Accelerating Deep Packet Inspection with SIMD-based Multi-literal Matching Engine

Hao Xu, Harry Chang, Kun Qiu, *Member, IEEE*, Yang Hong, Wenjun Zhu, Xiang Wang, Baoqian Li and Jin Zhao, *Senior Member, IEEE*

***Abstract*—Deep Packet Inspection (DPI) has been one of the most significant network security techniques. It is widely used to identify and classify network traffic in various applications such as web application firewall and intrusion detection. Different from traditional packet filtering that only examines packet headers, DPI detects payloads as well by comparing them with an existing signature database. The literal matching engine, which plays a key role in DPI, is the primary determinant of the system performance. FDR, an engine that utilizes 3 SIMD operations to match 1 character with multiple literals, has been developed and is currently one of the fastest literal matching engines. However, FDR has significant performance drop-off when faced with small-scale literal rule sets, whose proportion is more than 90% in modern databases. In this paper, we designed Teddy, an engine that is highly optimized for small-scale literal rule sets. Compared with FDR, Teddy significantly improves the matching efficiency by a novel shift-or matching algorithm that can simultaneously match up to 64 characters with only 15 SIMD operations. We evaluate Teddy with real-world traffic and rule sets. Experimental results show that its performance is up to 43.07x that of Aho-corasick (AC) and 2.17x that of FDR. Teddy has been successfully integrated into Hyperscan, together with which it is widely deployed in modern popular DPI applications such as Snort and Suricata.**

***Index Terms*—network security, DPI, SIMD, parallel computing**

I. Introduction

With the development of network security, features such as traffic classification, intrusion detection systems (IDS) [2], [3], and web application firewalls (WAF) [4], [5] have been broadly deployed in network systems [6]. The core of these features is traffic identification, which is provided by Deep Packet Inspection (DPI) [7]–[10]. Thus DPI technology has been widely used in applications such as Linux L7-Filter, Snort [2], ModSecurity [5], Zeek [11] and Suricata [3]. It makes the identification by examining packet payloads, searching them for specific regular expression (regex) rules, with a regex matching engine [12]–[15].

This work was supported by the National Natural Science Foundation of China under Grant No. 61972101. This work was presented in part at the ICPP 2021 conference [1]. *(Corresponding author: Jin Zhao.)*

Hao Xu, Kun Qiu, Wenjun Zhu and Jin Zhao are with the School of Computer Science, Fudan University, Shanghai 200438, China, and also with the Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 200438, China (email: xuhao21@m.fudan.edu.cn; qkun@fudan.edu.cn; wenjun@fudan.edu.cn; jzhao@fudan.edu.cn).

Harry Chang, Yang Hong, Xiang Wang and Baoqian Li are with Intel Asia-Pacific Research & Development Ltd., Shanghai 200240, China (email: fatchanghao@sina.com; hongyang729@msn.cn; xiang.w.wang@intel.com; baoqian.li@intel.com).

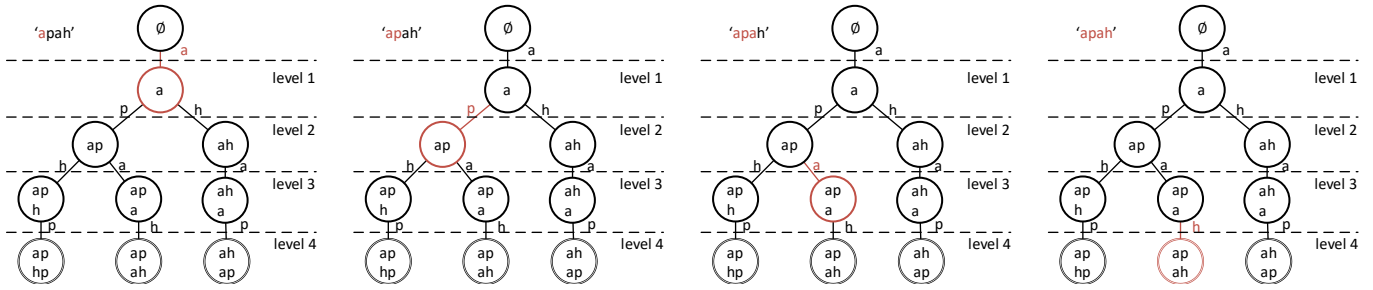
There is no doubt that the efficiency of the regex matching engine determines the performance of DPI system. To speed up regex matching, there is a trend toward using accelerators, such as GPU/NPU/FPGA [6], [16]–[25]. Nevertheless, most of these solutions suffer from one or more following shortcomings: 1) high capital costs, 2) insufficient memory to hold a large number of rules and 3) hard to be virtualized and provided as cloud-based middlebox services. While performance does improve, these solutions are rarely deployed. It is still the software regex matching algorithms running on CPUs that are adopted in most real scenarios.

However, in spite of continuous efforts, the performance of software regex matching algorithms [20], [21], [26]–[30] is still not satisfying in today's high bandwidth networks [31]–[37]. A recent study has shown that PCRE, a regex matching engine that is widely adopted by certain popular DPI applications such as Snort and Suricata, needs more than $6,942$ seconds to perform 1GB traffic matching, while this traffic has only $818,682$ packets [38]. Thus, pure regex matching is not a workable method, and taking string matching to pre-filter the packets has been proved to be the best remedy [39], [40], because string matching, or formally called literal matching, is two orders of magnitude faster than regex matching [41]. Nowadays, most real-world DPI applications has adopted this pre-filtering strategy [40]. They match specified literals before matching the regex, and the regex matching starts only if the literals are matched successfully. Therefore, literal matching also plays an important role in DPI system [42], [43], [43]–[45].

Literal matching, also called string searching or string matching, is a typical class of string algorithms. There are many literal matching algorithms, among which AC is the most classic one and FDR is the fastest one running on CPUs. FDR is integrated in Hyperscan [38], an efficient regex matching engine, and works at pre-filtering stage. It is based on the well-known bitwise-based Shift-Or algorithm [46]–[48] and can support matching multiple literals simultaneously, which we call multi-literal matching.

Nonetheless, FDR has a significant issue in small-scale literal rule sets. A small-scale literal rule set is empirically defined as a rule set whose number of literals is less than 60 and where most of the literals are shorter than 8 bytes. It is reported that small-scale literal rule sets appear frequently in the field of DPI. As an example, more than 90% of the rule sets in ModSecurity are small-scale ones [49].

When confronted with small-scale literal rule sets, FDR has an obvious problem. Since FDR has to do 8 times of *shift* and



(a) The first letter is *'a'*, which can be matched by the edge $\emptyset \rightarrow a$, then node a is activated.

(b) The second letter is *'p'*, which can be matched by the edge $a \rightarrow ap$, then node ap is activated.

(c) The third letter is *'a'*, which can be matched by the edge $ap \rightarrow apa$, then node apa is activated.

(d) The last letter is *'h'*, which can be matched by the edge $apa \rightarrow apah$, then node $apah$ is activated. And this result will be reported.

Fig. 1. An example of trie with literals *'aphp'*, *'ahap'*, *'apah'*. The nodes with double circles contain the literals. The nodes with single circle contain the prefixes of the literals. The edges between nodes indicate the search conditions. Level p indicates that p characters have been matched. We also give the matching process in detail when the input string is *'apah'*. We initially activate the root node $\emptyset$. During matching *'apah'* with the literals, node a, ap, apa, $apah$ will be activated in sequence. We use the red circle to indicate that the node is activated right now.

or operations no matter whether or not a literal is shorter than 8 bytes, it introduces a considerable amount of unnecessary computation. The unnecessary computation finally leads to a low data-level parallelism that FDR can only match 1 input byte per 3 SIMD operations.

In order to solve this problem, we proposed Teddy, a novel multi-literal matching engine with a new SIMD-based matching algorithm specializing in small-scale literal rule sets. Teddy has a notable performance, up to 43.07x that of AC and 2.16x that of FDR, when dealing with small-scale literal rule sets.

The main contributions of this paper are presented as follows.

- We have proposed a new suffix-based matching strategy that can largely reduce the number of *shift* and *or* operations.
- We have designed a high-performance SIMD-based shift-or matching algorithm that can simultaneously match up to 64 characters with only 15 SIMD operations.
- We have proposed a half-byte encoding method with which Teddy can be implemented on commodity CPUs.
- We found that a drawback of the *shift* instruction in AVX512 SIMD instruction set may cause plenty of false positives that influence Teddy's performance. And we introduced reinforced mask to reduce these false positives, which significantly improved Teddy's performance by up to 74% in small-scale literal rule sets.
- We implemented Teddy with AVX512 SIMD instruction set and evaluated it with real-world network traffic and DPI literal rules. We integrated Teddy into Hyperscan, together with which it has been successfully deployed in several famous DPI applications such as Snort and Suricata.

This work is an extended version of our prior conference paper [1]. The content extended in this paper includes:

- We found the underlying drawback of the *shift* instruction in AVX512 SIMD instruction set and the false positives it caused. We then introduced reinforced mask to reduce these false positives and improved Teddy's performance by up to 74%.
- We conducted two more groups of experiments. One was to compare the number of false positives with and without our new grouping strategy, so we could show the effectiveness of the strategy. The other was to compare Teddy with FDR from more dimensions, not only throughput but also false positives, shift-or matching time and exact matching time. This comprehensive analysis clearly demonstrates why Teddy outperforms FDR in terms of speed.

The rest of the paper is organized as follows. Section II introduces related work. Section III gives the overview design of Teddy. Section IV describes the new suffix-based matching strategy and grouping algorithm. Section V explains the SIMD-based matching algorithm and the half-byte encoding method. Section VI gives a detailed description of reinforced mask. Section VII evaluates Teddy. Finally, Section VIII gives a conclusion.

II. Related Work

A. Prefix-based literal matching

Before we give detailed information of Shift-Or, we introduce the prefix-based literal matching first. Literal matching [50]–[52], also called string searching or string matching, is an important class of string algorithms. Literal matching tries to find a place where strings (or called literals) are found within a longer input string. For example, let's search for the pattern *ap'* within an input string *aphpap'*. The first occurrence of *'ap'* is at the first byte, and the second occurrence is at the fifth byte. The naïve literal matching algorithm, which checks the place where the literal occurs inside the input string one byte by one, is an inefficient literal matching algorithm.

