Chapter 4

FNP: A Pattern Matching Algorithm for Network Processor Platforms

4.1 Introductions on NIDSes and NPUs

Typically, firewalls are employed to increase network security by restricting access to designated points on the network. However, a firewall is unable to detect or prevent malicious activities originating from within the network. Network Intrusion Detection Systems (NIDS) have been designed to complement firewalls, and are designed to identify attacks against networks or a host that are invisible to firewalls, thus providing an additional layer of security. Rapid growth of network traffic has increased the importance of NIDS performance. Generally two main methods are used for intrusion detection, namely Pattern Matching and Statistical Analysis. The former method applies a static set of patterns and alerts to traffic sequences with known signatures. Meanwhile, the latter method detects anomalous events statistically by gathering protocol header information and comparing this traffic to known attacks, as well as by sensing anomalies. Pattern matching tools are excellent at detecting known attacks, but perform poorly when facing a fresh assault or a modification of an old assault. NIDSes that use statistical analysis perform worse at sensing known problems, but much better at reporting unknown assaults. Improved implementation of a NIDS should combine these two methods to improve network protection. Either way, NIDSes relies on exact string matching from network packet payloads against thousands of intrusion signatures. The performance of signature-based NIDSes has been shown to be dominated by the speed of the string matching algorithms used to compare packets with signatures [13]. Implementing a different algorithm achieves an up to 500 percent increase in performance for snort 2.0. A NIDS must employ an efficient string matching algorithm since an under-performing passive system drops many packets and may miss many attacks, while an under-performing inline system creates a bottleneck for network performance [13].

The rulesets are basically souls of a NIDS. This work analyzes and experiments with the popular Snort [48] ruleset. Snort is an open-source NIDS and probably contains more signatures than any commercial and open-source competitors. Numerous NIDS manufacturers convert Snort signatures into their own rulesets [56]. This work uses the Snort ruleset owing to its popularity and complexity. Since this work mainly focuses on the string matching algorithm, the processing complexity increases with number of patterns in the ruleset. Notably, string matching in NIDS is domain-specific in numerous ways. First, rule patterns occur in various sizes and each rule may specify multiple string patterns. Second, strings may have case sensitivity requirements. A rule simultaneously may involve both case-sensitive patterns and non-case sensitive patterns. Third, most rule patterns are ASCII characters and not fair distributions of characters, while the network traffic mostly involves binary data. Fourth, priorities among signatures and a matched signature with the highest priority are selected during multiple matches. All these differences increase the importance of designing an efficient and domain-specific string matching algorithm for NIDSes.

Network Processors deliver significant improvements in networking device (e.g. switches and routers) time-to-market, product lifetime, and system capabilities [38]. Initially networking devices were built with general purpose CPUs, and the software-based nature of these devices was the key to adapting to new protocol standards and additional network functionality requirements. Over time, to accommodate increasing network traffic, simpler and fixed-function devices that could be built with ASICs (Application Specific Integrated Circuits) were developed. These devices traded-off the programmability of software-based designs for speed derived from hardware. Network Processors, currently on the market, deliver hardware-level performance to software programmable systems. Network Processors retain the system development flexibility of software-based devices while also meeting high-speed requirements. A network processor is a highly integrated structure comprising micro-coded or hardwired accelerated engines, memory sub-systems, and high-speed interconnect and media interfaces for rapid packet processing. Network processors generally use pipelining, parallelism, and multi-threading to hide latency. Network processors also employ hardware accelerators for hashing function, tree searches, frame forwarding, filtering, and alteration [38, 52]. The increase in network utilization and the weekly expansion in number of critical application layer exploits means NIDSes designers must develop ways to accelerate their attack analysis techniques when monitoring a fully-saturated network, and moreover must maintain a good false positive to false negative ratio. Besides developing an ASIC specifically for NIDS, this work also considers adapting Network Processors to implement NIDS, an approach that combines flexibility with good performance.

The design presented here employs multi-thread for parallel processing and hardware accelerated hashing engine to identify matching entries via a linked list in the event of hash collision to save processor power. Hashing engine checks the linked entries individually from a given starting address until it identifies a matched entry or reaches the end of the linked list. As previously described, searching entries by hashing engine hides latency and improves performance owing to context switching before a search result is returned.

4.2 Previous Pattern Matching Algorithms in NIDS

Extensive research exists on general pattern matching algorithms. The Boyer-Moore [5] algorithm is widely used because of its efficiency in single-pattern matching problems. This algorithm uses two heuristics to reduce the number of character comparisons required for pattern matching. Both heuristics are triggered by mismatches. The first heuristic, commonly referred to as the *bad character* heuristic, works as follows: if the search pattern contains the mismatching character, the pattern is shifted so that the mismatching character is aligned with the rightmost position at which it appears in the pattern. Meanwhile, if the mismatching character does not appear in the search pattern, the pattern is shifted so that the first character in the pattern is one position later than that of the mismatching character in the given text. The second heuristic, commonly referred to as the *good suffixes* heuristic, works as follows: if a mismatch is found in the middle of the pattern, the search pattern is shifted to the next occurrence of the suffix in the pattern. The Boyer-Moore algorithm was designed for searching for a single pattern from a given text and performs well in this role. However, the current implementation of Boyer-Moore in Snort is not efficient in seeking multiple patterns from given payloads [3, 13].

Aho and Corasick [1] proposed an algorithm for concurrently matching multiple strings. Aho-Corasick (AC) algorithm used the structure of a finite automation that accepts all strings in the set. The automation processes the input characters individually and tracks partially matching patterns. The AC algorithm has proven linear performance, making it suitable for searching a large set of rule signatures [13]. Two different implementations exist for Snort, implemented by Mike Fisk and Marc Norton, respectively. This work tested both implementations and employed the latter in the present experiments because of its superior performance. However, the Norton implementation requires considerably more memory than the Fisk implementation (1024 versus 260 bytes per pattern character). By tracing the source codes, this work found ultimately that the Fisk implementation is based on the so-called failure-function AC algorithm while the Norton implementation is based on the so-called optimized AC algorithm. The running time of the optimized AC algorithm is independent of the pattern set, and depends on the length of the longest pattern in the ruleset [54] for the failure-function AC algorithm.

