Towards Faster Pattern Matching: Analysis and Improvement to Extended Shift-or Algorithm

Jincheng Zhong, Shuhui Chen *

School of Computer, National University of Defense Technology, Changsha Deya Road 109, Hunan, China
Email: shchen@nudt.edu.cn

Pattern matching is the foundation of various network security applications and achieving fast pattern matching is of great importance. In recent years, pre-filtering methods for pattern matching have achieved great success. The current most efficient pattern matching library, hyperscan, is based on one pre-filtering algorithm, the extended shift-or algorithm. Benefited from the cache-friendly data structure and simple operations required, the extended shift-or algorithm can achieve a high matching throughput of several to even more than ten Gbps employing only one CPU core. However, since the extended shift-or algorithm may generate false positives when performing multi-string pattern matching, a high false positive ratio (FPR) can heavily affect the overall matching performance. As far as we know, prior works have not analyzed the FPR of the extended shift-or algorithm. In this paper, we first analyze the FPR of the extended shift-or algorithm in theory and show that the extended shift-or algorithm can support up to hundreds of thousands of patterns within a small FPR. Besides, this paper proposes a novel pattern grouping method for the extended shift-or algorithm to improve the pre-filtering effectiveness. Experiments show that employing the proposed novel pattern grouping method, the FPR of the extended shift-or algorithm can be reduced by up to 75%.

Keywords: pattern matching, network security, shift-or, pre-filtering method, DPI

1. INTRODUCTION

Pattern matching, which includes exact string matching and regular expression matching, is the foundation of various security applications such as antivirus programs and Network Intrusion Detection System (NIDS). Given a rule-set containing multiple patterns and an incoming stream of data, pattern matching is the process of inspecting every byte of the data stream to find the occurrences of each pattern in the rule-set. Since the size of the rule-set can reach tens of thousands, pattern matching is a heavy computing overhead and easily becomes the bottleneck of various security applications [1, 2]. Therefore, achieving high-performance pattern matching is of great importance and has been the focus of research for over two decades.

There are a number of works on proposing efficient pattern matching algorithms. Only limited to multi-string pattern matching, well-known algorithms include Aho Corasick (AC) algorithm [3, 4], and heuristic-based algorithms, such as Wu-Manber algorithm [5]. In recent years, pre-filtering methods [6, 7, 8, 9, 10, 11, 12, 13] have been proven to have better performance in practical applications.

The core idea of all pre-filtering methods is to utilize an efficient pre-filter to bypass most of the input traffic. Only a small proportion of traffic, which passes the pre-filter, requires further verification provided by the relatively slow classical pattern matching algorithms. Consequently, pre-filtering methods are able to perform fast pattern matching and are widely used in practice. The state-of-the-art pre-filter-based pattern matching library, hyperscan [14, 11], is claimed to outperform AC [3] by 3.2 to 8.2 times. The high matching performance of hyperscan is mainly achieved by the use of one pre-filtering algorithm, the extended shift-or algorithm [11].

For large-scale rule-sets containing thousands of patterns, the extended shift-or algorithm can deliver a superior matching throughput of more than ten Gbps with one CPU core. However, since it is a pre-filtering algorithm, the extended shift-or algorithm will produce false positives when performing matching. One false positive is one match that is accepted by the pre-filter but is rejected by further verification. For that false positives require further verification, to reduce verification overhead, the number of false positives caused by the pre-filter should be limited. In

other words, the pre-filter should have high pre-filtering effectiveness, which indicates the match probability [1] should be limited to a small value (e.g., 0.01). However, the match probability of the extended shift-or algorithm is not analyzed in prior works [11].

This paper theoretically analyzes the pre-filtering effectiveness[2] of the extended shift-or algorithm in detail. Our analysis points out that the extended shift-or algorithm can support up to hundreds of thousands of patterns when limiting the match probability to be smaller than 0.01. Besides, the extended shift-or algorithm is compared to the other two common pre-filters, the bloom filter [15] and the direct filter [8]. Given the same pre-filtering effectiveness, the extended shift-or algorithm requires fewer memory accesses and operations when matching each character, so it is much more efficient.

Further, based on the theoretical analysis of the pre-filtering effectiveness, it is found that the original pattern grouping method used in the extended shift-or algorithm can be improved. Therefore, a novel pattern grouping method is put forward in this paper. The experiments show that our proposed new grouping method can improve the pre-filtering effectiveness of the extended shift-or algorithm by up to 4 times.

In a nutshell, the main contributions of this paper are as following:

1. As far as we know, this paper first analyses the pre-filtering effectiveness of the extended shift-or algorithm in theory.
2. The extended shift-or algorithm is compared to the other two common pre-filters in this paper. The comparison shows that the extended shift-or algorithm outperforms others.
3. A novel pattern grouping method is proposed in this paper and greatly improves the pre-filtering effectiveness of the extended shift-or algorithm.
4. Employing the proposed novel pattern grouping method, we have implemented the optimized version of the extended shift-or algorithm and open-sourced the code at [16].

The remaining of this paper is organized as follows. Section 2 introduces the extended shift-or algorithm. Then, the theoretical analysis of the extended shift-or algorithm is given in Section 3. Finally, Section 4 optimizes the extended shift-or algorithm and Section 5 concludes this paper.

2. SHIFT-OR ALGORITHM

In this section, we first introduce how the standard shift-or algorithm works [17, 18, 19]. Then the extended shift-or algorithm proposed in hyperscan [11] are elaborated.

2.1. Standard shift-or algorithm

The standard shift-or algorithm is based on finite automata theory. More specifically, the standard shift-or algorithm works through simulating the Nondeterministic Finite Automaton (NFA) matching procedure using bit vectors.

Let p be a string pattern of length m, and c_i be the ith character of p (counted from left to right). The NFA A_p for matching p is shown in Fig.1. The standard shift-or algorithm uses a bit vector *state-vector* of length m to indicate whether each state in A_p is active. The ith bit of *state-vector* (counted from right to left), *state-vector*$[i]$, corresponds to state i of A_p. If *state-vector*$[i]$ is 0, it indicates that state i is active. Otherwise, it indicates that state i is inactive. For example, *state-vector* = 1...110 means that state 0 and 1 are active while other states are inactive. Note that as state 0 of A_p is always active, it does not need to correspond to a bit.