The prefix-based matching algorithms, such as Knuth-Morris-Pratt (KMP) [53], Shift-Or [46] and Aho-Corasick (AC) [50], are proven as successful strategies in practice. Formally, the prefix of a string is p letters ($p \leq m$ where m is the length of the string) that begins from the first byte of the string. As an example, *'aphpap'* has the following

TABLE I
TERMS OF DEFINITION

Notation	Description
n	The number of literals
m	The length of input string s
l	The maximum length of literals
w_i	The i-th($i \leq n$) literal
l_i	The length of literal w_i
s_i	The i-th($i \leq m$) character of input string s
Λ	The set of ASCII characters
$w_{i,j}$	The j-th($j \leq w_i$) character in literal w_i
p	The number of buckets
q_v	The number of literals in v-th($v \leq p$) bucket
r_v	The maximum length of literals in v-th($v \leq p$) bucket
λ_j^v	The set of j-th($j \leq r_v$) characters in v-th($v \leq p$) bucket

prefixes: *'a', 'ap', 'aph', 'aphp', 'aphpa'*. The prefix tree, also called trie, is a kind of ordered search tree that is used to store prefixes and perform prefix-based literal matching. Prefixes are stored in a top-to-bottom manner. We give a trie example with input string *'apah'* and 3 literals *'aphp', 'ahap', 'apah'* in Figure 1 to show a prefix-based literal matching.

We give the terms of definition in Table I.

B. Classical Shift-Or algorithm

Shift-Or algorithm uses bitwise techniques to check whether or not the given literal is present in the input string. Both FDR and Teddy are based on it.

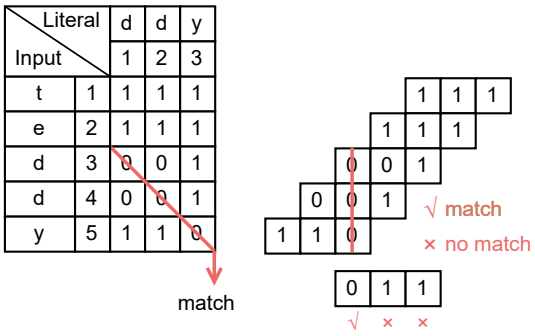
Suppose the input string s is *'teddy'* and the literal w is *'ddy'*. Shift-Or firstly builds, according to the literal, a mask table as shown in Figure 2.

Literal	d	d	y
Offset / ASCII	1	2	3
0x00	1	1	1
...	1	1	1
d	0	0	1
...	1	1	1
y	1	1	0
...	1	1	1
0xff	1	1	1

Fig. 2. Mask table. Each bit in this table indicates whether or not an ASCII character is at certain position of the literal. For example, the bit 0 in that orange cell indicates that character *'d'* is the second character of literal *'ddy'*, the bit 1 in that purple cell indicates that character *'y'* is not the first character of *'ddy'*.

We write this table as T. For c ($c \in \lambda$ where Λ is the set of ASCII characters), we say that T_c is the mask of c. If $c = w_k$, we set $T_{c,k} = 0$, else we set $T_{c,k} = 1$ ($k \in [1, l]$, where l is the length of literal w). $T_{c,k} = 0$ indicates that c is the k-th character of literal w.

For each input character $s_i (i \in [1, m]$, where m is length of the input string), we *load* T_{s_i} to make a match table. This table indicates the match result, as shown in Figure 3(a).



(a) Match table. It can be seen that a diagonal line passes zero bits which are $T_{s_3,1}$, $T_{s_4,2}$, $T_{s_5,3}$. This means that the third, fourth and fifth characters of input string are same as the first, second and third characters of the literal. Thus the literal is matched successfully.

(b) From top to bottom, we shift 3 masks 4, 3, 2, 1, 0 times respectively to align the corresponding bits on the same diagonal. Then we perform *or* operations to get the result vector. The zero bit in the result vector indicates a match.

Fig. 3. Match table and the shift-or process.

From this figure we know that the literal is matched successfully if l zero bits are on a diagonal line, where l is the literal length. To recognize this situation, we can *shift* masks in the match table to align the zero bits on the same diagonal and perform *or* operations to find the match position, as shown in Figure 3(b).

C. FDR

FDR extends Shift-Or algorithm to support multi-literal matching. For mask table, it sets $T_{c,k} = (b_1 b_2 \dots b_n)_2$ and $b_j = 0$ only if $c = w_{j,k} (j \in [1, n], k \in [1, l]$, where n is the number of the literals and l is the maximum length of the literals). $T_{c,k,j} = 0$ indicates that c is the k-th character of the j-th literal.

After building the mask table, *load*, *shift* and *or* operations are performed. This procedure is much like that of single-literal matching. Difference is that the mask has $n \cdot l$ bits and there are n bits being shifted each time, where n is the number of the literals and l is their maximum length.

FDR fixes values of both n and l to 8, so a mask would contain 64 bits, which is not too long to lose some bits during *shift* operations. And if $n > 8$ or $l > 8$, it takes the following two strategies:

1) **Grouping:** if $n > 8$, FDR will group them into 8 buckets.
2) **Truncation:** if $l > 8$, FDR will truncate the literal and only match its 8-byte-long suffix during shift-or matching process.

The grouping method is straight forward. We give an example in Figure 4. Theoretically, FDR takes 8 buckets, each of which has $q_1, q_2, \ldots, q_8$ literals. We define $w_1^v, w_2^v, \ldots, w_{q_v}^v$ are literals in the v-th bucket, and we define a character set $\lambda_k^v = \cup_{i=1}^{q_v}(w_{i,k}^v)$, which contains the k-th characters of all literals in the v-th bucket v. Accordingly, FDR defines the mask table as $T_{c,k} = (b_1 b_2 \ldots b_8)_2$, where $b_v = 0$ only if $c \in \lambda_k^v$.

Regardless of the number of the literals or the maximum literal length, masks in FDR will always be of $8 \times 8 = 64$ bits, with the grouping strategy allocating the literals into 8 buckets and the truncation strategy taking only the 8-byte-long suffixes of those literals for shift-or matching.

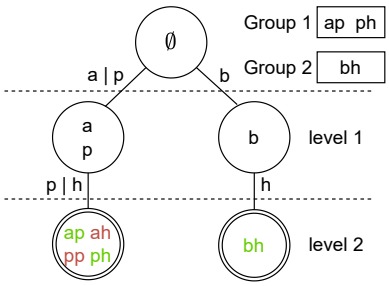


Fig. 4. An example of grouping. We group *'ap'* and *'ph'* into Bucket 1, *'bh'* into Bucket 2. The left node at level 1 can accept both characters *'a'* and *'p'*, and the left node in level 2 can accept both characters *'p'* and *'h'*. Therefore, all of *'ap'*, *'pp'*, *'ah'* and *'ph'* can be matched into Bucket 1. However, only *'ap'*,*'ph'* are true positive results, *'pp'*,*'ah'* are false positive results.

D. False-positive results

Obviously, the aforementioned two strategies may generate false positive results if we define the real matching results as *Ground Truth*:

1) **Grouping** As shown in Figure 4, false positive results will be probably generated after grouping.
2) **Truncation** The 8-byte-long suffix of a literal is matched but the prefix is not matched. This can also generate false positives.

When a positive result occurs in certain bucket, the positive piece of string will be compared with all literals in that bucket to check whether or not it is a true positive. In contrast to shift-or matching, this process is referred to as exact matching. Therefore if the positive is false, exact matching will be a waste of time. Obviously, a well-designed grouping strategy and truncation length can decrease the number of false positives and increase system's overall performance.

E. Motivations

FDR includes two stages of literal matching, which are shift-or matching and exact matching. In shift-or matching process, it matches the 8-byte-long suffixes of the literals. If a literal is shorter than 8, FDR would pad it with wildcards. Therefore it always needs 8 *load*, 8 *shift* and 8 *or* operations per iteration. When confronted with small-scale literal rule sets where most of the literals are shorter than 8 bytes, this can be quite a waste of computation resources. So we would like to take a σ-suffix-based matching strategy, which means that in shift-or matching process we match only the σ-byte-long suffixes of the literals. With small-scale literal rule sets, we would adjust σ from 1 to 4 and conduct experiments to find the best value of σ.

Also, after changing the shift-or matching strategy, the corresponding grouping strategy should also be adjusted because a well-designed grouping method could reduce the false positives. The grouping strategy of FDR follows two guidelines: i) group the literals of a similar length into the same bucket. ii) avoid grouping too many short literals into one bucket. Apparently, both of them are for general literal rule sets. For small-scale rule sets, we would design a suffix-based grouping strategy, which allocates literals that have similar σ-byte-long suffixes into the same bucket. We will demonstrate the correctness of this grouping strategy and conduct experiments to showcase its effectiveness in reducing false positives.

III. The Teddy Architecture

The core objective of Teddy is to offer an efficient multi-literal matching engine that has excellent performance in dealing with small-scale literal rule sets. In this section, we will show the overview design of Teddy and give a brief introduction to its components.

Teddy's overview architecture is shown in Figure 5. Teddy works through two stages: compile time and run time. During compile time, Teddy groups literals into 8 buckets and builds two tables, the mask table and the reinforced mask table, which will be used in shift-or matching process. During run time, Teddy experiences two matching stages, shift-or matching and exact matching. Shift-or matching is used to identify all possible occurrences of matches, which we refer to as positives. In contrast, exact matching is employed to filter out the false positives. In the overall workflow and the exact matching part, Teddy is identical to FDR. The differences lie in Teddy's newly designed suffix-based grouping strategy, SIMD-based shift-or matching algorithm, encoding method and reinforced mask.

A. The grouping strategy

Different from FDR's grouping algorithm, Teddy's grouping strategy groups literals by the similarity of σ-byte-long suffixes of literals. For example, suppose $\sigma = 2$, *'abcde'* is more similar to *'efgde'* than *'abcgf'*. So *'abcde'* and *'efgde'* will be assigned into the same bucket. Thus, Teddy uses σ as the truncation length and matches σ-byte-long suffixes in shift-or matching process. In small-scale rule sets, this grouping strategy can significantly reduce false positives.