Coit, Stainford, and McAlerney implemented Gusfield's version of the Commentz-Walter algorithm called AC_BM [7] which uses suffix trees for the good suffix heuristic. The algorithm is a Boyer-Moore like algorithm applied to a set of keywords held in an Aho-Corasick like keyword tree overlaying the common prefixes of the keywords. AC_BM searches multiple-pattern simultaneously and operates from 1.02 to 3.32 times as fast as the Boyer-Moore implementation on Snort. However, several unresolved issues have hampered their works with the full Snort ruleset. First, AC_BM reorders the rules despite the implicit ordering of Snort rules, meaning rules are not supported by priorities. Second, AC_BM requires additional structures to search non-case sensitive patterns. Third, AC_BM can only be applied to rules with a single content string to be matched. Furthermore, the efficiency of AC_BM depends heavily on the length of the shortest pattern being searched for, since the maximum number of shifts is bound to this value. Several Snort rules have a content length of just one (For example, sid 614, BACKDOOR hack-a-tack attempt [48]), which strongly affects AC_BM performance.

Concurrently with Coit's work, Fisk and Varghese designed a set-wise Boyer-Moore-Horspool (SBMH) algorithm [13], adapting the Boyer-Moore-Horspool [17] algorithm to simultaneously match a rule set. The set of patterns can be compared to any position in the text quickly by storing the reversed patterns in a trie.

Their experiments showed that this algorithm is faster than both the Aho-Corasick and Boyer-Moore algorithms for medium-size pattern sets. However, like AC_BM, the maximum number of shifts also is bounded by the length of the shortest pattern (denoted as *LSP*) in the pattern set.

Markatos, Antonatos, Polychronakis, and Anagnostakis have designed an exclusion-based signature matching algorithm known as ExB [33]. ExB is based on a simple logic, namely: If pattern *P* contains at least one character not in text *T*, then *P* is not in *T*. Every time a new payload arrives, the payload is preprocessed to construct an occurrence bitmap to record the occurrence of distinct characters within the payload. ExB then identifies the patterns individually to check whether any characters appear in the pattern but not the payload. If such characters do exist then the pattern is skipped since matching is impossible. Otherwise, the Boyer-Moore algorithm is invoked to search the pattern in the given payload. The basic concept of E^2xB is the same as that of ExB, with the two methods differing only in the method used to denote a match [3]. The effectiveness of both ExB and E^2xB decreases significantly as the number of rules exceeds 1000, a phenomenon possibly caused by the effect of increasing false-match rates. ExB and E^2xB, essentially linear matching algorithms, were designed when Snort used the linear Boyer-Moore algorithm as its default search engine. The Snort 2.0 has given up such linear matching algorithms and uses the well-known MWM [57] and AC algorithms as its default search engine. Additionally, [3] demonstrated that the implementation performance of these algorithms is better using a Pentium-3 1 GHz processor with 512KB L2 cache than a Pentium-4 1.7 GHz processor with 256KB L2 cache. Their experimental results demonstrated that ExB performance depends significantly on the cache size. Accordingly, when these algorithms are implemented over the network-processor platform, performance suffers due to large memory access latency, since generally internal memory size is extremely small when the cache memory is absent.

The MWM algorithm is another widely used multi-pattern matching algorithm designed by Wu and Manber [57]. The current implementation of Snort uses this algorithm as the default engine if the search-set size exceeds ten. The MWM algorithm uses the Boyer Moore algorithm with a two-byte shift table established by pre-processing all patterns. This algorithm performs a hash on the two-byte prefix into a group of patterns, which then are checked beginning from the final character when partially matching occurs. The MWM algorithm is used in agrep [57] and has been shown to deal with large amounts of patterns efficiently. However, like AC_BM and SBMH, the performance of the MWM algorithm also depends considerably on the *LSP*, because the maximum number of shifts equals this value minus one.

4.3 Design and Implementation of FNP

This work addresses the string matching problem formally before introducing the proposed *FNP* algorithm.

Given an input text $T = t_0, t_1, \ldots, t_n$, and a finite set of strings $P = \{P_1, P_2, \ldots, P_r\}$, the string matching problem involves locating and identifying the substring of T which is identical to $P_j = a_0^j, a_1^j, \ldots, a_{m-1}^j$, $1 \le j \le r$, where

$t_{s+i} = a_i^j$, $0 \le i \le m-1$. And this equation can be also denoted as

$t_s \ldots t_{s+m-1} = a_0^j \ldots a_{m-1}^j$

4.3.1 FNP Algorithm

The proposed *FNP* algorithm is exclusion-based and its implementation is extremely straightforward. *FNP* is based on the following simple reasoning: For an arbitrary pattern P_j = $a_0^j, a_1^j, \ldots, a_{m-1}^j$ if w sequential bytes of T can be found from location s, where

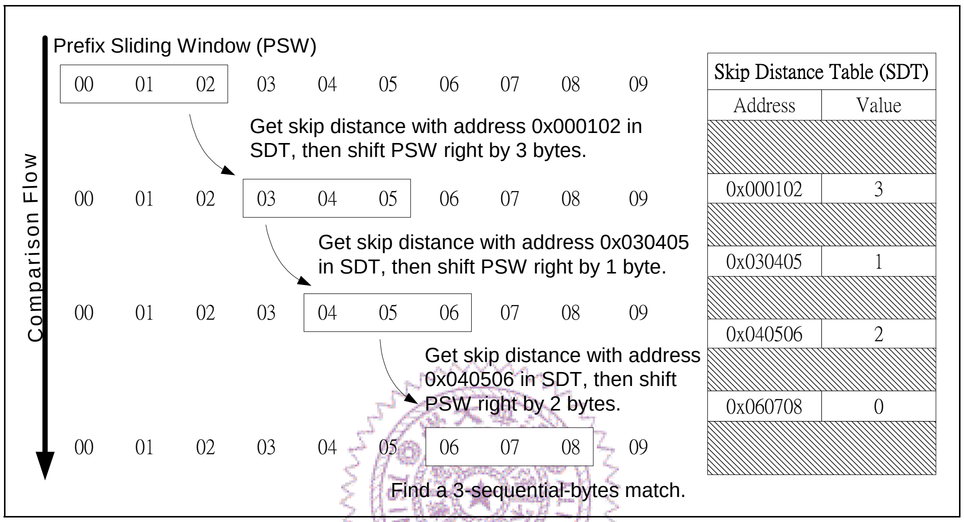
$$t_{s+i} \ldots t_{s+w-1} \neq a_0^j \ldots a_{w-1-i}^j \quad , i = 0, 1, 2, \ldots , w-1$$

Then, the w sequential bytes do not contain any i-bytes prefix of P_j as its suffix, where $1 \leq i \leq w$. Therefore, the w sequential bytes can be skipped during searching. On the other hand, if w sequential bytes can be found in T that also is a prefix of P_j, then comparing the remaining $(m - w)$ bytes from this position is worthwhile.

