table 4. Number of states at different depth ranges.

Depths	Snort	ClamAV Type 1	ClamAV Type 3
$\leq$2	2039	3499	675
$\leq$3	6204	7813	1765
$\leq$4	11,100	12,349	3110
$\leq$5	16,340	17,093	4652
$\leq$6	21,564	21,919	6348
$\leq$7	26,785	26,779	8145
$\leq$8	31,955	31,642	10,050

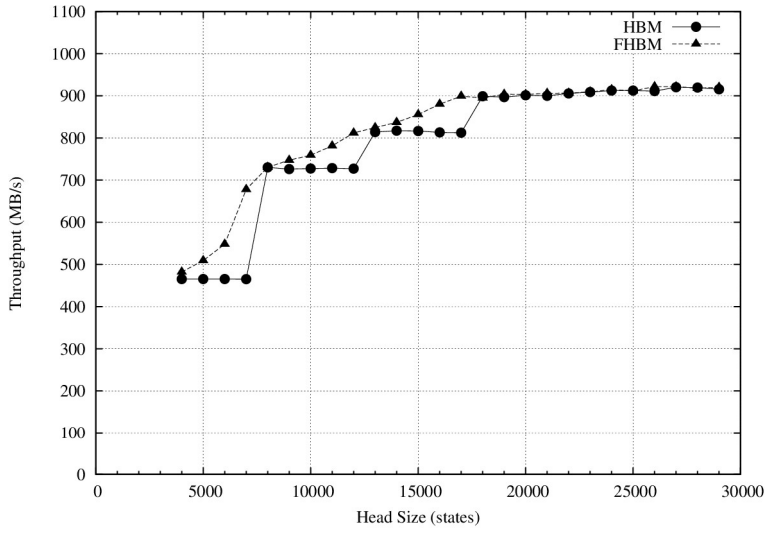


(a) Match ratio=1%

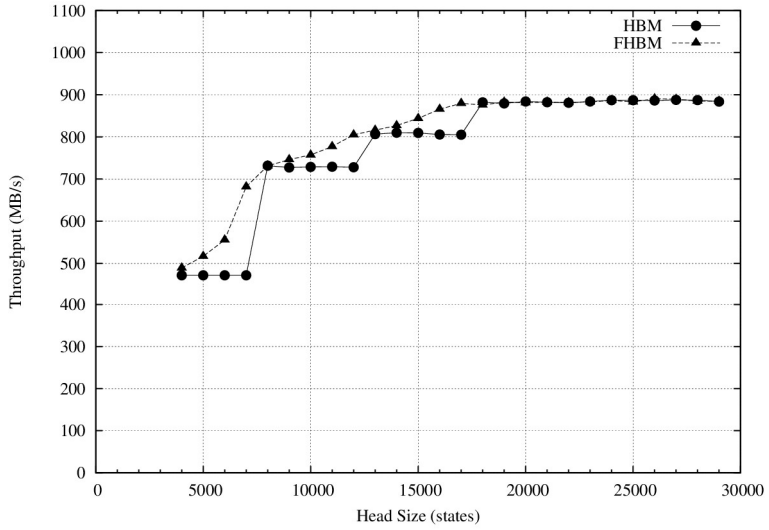
(b) Match ratio=8%

(c) Match ratio=32%

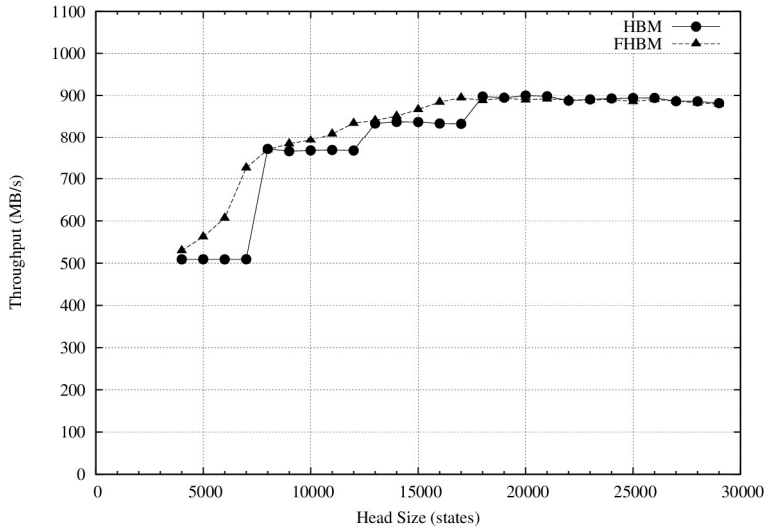
Figure 4. Throughput value plotted against head size for the Snort pattern set.