B. The SIMD-based shift-or matching algorithm

During run time, Teddy matches the literals with the input string. It will use a SIMD-based shift-or matching algorithm to match the literals in parallel. For efficiency concerns, the matching algorithm only matches the σ-byte-long suffixes of the literals, i.e., the truncation length is σ. During shift-or

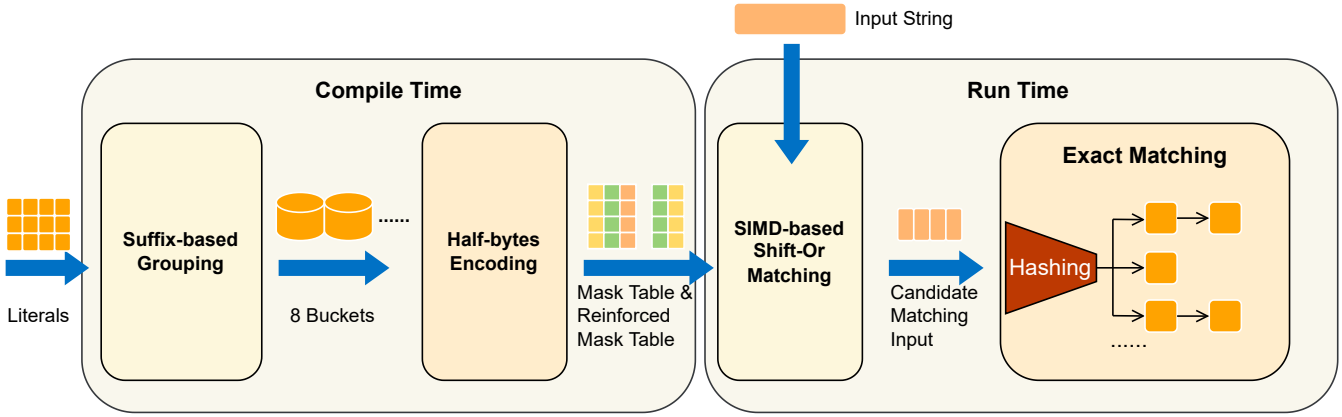


Fig. 5. An overview design of Teddy. Teddy works through two stages: compile time and run time. During compile time, Teddy groups the literals into 8 buckets and builds two tables, a mask table and a reinforced mask table. During run time, Teddy matches multiple literals in parallel with the input string. The input string first goes through shift-or matching, leaving the possible matched pieces, then exact matching, filtering out the false positives.

matching process, FDR matches 1 character per 3 SIMD operations, while Teddy can simultaneously match up to 64 characters per 5σ SIMD operations.

C. The encoding method

Although Teddy's high-efficiency shift-or matching algorithm enhances data-level parallelism, we encountered challenges while attempting to implement it with modern SIMD instruction sets. The primary issue is that an SIMD vector is not long enough to accommodate a column vector of Teddy's mask table. Thus we propose a novel encoding method, which we referred to as half-byte encoding, to compress the mask table.

D. The reinforced mask

Due to the drawback of the *shift* instruction in AVX512 SIMD instruction set, Teddy would lose some bits during shifting the vectors, which then causes lots of false positives that influence the overall performance. Therefore, we propose an innovative method to reduce these false positives by introducing reinforced mask to fill in the missing bits of information. The reinforced mask significantly reduces false positives and improves Teddy's performance by up to 74%.

IV. The Grouping Strategy

As we have mentioned above, Teddy significantly reduces false positives in small-scale literal rule sets by designing a new suffix-based grouping algorithm. In this section, we will describe this grouping algorithm in detail. We will first describe the scoring algorithm, which can compute the score of a bucket that includes several literals. Then we will give the heuristic greedy algorithm based on the score. At last, we will give proof of the correctness of our grouping strategy.

A. The score of literals in a bucket

We use a score to indicate the difference degree among literals in a bucket. The score of a bucket will be increased if another literal is put into the bucket. The easiest way to define the score is to count the number of bit 1 for each character in the σ-byte-long suffix and multiply these numbers to get the score value. When a new literal is put into the bucket, we use bit-*or* operation '|' to calculate the new score of the bucket. We present the algorithm in Algorithm 14. Also, we give an example in Figure 6 to show how to calculate the score.

Algorithm 1: SCORE(v)
calculating the score in bucket v

Input: q literals in the bucket
Output: S the score of the bucket
$w_{i,j}$ is the j-th character in literal w_i
l_i is the length of literal w_i
σ is the length of suffix
| is the bit-*or* operation
COUNT is the number of BIT 1 in a character
$S \leftarrow 1$
$B_1, \ldots, B_\sigma \leftarrow 0$
for i *from* 1 *to* q **do**
  for j *from* 1 *to* σ **do**
    $B_j \leftarrow B_j \mid w_{i,l_i-\sigma+j-1}$
for i *from* 1 *to* σ **do**
  $c \leftarrow \text{COUNT}(B_i)$
  $S \leftarrow S * c$
return S

B. The greedy-based merging algorithm

We can use SCORE(v) to calculate the score of a bucket. In the beginning, n literals are mapped into n buckets. Then, the heuristic algorithm will iteratively merge n buckets into p buckets (usually $p = 8$). In each iteration, the algorithm always merges two buckets that can result in the minimum score increasing. Theoretically, suppose there are p buckets in the current iteration, and each bucket is $v_i, i \in [1, p]$. We use $v_{i,j}$ to indicate the merged bucket with v_i and v_j. The algorithm will choose two suitable (i, j) that have

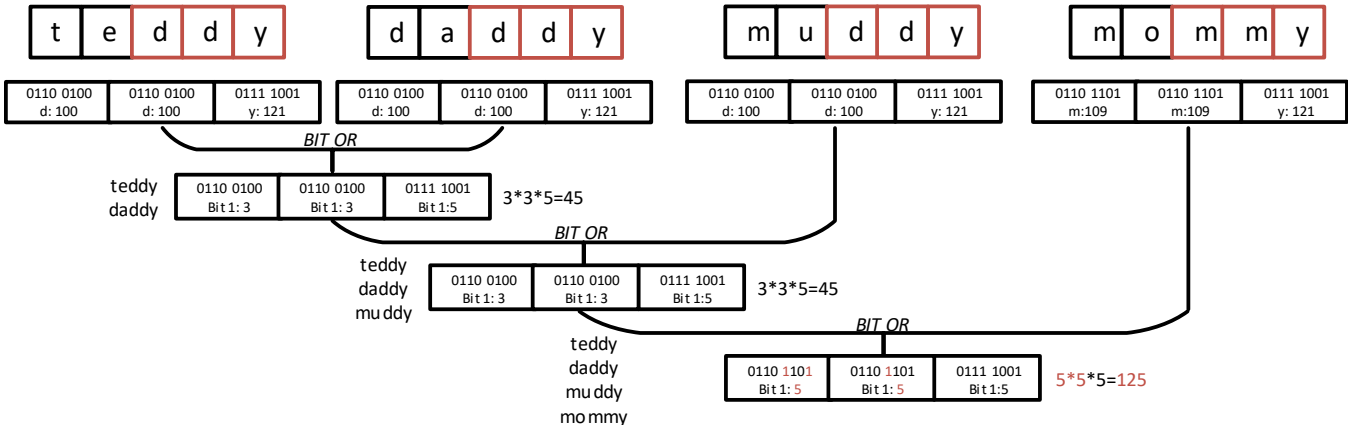


Fig. 6. An example of computing the score of literals in a bucket. We group 4 literals:*'teddy'*, *'daddy'*, *'muddy'*, and *'mommy'* into 1 bucket. We suppose $\sigma = 3$. We first group *'teddy'* and *'daddy'*. The suffix of these two literals is both *'ddy'*, and the bitwise presentation of *'d'* and *'y'* is 01100100_2 and 01111101_2. The BIT 1 numbers of *'d'* and *'y'* are 3 and 5. We multiply them and get the score $3 * 3 * 5 = 45$. Then, we group *'teddy'*, *'daddy'* with *'muddy'*. Since the suffix of the newly-added literal is the same as the existing suffix, the score is still 45. At last, we group them with *'mommy'*. The bitwise presentation of *'m'* is 01101101_2, and the result of bit-or between *'d'* and *'m'* is 01101101_2. It makes the number of BIT 1 increase to 5. Thus, the score of the bucket is $5 * 5 * 5 = 125$.

the minimum SCORE($v_{i,j}$), $\forall i, j \in [1, p]$. We give detailed information in Algorithm 21.

Algorithm 2: Grouping n literals into p buckets

Input: n literals
Output: p buckets

```
1  t is the number of buckets in current iteration
2  b is the minimum score in current iteration
3  t ← n
4  while t > p do
5      best ← ∞
6      for i from 1 to t do
7          for j from i to t do
8              new ← SCORE(v_{i,j})
9              old ← SCORE(v_i) + SCORE(v_j)
10             if new < old then
11                 i_min ← i
12                 j_min ← j
13                 break
14             if best > new − old then
15                 best ← new − old
16                 i_min ← i
17                 j_min ← j
18     if find i_min and j_min in this iteration then
19         merge v_{i_min} with v_{j_min} into a bucket
20         t ← t − 1
21 return p buckets
```

C. Proof of correctness

We can simply prove that the score can indicate the difference degree among literals in the bucket.

Proposition 1. Suppose there are two characters α and β. COUNT is the number of BIT 1 in a character. COUNT($\alpha \mid \beta$) $\geq$ COUNT(α) and COUNT($\alpha \mid \beta$) $\geq$ COUNT(β).

Proof 1. If $\alpha = \beta$, COUNT($\alpha|\beta$) = Count(α) and COUNT($\alpha|\beta$) = Count(β). If $\alpha \neq \beta$, according to the definition of 'bit-or', BIT 1 in α but not in β and BIT 1 in β but not in α will all be counted in the COUNT($\alpha \mid \beta$), which makes COUNT($\alpha \mid \beta$) $\geq$ COUNT(α) and COUNT($\alpha \mid \beta$) $\geq$ COUNT(β). Thus, Proposition 1 is proved.

Proposition 2. Suppose t is the number of buckets. $\forall i, j \in [1, t]$, we have SCORE($v_{i,j}$) $\geq$ SCORE(v_i), and SCORE($v_{i,j}$) $\geq$ SCORE(v_j).