To clarify this point, this study uses a Prefix Sliding Window (denoted as *PSW*) with length w which shifts from the leftmost byte to the rightmost byte of T. Every time the *PSW* shifts, an attempt is made to determine whether S, the w sequential bytes covered by *PSW*, contains $a_0^j \ldots a_{k-1}^j$ of pattern P_j, where $1 \leq k \leq w$. If no such pattern exists whose first k sequential bytes are contained by S, then the *PSW* can be shifted w bytes to the right. On the other hand, if a pattern P_j exists, where

$$S_{w-k} \ldots S_{w-1} = a_0^j \ldots a_{k-1}^j$$

then the *PSW* is shifted right by $w - k$ bytes. Even if w sequential bytes in T are found that also is a prefix of P_j, false matches certainly will still exist (even if the first w bytes match, the remaining $m - w$ bytes still may fail to match). However, later this work shows the numerous unnecessary comparisons that this approach can eliminate, making the loss of false matches affordable. The remaining problem is how to determine whether S contains the prefix of a pattern P_j. If w is sufficiently small, for example three, a table named the Skip Distance Table (denoted as *SDT*) can list all possibilities of three sequential bytes. This approach is quite intuitive and also effective. For example, if a rule signature exists with the content string ‘abcd’, identical to {0x61, 0x62, 0x63, 0x64} in ASCII code, then the table entry of address 0x000061 to 0xFFFF61 (last byte remains unchanged) is set to two. Accordingly, when *S* falls into this range, it should shift right by two bytes so that the first byte of its new location will be aligned with ‘a’. Table entries of address 0x006162 to 0xFF6162 (last two bytes remain unchanged) then are set to 1.



Prefix Sliding Window (PSW)
Comparison Flow
00 01 02 03 04 05 06 07 08 09
Get skip distance with address 0x000102 in SDT, then shift PSW right by 3 bytes.
Get skip distance with address 0x030405 in SDT, then shift PSW right by 1 byte.
Get skip distance with address 0x040506 in SDT, then shift PSW right by 2 bytes.
Find a 3-sequential-bytes match.
Skip Distance Table (SDT)
Address
Value
0x000102
3
0x030405
1
0x040506
2
0x060708
0

Figure 14. *PSW* Movement

Accordingly, when *S* falls into this range, it should shift right by one byte so that the first two sequential bytes of its new location will be aligned with ‘ab’. Subsequently, the table entries of address 0x616263 are set to zero. When *S* is identical to 0x616263, it matches the first three sequential bytes of the content string ‘abcd’. Other table entries are set to 3. If *S* falls into this category, it can be shifted three bytes to the right safely. Figure 14 illustrates how *PSW* moves with the entry values in *SDT*.

The following details the design of the *FNP* algorithm. This work first explains the table structures adapted in the present design, and then divides the algorithm into off-line pre-processing and runtime processing. The off-line pre-processing constructs necessary rule tables and lookup tables while the runtime processing processes the payload and identifies the matches. For simplicity this work assumes $w = 3$.

4.3.2 Table Definitions

The Skip Distance Table (*SDT*) is used to lookup the number of bytes that *PSW* should shift right, and also to check whether *S* matches the pattern prefix. The size of *SDT* is 2^{w*8}, namely 16M if $w = 3$. During comparison the instant integer value of *S* is taken as the entry address for looking up the skip distance. For example, if *S* is 0x006162, then the value stored in address 0x006162 is returned as the skip distance. If the returned value is zero then three bytes are matched, otherwise *PSW* shifts right by the returned-value bytes.

The Rule Hashing Table (denoted as *RHT*) preserves the signature content strings. The *FNP* is designed to perform multiple-pattern matching simultaneously, and hashing is used to distribute pattern content strings. The *RHT* is a one-dimension hashing table which stores the link pointers to collision entries, the hashing key for matching, content string length, Rule-ID, and the remainder of the content string. The hashing key is the first four sequential bytes ($a_0^j \ldots a_3^j, 1 \le j \le r$) of the content string. If multiple rules have the same hashing value and a collision occurs, a linked list is maintained to preserve the collision entries, and here the lookup coprocessor is used to accelerate the search. As previously described, the lookup coprocessor traverses the linked list from the given starting address to identify an entry with a matched hashing key without compromising CPU power owing to the occurrence of a context switch. If a match is identified then this work begins to compare the remaining $m - 4$ bytes.

Rule Status Table (denoted as *RST*) is designed for multiple purposes: to record whether a pattern has been matched previously, to accommodate multiple-content patterns, and to maintain rule priorities. *RST* is exactly the same size as the number of content string entries in *RHT*, and the order of *RST* entries is exactly the same as the Rule-ID in *RHT*. Rule priority increases with decreasing Rule-ID. An *RST* entry comprises a MATCH flag, a HEAD flag, and several link pointers to other entries. Multiple-content strings are taken apart so that every content string occupies an entry in *RHT*. The longest string then is selected as HEAD from a multiple-content rule, and the HEAD entry preserves links to NON-HEAD entries belonging to the same rule. Single-content strings individually are marked as HEADs. The *RST* actually is a linked list on which every entry is connected individually, and thus the lookup coprocessor can be employed again to seek the highest priority entry.

4.3.3 Off-line Pre-Processing

This stage involves constructing *SDT, RHT, and RST.*
During initialization, all entries in *SDT* are set to 3. For patterns with length of 4 or greater, the corresponding entries of their first three sequential bytes ($a_0^j \ldots a_2^j$) and second three sequential bytes ($a_1^j \ldots a_3^j$) in *SDT* are set to zero. Every table entry whose last (rightmost) 8-bits of address is identical to any one-byte prefix of the patterns and whose entry value is not zero is set to 2, and every table entry whose last (rightmost) 16-bits of address is identical to any two-byte prefix of the patterns and whose entry value is not zero is set to 1. Notably, for patterns with length of 3 or below the corresponding entries can be marked with a special flag (say, its Rule-ID) to denote a match. For example, for patterns containing only '0x7B' as their content string, then the entries whose first, second, or last 8-bits of addresses are 0x7B are marked. This method can identify matching patterns with length of 3 or below and only one memory access.

The next step is to insert rules into *RHT*. The multiple-content signatures are taken apart so that every content string has its own entry. The first four sequential bytes of every content string are used to derive the hashing value of that string. In the event of a collision, a linked list is maintained to preserve the collision entries. Other fields like pattern length, content strings, and rule ID also are filled in the *RHT*.