(a) Match ratio=1%

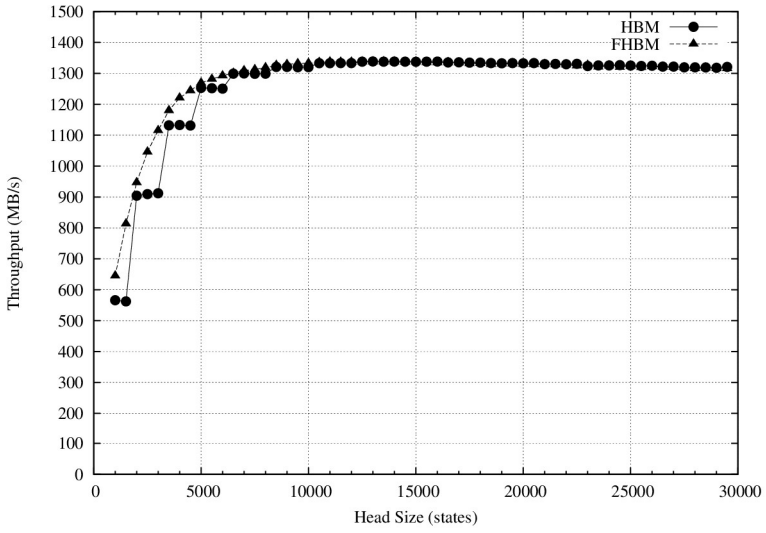


(b) Match ratio=8%

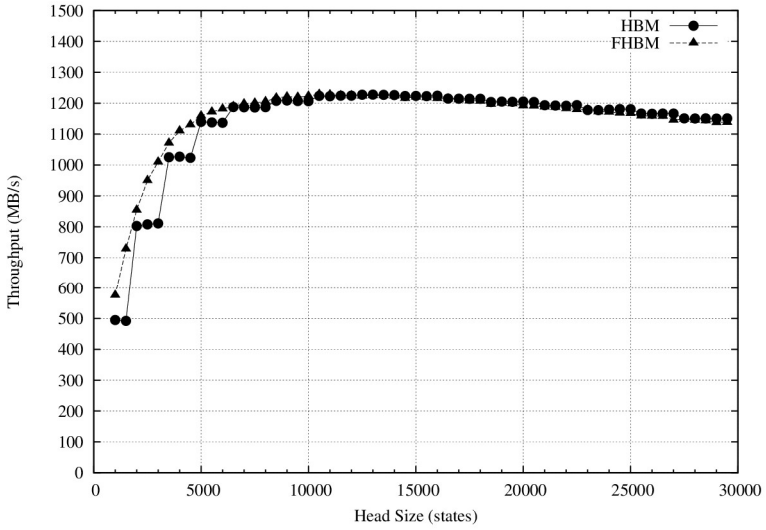


(c) Match ratio=32%

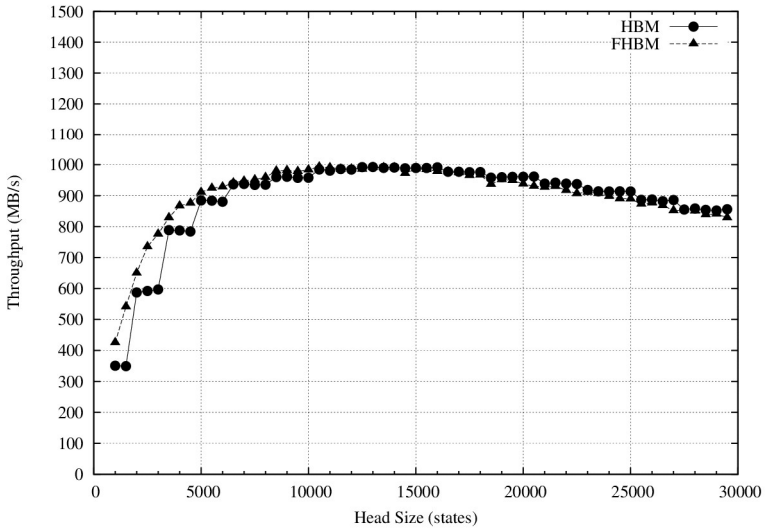
Figure 5. Throughput value plotted against head size for the ClamAV type 1 pattern set.