Proof 2. According to Proposition 1 and the line 10 in the Algorithm 1, $B_1, \ldots, B_\sigma$ in bucket $v_{i,j}$ will be greater than or equal to that in v_i and v_j. Also, according to the line 11 to 13, S in bucket $v_{i,j}$ will be greater than or equal to v_i and v_j. Thus, Proposition 2 is proved.

In brief, Proposition 1 and Proposition 2 show that the score of buckets is monotonous increasing during the grouping process. Also, the fewer differences between the two buckets, the less score increased after their merging. If all literals in one bucket have the same suffix, the score will never change during the grouping process. Thus, we can use this score as a metric in our greedy-based merging algorithm.

V. THE ENCODING METHOD AND SIMD IMPLEMENTATION

A. Classical mask table

We have given the definition of FDR's mask table in §II-C. Teddy's mask table T is quite similar to that of FDR: we set $T_{c,k} = (b_1 b_2 \ldots b_8)_2$ and $b_j = 0$ only if $c \in \lambda_k^j (c \in \Lambda, j \in [1, 8], k \in [1, \sigma]$ where Λ is the set of ASCII characters).

B. SIMD parallelizing

After building the mask table, we can propose a SIMD-based shift-or matching algorithm to match input string with 8 buckets. We give an example in Figure 7. Note that from now on we only show 1 bucket in our examples and figures, but don't forget that there are actually 8 buckets, which means that every cell in mask table and match table should have 8 bits, i.e., 1 byte.

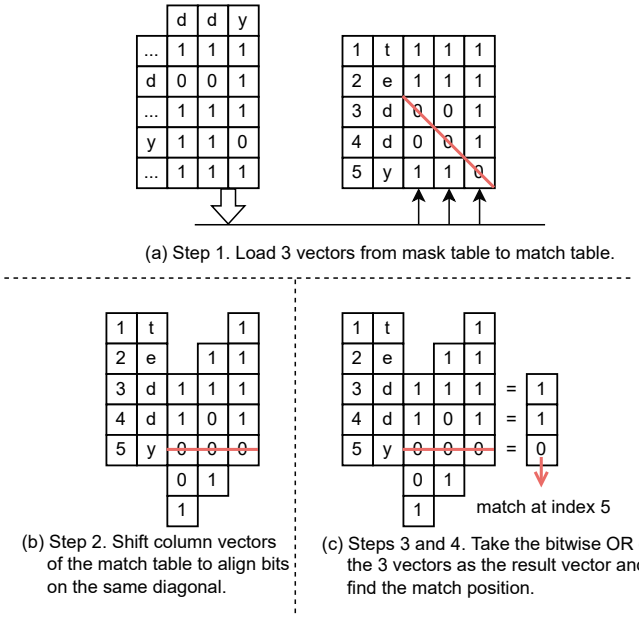


Fig. 7. An example of SIMD implementation. Suppose the input string is *'teddy'*, $\sigma = 3$ and there is only 1 literal, whose 3-byte-long suffix is *'ddy'*, in the bucket. There are 4 steps: (1) we load $\{T_{t,k}, T_{e,k}, T_{d,k}, T_{d,k}, T_{y,k}\}$ for $k = 1, 2, 3$ to get the match table. Briefly speaking, we load vector $\{1, 1, 0, 0, 1\}$ for $k = 1$ and $k = 2$, and we load $\{1, 1, 1, 1, 0\}$ for $k = 3$. (2) we shift the first vector (the one loaded for $k = 1$) 2 times, and shift the second vector (the one loaded for $k = 2$) 1 time. (3) we take the bitwise OR of the 3 vectors. (4) we find bit 0 in the result vector. The position of bit 0 indicates the place where suffix *'ddy'* is matched.

In the above example, the input string *'teddy'* is quite short, just 5 bytes, and a vector of the match table has only $5 \times 8 = 40$ bits. Usually it can be processed with just 1 iteration of (load, shift, or) process only if the SIMD vector is longer than 40 bits. Generally speaking, the number of input bytes that can be processed in an iteration is dependent on the SIMD vector length. If the SIMD vector can hold L bits, then Teddy can process $\frac{L}{8}$ input bytes in an iteration. Given an m-byte-long input string and σ-byte-long suffix, the shift-or matching steps are: (1) read $\frac{L}{8}$ bytes from the input string and take them as indices to load σ vectors from mask table; (2) shift $\sigma - 1$ vectors; (3) take bitwise OR of the σ vectors; (4) find the 0 bits in the result vector which reflect matching positions; (5) repeat steps (1) to (4) until the whole input string has been processed.

C. Half-byte encoding

The aforementioned algorithm can largely reduce the *load*, *shift* and *or* operations. However, there are lots of implementation issues with deploying it on modern CPUs. In most CPU architectures, the instruction width of SIMD is limited (e.g., SSE only supports 128 bits per vector, AVX2 only supports 256 bits per vector, and AVX512 only supports 512 bits per vector). This brings challenges to step (1), where we have to load vectors from columns of the mask table. We give a more detailed example of this step in Figure 8.

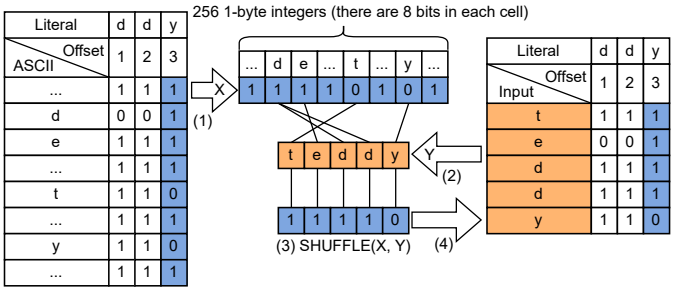


Fig. 8. An example of loading a vector from mask table with SIMD instruction SHUFFLE. SHUFFLE has two parameters: *src* and *ctl*. We use X to indicate *src* and Y to indicate *ctl* in this figure. The function of SHUFFLE is parallelly reading elements in X with indices in Y and putting them into a result vector. In this example, we use SHUFFLE to load the elements whose indices are *'t'*, *'e'*, *'d'*, *'d'* and *'y'* from the mask table. We set the last column vector of the mask table as X and $\{$*'t'*, *'e'*, *'d'*, *'d'*, *'y'*$\}$ as Y, then we use SHUFFLE to get the result vector $\{1, 1, 1, 1, 0\}$.

It can be observed that columns of the mask table contain $256 \times 8 = 2048$ bits. However, there are no SIMD instructions that can load elements from a 2048-bit SIMD vector currently (2048-bit SIMD vector is not supported by current CPUs). So, we have to design a new encoding method to compress the mask table.

As shown in Figure 9, we use two 4-bit integers, also called half-bytes, to indicate a character. Since the range of 4-bit integer is $0 \sim 15$, the mask table has been changed to $T'_{c,k}, (c \in [0, 15], k \in [1, 2\sigma])$. If $T_{c,k,j} = 0$, we have $T'_{c_l,k_l,j} = 0, T'_{c_h,k_h,j} = 0$. c_l is the lower 4 bits of character c and c_h is the higher 4 bits of character c. With the new mask table $T'_{c,k}$, we can use SHUFFLE to load elements in the half-byte-encoded mask table, as shown in Figure 10.

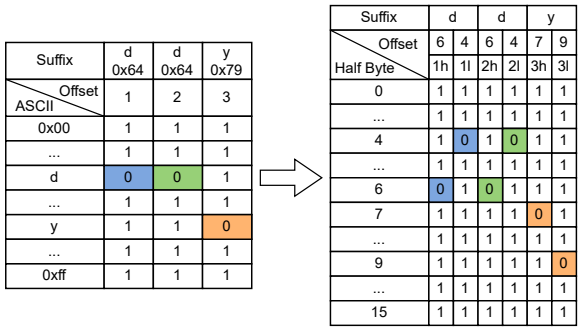


Fig. 9. An example of half-byte encoding. We use two 4-bit integers, also called half-bytes, to indicate a character. Thus, the 3-byte-long suffix *'ddy'* has been converted to 6-integer-long suffix *4,6,4,6,9,7*. Also, the elements in mask table need to be changed. For example, because $T_{d,1} = 0$, we have $T'_{4,1_l} = 0, T'_{6,1_h} = 0$.

Since column vectors of the mask table has been shortened to $16 \times 8 = 128$ bits. Current CPUs can use SHUFFLE instruction to load corresponding elements from the mask

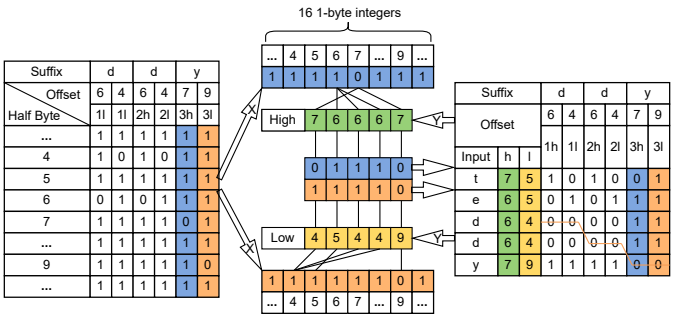


Fig. 10. An example of loading vectors from the new mask table with instruction SHUFFLE. We use SHUFFLE twice to load from mask table the elements whose indices are half-bytes of *'t'*, *'e'*, *'d'*, *'d'* and *'y'*. For the first time, we load from the vector of offset 3_h in mask table and indices are the higher half-bytes of {*'t'*, *'e'*, *'d'*, *'d'*, *'y'*}({7, 6, 6, 6, 7}), then we use SHUFFLE to get the result vector $\{0, 1, 1, 1, 0\}$. For the second time, we load from the vector of offset 3_l in mask table and indices are the lower half-bytes of {*'t'*, *'e'*, *'d'*, *'d'*, *'y'*}({4, 5, 4, 4, 9}), then we use SHUFFLE to get the result vector $\{1, 1, 1, 1, 0\}$. At last, we bit-or the two result vectors to get the final result vector of offset 3, i.e., $\{1, 1, 1, 1, 0\}$.

table. For each byte of the suffix, we have to use SHUFFLE twice: load the vector of higher half-byte and load the vector of lower half-byte. Then, take bitwise OR of the two vectors. Using half-byte-encoded mask table will bring additional σ *or* operations since we have to take bitwise OR of two half-byte vectors. Therefore, we need 2σ *load* operations to load σ pairs of vectors from the new mask table, σ *or* operations to be performed on the σ pairs of vectors and get σ vectors, $\sigma-1$ *shift* operations to shift the σ vectors and $\sigma-1$ *or* operations to take the bitwise OR of the shifted vectors.