The MATCH flag of all *RST* entries is set to false, and entries corresponding to single-content rules are marked as HEADs. For multiple-content rules only the longest content entries are marked as HEADs. When several entries belong to the same multiple-content rule then the HEAD entry will contain pointers to other entries. During comparison, if a pattern is found in *T*, the MATCH flag of the corresponding entry is set to true so that compare this entry again is unnecessary for the remainder of *T*. Once *T* has been screened out entirely, hash coprocessor is used again to search for the HEAD entry with MATCH flag set so that there is no need to perform a linear search to identify the highest priority matched entry. After finding such an entry, whether or not that entry points to other entries is checked. A multiple-content rule is matched if every content matches *T*. Figure 15 illustrates an example of inserting a multiple-content rule into *RST* and *RHT*. This multiple-content rule contains three content strings: "*abcdefgh*", "*123456*", and "*123400*". The hashing results of keys *abcd* and *1234* are assumed to be 1 and 3, respectively. In *RST*, string "*abcdefgh*" is chosen as the HEAD entry, with pointers linked to "*123456*" and "*123400*". Meanwhile, "*123456*" and "*123400*" are linked in the same linked list because their keys (*1234*) are identical in *RHT*.

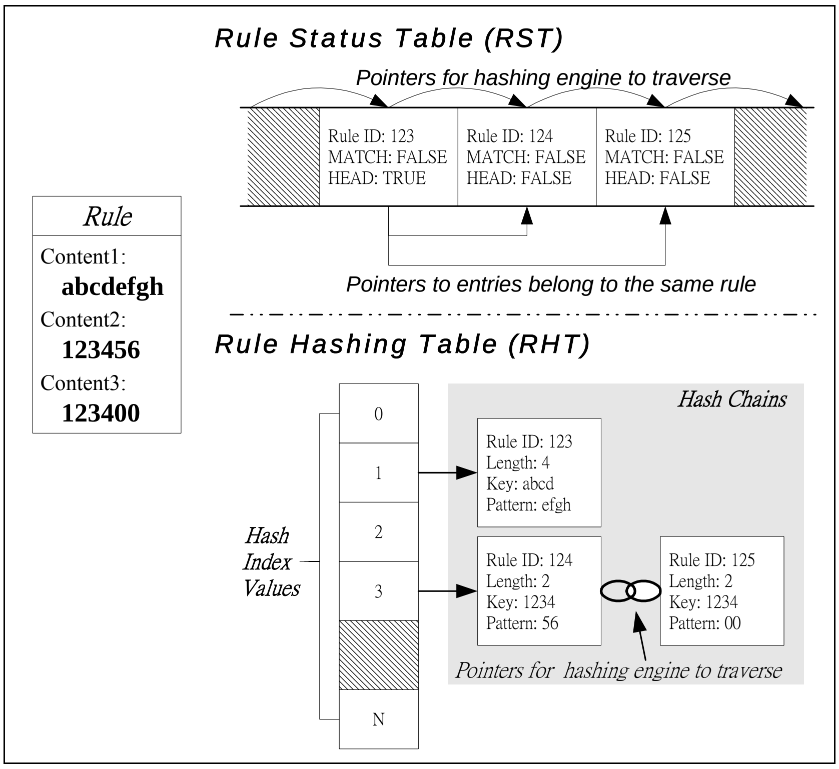


Figure 15. An Example of the RST and RHT.

4.3.4 Runtime Processing

The matching procedure of the proposed *FNP* algorithm is quite simple. Initially *PSW* is aligned with the first byte of the incoming payload. The string within the *PSW* S ($t_0 ... t_2$) then is fetched, and its skip distance is looked up in the *SDT*. If skip distance N does not equal to zero, then *PSW* is shifted right by N bytes in the next round. If N is zero, an attempt is made to fetch the next three sequential bytes ($t_1 ... t_3$) and lookup its skip distance. If the skip distance for the second three sequential bytes, M, is not zero, *PSW* is shifted right by $M+1$ bytes for the next round. If M is zero, then two consecutive 3-sequential-bytes both appear in the signature content. From the present experiment, checking the 2nd three sequential bytes reduces false matches by over 80%. These two 3-sequential-bytes then are combined to one word and thrown to the Hashing Lookup Engine for further searching. Notably, Network Processor, like most CPUs, accesses memory on a word (32-bits) basis, so that two consecutive

3-sequential-bytes can be obtained via a single memory access. When the returned skip distance is zero or one, fetching the next 3-sequential-bytes from memory is unnecessary since they are already stored in the register. If a word needs to be searched in *RHT*, this job is left to the lookup co-processor, after which context switching is performed. After this thread wakes up again, the lookup co-processor either returns the address of the matching entry or sets a bit indicating the failure of matching. If lookup fails, *PSW* is shifted one byte right to continue. If lookup succeeds, then whether the MATCH flag of this entry has been set in *RST* is checked, because this avoids the need to waste time on rechecking matching entries. If the entry found previously has not been matched then an exact matching is conducted between the payload and the remaining content. If the remaining content matches the payload, then the MATCH flag of the corresponding *RST* entry is set to TRUE to indicate a match. Meanwhile, if no match exists the following entries are searched again using a lookup coprocessor until all of the collision entries have been checked. The primary payload matching procedure of the *FNP* algorithm is as follows:

FNP Algorithm

Input: A text string $T = t_1, t_2, .., t_n$. SDT, RHT, and RST.

Output: *RST* with matched entries.

```
begin
    $i \leftarrow 0$
    **while** $(i < (n - 2))$ **do**
        begin
            $s \leftarrow SDT[t_i \ldots t_{i+2}]$
            $d \leftarrow s >> 8$      {upper byte is the Rule-ID of the short pattern}
            **if** $(d > 0)$ **then**
```

```
                        RST[d].Matched ← true
                    $N$ ← s & 0xff        {the lower byte is the skip distance}
                    **if** ($N > 0$) **then**
                        i ← i + N
                    else
                        begin
                            M ← SDT[$t_{i+1}...t_{i+3}$] & 0xff
                            **if** ($M > 0$) **then**
                                i ← i + M + 1
                            **else**
                                Search in *RHT* with key $t_i...t_{i+3}$
                        **end**
                **end**
        end
```

After going through the entire payload, a matching entry with the highest priority is selected from *RST*. If no matching entry exists, then the payload is clean. Notably, the work of searching in *RST* can be performed by the lookup co-processor. Consequently, the whole table does not need to be traversed to select the entry with the highest priority by CPU.

If the ruleset contains non-case sensitive rules, the keys in *SDT* and *RHT* are converted into lower case during the pre-processing time and a flag is used to denote the case attribute. During comparison *S* is converted into lower case to lookup in *SDT* and *RHT*. If the lookup succeeds then the case attribute is checked to confirm whether or not the payload needs to be compared.