(a) Match ratio=1%



(b) Match ratio=8%



(c) Match ratio=32%

Figure 6. Throughput value plotted against head size for the ClamAV type 3 pattern set.

Table 5. Number of accesses to states at each depth for the Snort pattern set (1 unit = 1000).

Depth	Match Ratio					
	1%	**2%**	**4%**	**8%**	**16%**	**32%**
1	1726	1710	1680	1617	1494	1244
2	1919	1902	1868	1801	1666	1393
3	612	607	599	581	546	475
4	158	159	159	160	161	164
5	34	35	38	44	55	78
6	16	17	20	26	39	63
7	5	7	10	16	29	53
8	2	3	6	12	23	46

As shown in Figure 5, the HBM and FHBM algorithm results for ClamAV type 1 exhibited a throughput-head size relationship similar to that for Snort. This is explained by the similar numbers of states at different depths for the two pattern set types (Table 4). Note that both algorithms achieved higher throughputs for the ClamAV type 1 pattern set. Given a match ratio of 1% and head size of 20,000 states, FHBM throughput was 903 MB/s for ClamAV type 1 and 775 MB/s for Snort. As shown in Table 3, the percentage of patterns exceeding 16 bytes in the ClamAV type 1 pattern set (i.e., 93.7%) was much higher than that in the Snort pattern set (i.e., 43.9%). Thus, for any match ratio, the input stream generated using the ClamAV type 1 pattern set contained fewer patterns than that using the Snort pattern set, resulting in greater access to states at lower depths for the ClamAV type 1 pattern set and higher throughput values.

As shown in Figure 6, the HBM and FHBM algorithms for the ClamAV type 3 pattern set exhibited a different throughput-head size relationship compared to the other two pattern sets. Maximum throughputs for both algorithms were initially obtained at a head size of 10,000 states—much smaller than that shown in Figures 4 and 5. This is explained by the approximately 10,000 states found at the lowest eight depths for the ClamAV type 3 pattern set. As discussed earlier, most state access occurred at lower depths, therefore head size increase exerted little impact on throughput when head sizes exceeded 5000 states. Second, when the match ratios were 8% or 32%, a large head size resulted in decreased throughput—a decrease that was more obvious at 32%. As shown in Table 4, 94.8% of the ClamAV type 3 patterns were long. For high match ratios, states at higher depths were accessed more frequently, resulting in main memory access when the head size exceeded the cache size. Last, for the ClamAV type 3 pattern set, both algorithms achieved higher throughput values than the other two at the same head size, since the head part was capable of storing states at higher depths. Accordingly, most state access occurred in the head part rather than the body part, resulting in shorter access time.

5. Conclusion and Future Work

In this paper, we described our proposal for a flexible head-body matching (FHBM) algorithm for use with NIDSs and multi-core processors. Unlike the HBM algorithm, which statically constructs head-body finite automata according to pre-defined depth values, our proposed algorithm partitions head and body parts based on head size, thereby constructing more efficient HBFAs compared to the HBM algorithm. According to our results, the FHBM algorithm achieved up to 58% higher throughput for the Snort pattern set (536 MB/s vs. 339 MB/s when the match ratio is 1% and the head size is 6000 states), 46% for the Clam AV type 1 pattern set (678 MB/s vs. 465 MB/s when the match ratio is 1% and the head size is 7000 states), and 55% for the Clam AV type 3 pattern set (541 MB/s vs. 349 MB/s when the match ratio is 32% and the head size is 1500 states). Although the FHBM algorithm can partition an AC-DFA more flexibly than the HBM algorithm, there is still space to improve. Given an AC-DFA, the boundary between the H-DFA and B-NFA constructed by the FHBM algorithm lies at depth i or depth $I + 1$. In our future work, we plan to further explore the relationship between throughput and

H-DFA/B-NFA boundary, and design an algorithm that can achieve higher throughput by partitioning an AC-DFA without the H-DFA/B-NFA boundary limitation with the FHBM algorithm.

Acknowledgments: This work was supported in part by the High Speed Intelligent Communication (HSIC) Research Center of Chang Gung University, Taiwan, and by grants from the Ministry of Science and Technology of Taiwan (NSC-101-2221-E-182-074 and MOST-104-2221-E-182-005) and Chang Gung Memorial Hospital (BMRP 942).