In conclusion, Teddy needs about 5σ operations to process $\frac{L}{8}$ input bytes (L is the SIMD vector length, i.e., the number of bits an SIMD vector can hold) in an iteration. Specifically, with AVX512 SIMD instruction set whose SIMD vector can hold 512 bits and with $\sigma = 3$ which has been proved to be the optimal value of σ by our practical experience, Teddy needs 15 SIMD operations to process 64 input bytes, which is much more efficient than FDR, who needs 3 SIMD operations to process 1 byte.

VI. The Reinforced Mask

After deploying Teddy in a production environment and applying it to small-scale literal rule sets over a period of time, we were perplexed to discover that the number of false positives in its shift-or matching process was unexpectedly high, exceeding our initial expectations. Diving into details, we found that these additional false positives were caused by the drawback of the *shift* instruction in AVX512 SIMD instruction set.

In this section, we would explain why the *shift* instruction leads to additional false positives and how Teddy deals with it.

A. Additional false positives

Teddy processes a chunk of input bytes in an iteration. For each chunk, it performs $\sigma - 1$ *shift* operations. Because the existing *shift* instruction of AVX512 SIMD instruction set cannot shift bits across 16-byte boundary, some bits will be lost after shifting. These lost bits may lead to additional false positives.

Suppose the literal suffix is *'ddy'*, the last two characters of previous chunk are *'a'* and *'b'*, the first character of current chunk is *'y'*. Lost bits would lead to a false positive at index 1 in current chunk, as shown in Figure 11. In this case, 3 mask bits of *'a'* and *'b'* get lost during shifting. This actually means that only part of *'aby'*, i.e., *'y'*, is matched.

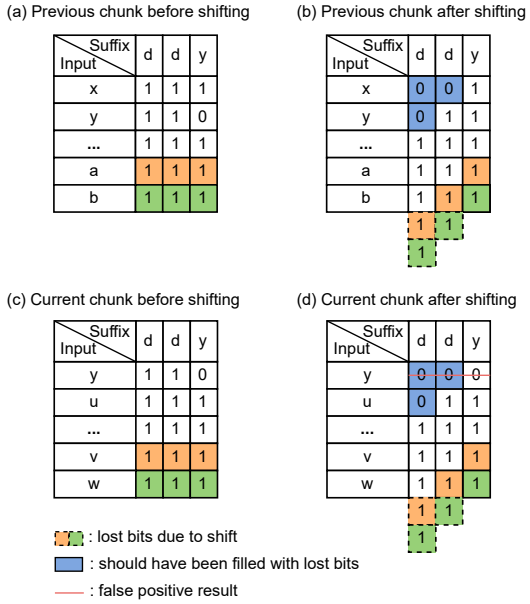


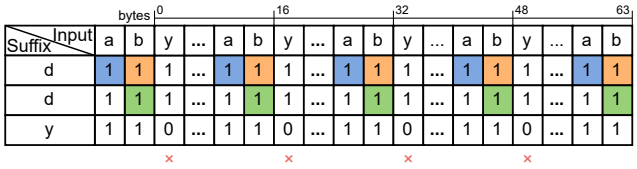
Fig. 11. The sequence *'aby'* scratches across two chunks with *'ab'* in previous chunk and *'y'* in current chunk. Obviously, *'aby'* doesn't match with *'ddy'*. However, after shifting, 3 mask bits of *'a'* and *'b'* in previous chunk, which should have been shifted to current chunk, are lost. Meanwhile, 3 zero bits are filled in current chunk to replace the lost bits. Finally, a false positive occurs at index 1 in current chunk and it matches *'aby'* wrongly with *'ddy'*.

With AVX512 SIMD instruction set, Teddy can handle 64 input bytes in an iteration. During each iteration, the *shift* operation is implemented by instruction VPSLLDQ. Unfortunately, this instruction has a drawback that it cannot shift bits across 16-byte boundary. Since the chunk size of AVX512 Teddy is 64 bytes, a chunk includes 4 16-byte boundaries where some bits will be lost after shift and false positives may appear. Suppose the input string has a sequence *'aby'* appearing at each 16-byte boundary, with *'ab'* in the left 16-byte region and *'y'* in the right region, the match table and shift-or results are shown in Figure 12.

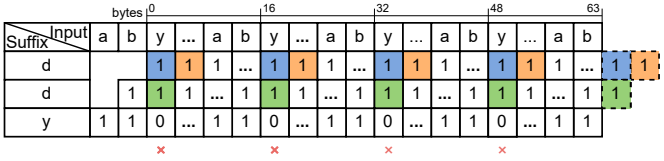
At offset 0, 16, 32 and 48, Teddy only matches the last character *'y'* of *'aby'*, so it matches *'aby'* wrongly with *'ddy'*. In practice, AVX512 Teddy has plenty of additional false positives caused by the *shift* instruction and they will severely influence the overall system performance.

B. The reinforced mask

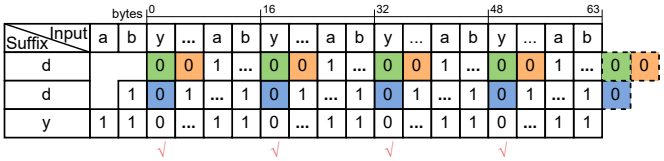
Through experiments, we found that these additional false positives account for the majority of total false positives and may have a great influence on overall system performance. So



(a) Match table. Look up the mask table and load masks to build this match table. Obviously, There are no positives at offset 0, 16, 32 and 48.



(b) Expected shift result. At each 16-byte boundary, bits in colored cells are those shifted from the left 16-byte regions. They should be mask bits of *'a'* and *'b'*. There are no positives at offset 0, 16, 32 and 48.



(c) Actual shift result. Actually, bits in colored cells are all 0 bits because VPSLLDQ cannot shift bits across 16-byte boundary and 0 bits take the place. Finally, false positives appear at offset 0, 16, 32 and 48.

Fig. 12. Because *'aby'* doesn't match with *'ddy'*, there should be no positives at offset 0, 16, 32 and 48, just like (b). However, due to the fact that VPSLLDQ cannot shift bits across 16-byte boundary, the actual shift-or result is like (c). At offset 0, 16, 32 and 48 positives wrongly appear. They are all false positives.

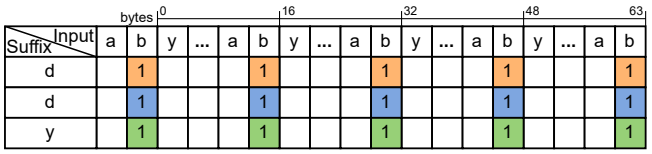
Teddy tries to reduce their amount by filling in the lost bits during shift-or process, as shown in Figure 13.

We call the mask that contains the lost bits of a 16-byte region's last character as reinforced mask. And we call the newly joined vector that contains the 4 reinforced masks as reinforced vector. Teddy lets the reinforced vector join the *or* operation together with the 3 shifted vectors. In this way, it fills in the lost bits of the last characters in each 16-byte region. Before doing this, Teddy only matches the last character of the suffix at 16-byte boundaries, so *'aby'* matches wrongly with *'ddy'*. But now Teddy will match the last two characters of the suffix, so *'aby'* doesn't match with *'ddy'* because *'by'* doesn't match with *'dy'*.

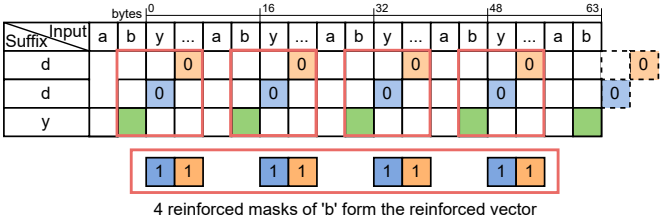
In practice, Teddy would build a reinforced mask table at compile time, as shown in Figure 14. At run time, Teddy would load necessary reinforced masks to form the reinforced vector and let it join the *or* operation together with those shifted vectors.

C. How strong should it be reinforced to?

As we have mentioned, building a reinforced mask table and filling in the lost bits of the last bytes in each 16-byte region could help Teddy to match one more byte at 16-byte boundaries, and thus reduce the additional false positives. However, this cannot eliminate all additional false positives, as shown in Figure 15.



(a) Match table before shifting. Before shifting, the mask bits of *'b'* are arranged vertically.



(b) Match table after shifting. In each square is the shift result of *b*'s mask bits. After shifting, mask bits of *'b'* are arranged diagonally. However, the first two bits of *'b'* are lost. The light-colored cells are padded with 0 bits rather than bits of *'b'*. But now that the bitwise *or* operation is going to be performed on those shifted vectors, we can still take the lost bits to form a new vector to join the *or* operation. The lost bits can be easily gained from the mask table. We call the mask that contains the lost bits of the last character in a 16-byte region as reinforced mask. And we call the newly joined vector that contains the 4 reinforced masks as reinforced vector.

Fig. 13. After shifting, two mask bits of *'b'* are lost, but these lost bits can still be filled in.

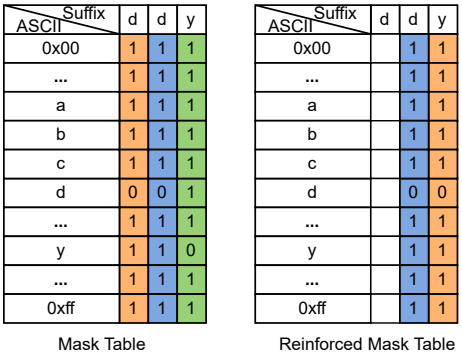


Fig. 14. Reinforced mask table is built by taking the necessary part of the original mask table. When performing the BIT-OR operations, load the reinforced masks of those last characters in each 16-byte region to form the reinforced vector join the BIT-OR together with previous shifted vectors.