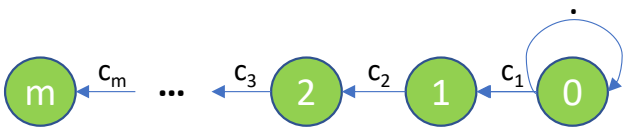

FIGURE 1: NFA for matching pattern p.

Besides *state-vector*, shift-or algorithm will generate one $shift\text{-}or\text{-}vector$ of length m for each character of the alphabet. The ith bit of $shift\text{-}or\text{-}vector$ for character c, $shift\text{-}or\text{-}vector[c][i]$, is used to indicate that if character c exists in the transition from state $i-1$ to state i. For example, if c_1 is different from c_k $(2 \leq k \leq m)$, then $shift\text{-}or\text{-}vector[c_1]$ = 1...110, which means that character c_1 exists only in the transition from state 0 to state 1.

With the two data structures, *state-vector* and $shift\text{-}or\text{-}vector$, the matching procedure of the standard shift-or algorithm is shown in Algorithm 1. For matching input text T of length n, *state-vector* is initialized to 1..111 first as only state 0 of A_p is initially active. For each character c of T, shift-or algorithm first performs one bitwise left shift operation (i.e. $<<$) to *state-vector*, which is equivalent to perform transitions from state $i-1$ to state i $(1 \leq i \leq m)$ for A_p. Then it performs one bitwise or (i.e., $|$) operation on *state-vector* and $shift\text{-}or\text{-}vector[c]$, which supresses the transitions without character c. Through the two bitwise operations performed on character c, *state-vector* maintains the consistency with the states of A_p. In order to determine whether a match occurs, the standard shift-or algorithm will test whether *state-vector*$[m]$ is equal to 0, which means that the accept state of A_p is active.

[1] The probability that an arbitrary text is accepted by the pre-filter.

[2] The pre-filtering effectiveness is measured using the match probability. The smaller the match probability, the higher the pre-filtering effectiveness.

Input: p, a string pattern of length m; T, input of length n to be searched

```
1 Initialize state-vector to 1...111;
2 Initialize shift-or-vector;
3 for i ← 1 to n do
4 |   c ← the ith character of T;
5 |   state-vector = (state-vector << 1) |
  |    shift-or-vector[c];
6 |   if state-vector[m] equals to 0 then
7 |   |   Report T matches p at position i;
8 |   end
9 end
```

Algorithm 1: Matching procedure of shift-or algorithm

To clarify the matching procedure of the standard shift-or algorithm, let the pattern to be matched be $p = acbc$ and the input text be $T = acacbc....$ Fig.2 shows the matching process. In the initialization stage, $shift\text{-}or\text{-}vector[a]$ is initialized to 1110 as character "a" appears at the first position of p. Following the same rule, $shift\text{-}or\text{-}vector[b]$ is initialized to 1011 and $shift\text{-}or\text{-}vector[c]$ is initialized to 0101. In the matching stage, for the first input character "a", *state-vector* is shifted one bit to the left and gets 1110. Then bitwise or operation is performed on *state-vector* and $shift\text{-}or\text{-}vector[a]$ and the result is 1110, which means character "a" activates state 1 of the NFA corresponding to the pattern p. Next, for character "b", after one shift and or operations, *state-vector* becomes 1101, which indicates that state 2 of the NFA is activated. However, as the third character "a" does not exist in the transition from state 2 to state 3, *state-vector* comes back to 1110 after matching "a". Following the matching procedure, when matching the sixth character "c" of T, the value of *state-vector* changes to 0111, which means that the accepting state (i.e., state 4) of the NFA is activated. Therefore, one match is reported.

Assume that the computer word size is w, which can be up to 512 using Single Instruction Multiple Data (SIMD) instructions supported by modern CPUs. The matching time complexity of the standard shift-or algorithm is $O(\lceil \frac{m}{w} \rceil n)$. In practice, m tends to be smaller than w. Therefore, the matching time complexity is $O(n)$. Assume that the input alphabet is Σ, which is usually the ASCII alphabet in practice. Then the space complexity of the standard shift-or algorithm is $O(m|\Sigma|)$.

2.2. Extended shift-or algorithm

Although the standard shift-or algorithm runs fast with low space complexity, it only supports the matching of a single string pattern. To support multiple string patterns, the shift-or algorithm is extended by hyperscan [11].

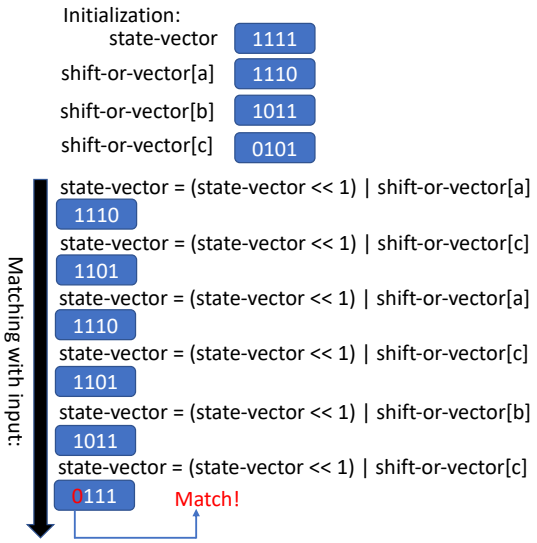

FIGURE 2: Shift-or algorithm matching example.

2.2.1. *Multiple patterns*

The standard shift-or algorithm can be directly applied to the matching of multiple patterns with the same length. However, a large number of mismatches (i.e., false positives) can be caused this way.

For example, employing the standard shift-or algorithm to match two patterns, $p_1 = ac$ and $p_2 = bd$, the equivalent NFA will be like that shown in Fig.3. As shown in the figure, the equivalent NFA not only accepts "ac" and "bd" but also accepts "ad" and "bc", which are two mismatches caused by the shift-or algorithm.