Author Contributions: Chun-Liang Lee and Tzu-Hao Yang conceived and designed the experiments; Tzu-Hao Yang performed the experiments; Chun-Liang Lee and Tzu-Hao Yang analyzed the data; Chun-Liang Lee and Tzu-Hao Yang contributed reagents, materials and analysis tools; Chun-Liang Lee wrote the paper. All authors have read and approved the final manuscript.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Handley, M.; Paxson, V.; Kreibich, C. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In Proceedings of the Symposium on USENIX Security, Washington, DC, USA, 13–17 August 2001; pp. 115–131.
2. Kruegel, C.; Valeur, F.; Vigna, G.; Kemmerer, R. Stateful intrusion detection for high-speed networks. In Proceedings of the Symposium on Security and Privacy, Oakland, CA, USA, 12–15 May 2002; pp. 285–293.
3. Paxson, V. Bro: A system for detecting network intruders in real-time. *Comput. Netw.* **1999**, *31*, 2435–2463. [CrossRef]
4. Tian, D.; Liu, Y.H.; Xiang, Y. Large-scale network intrusion detection based on distributed learning algorithm. *Int. J. Inf. Secur.* **2009**, *8*, 25–35. [CrossRef]
5. Beghdad, R. Critical study of neural networks in detecting intrusions. *Comput. Secur.* **2009**, *27*, 168–175. [CrossRef]
6. Wu, J.; Peng, D.; Li, Z.; Zhao, L.; Ling, H. Network intrusion detection based on a general regression neural network optimized by an improved artificial immune algorithm. *PLoS ONE* **2015**, *10*, e0120976. [CrossRef] [PubMed]
7. Antonatos, S.; Anagnostakis, K.G.; Markatos, E.P. Generating realistic workloads for network intrusion detection systems. In Proceedings of the 4th international workshop on Software and performance (WOSP'04), Redwood Shores, CA, USA, 14–16 January 2004; pp. 207–215.
8. Cabrera, J.B.; Gosar, J.; Lee, W.; Mehra, R.K. On the statistical distribution of processing times in network intrusion detection. In Proceedings of the Conference on Decision and Control, Woburn, MA, USA, 14–17 December 2004; Volume 1, pp. 75–80.
9. Erdem, O. Tree-based string pattern matching on FPGAs. *Comput. Electr. Eng.* **2016**, *49*, 117–133. [CrossRef]
10. Kim, H.; Choi, K.-I. A pipelined non-deterministic finite automaton-based string matching scheme using merged state transitions in an FPGA. *PLoS ONE* **2016**, *11*, e0163535. [CrossRef] [PubMed]
11. Kim, H. A failureless pipelined Aho-Corasick algorithm for FPGA-based parallel string matching engine. *Lect. Notes Electr. Eng.* **2015**, *339*, 157–164.
12. Chen, C.C.; Wang, S.D. An efficient multicharacter transition string-matching engine based on the Aho-Corasick algorithm. *ACM Trans. Archit. Code Optim.* **2013**, *10*, 1–22. [CrossRef]
13. Kaneta, Y.; Yoshizawa, S.; Minato, S.I.; Arimura, H.; Miyanaga, Y. A Dynamically Reconfigurable FPGA-Based Pattern Matching Hardware for Subclasses of Regular Expressions. *IEICE Trans. Inf. Syst.* **2013**, *E95-D*, 1847–1857. [CrossRef]
14. Tsai, H.J.; Yang, K.H.; Peng, Y.C.; Lin, C.C.; Tsao, Y.H.; Chang, M.F.; Chen, T.F. Energy-efficient TCAM search engine design using priority-decision in memory technology. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 962–973. [CrossRef]
15. Peng, K.; Tang, S.; Chen, M.; Dong, Q. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM. In Proceedings of the ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, Brooklyn, NY, USA, 3–4 October 2011; pp. 24–35.
16. Liu, R.T.; Huang, N.F.; Chen, C.H.; Kao, C.N. A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Trans. Embedded Comput. Syst.* **2004**, *3*, 614–633. [CrossRef]