Based on what we have illustrated above, it's easy to understand that this time the false positives are due to the lost bits of the second to last characters in each 16-byte region. Of course we can take the same method by building another reinforced mask table and loading the reinforced masks of those next to last characters in each 16-byte region. Theoretically, there are $\frac{m}{16}$ 16-byte boundaries in m input bytes. We take a boundary as $s_i s_{i+1} s_{i+2}$, with s_i and s_{i+1} in left region and s_{i+2} in right region. These boundaries can be classified into five categories:

- s_{i+2} doesn't match with any group. These boundaries won't produce false positives.
- s_{i+2} matches with certain group but s_{i+1} and s_i do not. These boundaries would produce false positives unless we load reinforced vectors of s_i and s_{i+1}.
- s_{i+2} and s_i match with certain group. These boundaries

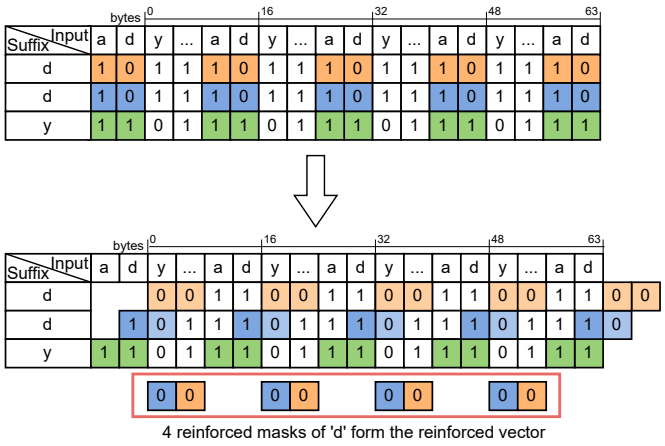


Fig. 15. Suppose *'ady'* appears at 16-byte boundaries with *'ad'* in the left 16-byte region and *'y'* in the right region. Although the lost bits of *'d'* have been made up, false positives still appear at offset 0, 16, 32 and 48 because filling in lost bits of *'d'* only guarantees *'dy'* to be matched properly. Therefore *'ady'* still matches wrongly with *'ddy'*.

would produce false positives unless we load reinforced vector of s_{i+1}.

- s_{i+2} and s_{i+1} match with certain group. These boundaries would produce false positives unless we load reinforced vector of s_i.
- All of s_{i+2}, s_{i+1} and s_i match with certain group. These boundaries are originally true positives.

Assuming the frequencies of occurrence for these five boundaries are denoted as f_k ($k \in [1,5]$), the number of loaded reinforced vectors is denoted as ξ and the number of false positives produced by *shift* instruction is denoted as N_{fp}, their relation can be represented as:

$$N_{fp} = \begin{cases} \frac{1}{16}m(f_2+f_3+f_4) & \xi = 0 \\ \frac{1}{16}m(f_2+f_4) & \xi = 1 \\ 0 & \xi = 2 \end{cases} \quad (1)$$

Besides, as we have mentioned in Section V-C, Teddy without reinforced mask would need 5σ SIMD instructions to process $\frac{L}{8}$ input bytes. For Teddy with reinforced mask, it would need $(5\sigma+2\xi)$ instructions per $\frac{L}{8}$ input bytes. Therefore, its instruction complexity is:

$$I(m) = \frac{40}{L}m\sigma + \frac{16}{L}m\xi \quad (2)$$

where L is SIMD vector length, m is input string length, σ is literal suffix length, ξ is the number of reinforced vectors and $I(m)$ represents the number of SIMD instructions needed to process m input bytes.

From equations 1 and 2, we know that increasing ξ would decrease the number of false positives, but it would also consume more CPU resources as it introduces extra SIMD instructions. Therefore, there may exist a best value of ξ. Through experiment, we find that Teddy will achieve the best performance with $\xi = 1$.

VII. Evaluation

A. Environment

We evaluated Teddy on commodity hardware and conducted experiments on a Linux server (Ubuntu 18.04) with Intel Xeon Gold 6364 (2.6GHz, dual sockets, 28 cores per socket) and 64GB DDR4 memory. Grouping algorithm and encoding method are implemented in C++, SIMD-based matching algorithm is implemented in C. We used GCC 7.5 to compile them.

B. Dataset

To evaluate the efficiency of Teddy, we conducted experiments with a variety of real-world DPI literal rules and network traffic. As shown in Table III, literals were collected from the rule sets of ModSecurity, which is a DPI-based Web Application Firewall (WAF). These rule sets range from small-scale ones (rule sets $1 \sim 11$), which contain less than 60 literals, to medium-scale ones that include hundreds or even more than a thousand literals. Input strings were collected from Alexa HTTP requests and IXIA HTTP responses. Also, we randomly generated an input string for evaluation. We present them in Table II.

TABLE II
Input String

Type	Size
Random	763 KB
IXIA HTTP Responses	121 MB
Alexa HTTP Requests	2.12 MB

TABLE III
Literal Rule Sets ($1 \sim 11$ are small-scale ones)

No.	Name	Dataset	n	Literal Type
1	Scan1	Scanners-headers	8	Scanners
2	Log2	Java-errors	10	Log and Error
3	Mali7	Scripting-user-agents	11	Malicious Code
4	Log1	IIS-errors	13	Log and Error
5	Craw	Crawlers-user-agent	16	Crawlers
6	Scan2	Scanners-urls	17	Scanners
7	Atta3	Restricted-upload	17	Malware Attack
8	Mali1	Java-code-leakages	17	Malicious Code
9	Mali6	PHP-variables	19	Malicious Code
10	Mali2	Java-classes	43	Malicious Code
11	Mali4	PHP-function-names	44	Malicious Code
12	Log4	SQL-errors	80	Log and Error
13	Scan3	Scanners-user-agents	87	Scanners
14	Mali8	UNIX-shell	115	Malicious Code
15	Atta2	Restricted-files	127	Malware Attack
16	Log3	PHP-errors	218	Log and Error
17	Mali9	Windows-Powershell	253	Malicious Code
18	Mali3	PHP-config-directives	276	Malicious Code
19	Atta1	Lfi-os-files	1,090	Malware Attack
20	Mali5	PHP-function-names	1,264	Malicious Code

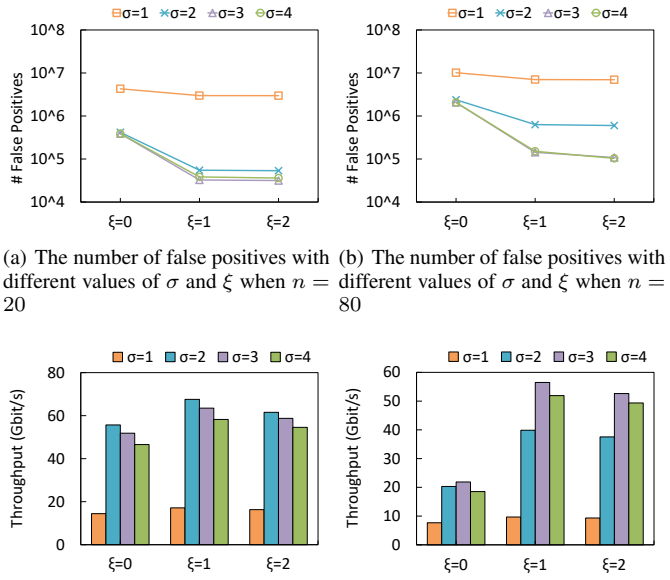
C. Find the optimal values for σ and ξ

We have mentioned that in order to deal with small-scale literal rule sets more efficiently, Teddy takes a shorter suffix for shift-or matching. We record the suffix length as σ and Teddy would need about $(5\sigma + 2\xi)$ SIMD operations (*load*,

shift and *or*) each iteration in shift-or matching process. This tells us that with shorter suffix (smaller σ), there would be less time spent on shift-or matching. However, shorter suffix also implies more false positives, which would then demand more time in exact matching to be filtered out. Therefore, a proper value of σ should be determined through experiments.

Also, we have talked about ξ, the number of reinforced vectors. Larger ξ would introduce less false positives but also need more *or* operations. In other words, larger ξ would reduce the time spent on exact matching but increase the time spent on shift-or matching. Therefore, we should also determine a proper value of ξ.

Hence, to find out the optimal values of σ and ξ, we carried out 2 groups of experiments with 20 and 80 literals respectively. These literals were chosen from ModSecurity rule sets. And we took Alexa HTTP Requests as input string. For each group of experiments, we increased σ from 1 to 4 and ξ from 0 to 2. We recorded the **number of false positives** and the overall **throughput**. We repeated each single experiment 1000 times and got the average result, as shown in Figure 16.



(a) The number of false positives with different values of σ and ξ when $n = 20$

(b) The number of false positives with different values of σ and ξ when $n = 80$

(c) Throughput with different values of σ and ξ when $n = 20$

(d) Throughput with different values of σ and ξ when $n = 80$

Fig. 16. The number of false positives and the throughput with different values of σ and ξ.

From Figure 16(c), we see that Teddy has best performance when $(\sigma, \xi) = (2, 1)$, its performance with reinforced mask is 21% better than without reinforced mask (i.e., $(\sigma, \xi) = (2, 0)$). From Figure 16(d), we see that Teddy has best performance when $(\sigma, \xi) = (3, 1)$, its performance with reinforced mask is 158% better than without reinforced mask (i.e., $(\sigma, \xi) = (3, 0)$). This result shows the **effectiveness of reinforced mask**.

According to our practical experience, we usually take $\boldsymbol{(\sigma, \xi) = (3, 1)}$. This can be attributed to the pattern observed in the number of false positives. From Figure 16(a) and 16(b), it is evident that there is a notable decrease in the number of false positives as ξ increases from 0 to 1. However, the reduction becomes less prominent as ξ increases further from 1 to 2. Similarly, when considering a fixed value of ξ, the reduction in false positives is apparent when σ increases from 1 to 3, but becomes subtle or even results in an increase when σ further increases from 3 to 4.

In the production environment, we consistently use $(\sigma, \xi) = (3, 1)$ as our chosen parameters. Hence, for the remaining evaluation, we will continue to utilize $\sigma = 3$ and $\xi = 1$ as our designated values.

D. The effectiveness of grouping strategy

To deal with small-scale literal rule sets more efficiently, Teddy matches the 3-byte-long suffixes during the process of shift-or matching. Also, it takes a new grouping strategy that is different from FDR's grouping to reduce false positives. To prove the effectiveness of Teddy's grouping strategy, we conducted two sets of experiments, with IXIA HTTP Responses as input string and ModSecurity rules in Table III as literal rule sets. One set of experiments is to let Teddy use its own grouping strategy, while the other set of experiments is to let Teddy use FDR's grouping strategy. For each single experiment, we counted the number of false positives, as shown in Figure 17.