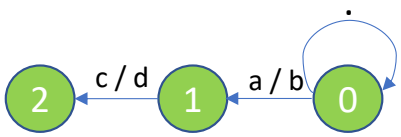

FIGURE 3: The equivalent NFA for matching pattern p_1 and p_2.

Let P be the pattern set to be matched by the standard shift-or algorithm, and m be the length of each pattern. Assume that n_i $(1 \leq i \leq m)$ is the number of unique characters at the ith position of each pattern in P. Then the number of all mismatches is up to $\prod_{i=1}^{m} n_i - |P|$.

To reduce mismatches when supporting multiple patterns, two extensions to the standard shift-or algorithm are proposed in hyperscan. One is "multiple buckets" and the other is "super characters". The standard shift-or algorithm with these two extensions is called the extended shift-or algorithm in this paper.

2.2.2. *Multiple buckets*

The first extension is "multiple buckets", which is equivalent to running multiple standard shift-or instances in parallel. First, the set of string patterns

are grouped into B distinct buckets, which is numberd from 1 to B. For now, each pattern belongs to one of the B buckets. The pattern grouping procedure will be discussed in Secion 4. Second, in order to support the parallel excecution of B shift-or instances, the sizes of *state-vector* and each *shift-or-vector* are increased to B times. Assume that the size of original *state-vector* and *shift-or-vector* are both m. For now, the size of *state-vector* and *shift-or-vector* both become Bm, and the ith bit in the kth B bits of *state-vector* or *shift-or-vector* corresponds to the kth bit in the original *state-vector* or *shift-or-vector* belonging to bucket i.

Fig.4 shows one example with two patterns, $p_1 = ac$ and $p_2 = bd$, being grouped into bucket 1 and bucket 2. As character "a" appears only at the first position of p_1 in bucket 1, *shift-or-vector*$[a]$ zeroes the first bit in the first B bits. For character "b", since it only appears at the first position of p_2 in bucket 2, *shift-or-vector*$[b]$ zeroes the second bit in the first B bits. The ith bit in each B bits of *state-vector* and *shift-or-vector* indicates bucket i.

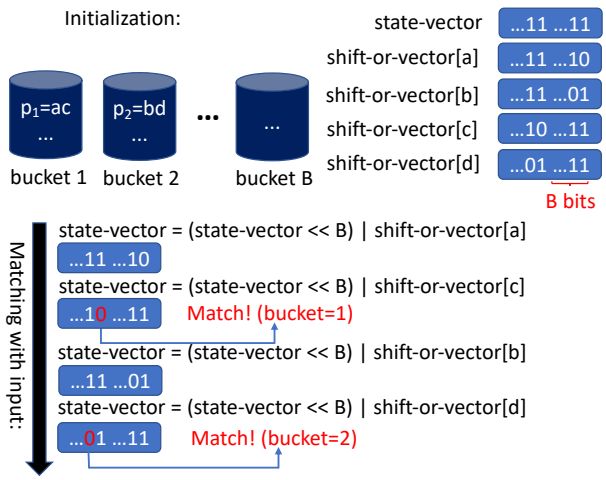

FIGURE 4: Shift-or algorithm with "multiple buckets" extension.

As for the matching procedure with "multiple buckets", instead of shifting *state-vector* left by 1 bit, the shift-or algorithm with "multiple buckets" extension shift *state-vector* left by B bits per character[3]. In this way, the "multiple buckets" extension realizes the parallel execution of multiple standard shift-or instances. Since each bucket equals an independent standard shift-or instance, any two patterns in different buckets will not produce any mismatches. In addition, each bucket supports string patterns of one length. Consequently, string patterns with at most B different lengths can be supported by the "multiple buckets" extension. In the implementation of hyperscan, B is set to 8, and pattern lengths from 1 to 8 are supported.

[3]The description is kind of different from hyperscan. However, they are functionally equivalent.

2.2.3. *Super characters*

With the "multiple buckets" extension, although the mismatches between any two patterns in different buckets are eliminated, there are still a large number of mismatches produced by the patterns in each bucket. Therefore, the second extension, "super characters", is proposed to suppress the mismatches in each bucket. The core idea of "super characters" is to reflect some information of the next character in a pattern when building *shift-or-vector* for the current character.

Assume that the input alphabet is ASCII character set and S-bit super characters are used. Since we can at most borrow all information (8 bits) from the next character, S should be in the range of 8 to 16. Let $S = 10$, which indicates that 2 bits information is borrowed from the next character when building *shift-or-vector* for the current character. As shown in Fig.5, for two patterns ($p_1 = ac$ and $p_2 = bd$) in one bucket, instead of building *shift-or-vector* for characters "a", "b", "c" or "d", "super characters" extention builds *shift-or-vector* for super characters α, β, γ, and δ. Since character "c" is the next character of "a", the super character for "a" is α = (low-order 2 bits of "c" << 8) | "a". For character "c", as there is no next character following it, its corresponding super character γ = (arbitrary 2 bits << 8) | "c", which indicates that γ can match 4 values.

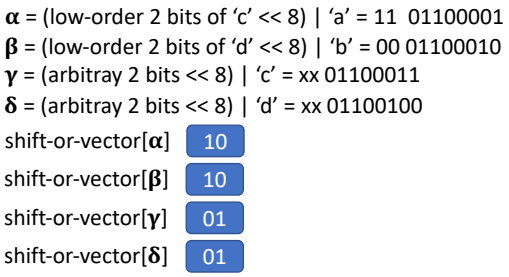

FIGURE 5: Super characters and their *shift-or-vector*.

Employing the "super characters" extension, the equivalent NFA is shown in Fig.6. When the input is $T = ad$, as super character θ = (low-order 2 bits of "d" << 8) | "a" can not match α or β, "ad" will not be matched by the NFA. Similarly, the false positive "bc" is also suppressed by "super characters". Consequently, most false positives can be avoided through "super characters" extension.