4.4 Analysis of FNP

Interestingly, matches are rare in multiple-pattern searching of NIDSes. From our observations, the distribution of characters in the current Snort signature content is not uniform. Among 256 possible characters, only 149 distinct characters appeared in the Snort full ruleset released on Aug 10, 2003. Regarding occurrence frequency, 95% of characters in the Snort ruleset are 7-bit ASCII codes. The fact of that most signatures comprise ASCII characters while most real network traffic comprises uniform distribution characters indicates that matches are unlikely to be frequent. A linear-time algorithm like AC is optimal in the worst case, but in the typical case it is more desirable to keep the algorithm simple [17] and skip a large portion of the text during searching [9, 24]. Since matches are infrequent, algorithms that skip as much of payloads as possible perform better than others. Both SBMH, AC_BM are designed to compare from right to left and thus maximize skip number. However, their maximum numbers of shifts are bound to LSP. Figure 16 illustrates the distribution of content length in the full Snort ruleset, and surprisingly the figure contains 69 signatures (multiple-content signatures included) with a content length of one. Consequently, algorithms whose performance depends on *LSP* are inefficient if the LSP of search set is small. As for the MWM algorithm, the maximum number of shift bytes equals the value of *LSP* minus 1 [57], which also reduces the desirability of the MWM if the *LSP* is very small.

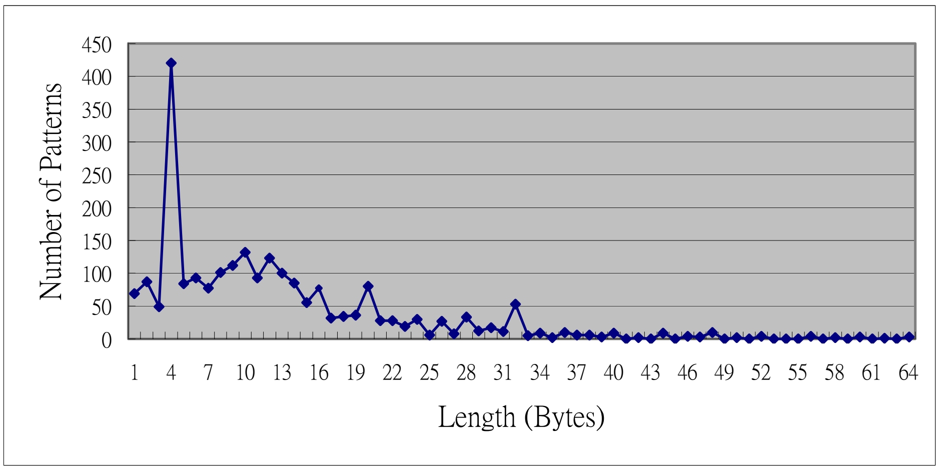


Figure 16. Distribution of pattern length in the current Snort ruleset

Since *FNP* is designed mainly for network processors, the key to evaluating a pattern matching algorithm is the number of memory accesses during searching. Memory accesses are very expensive on Network Processors, since memory access latency influences implementation speed markedly. Multi-threading is the most common approach to hiding latency [44]. However, in a computationally intensive application like multi-pattern matching, all contexts tend to access the memory simultaneously. The following aims to show that the algorithm presented here can reduce the number of memory accesses and thus improve overall performance significantly.

This work applies a probabilistic model to examine the average performance of this *FNP* algorithm. Let N_k denote the number of k occurring in *SDT*, $k = 0, 1, 2, 3$. Additionally, let A denote alphabetical space size, which is 2^{24} when $w = 3$. The expected value of skip distance for each move thus is:

$$E = ((N_1 + (2 * N_2) + (3 * N_3)) / A) + ((N_0 / A) * (((2 * N_1) + (3 * N_2) + (4 * N_3)) / A))$$

In the current Snort full ruleset, when A is 2^{24}, $N_0 = 1796$, $N_1 = 138503$, $N_2 = 6742046$ and $N_3 = 9894871$, then E is 2.58. If no match exists, only two memory accesses are required for shift execution, one for payload fetching and another for lookup in *SDT*.

Thus if no two consecutive 3-sequential-bytes are matched in a 750 bytes packet, less than 581 memory accesses are required, better than the 1,500 memory accesses required using AC algorithm.

If a match occurs, since the hashing engine of the Network Processor is used to identify entries with matching keys, this work only counts memory access for exact string comparison, as follows

*(AVERAGE_PATTERN_LENGTH/4) * 2*

Figure 16 reveals that over 95% of patterns have a length of below 32, while over 75% patterns have a length of below 16. Assuming an average pattern length of 16 bytes, and comparing 4 bytes at a time, eight memory accesses are required for string matching. If a 750 bytes payload contains *p* matches, then the total number of memory accesses is less than 581 + (8 * *p*). Since matching is rare, the average number of memory accesses of *FNP* is better than the 1,500 of AC algorithm.

4.5 Experiments over FNP

To verify the effectiveness of the proposed *FNP* algorithm, its performance was evaluated against the previously mentioned *SBMH, AC*, and *MWM* algorithms. Because of the difficulty of implementing all these four algorithms with Network Processor micro codes, some experiments were implemented on general PCs to simulate the network-processor environment. Nevertheless, the *FNP* was implemented using the Vitesse IQ2000 [52] Network Processor. The relations between the program performance and the number of memory accesses during execution also were assessed. Combining the results of simulations and network-processor experiments substantiates the efficiency and practicability of the *FNP* algorithm. Additionally, this work demonstrated that the *LSP* influences multi-pattern matching algorithm performance significantly, and moreover that the *FNP* algorithm is more efficient and faster than the other three algorithms provided $LSP \leq 4$.

The current Snort ruleset, containing 1,942 rules with 2,475 patterns, was employed as the default searching pattern. The full-packet traces can be derived from the "Capture the Capture The Flag" (CCTF) project held in DEFCON [10] annually. The DEFCON9 packet traces used in the present experiments were the most up-to-date available.