17. Bacon, D.F.; Rabbah, R.; Shukla, S. FPGA programming of the masses. *Commun. ACM* **2013**, *56*, 56–63. [CrossRef]
18. Scarpazza, D.P.; Villa, O.; Petrini, F. Exact multi-pattern string matching on the cell/B.E. processor. In Proceedings of the Conference on Computing Frontiers, Ischia, Italy, 5–7 May 2008; pp. 33–42.
19. Schuff, D.L.; Choe, Y.R.; Pai, V.S. Conservative vs. optimistic parallelization of stateful network intrusion detection. In Proceedings of the International Symposium on Performance Analysis of Systems and Software, Philadelphia, PA, USA, 20–22 April 2008; pp. 32–43.
20. Vallentin, M.; Sommer, R.; Lee, J.; Leres, C.; Paxson, V.; Tierney, B. The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In Proceedings of the International workshop on Recent Advances in Intrusion Detection, Queensland, Australia, 5–7 September 2007; pp. 107–126.
21. Lee, C.L.; Lin, Y.S.; Chen, Y.C. A hybrid CPU/GPU pattern-matching algorithm for deep packet inspection. *PLoS ONE* **2015**, *10*, e0139301. [CrossRef] [PubMed]
22. Lin, Y.S.; Lee, C.L.; Chen, Y.C. A capability-based hybrid CPU/GPU pattern matching algorithm for deep packet inspection. *Int. J. Comput. Commun. Eng.* **2016**, *5*, 321–330. [CrossRef]
23. Lin, Y.S.; Lee, C.L.; Chen, Y.C. Length-bounded hybrid CPU/GPU pattern matching algorithm for deep packet inspection. *Algorithms* **2017**, *10*. [CrossRef]
24. Tran, N.P.; Lee, M.; Choi, D.H. Cache locality-centric parallel string matching on many-core accelerator chips. *Sci. Program.* **2015**, *2015*, 1–20. [CrossRef]
25. Zha, X.; Sahni, S. GPU-to-GPU and host-to-host multipattern string matching on a GPU. *IEEE Trans. Comput.* **2013**, *62*, 1156–1169. [CrossRef]
26. Tumeo, A.; Villa, O.; Chavarria-Miranda, D.G. Aho-corasick string matching on shared and distributed-memory parallel architecture. *IEEE Trans. Parallel Distrib. Syst.* **2012**, *23*, 436–443. [CrossRef]
27. Yang, Y.H.; Prasanna, V.K. Robust and scalable string pattern matching for deep packet inspection on multicore processors. *IEEE Trans. Parallel Distrib. Syst.* **2013**, *24*, 2283–2292. [CrossRef]
28. Knuth, D.E.; Morris, J.; Pratt, V. Fast pattern matching in strings. *SIAM J. Comput.* **1977**, *6*, 127–146. [CrossRef]
29. Boyer, R.S.; Moore, J.S. A fast string searching algorithm. *Commun. ACM* **1977**, *20*, 762–772. [CrossRef]
30. Aho, A.V.; Corasick, M.J. Efficient string matching: An aid to bibliographic search. *Commun. ACM* **1975**, *18*, 333–340. [CrossRef]
31. Manber Wu, S.; Manber, U. *A Fast Algorithm for Multi-Pattern Searching*; Technical Report TR-94-17; Department of Computer Science, University of Arizona: Tucson, AZ, USA, 1994; Available online: http://webglimpse.net/pubs/TR94-17.pdf (accessed on 24 May 2017).
32. Tuck, N.; Sherwood, T.; Calder, B.; Varghese, G. Deterministic memory-efficient string matching algorithms for intrusion detection. In Proceedings of the IEEE INFOCOM, Hong Kang, China, 7–11 March 2004; pp. 333–340.
33. Bremler-Barr, A.; Hay, D.; Koral, Y. CompactDFA: Scalable Pattern Matching Using Longest Prefix Match Solutions. *IEEE/ACM Trans. Netw.* **2014**, *22*, 415–428. [CrossRef]
34. Liu, C.; Pan, Y.; Chen, A.; Wu, J. A DFA with extended character-set for fast deep packet inspection. *IEEE Trans. Comput.* **2014**, *63*, 1925–1937. [CrossRef]
35. Head-body String Matching. Available online: http://sourceforge.net/projects/hbsm (accessed on 23 February 2017).
36. Snort.Org. Available online: http://www.snort.org (accessed on 23 February 2017).
37. ClamAV. Available online: http://www.clamav.net (accessed on 23 February 2017).