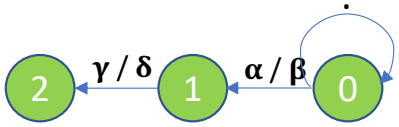

FIGURE 6: The NFA consistent with shift-or algorithm with "super characters" extension.

2.2.4. Time and space complexity

Let B be the number of buckets, S be the number of super character bits, L be the maximum pattern length supported by each bucket, and w be the computer word size.

The search time complexity of the extended shift-or algorithm is $O(\lceil \frac{BL}{w} \rceil n)$ when matching text with a length of n. Since w can reach up to 512 when employing SIMD instructions of modern CPUs, for $B = 8$ and $L = 8$ in the hyperscan implementation, the search time complexity of the extended shift-or algorithm is $O(n)$. Even if a larger B or L is used, the matching time complexity of each input character can still be maintained as $O(1)$.

Since there are 2^S super characters, and each super character corresponds to a *shift-or-vector* with a length of BL, the space complexity of the extended shift-or algorithm is $O(2^S BL)$.

3. PRE-FILTERING EFFECTIVENESS OF THE EXTENDED SHIFT-OR ALGORITHM

Although a large proportion of mismatches can be suppressed by "multiple buckets" and "super characters" extensions, there will still be many false positives. When applying the extended shift-or algorithm to multiple-string matching, a further verification stage is required to remove all false positives. A number of exact string matching algorithms can be used for the verification stage. In hyperscan, the verification phase consists of hashing and exact string comparison. Regardless of the algorithm used in the verification phase, it is crucial that the extended shift-or algorithm can filter as much input text as possible, which determines the overhead of the further verification phase.

However, the pre-filtering effectiveness of the extended shift-or algorithm is not analyzed in the hyperscan paper or any other work. In this section, we will theoretically analyze the pre-filtering effectiveness of the extended shift-or algorithm in detail.

3.1. Measurement of pre-filtering effectiveness

Given an arbitary text t of length L, let PM be the probability that the extended shift-or algorithm matches t. Obviously, the smaller the PM, the better the pre-filtering effectiveness. Let PM_i $(1 \le i \le B)$ be the probability that bucket i of the extended shift-or algorithm matches t. Since each bucket is independent from other buckets when performing matching, $PM = 1 - \prod_{i=1}^{B}(1 - PM_i)$ holds. Furthermore, if PM_i is small for $1 \le i \le B$ (e.g., $PM_i \le 0.01$), then $PM \approx \sum_{i=1}^{B} PM_i$ holds.

To evaluate PM_i, we can first calculate the number of all possible matching texts, $|TM|$, of bucket i. Then $PM_i = \frac{|TM|}{|T|}$, where $|T|$ is the number of all possible texts with a length of L and $|T| = 256^L$.

We use *matching graph* to calculate the number of all possible matching texts, $|TM|$, for a bucket. A *matching graph* consists of a *start* node, a *final* node, and L levels of *char nodes*. For the ASCII character set, each level has 256 char nodes, in which each char node corresponds to an ASCII character. There are directed edges between the start node and level 1 char nodes, between level l and level $l+1$ char nodes, and between level L char nodes and the final node. Each *matching path* from the start node to the final node corresponds to a matching text which can be accepted by the bucket. Therefore, the number of all matching paths from the start node to the final node equals $|TM|$ of the bucket.

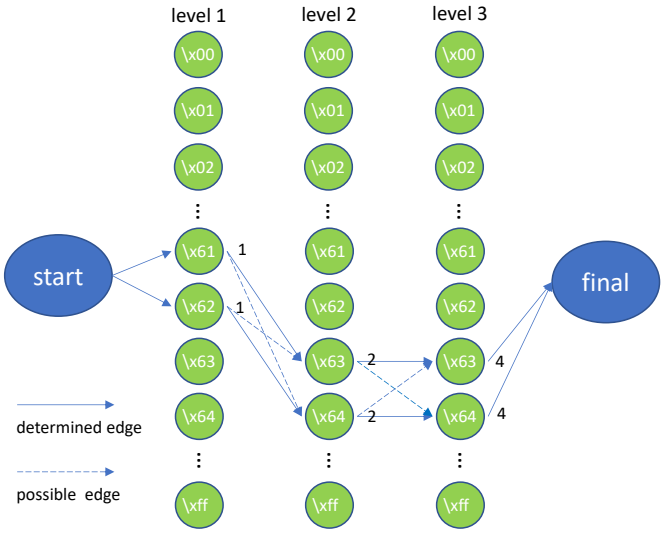

FIGURE 7: Match graph of the bucket containing two patterns: "acc" and "bdd".

3.1.1. Example

Fig.7 shows the example matching graph for one bucket containing only two patterns, i.e., "acc" and "bdd". Let $G[l][c]$ be the number of matching paths from the start node to char c node in level l. If char c exists in the first position of some pattern, we can figure out that $G[1][c] = 1$. Otherwise, $G[1][c] = 0$. Therefore, as shown in the figure, among the char nodes in level 1, only $G[1]["a"]$ and $G[1]["b"]$ equal to 1.

For char c node in level l $(l \ge 2)$, if char c exists in the lth position of at least one pattern in the pattern set, the number of all paths to it should be the sum of the number of paths that all char nodes in the previous level are connected to it, i.e., $G[l][c] = \sum_{k=0}^{255}(G[l-1][k] \times f(k,c,l))$. Otherwise, $G[l][c] = 0$.

$f(k,c,l))$ is a binary funciton, indicating whether char c node in level l is connected with char k node in the previous level. If char c node is connected with char k node in the previous level, $f(k,c,l) = 1$. Otherwise, $f(k,c,l) = 0$. Whether two char nodes in matching graph are connected or not is determined by the number

of super character bits, S. According to Section 2.2.3, the transitions of the equivalent NFA reflet $(S-8)$ bits information of the next charcter. Therefore, only if there exists a pattern p with $p[l-1]=k$ and the low-order $(S-8)$ bits of $p[l]$ equals to the low-order $(S-8)$ bits of char c, char c node is connected with char k node.