4.5.1 Evaluation of the number of memory accesses

As previously described, the proposed *FNP* algorithm requires fewer memory accesses given small *LSP*. The four algorithms are evaluated using different search set sizes and *LSPs* by counting number of memory accesses. The packet trace (900MB) defcon_eth0.dump2 [10] was employed to generate the test traffic more realistically. Trace defcon_eth0.dump2 was selected because of its low compression rate compared to other packet traces, and because the content of this trace is considerably more complicated, thus increasing test fairness. Figure 17 illustrates the results of these four algorithms for different search set sizes and *LSP*s, where *FNPw3* denotes *PSW* size of 3. The case involving the MWM algorithm with $LSP = 1$ was not assessed because the MWM algorithm does not support this situation. The *FNP* algorithm clearly outperformed the other three algorithms for $LSP \leq 4$. Notably, 35% of the patterns in the latest Snort ruleset fall into this category. In this experiment, approximately 740M memory accesses are required for *FNP* to process 900M data. This experimental result is quite close to previous analyses in which two memory accesses were required for processing 2.58-bytes data. On the other hand, the *FNP* algorithm markedly reduces the number of false checks, which generally increases with search-set size, because lookup in *RHT* is invoked only if two consecutive 3-sequential-bytes matches occur.

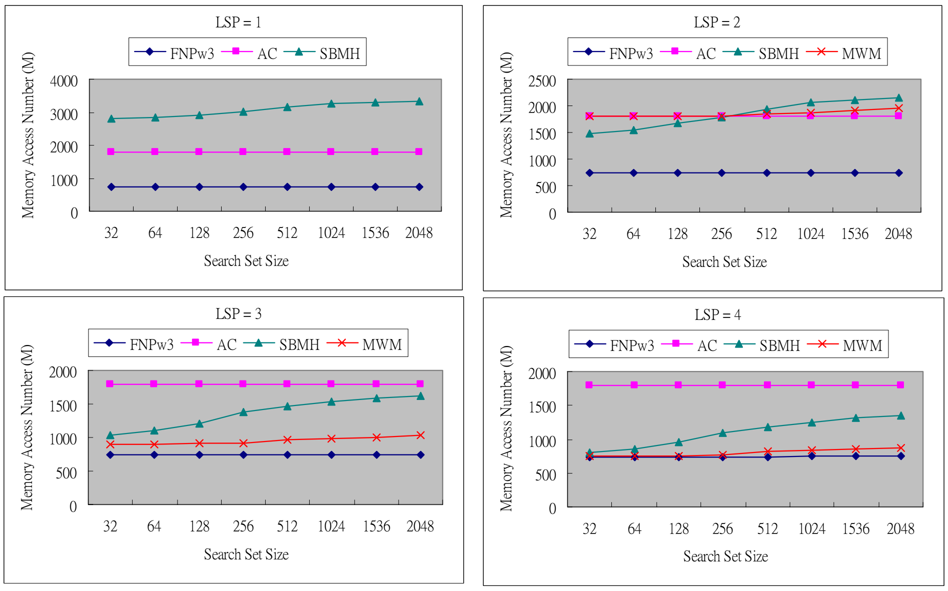


Figure 17. Number of memory accesses during pattern matching processing

Notably, two major influences affect the performance of multi-pattern matching algorithms in the NIDS, namely: *LSP* value and the pattern ruleset size. Interestingly, previous works focused on the latter factor only, while neglecting the former factor. Figure 17 reveals that search-set size does not influence the number of memory accesses required for the MWM algorithm to complete the multi-pattern matching, but for *LSP* = 2,3,4 the required number of memory accesses is approximately 1800M, 950M, 800M, respectively. The SBMH algorithm displays the same phenomenon. This phenomenon indicates that value of *LSP* is even a major influence on the performance of multi-pattern matching algorithms.

4.5.2 Performance Evaluation on a general PC with disabled cache memory

Most Network Processors lack cache memory and internal memory size is usually small. To simulate this condition the present experiments were run on a general PC with the L1/L2 cache memory turned off. A Pentium-4 processor PC running at 1.7 GHz with 512 MB of DDR memory was employed for the experiments. The host operating system was Windows XP, and the packet trace was defcon_eth0.dump2 in DEFCON9. Figure 18 shows the total processing time required for these four algorithms to search the defcon_eth0.dump2 with different search-set sizes and *LSP*s. The timer counts the pattern matching procedure only and excludes the file-loading time and other operations.

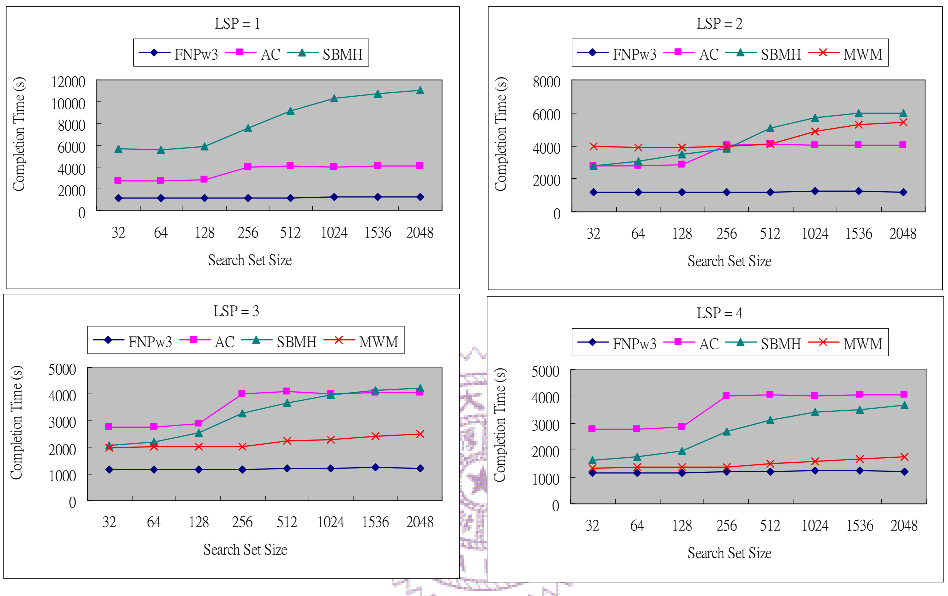


Figure 18. Completion time comparison using a cache-disabled PC

Figure 18 reveals that the processing time of the *FNP* algorithm is less than that of the other three algorithms given *LSP* ≤ 4. This phenomenon again highlights the importance of *LSP* to the performance of multi-pattern matching algorithms, such as MWM and SBMH. The MWM algorithm should outperform *FNP* when *LSP* ≥ 5. However, given that 35% of the patterns in the current Snort ruleset have lengths of 1, 2, 3, or 4. Although the NIDS signatures are usually partitioned into groups by L3/L4 header fields (for example IP pairs and port pairs), most likely the *LSP* ≤ 4 for each of the groups. Moreover, most NIDSes accept user-defined signatures and these signatures may lower the *LSP* further. Such a fact implies that the *FNP* performs better than other multi-pattern matching algorithms in most cases. Actually, a hybrid algorithm can be employed where *FNP* is applied for groups with *LSP* ≤ 4, and another algorithms (such as MWM) can be employed for groups with *LSP* ≥ 5.