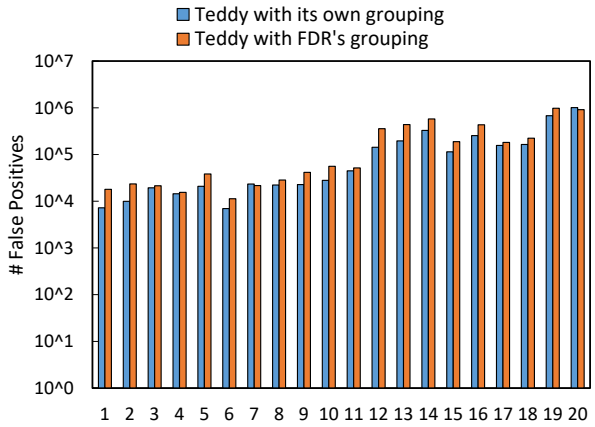


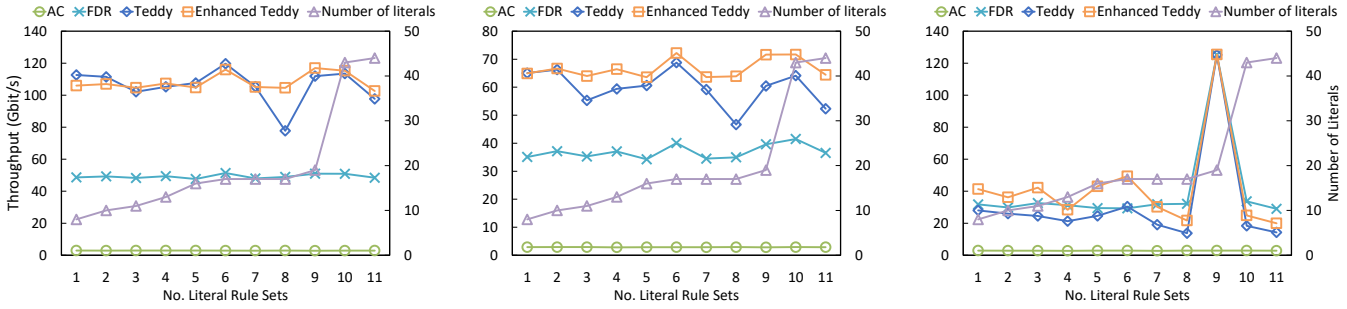
Fig. 17. The number of false positives with different grouping strategy.

We can see that in most cases Teddy's grouping strategy introduces less false positives than FDR's grouping strategy.

E. Overall performance compared with AC and FDR

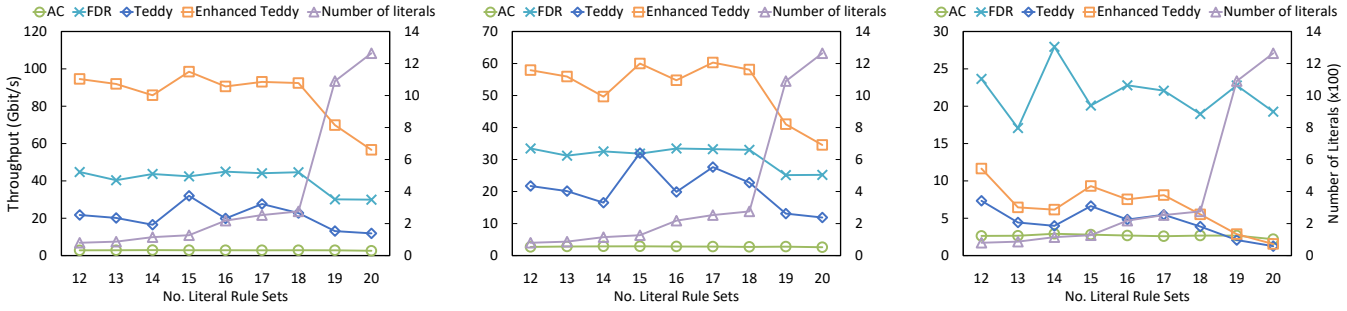
To compare the overall performance among AC, FDR and Teddy, we conducted three groups of experiments with random string, IXIA HTTP Responses and Alexa HTTP Requests as input strings respectively. We also tested the performance of Teddy without reinforced mask. To distinguish it from Teddy with reinforced mask, we refer to the latter as Enhanced Teddy. For each group of experiments, we took ModSecurity small-scale literal rule sets, those $1 \sim 11$ in Table III, as literal rules. The result is shown in Figure 18.

Although Teddy is designed to deal with small-scale literal rule sets, we also conducted another three groups of experiments to feed Teddy with medium-scale literal rule sets, those $12 \sim 20$ in Table III, as shown in Figure 19.



(a) Random (b) IXIA HTTP Responses (c) Alexa HTTP Requests

Fig. 18. Compare the performance of AC, FDR, and Teddy with ModSecurity small-scale literal rule sets ($1 \sim 11$ in Table III). In most cases Enhanced Teddy is faster than AC, FDR and Teddy. In (a), with random string as input, Enhanced Teddy's performance is up to 2.17x that of FDR, 34.32x that of AC and 1.34x that of Teddy. In (b), with IXIA HTTP Responses Enhanced Teddy's performance is up to 1.76x that of FDR, 21.24x that of AC and 1.37x that of Teddy. In (c), with Alexa HTTP Requests as input, Enhanced Teddy's performance is up to 1.68x that of FDR, 43.07x that of AC and 1.74x that of Teddy.



(a) Random (b) IXIA HTTP Responses (c) Alexa HTTP Requests

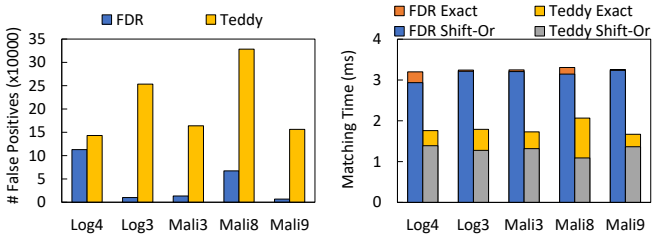
Fig. 19. Compare the performance of AC, FDR, and Teddy with ModSecurity medium-scale literal rule sets ($12 \sim 20$ in Table III). In many medium-scale cases, Teddy can still perform very well, as shown in (a) and (b). But in other cases, Teddy doesn't catch up with FDR, as shown in (c).

F. Detailed comparison between Teddy and FDR

Both Teddy and FDR would produce false positives during the process of shift-or matching. The number of false positives may depend on their grouping strategies and suffix length. Meanwhile, both of them have two matching stages, **shift-or matching** and **exact matching**. Shift-or matching is to find the potential matching positions and exact matching is to filter out the false positives. Teddy and FDR take different shift-or matching algorithms but share the same exact matching process.

Through the above experiments we observe that Teddy is much faster than FDR in most cases. To provide a clearer explanation for Teddy's high performance, we conducted further experiments comparing Teddy with FDR through more dimensions, including the number of false positives, shift-or matching time and exact matching time. We randomly chose 5 literal rule sets from Table III and took IXIA HTTP Responses as input strings for our experiments.

The **comparison of false positives** is shown in Figure 20(a). We observe that Teddy has much more false positives than FDR, which is reasonable because Teddy takes a 3-byte-long suffix for shift-or matching while FDR takes an 8-byte-long suffix. In the process of shift-or matching, FDR can guarantee the last 8 bytes of literals to be matched correctly but Teddy can only guarantee the last 3 bytes to be exactly matched.



(a) Number of False Positives (b) Shift-Or and Exact Matching Time

Fig. 20. Detailed comparison between Teddy and FDR.

The **comparison of shift-or matching and exact matching** is shown in Figure 20(b). We can observe that: i) Teddy spends more time on exact matching than FDR. This is because Teddy has more false positives that need to be filtered out in exact matching, as shown in Figure 20(a). ii) Teddy spends much less time on shift-or matching than FDR. This is because Teddy matches only the last 3 bytes of the literals in shift-or matching, while FDR matches the last 8 bytes. This experiment tells us that Teddy beats FDR by maintaining a more reasonable balance between shift-or matching and exact matching when confronted with small-scale literal rule sets.

VIII. Conclusion

In this paper, we have proposed Teddy as a solution to the scalability issue that the efficiency of FDR in small-scale literal rule sets is not satisfying. Compared with FDR, Teddy works much more efficiently when confronted with small-scale literal rule sets. Specifically, Teddy integrates the following features into its overall design: (1) Teddy proposes a suffix-based matching strategy to match a shorter suffix during the process of shift-or matching so that it can reduce the number of *shift* and *or* operations; (2) Teddy takes a new SIMD-based shift-or matching algorithm that can parallelly match up to 64 characters with only 15 SIMD operations; (3) Teddy finds an appropriate encoding method to implement the algorithm on real-world SIMD platforms; (4) Teddy reduces a lot of false positives and improves the overall performance significantly by introducing the reinforced mask. Our evaluation results show that Teddy can achieve up to 43.07x performance of AC, 2.16x performance of FDR. Comparing Teddy with state-of-the-art solutions, the results demonstrate the effectiveness of our engine. Teddy has been integrated into Hyperscan, together with which it has been widely deployed into predominant DPI applications such as Snort and Suricata.

References

[1] K. Qiu, H. Chang, Y. Hong, W. Zhu, X. Wang, and B. Li, "Teddy: An efficient simd-based literal matching engine for scalable deep packet inspection," in *Proc. 50th Int. Conf. Parallel Process.*, 2021.

[2] (2021) Snort intrusion detection system. [Online]. Available: https://snort.org

[3] (2021) Suricata open source ids. [Online]. Available: http://suricata-ids.org/

[4] Shorewall. [Online]. Available: http://shorewall.net/

[5] (2021) Modsecurity web application firewall. [Online]. Available: https://www.modsecurity.org/

[6] M. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: A highly-scalable software-based intrusion detection system," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2012, pp. 317–328.

[7] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep packet inspection as a service," in *Proc. ACM Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2014, pp. 271–282.