In Fig.7, the dotted lines indicate the paths that may be supressed by "super character" extension. Assuming that those paths are not supressed, then $G[2]["c"] = G[2]["d"] = G[1]["a"] + G[1]["b"] = 2$, and $G[3]["c"] = G[3]["d"] = G[2]["c"] + G[2]["d"] = 4$. Since the number of all matching paths equal to $\sum_{c=0}^{255} G[L][c]$, the number of all possible matching texts is $|TM| = G[3]["c"] + G[3]["d"] = 8$.

3.1.2. General cases

For general cases, assuming that a pattern set P with all patterns having the length of L is in one shift-or bucket, the number of its all possible matching texts, $|TM|$, can be figured out using Algorithm 2. Consequently, the probability that an arbitrary text of length L matches the shift-or bucket is $|PM| = \frac{|TM|}{256^L}$.

Input : P, a pattern set with all patterns having the length of L
Output: $|TM|$, the number of all possible matching texts

```
Initialize G[l][c] ← 0, (1 ≤ l ≤ L, 0 ≤ c ≤ 255);
for char c ← 0 to 255 do
    if c exists in the first position of some pattern in P then
        G[1][c] ← 1;
    end
end
for level l ← 2 to L do
    for char c ← 0 to 255 do
        if c exists in the lth position of some pattern in P then
            G[l][c] = Σ_{k=0}^{255} (G[l-1][k] × f(k,c,l));
        end
    end
end
|TM| = Σ_{c=0}^{255} G[L][c];
return |TM|;
```

Algorithm 2: Calculating the number of all possible matching texts of one shift-or bucket

3.2. Pre-filtering effectiveness of random rule-sets

3.2.1. Calculation

It can be inferred from the previous subsection that the pre-filtering effectiveness of the extended shift-or algorithm is closely related to the pattern set. In order to theoretically analyze the pre-filtering effectiveness, we assume that one random pattern set P with N patterns is contained in one shift-or bucket.

Let L be the length of all patterns in P, and S be the number of super character bits. We also employ matching graph to calculate the expected number of all matching texts, EM, accepted by the shift-or bucket. Let $E[l][c]$ be the expected number of matching paths from the start node to char c node in level l, then we have

$$EM = \sum_{c=0}^{255} E[L][c] \tag{1}$$

For each char c node in level 1, the expected number of matching paths to it should be

$$E[1][c] = 1 \times (1 - \frac{255}{256}^N) \tag{2}$$

Note that $1-\frac{255}{256}^N$ is the probability that char c exists in the first position of at least one pattern in P.

For each char c node in level l $(l \geq 2)$, the expected number of matching paths to it should be

$$E[l][c] = (1 - \frac{255}{256}^N) \times \sum_{k=0}^{255} (E[l-1][k] \times ef(k,c,l)) \tag{3}$$

Note that $1-\frac{255}{256}^N$ is the probability that char c exists in the lth position of at least one pattern in P and $ef(k,c,l)$ is one function calculating the probability that char c node in level l is connected with char k node in level $l-1$. Let $ef(k,c,l)$ be the probability that char c node is not connected with char k node, then we have

$$\begin{aligned} ef(k,c,l) &= 1 - \widetilde{ef(k,c,l)} \\ &= 1 - (\frac{255}{256} + \frac{1}{256} \times (1 - 2^{8-S}))^N \end{aligned} \tag{4}$$

From Equation 1, 2, 3 and 4, we can get that the number of all matching paths of one shift-or bucket for pattern set P is

$$EM = 256^L \times (1 - \frac{255}{256}^N)^L \times (1 - (1 - 2^{-S})^N)^{L-1} \tag{5}$$

Consequently, the probability that an arbitrary text matches the shift-or bucket is

$$\begin{aligned} PM &= \frac{EM}{256^L} \\ &= (1 - \frac{255}{256}^N)^L \times (1 - (1 - 2^{-S})^N)^{L-1} \end{aligned} \tag{6}$$

3.2.2. Analysis

To graphically show the pre-filtering effectiveness of the extended shift-or algorithm, let L (i.e., the length of all patterns in one shift-or bucket) be fixed to 8 first. According to Equation 6, we can draw the change curve of PM with N and S as shown in Fig.8. It can be

seen from the figure that the PM of a shift-or bucket increases as the number of patterns (N) increases, and decreases as the number of super character bits (S) increases. If the PM of one shift-or bucket is limited to be smaller than 0.001, the maximum number of the patterns that can be supported within one shift-or bucket varies from 255 to 30565 for $8 \leq S \leq 16$. It indicates that when choosing a large S, the extended shift-or algorithm has high pre-filtering effectiveness even for large pattern sets with tens of thousands of rules.

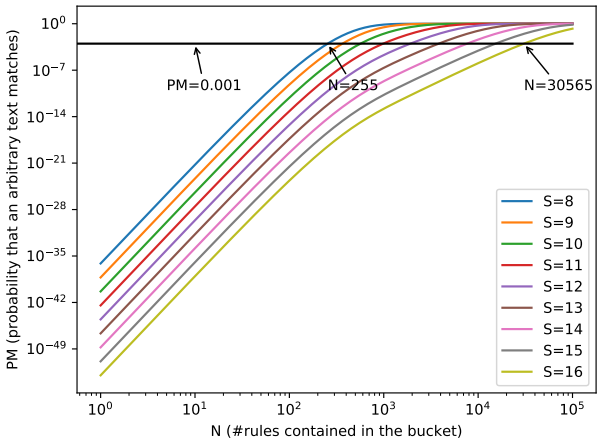

FIGURE 8: The change curve of PM with N and S ($L = 8$).

To show the relationship of the pre-filtering effectiveness and the pattern length L, let S be fixed to 4. Similarly, the change curve of PM with N and L can be drawn in Fig.9. It shows that when S and N are fixed, the PM of a shift-or bucket decreases as the pattern length (L) increases. When L is small, one shift-or bucket can only support a small number of patterns. For example, when $L = 2$ and $S = 4$, one shift-or bucket can accommodate at most 71 patterns to limit the match probability to less than 0.001. Therefore, to ensure the pre-filtering effectiveness of the extended shift-or algorithm, the number of short patterns should be as small as possible.