Figure 18 also demonstrates the scalability of the *FNP* algorithm. Search set size clearly does not significantly influence *FNP* algorithm performance. Surprisingly, *AC* algorithm performance is not independent of search-set size. The reason for this phenomenon appears to be the large memory consumption of the *AC* algorithm (1024 bytes per pattern character). For larger search sets the state automation structure may exceed the size of a memory page within a DRAM cell, meaning that extra row-precharging time is required.

4.5.3 Performance Assessment with a Randomly Generated Ruleset

To evaluate the scalability of *FNP* algorithm, a test by searching the defcon_eth0.dump2 packet trace with randomly generated rulesets is also conducted on the PC with turned-off cache. The length of the ruleset patterns varies between 1 to 128 bytes. Table IV lists the relationship between searching time, search-set sizes, and expected values of skip distance (*E*). The searching time increases very slowly with search-set size. This phenomenon shows that the *FNP* algorithm is very efficient and can accommodate a large set of rules. Notably, the searching time in Table IV is longer than in Figure 18. This phenomenon occurs because the rulesets adapted in Table IV are generated randomly, while the Snort ruleset mainly comprises ASCII codes, so that the expected skip distance values in Table IV are smaller than those in Figure 18.

Table IV. Scalability test for the FNP algorithm with randomly generated rulesets

Ruleset	Time (s)	N_0	N_1	N_2	N_3	E

Size						
1024	1573	1962	257740	16193787	323727	2.003934
2048	1590	3902	509269	16133127	130918	1.977443
4096	1602	7696	990620	15648039	130861	1.94873
8192	1638	15276	1898969	14732145	130826	1.894513
16384	1741	30351	3526308	13089739	130818	1.79724
32768	1873	60453	6249190	10336756	130817	1.633977

4.5.5 Implementing FNP on a Network Processor Platform

To further evaluate the practical performance of the *FNP* algorithm, this work implements it on the Vitesse IQ2000 Network Processor platform. The IQ2000 has four 200 MHz RISC Packet Processing Engines (*PPE*s), each containing five sets of 32-bit registers, allowing up to four separate contexts to be active simultaneously. Each PPE also contains a lookup co-processor used to search for a given key in a specified linked-list. This facility can be used to search both the *RST* and *RHT*. Each PPE contains 2K-byte internal memory, and 512 bytes are assigned to each context. The system also has 512MB Direct Rambus DRAMs (RDRAMs) as the main memory.

To write the micro-code program efficiently, the IQ2000 technical documents suggest reducing the number of direct RDRAM accesses and trying to move data into the internal memory instead. In the present case, since the other tables such as *SDT* are too large to fit into the 2K-byte local memory, manipulating the packet payloads is the only way to reduce the number of direct RDRAM accesses. To confirm the impact of memory access number on performance, this work implements the *FNP* algorithm using two different methods. The first method (Exp1) is to access the packet payloads from RDRAM directly eight bytes at a time, with the next eight bytes being fetched each time *PSW* moves beyond the boundary of the current 8-byte payload. Meanwhile, the second method (Exp2) involves first fetching the payloads via DMA into internal memory 384 bytes a time, then fetching the next 384-byte of data through DMA if the *PSW* exceeds the boundary of 256 bytes. As the lookup co-processor, the context switch occurs during data transfer using DMA co-processor to hide the latency. Notably, the reason the boundary is set in the 256th byte is a heuristic for handling the situation in which a match occurs near the boundary, and this heuristic guarantees that the matched payload must already have existed in embedded local memory.

The network processor platform is designed to handle traffic of several hundreds Mbps and it is difficult to replay the Defcon9 traces in such high speed. Therefore, the SmartBits 6000B [51] and SmartApplication [51] are employed to generate the UDP traffic in Gigabit rate. Figure 19 illustrates the throughput of both experiments. Notably, the performance measurement results presented here are inline forwarding rates, not passive processing rates. The SmartApplication generates UDP packets in different sizes, and obviously the performance of the *FNP* program is better for small packets than for large packets. This phenomenon appears related to the fact that the program presented here ignores the header parts (MAC header, IP header, and UDP header) of a packet, and the proportion of the payload in a small-size packet is smaller than that in a large-size packet. Figure 19 illustrates that the program in Exp 2 is more efficient than that in Exp1. The only difference between the programs in Exp 1 and Exp 2 is the method of moving packet payloads. Take the 1514-byte UDP packet in our test for example; the program in Exp 2 eliminates 365 RDRAM accesses by using a six times DMA transfer, the latency of which can be hidden to achieve an approximately 30% improvement in performance. The testing results demonstrate the point that reducing the number of memory accesses during processing significantly improves program performance.

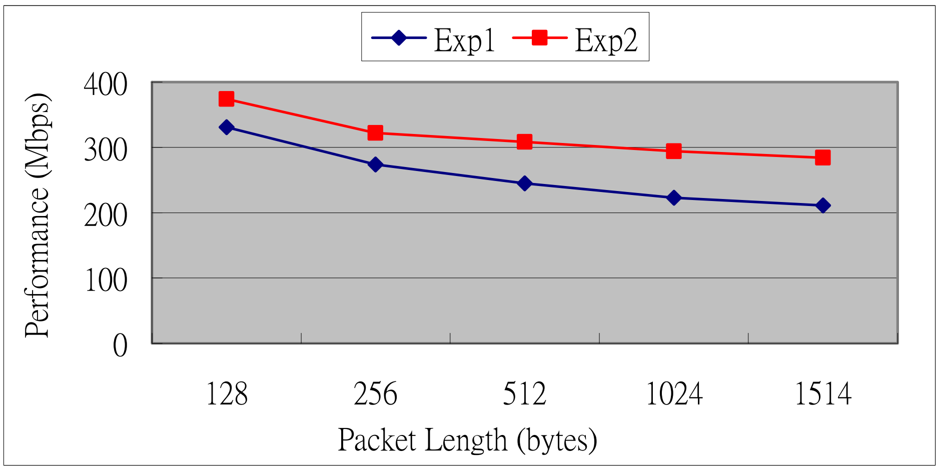


Figure 19. The performance of the FNP algorithm with different packet lengths

Coding in micro-code language is not easy because of its poor readability and its dependency on hardware characteristics. For example, the 8-byte data in Exp1 could be moved either through DMA or through two Load-Word (LW) instructions. This work demonstrates that moving data through DMA is more efficient if the data length exceeds 24 bytes, however in the present case it is faster to use the 2-LW instructions. This change achieves nearly a 20% performance improvement. The experimental results and the coding experience suggest that manipulating number of memory accesses is sensitive to the success of the program performance. Figure 17 reveals that the *FNP* algorithm is more efficient in number of memory accesses than the other three algorithms when $LSP \leq 4$. Therefore the *FNP* algorithm appears more suitable and efficient than alternatives in this situation.

