# Chapter 5

# FNP$^2$: A MWM-like Pattern Matching Algorithm

## 5.1 Introduction of FNP$^2$

This work presents a multiple pattern matching algorithm which works efficiently whatever the number of ruleset or the minimal pattern length is. Moreover, this design utilizes the hardware accelerators of network processors to hide the search latency and speed up the performance. Network processors are part of an emerging class of programmable ICs based on system-on-a-chip technology that perform communications-specific functions more efficiently than general-purpose processors. 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.

## 5.2 Design/Implementation of FNP$^2$

This work addresses the string matching problem formally before introducing the proposed FNP$^2$ algorithm.

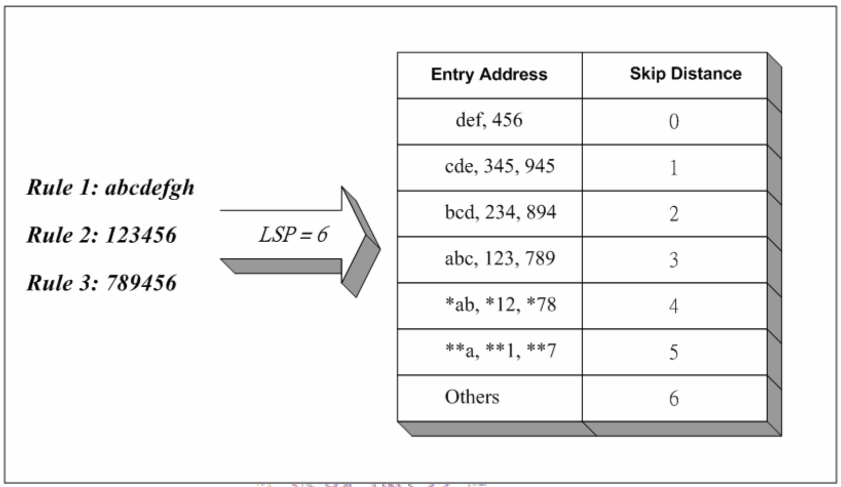
Given an input text $T = t_0, t_1, \ldots, t_n$, and a finite set of strings $P = \{P_1, P_2, \ldots, P_r\}$, the string matching problem involves locating and identifying the substring of $T$ which is identical to $P_j = a_0^j, a_1^j, \ldots, a_{m-1}^j$, $1 \le j \le r$, where $t_s \ldots t_{s+m-1} = a_0^j \ldots a_{m-1}^j$. And this equation can be also denoted as $t_s \ldots t_{s+m-1} = a_0^j \ldots a_{m-1}^j$.

*FNP²* is a *MWM*-like algorithm and 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

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

Then, $P_j$ doesn’ t contain the $t_s \ldots t_{s+w-1}$ so that the $w$ sequential bytes can be skipped safely during searching. On the other hand, if the $w$ sequential bytes $t_s \ldots t_{s+w-1}$ is identical to $a_i^j \ldots a_{i+w-1}^j$, where $0 \le i \le m\text{-}w$, then it furthermore verifies whether the last $w$ sequential bytes of $P_j$ $(a_{m-w}^j \ldots a_{m-1}^j)$ is identical to $t_{s+m-i-w} \ldots t_{s+m-i-1}$. Once the $w$-bytes suffix of $P_j$ is matched in certain position in $T$, an exact match will be performed. 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 $a_i^j \ldots a_{i+w-1}^j$ of pattern $P_j$, where $0 \le i \le m\text{-}w$. The following details the design of the *FNP²* algorithm. 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 $w = 3$ for better performance

This stage involves constructing Skip Distance Table (*SDT*), Rule Hashing Table (*RHT*), and Rule Status Table (*RST*) during initialization of this engine. *SDT* is used to determine how many sequential bytes can be skipped safely in searching phase. During initialization, all entries in *SDT* are set to *LSP*. Every table entry whose last (rightmost) 8-bit of address is identical to any one-byte prefix of the patterns is set to *LSP* - 1, and every table entry whose last (rightmost) 16-bit of address is identical to any two-byte prefix of the patterns is set to *LSP* - 2. Every table entry whose address is identical to $a^{j}_{LSP-3-i} \ldots a^{j}_{LSP-1-i}$ is set to *i*, $1 \leq j \leq r$ *and* $LSP\text{-}3 \geq i \geq 0$. Figure 21 demonstrates an example of the construction of *SDT*. The *LSP* in this example is 6 and the maximal skip distance is 6 also. The skip distance of other NIDS pattern matching algorithms is bound to *LSP* while the minimal skip distance of *FNP*$^2$ is 3. On the other hand, the construction and design of RST and RHT can be referenced from previous chapter.