3.3. Comparison with other pre-filters

In this subsection, the extended shift-or algorithm is compared to the other two common pre-filters, the bloom filter[15, 6] and the direct filter[8].

One direct filter (DF) is a bitmap indexed by several bytes of input text. For example, a 2B DF (2 byte DF) is a bitmap with a size of 2^{16} bits. In order to test whether a text T of length 2 matches it, the direct filter treats T as a 16-bit unsigned integer and uses it as a bit index for the 2B DF. If the corresponding bit of the DF is set, it indicates a match occurs. To apply to string patterns with a length of L, we use a total of $L/2$ 2B

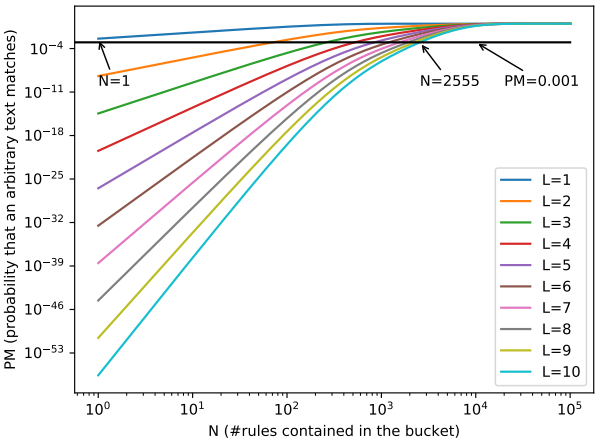

FIGURE 9: The change curve of PM with N and L ($S = 4$).

DFs for the direct filter. Assuming that $L = 8$, only when a text T with a length of 8 sequentially passes the tests of all the 4 DFs, i.e., $T[2i : 2i + 1]$ (for $0 \leq i \leq 3$) passes the test of the $i + 1$th DF, is it reported as a match.

To measure the pre-filtering effectiveness of the bloom filter, assuming that the match probability is PM, the optimal number of hash functions is set to $-log_2(PM)$ [20].

We limit the maximum match probability, PM, to 0.01, 0.001, and 0.0001, respectively. To make L divisible by 2, let L be 4, 6, and 8. Since the direct filter uses $L/2$ 2B DFs and occupies $L/2 \times$ 8KB of memory, the maximum space by all pre-filters are limited to 16KB ($L = 4$), 24KB ($L = 6$), and 32KB ($L = 8$). Under these restrictions, the maximum number of patterns supported, required operations, and memory accesses for each input byte are compared. The comparison results is shown in Table 1.

From the table, it is shown that the pre-filtering effectiveness of the bloom filter is the highest. However, the bloom filter is slow for that it needs to perform up to $-log_2(PM)$ hash calculations and at most $-log_2(PM)$ memory accesses for matching per character. When limiting the match probability (PM) to less than 0.01, at most 10 hash calculations and 10 memory accesses are required. Compared to the bloom filter, under the same low space occupation and high pre-filtering effectiveness, although the extended shift-or algorithm supports fewer patterns, it is much faster for it only requires 2 bitwise operations and 1 memory access per character. Compared to the direct filter, the extended shift-or algorithm outperforms in all aspects.

TABLE 1: Comparison with the bloom filter and the direct filter.

pattern length	pre-filter type	#patterns supported (maximum)			#operations (maximum)	#memory acesses (maximum)
		PM=0.01	PM=0.001	PM=0.0001		
L = 4	bloom filter	13665	9115	6830	$-log_2(PM)$ hashs	$-log_2(PM)$
	direct filter	6905	2105	659	6 bitwise operations	2
	shift-or	7950	3450	1560	2 bitwise operations	1
L = 6	bloom filter	20550	13700	10275	$-log_2(PM)$ hashs	$-log_2(PM)$
	direct filter	15900	6905	3115	9 bitwise operations	3
	shift-or	16635	9480	5655	2 bitwise operations	1
L = 8	bloom filter	27392	18267	13703	$-log_2(PM)$ hashs	$-log_2(PM)$
	direct filter	24910	12830	6905	12 bitwise operations	4
	shift-or	23910	15285	10235	2 bitwise operations	1

4. OPTIMIZATION OF THE EXTENDED SHIFT-OR ALGORITHM

In Section 2.2.2, the "multiple buckets" extension has been introduced in detail. In the procedure of the "multiple buckets" extension, the pattern set is first divided into multiple groups, each of which is placed into one bucket later. The strategy for grouping patterns affects the pre-filtering effectiveness of the extended shift-or algorithm, thereby affecting the matching performance. Due to the observation that the original grouping method behaves badly in some cases, we try to propose a novel and more efficient grouping method.

In this section, we first introduce the original pattern grouping method used in [11]. Then a new grouping method is proposed to improve the pre-filtering effectiveness of the extended shift-or algorithm. Finally, experiments are conducted to show significant improvement.

4.1. Original pattern grouping method

The original pattern grouping method is designed based on two guidelines. First, for that one shift-or bucket only supports one pattern group with the same length, the patterns with a similar length are grouped into the same bucket to minimize the information loss of longer patterns. Second, since short patterns are likely to increase false positives, it avoids grouping too many short patterns into one bucket.

In order to meet the above two requirements, the original grouping algorithm first sorts all patterns in the ascending order of length, and assign an id of 0 to $N-1$ to each pattern. Then, a dynamic programming method is used for calculating the minimum cost of grouping N patterns into B buckets. The dynamic programming method can be summarized as the following two equations:

1. $t[i][j] = \min_{k=i+1}^{N-1}(cost_{ik} + t[k+1][j-1])$, where $t[i][j]$ is the minimum cost of grouping pattern i to $N-1$ into $j+1$ buckets.
2. $cost_{ik} = \frac{(k-i+1)^{\alpha}}{length_i^{\beta}}$, where $cost_{ik}$ is the cost of grouping pattern i to k into one bucket, $length_i$ is the length of pattern i, α and β are two parameters[4].