Since the number of memory accesses significantly influences program performance, the present design also should benefit greatly from the hashing engine. Without the hashing engine, the program must traverse the *RH*T and *RST* several times when processing a packet by accessing RDRAM directly, and result in significantly downgrading performance. The present design not only uses the hashing engine to improve throughput, but also maintains rule priority without sacrificing performance.

Notably, program performance obviously depends on hardware capacity. We believe that performance can be improved markedly by using more high-end Network Processor Units, like Vitesse IQ2200 [52] (with four 400 MHz PPEs), Intel IXP2400 [22] (with eight 600 MHz PPEs), or even Intel IXP2800 [22] (with 16 1.4 GHz PPEs).

4.5.6 FNP on general PCs

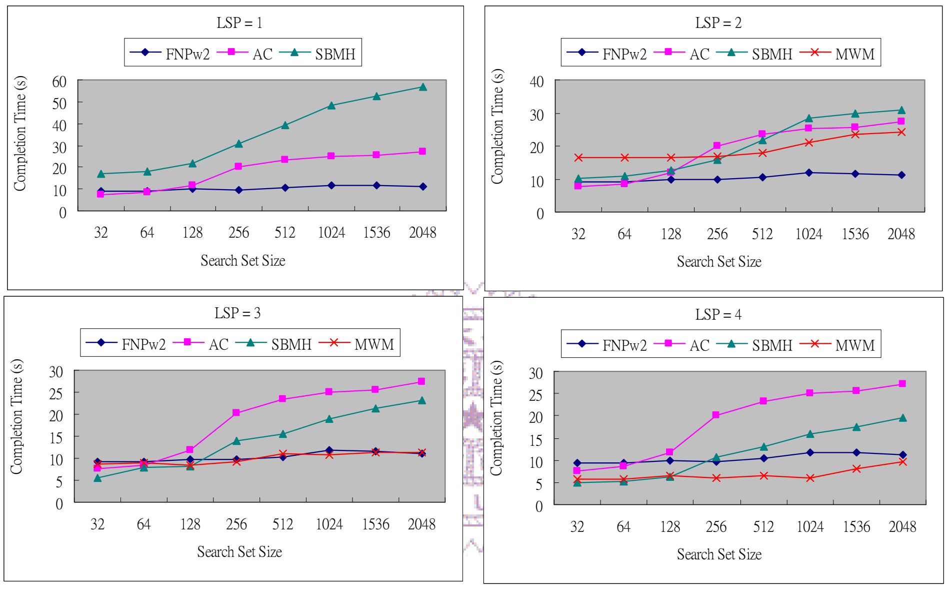


Figure 20. Completion time comparison by using general PC with cache turned on

Current Snort (2.0) takes AC and MWM algorithms as default pattern matching engines, and this section aims to demonstrate that the *FNP* algorithm is more suitable than AC algorithm in most cases. To accommodate the cache size of normal PCs, the size of *PSW* was set to 2 in this experiment, and thus the size of *SDT* was 64K. The experiment was run on a general PC with a Pentium 4 processor running at 1.8 GHz, with L1 data cache of 8KB, L2 cache of 512 KB, and 512 MB of DDR memory. The host operating system was Windows XP. The packet trace in this test was still defcon_eth0.dump2 in DEFCON9. This work tests the *FNP* against the other three algorithms with both *LSP* and search set size the same as in the other experiments.

The *FNPw2* in Figure 20 indicates that the size of the *PSW* in this test is 2. With *LSP* < *3*, *FNP* processes packets faster than the other three algorithms when the search set size exceeds 64. With *LSP* = 3, the performance of *FNP* is almost identical to that of the MWM algorithm. Moreover, the *FNP* algorithm consumes much less memory than AC algorithm. For example, the *FNP* algorithm can accommodate 256 patterns using 256K, but AC algorithm requires over 2M to accommodate the same number of patterns. Furthermore, the *FNP* algorithm works more like MWM than AC, making the structures and tables easier to reuse. Figure 20 also illustrates the scalability of *FNP*. The increment of search-set size has almost no influence on the performance of the *FNP* algorithm when it is below 2048.

4.6. Summaries of FNP

This work examined the importance of the pattern matching algorithm for NIDS, and designed and implemented a fast and efficient algorithm named *FNP* for network processor platforms. *FNP* uses the characteristic of NIDS rulesets and the hardware facility of Network Processor to maximize performance.

Owing to the difficulty of implementing other multi-pattern matching algorithms (such as AC, SBMH, and MWM) by micro-code simultaneously, only the *FNP* algorithm is implemented on the Vitesse IQ2000 Network Processor platform to evaluate the relation between performance and the number of memory accesses for processing multi-pattern matching. On the other hand, the *FNP* algorithm is compared with the other three algorithms using general PCs. To simulate the Network Processor environment, both the L1/L2 caches are turned-off in this experiment. The experimental results reveal that the *FNP* outperforms the other three algorithms when *LSP* ≤ *4*. On a normal PC with the cache turned, the *FNP* also performs well for *LSP* ≤ *3*. Since 35% of the patterns in the current Snort ruleset have lengths of 1, 2, 3, or 4, then since NIDS rules are usually partitioned into groups based on L3/L4 header fields (for example IP pairs and port pairs) during the matching procedure, it is likely that *LSP* ≤ 4 for many groups. Besides, the user-defined signatures may even make a smaller *LSP* further. Consequently, *FNP* should perform better than other multi-pattern matching algorithms in most cases.

Pattern and payload characteristics affect the performance of multi-pattern matching algorithms in NIDSes, as in other applications. The NIDSes may partition the signatures into sub-groups based on L3/L4 header fields. The fastest matching algorithms differ among subgroups. From existing research, the assessment should depend on search-set size and *LSP* value. The *FNP* algorithm has been shown to be very efficient for small *LSP* regardless of search set size. According to our experiments, a hybrid multi-pattern matching algorithm comprising both *FNP* and *MWM* algorithms covers most cases and achieves better performance regardless the search-set size and the value of *LSP*.

Generally, the NIDS detection engine conducts flow classification, header-field comparison, and multi-pattern matching. Although multi-pattern matching is the most time-consuming task, a fast packet processing flow is desirable for integrated handling of these issues. Using the facilities provided by the Network Processor may be a good solution to this problem. This direction is left for future works to pursue.