| Entry Address | Skip Distance |
|---|---|
| def, 456 | 0 |
| cde, 345, 945 | 1 |
| bcd, 234, 894 | 2 |
| abc, 123, 789 | 3 |
| *ab, *12, *78 | 4 |
| **a, **1, **7 | 5 |
| Others | 6 |

Figure 21. An example of the construction of *SDT*

## 5.3 Experiments over FNP$^2$

To verify the effectiveness of the proposed *FNP*$^2$ algorithm, its performance was evaluated against the previously mentioned *SBMH*, *AC*, *MWM*, and *E*$^2$*xB* algorithms. Because of the difficulty of implementing all these five algorithms with Network Processor micro codes, the later four experiments were implemented on general PCs to simulate the network-processor environment. 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.

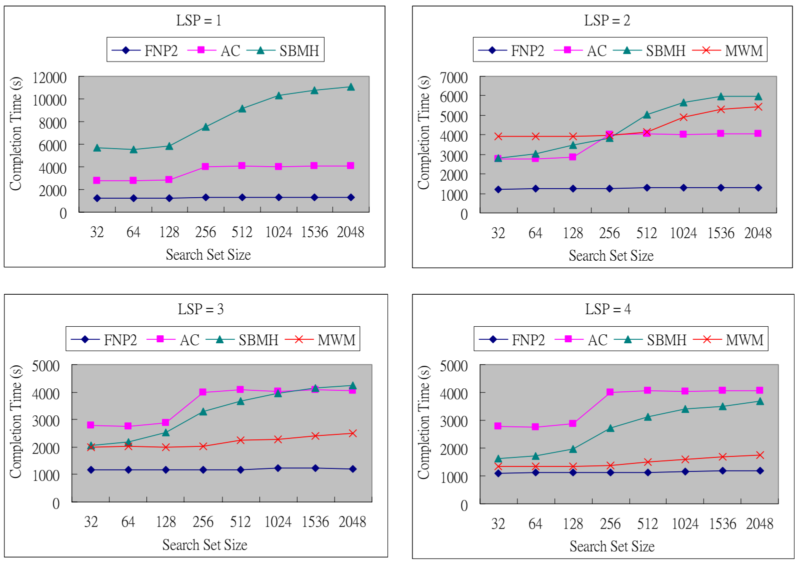


Figure 22. Number of memory accesses during pattern matching processing

As previously described, the proposed *FNP*$^2$ algorithm requires fewer memory accesses so that better performance can be achieved. The five 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 22 illustrates the results of four algorithms except *E*$^2$*xB* for different search set sizes and *LSPs*. The case involving the *MWM* algorithm with *LSP* = 1 was not assessed because the MWM algorithm does not support this situation. The experiment results of *E*$^2$*xB* is not listed in Figure 22 because of the big gap between its result and the other four. The number of memory accesses in *E*$^2$*xB's* simulation results is from 1112M to 23354M. It's clear to see that *E*$^2$*xB* doesn't work well when rule-set size is large and cache memory is unavailable like Network Processor platform. Figure 22 shows that *FNP*$^2$ algorithm clearly outperformed other algorithms in this way.

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 22 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.

## 5.4 Summaries of FNP$^2$

This work examined the importance of the pattern matching algorithm for NIDS, and designed and implemented a fast and efficient algorithm named *FNP*$^2$ for network processor platforms. *FNP*$^2$ 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*, $E^2xB$ and *MWM*) by micro-code simultaneously, we evaluate these algorithms by implementing in general PC platoform. The experimental results reveal that the FNP$^2$ outperforms the other algorithms in this matter.

Network Processors are known to be powerful to handle L3/L4 traffic and this design take use of the characteristic of Network Processor to process L7 payloads efficiently. Moreover, the searching algorithm benefits much more when LSP is small whereas it's the common case in NIDS application.

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.