According to the above two equations, the original algorithm can calculate the minimum cost of grouping all patterns into B buckets, i.e., $t[0][B-1]$. In the calculation process, each pattern is recorded to which group it belongs.

However, since the original grouping algorithm only considers the length of each pattern when calculating the grouping cost, it is far from the optimal grouping. For patterns with the same length, the original method treats them equally and will randomly group them into different buckets.

4.2. Our proposed grouping method

How to measure the pre-filtering effectiveness of the extended shift-or algorithm has been elaborated in Section 3.1. Based on the measurement, we propose a new grouping method targeted at maximizing the pre-filtering effectiveness of the extended shift-or algorithm.

However, the global optimal patterns k-grouping problem is difficult, and the time complexity is exponential. To limit the time cost of grouping, we propose a *local search* method to efficiently find a local optimal grouping solution. Algorithm 3 illustrates the proposed local search grouping method.

First, the patterns are divided using the original grouping method to generate an initial grouping. Then we try to randomly remove a pattern p from a group G_i.

[4]In hyperscan implementation, α is set to 1.05 and β is set to 3.

Input : P, a pattern set with N patterns; B, number of groups desired
Output: B pattern groups, G_1, G_2, ..., G_B

Divide all patterns into B groups using the original method;
$iterations \leftarrow 0$;
$nochange \leftarrow 0$;
while $iterations < T_1$ *and* $nochange < T_2$ **do**
  $iterations \leftarrow iterations + 1$;
  Randomly choose a pattern p from a group G_i;
  $cost_i \leftarrow M(G_i) - M(G_i - p)$; // $M()$ is a function calculating the pre-filtering effectiveness for a pattern group; $G_i - p$ means that removing pattern p from group G_i;
  for $j \leftarrow 1$ **to** B, $j \neq i$ **do**
    $cost_j \leftarrow M(G_j + p) - M(G_j)$; // $G_j + p$ means that putting pattern p into group G_j;
  end
  $t \leftarrow \arg\min_{k=1}^{B}(cost_k)$;
  if $t = i$ **then**
    $nochange \leftarrow nochange + 1$;
  else
    Move pattern p into G_t;
    $nochange \leftarrow 0$;
  end
end

Algorithm 3: Our proposed pattern grouping algorithm.

For each pattern group, the cost (i.e., the reduced pre-filtering effectiveness) of putting p into it is calculated. The pattern group G_t with the minimal cost is the target group where p is placed later. We repeat the previous step until the local optimal solution is reached or the number of iterations exceeds a limit, T_1. If the randomly selected pattern p has not been moved to another group (i.e., G_t is G_i itself) for T_2 consecutive times, it is considered that the local optimal solution has been reached. T_1 and T_2 are two parameters, which are set to the size of the input pattern set and 100 respectively. The function of calculating the pre-filtering effectiveness, $M()$ in Algorithm 3, is based on the measurement algorithm in Section 3.1.

Although the grouping solution generated through Algorithm 3 is just close to local optimal, it behaves much better compared to the original grouping method.

4.3. Experiments

Experiments are conducted to compare the original grouping method and our proposed grouping method. String patterns from ClamAV [21] (an open-source anti-virus application) and Snort [22] (an open-source network intrusion detection system) are used as the test rule-sets. The two grouping methods are implemented using C++ and the source code can be publicly accessed at [16]. All experiments are conducted on an x86 machine, where the operating system is macOS 10.14.

The number of shift-or buckets, B, is set to 8, which is the default setting of hyperscan. The size of the test rule-sets rises from one thousand to twenty thousand, while the number of super character bits S increases from 8 to 12. For the rule-sets extracted from ClamAV and Snort, the comparison results are shown in Fig.10 and Fig.11, respectively.

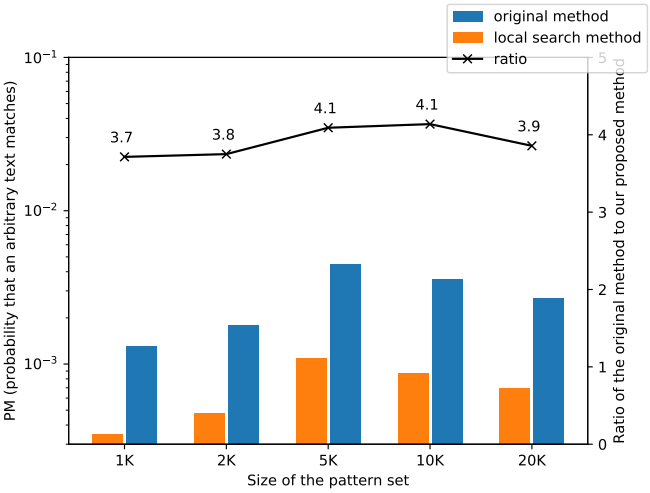

FIGURE 10: Comparison of the pre-filtering effectiveness of the two grouping methods on ClamAV rule-sets.

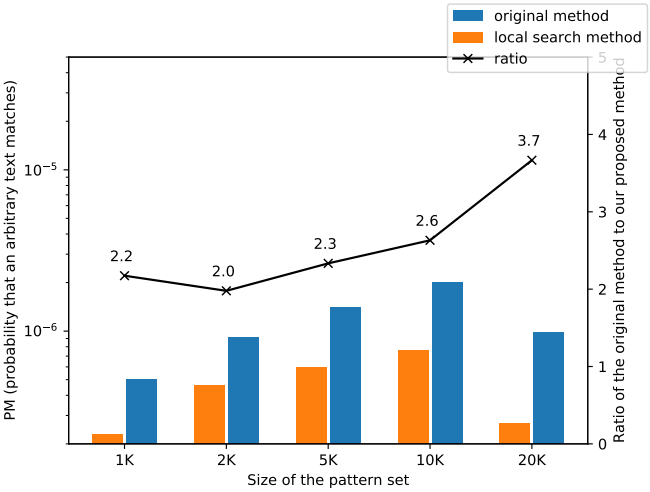

FIGURE 11: Comparison of the pre-filtering effectiveness of the two grouping methods on Snort rule-sets.