[8] T. Bujlow, V. Carela-Español, and P. Barlet-Ros, "Independent comparison of popular dpi tools for traffic classification," *Comput. Netw.*, vol. 76, pp. 75–89, 2015.

[9] nDPI. [Online]. Available: https://www.ntop.org/products/deep-packet-inspection/ndpi/

[10] R. T. El-Maghraby, N. M. Abd Elazim, and A. M. Bahaa-Eldin, "A survey on deep packet inspection," in *Int. Conf. Comput. Eng. Syst. (ICCES)*, 2017, pp. 188–197.

[11] Zeek. [Online]. Available: https://zeek.org/

[12] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. 2007 ACM CoNEXT Conf.*, 2007, pp. 1–12.

[13] M. Becchi and P. Crowley, "A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, Apr. 2013.

[14] X. Yu, B. Lin, and M. Becchi, "Revisiting state blow-up: Automatically building augmented-fa while preserving functional equivalence," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1822–1833, 2014.

[15] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. 2006 ACM/IEEE Proc. Symp. Archit. Netw. Commun. Syst.*, 2006, pp. 93–102.

[16] C. Clark and D. Schimmel, "Scalable pattern matching for high speed networks," in *Proc. 12th IEEE Symp. Field-Programmable Custom Comput. Mach.*, 2004, pp. 249–257.

[17] R. Sidhu and V. Prasanna, "Fast regular expression matching using fpgas," in *9th IEEE Symp. Field-Programmable Custom Comput. Mach. (FCCM'01)*, 2001, pp. 227–238.

[18] D. Sidler, Z. István, M. Owaida, and G. Alonso, "Accelerating pattern matching queries in hybrid cpu-fpga architectures," in *Proc. ACM Int. Conf. Manag. Data (SIGMOD)*, 2017, pp. 403–415.

[19] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "Infant: Nfa pattern matching on gpgpu devices," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 5, pp. 20–26, Oct. 2010.

[20] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," *SIGPLAN Not.*, vol. 50, no. 4, pp. 619–630, Mar. 2015.

[21] Z. Zhao, B. Wu, and X. Shen, "Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation," in *Proc. 19th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2014, pp. 543–558.

[22] Y.-H. Yang and V. Prasanna, "High-performance and compact architecture for regular expression matching on fpga," *IEEE Trans. Comput.*, vol. 61, no. 7, pp. 1013–1025, 2012.

[23] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character nfa," in *Proc. Int. Conf. Field Programmable Logic and Applications*, 2008, pp. 131–136.

[24] M. Avalle, F. Risso, and R. Sisto, "Scalable algorithms for nfa multi-striding and nfa-based deep packet inspection on gpus," *IEEE/ACM Trans. Netw.*, vol. 24, no. 3, pp. 1704–1717, 2016.

[25] E. Sadredini, D. Guo, C. Bo, R. Rahimi, K. Skadron, and H. Wang, "A scalable solution for rule-based part-of-speech tagging on novel hardware accelerators," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2018, p. 665–674.

[26] L. Vespa, N. Weng, and R. Ramaswamy, "Ms-dfa: Multiple-stride pattern matching for scalable deep packet inspection," *Comput. J.*, vol. 54, no. 2, pp. 285–303, 2011.

[27] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 86–98.

[28] Pcre: Perl compatible regular expressions. [Online]. Available: https://www.pcre.org/

[29] Google RE2. [Online]. Available: https://github.com/google/re2/

[30] J. Qiu, Z. Zhao, and B. Ren, "Microspec: Speculation-centric fine-grained parallelization for fsm computations," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2016, pp. 221–233.

[31] S. Wang, C. Liu, Y. Liu, G. Li, M. Zhang, Y. Wang, and M. Xu, "Fast multi-string pattern matching using pisa," in *Proc. 15th Int. Conf. Emerg. Netw. EXp. Technol. (CoNEXT)*, 2019, pp. 74–75.

[32] C. Xu, S. Chen, J. Su, S. M. Yiu, and L. C. K. Hui, "A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms," *IEEE Commun. Surv. Tutor.*, vol. 18, no. 4, pp. 2991–3029, 2016.

[33] C. E. Graves, W. Ma, X. Sheng, B. Buchanan, L. Zheng, S.-T. Lam, X. Li, S. R. Chalamalasetti, L. Kiyama, M. Foltin, J. P. Strachan, and M. P. Hardy, "Regular expression matching with memristor tcams for network security," in *Proc. 14th IEEE/ACM Int. Symp. Nanoscale Archit.*, 2018, p. 65–71.

[34] Capacity Planning for Snort IDS: Bilbous, Not Tapered. [Online]. Available: https://mikelococo.com/2011/08/snort-capacity-planning/

[35] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, "Operational experiences with high-volume network intrusion detection," in *Proc. 11th ACM Conf. Comput. Commun. Secur.*, 2004, pp. 2–11.

[36] "IEEE Standard for Ethernet - Amendment 10: Media Access Control Parameters, Physical Layers, and Management Parameters for 200 Gb/s and 400 Gb/s Operation," *IEEE Std. 802.3bs-2017 (Amendment to IEEE 802.3-2015 as amended by IEEE's 802.3bw-2015, 802.3by-2016, 802.3bq-2016, 802.3bp-2016, 802.3br-2016, 802.3bn-2016, 802.3bz-2016, 802.3bu-2016, 802.3bv-2017, and IEEE 802.3-2015/Cor1-2017)*, pp. 1–372, 2017.

[37] The new world of 400 gbps ethernet. [Online]. Available: https://www.accton.com/Technology-Brief/the-new-world-of-400-gbps-ethernet/

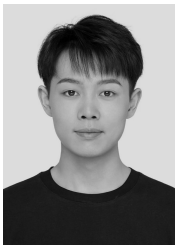
[38] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern cpus," in *Proc. NSDI*, 2019, pp. 631–648.

[39] T. Qiu, X. Yang, and B. Wang, "Filtering techniques for regular expression matching in strings," in *Database Systems for Advanced Applications*, C. Liu, L. Zou, and J. Li, Eds. Springer International Publishing, 2018, pp. 118–122.

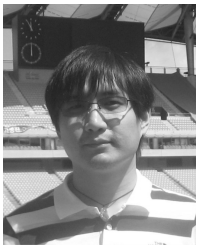
[40] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen, "Splitscreen: Enabling efficient, distributed malware detection," *J. Commun. Netw.*, vol. 13, no. 2, pp. 187–200, 2011.

[41] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "Dfc: Accelerating string pattern matching for network applications," in *Proc. NSDI*, 2016, pp. 551–565.

[42] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," in *Proc. 4th Int. Workshop Softw. Perform.*, 2004, pp. 207–215.
[43] P.-c. Lin, Z.-x. Li, Y.-d. Lin, Y.-c. Lai, and F. C. Lin, "Profiling and accelerating string matching algorithms in three network content security applications," *IEEE Commun. Surveys Tuts.*, vol. 8, no. 2, pp. 24–37, 2006.
[44] Scaling cloudflare's massive waf. [Online]. Available: http://www.scalescale.com/scaling-cloudflaresmassive-waf/
[45] H. Xu, H. Chang, W. Zhu, Y. Hong, G. Langdale, K. Qiu, and J. Zhao, "Harry: A scalable simd-based multi-literal pattern matching engine for deep packet inspection," in *Proc. IEEE Int. Conf. Comput. Commun.*, 2023.
[46] R. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35, no. 10, pp. 74–82, 1992.
[47] K. Fredriksson, "Shift-or string matching with super-alphabets," *Information Processing Letters*, vol. 87, no. 4, pp. 201–204, 2003.
[48] S.-I. Oh, I. Lee, and M. S. Kim, "Fast filtering for intrusion detection systems with the shift-or algorithm," in *Proc. Asia-Pacific Conf. Commun. (APCC)*, 2012, pp. 869–870.
[49] (2021) Modsecurity rule sets. [Online]. Available: https://owasp.org/www-project-modsecurity-core-rule-set/
[50] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, Jun. 1975.
[51] B. Commentz-Walter, "A string matching algorithm fast on the average," in *Int. Colloq. Automat. Lang. Prog.*, 1979, pp. 118–132.
[52] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, no. 2, pp. 249–260, 1987.
[53] M. Régnier, "Knuth-morris-pratt algorithm: an analysis," in *Int. Symp. Math. Found. Comput. Sci.*, 1989, pp. 431–444.



Hao Xu received his bachelor's degree in computer science from Northeastern University, Shenyang, China, in 2019. He is currently working toward the master's degree with the the School of Computer Science, Fudan University, Shanghai, China. His research interests include software-defined networking and deep packet inspection.



Harry Chang received his B.Eng. Degree from Shanghai Jiao Tong University in 2006. He now works for Intel as a senior software engineer. His research/work interests include computer network and performance optimization on Intel architecture.

Kun Qiu received his B.Sc. Degree from Fudan University in 2013, and received his Ph.D. Degree from Fudan University in 2019. He works for Intel as a software engineer from 2019 to 2023. Now he joined Fudan University in 2023 as an assistant professor. His research interests include computer networks and computer architecture. He is a member of IEEE, ACM and CCF.

Yang Hong received his B.S. Degree from Nanjing University in 2014, and received the M.E. Degree from Shanghai Jiao Tong University in 2017. He now works for Intel as a software engineer. His work interests include SIMD algorithm optimization and pattern matching.

Wenjun Zhu received his master's Degree from Nanjing University of Posts and Telecommunications in 2020. He works for Intel as a software engineer from 2020 to 2023. Now he joined Fudan University in 2023 as a software engineer. His research interests include computer networks and computer architecture.

Xiang Wang received his master's degree of electrical engineering from University of Missouri – Columbia, USA. He is currently a solution architect at Intel. His research interests include regular expression matching algorithms and network security systems.

Baoqian Li received his B.S. Degree from Tongji University in 2004, and received his Master in Software Engineering Degree from Fudan University in 2012. He now works for Intel as a software engineering manager. His research/work interests include computer network, cloudnative and performance optimization on IA.

Jin Zhao received the B.Eng. degree in computer communications from Nanjing University of Posts and Telecommunications, China, in 2001, and the Ph.D. degree in computer science from Nanjing University, China, in 2006. He joined Fudan University in 2006. His research interests include software defined networking and distributed machine learning. He is a senior member of IEEE.