From Fig.10, it shows that the match probability using the original method is about 4 times that of

TABLE 2: Compilation time (seconds) when using different grouping methods on Snort rule-sets.

rule-set size	1K	2K	5K	10K	20K
original method	0.08	0.27	1.68	6.62	26.84
proposed method	0.51	1.81	8.35	24.36	76.29

using the proposed local search method on ClamAV rule-sets. In other words, the number of false positives generated by the original grouping method is 4 times that of our proposed grouping method. Therefore, our proposed grouping method improves the pre-filtering effectiveness by about 4 times on the test rule-sets from ClamAV. While for the test rule-sets from Snort, the pre-filtering effectiveness has increased by 2 times to 3.7 times, which is also a great improvement.

For Snort rule-sets, the compilation time of the extended shift-or algorithm using different grouping methods is shown in Table 2. Since our proposed grouping method requires an additional local search procedure, the compilation is slower than using the original grouping method. However, the increased compilation time is limited and therefore acceptable. Even when the rule-set size reaches as large as twenty thousand, employing our proposed grouping method, the compilation of the extended shift-or algorithm can still be finished in 100 seconds. For ClamAV rule-sets, the compilation time results are similar.

5. CONCLUSION

The extended shift-or algorithm proposed in hyperscan [11] is an efficient pre-filtering algorithm for pattern matching and is applied in the current most fast matching library. In this paper, the theoretical analysis of the pre-filtering effectiveness of the extended shift-or algorithm is given. It is shown that the extended shift-or algorithm can maintain a low match probability even when performing the matching of hundreds of thousands of patterns at the same time. Then, the extended shift-or algorithm is compared to the other two common pre-filters, the bloom filter and the direct filter. Since much fewer operations and memory accesses are required for matching per character, the comparison indicates that the extended shift-or algorithm is superior to the other two pre-filters. Finally, a novel pattern grouping method is put forward to optimize the extended shift-or algorithm, and experiments show that our proposed optimization improves the pre-filtering effectiveness by up to 4 times. From above all, we believe that our work is of great help to the design of practical pattern matching engines using the extended shift-or algorithm.

FUNDING

This work is supported by National Key Research and Development Program of China [No.2018YFB0204301].

REFERENCES

[1] (2012) Kargus: a highly-scalable software-based intrusion detection system. *Proceedings of the 2012 ACM conference on Computer and communications security*.

[2] (2008) Gnort: High performance network intrusion detection using graphics processors. *International Symposium on Recent Advances in Intrusion Detection*.

[3] Aho, A. V. (1975) Efficient string matching: An aid to bibliographic searach. *Comm Acm*, **18**, 333–340.

[4] Norton, M. (2004) Optimizing pattern matching for intrusion detection. *Sourcefire, Inc., Columbia, MD* , **?**

[5] Sun, W. and Manber, U. (1992) Fast text searching: allowing errors. *Communications of the ACM*, **35**, 83–91.

[6] Moraru, I. and Andersen, D. G. (2012) Exact pattern matching with feed-forward bloom filters. *Journal of Experimental Algorithmics (JEA)*, **17**, 3–1.

[7] Liu, T., Yong, S., Liu, A. X., Li, G., and ang, B. F. (2012) *A prefiltering approach to regular expression matching for network security systems.* Applied Cryptography and Network Security.

[8] Choi, B., Chae, J., Jamshed, M., Park, K., and Han, D. (2016) {DFC}: Accelerating string pattern matching for network applications. *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pp. 551–565.

[9] Haghighat, M. H. and Li, J. (2018) Toward fast regex pattern matching using simple patterns. *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*.

[10] Liu, A. X. and Norige, E. (2019) A de-compositional approach to regular expression matching for network security. *IEEE/ACM Transactions on Networking*, **PP**, 1–13.

[11] Wang, X., Hong, Y., Chang, H., Park, K., Langdale, G., Hu, J., and Zhu, H. (2019) Hyperscan: a fast multi-pattern regex matcher for modern cpus. *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 631–648.

[12] Zhao, Z., Sadok, H., Atre, N., Hoe, J. C., Sekar, V., and Sherry, J. (2020) Achieving 100gbps intrusion prevention on a single server. *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 1083–1100.

[13] Cs, A., Ma, A., Olb, A., and Mp, A. (2020) Multiple pattern matching for network security applications: Acceleration through vectorization. *Journal of Parallel and Distributed Computing*, **137**, 34–52.

[14] Hyperscan.io - a high-performance regular expression matcher from intel. https://www.hyperscan.io/. Accessed April 25, 2021.

[15] Bloom, B. H. (1970) Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13.7**, 422–426.

[16] davemmd/so-matching: Shift or algorithm research code. https://github.com/daveMmd/so-matching. Accessed April 25, 2021.

[17] Dmlki, B. (1964) An algorithm for syntactic analysis. *Computational Linguistics*, **3**, 29–46.

[18] Baeza-Yates, R. A. and Gonnet, G. H. (1988) A new approach to text searching. *ACM SIGIR Forum*, **23**, 168–175.

[19] BaezaYates and Alberto, R. (1989) Efficient text searching. *Thesis Dept of Computer Science University of Waterloo* , **?**

[20] Starobinski, D., Trachtenberg, A., and Agarwal, S. (2003) Efficient pda synchronization. *IEEE Transactions on Mobile Computing*, **2**, 40–51.

[21] Clamavnet. https://www.clamav.net/. Accessed April 25, 2021.

[22] Snort - network intrusion detection & prevention system. https://www.snort.org/. Accessed April 25, 2021.

[23] The zeek network security monitor. https://zeek.org/. Accessed October 7, 2020.

[24] Xu, C., Chen, S., Su, J., Yiu, S. M., and Hui, L. C. K. (2016) A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, **18**, 2991–3029.

[25] Zhong, J. and Chen, S. (2019) Accelerating dfa construction based on hierarchical merging. *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, pp. 1360–1365. IEEE.