Harry: A Scalable SIMD-based Multi-literal Pattern Matching Engine for Deep Packet Inspection

Hao Xu[*†], Harry Chang[‡], Wenjun Zhu[‡], Yang Hong[‡], Geoff Langdale[‡], Kun Qiu[‡] and Jin Zhao[*†]
[*]School of Computer Science, Fudan University, China
[†]Shanghai Key Laboratory of Intelligent Information Processing, Shanghai, China
[‡]Intel Asia-Pacific Research & Development Ltd., Shanghai, China
xuhao21@m.fudan.edu.cn, jzhao@fudan.edu.cn, {harry.chang, wenjun.zhu, hong.a.yang, geoff.langdale, kun.qiu}@intel.com

***Abstract*—Deep Packet Inspection (DPI) is a significant network security technique. It examines traffic workloads by searching for specific rules. Since every byte of packets needs to be examined by many literal rules, multi-literal matching becomes the performance bottleneck of DPI. FDR, the fastest multi-literal matching engine on CPUs, takes advantage of Single-Instruction-Multiple-Data (SIMD) to alleviate this bottleneck and achieves a performance boost over the widely-used Aho-Corasick (AC) algorithm. However, FDR does not deeply exploit the data-level parallelism of SIMD and its SIMD vector utilization is only 50%. Besides, limited by certain SIMD shift instructions, it cannot benefit from advanced SIMD instruction sets. To overcome these issues, we propose Harry, a scalable and SIMD-based multi-literal matching engine. Harry adopts a column-vector-based matching algorithm to improve the data-level parallelism and SIMD vector utilization. To support the algorithm, it takes two encoding methods to compress the mask table. Also, it utilizes shuffle instruction to implement shift. We implement Harry on commodity CPU and evaluate it with real network traffic and DPI rules. Our evaluation shows that Harry reaches a throughput of 30∼70Gbit/s, up to 52x that of AC and 2.09x of FDR. It has been successfully deployed in Hyperscan.**

***Index Terms*—DPI, literal matching, SIMD, network security**

I. Introduction

Deep Packet Inspection (DPI) [1]–[3] is one of the fundamental techniques for many network security applications including Intrusion Detection/Prevention System (IDS/IPS) [4]–[7], Web Application Firewall (WAF) [8], [9] and application identification system [10]. Different from the traditional packet inspection that examines only the fixed 5 tuples in packet headers, DPI analyzes the content of the packets by searching for specific rules (i.e., patterns or signatures), which are usually regular expressions. Therefore, regular expression matching has been the core of DPI.

Nowadays, the network bandwidth is dramatically increasing [11]–[14] and many DPI-based applications demand real-time stream processing, which motivates researchers to improve the regex (Regular Expression) matching efficiency. To speed up regex matching, including pure string matching, there is a trend toward using accelerators, such as GPU/NPU/FPGA [7], [15]–[24] and programmable switching ASICs [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.

There have been plenty of optimized regex matching algorithms [19], [20], [26]–[31]. However, despite continuous efforts, their performance is still not satisfying [7], [32]. It has been reported that PCRE [28], a popular regex engine, and RE2 [29], Google's regex engine, need 12,800 and 1,116 seconds respectively to perform 1GB traffic matching, while this traffic has only 818,682 packets [33].

On the contrary, taking multi-literal matching (also known as multi-string matching) as a pre-filtering method has been proven to be the best practice [34], [35], because literal matching is two orders of magnitude outperforming regex matching [36]. Most DPI-based applications, such as Snort [4] and Suricata [5], match packets with multiple literal rules before the expensive regex matching. The well-known Hyperscan [33], currently the fastest regex matcher, decomposes regular expressions into literal and subregex components, and performs multi-literal matching first to avoid unnecessary regex matching, by which it needs only 133 seconds to complete the 1GB traffic matching. Therefore, multi-literal matching has been a fundamental building block, also the performance bottleneck, of DPI systems [32], [37], [38].

There are a significant number of multi-literal matching algorithms [33], [36], [39]–[41], among which the classic Aho-Corasick (AC) algorithm [39] is used by many DPI applications such as Snort [4], Suricata [5], ModSecurity [8] and CloudFlare's WAF [32]. Besides, FDR [33], a bitwise-based and SIMD-accelerated state-of-the-art multi-literal matching engine, is much faster than AC and has been successfully deployed by over 40 commercial projects, integrated into 37 open-source projects and in production used by tens of thousands of cloud servers in data centers.

However, FDR has three issues. First, it does not deeply exploit the data-level parallelism of SIMD. Second, its SIMD vector utilization is relatively low, only 50%. Third, it is limited by certain SIMD shift instructions and cannot be implemented with more advanced SIMD instruction sets. In prior work, Teddy [40] was presented to overcome the first two issues, but its strategy has determined that it is only effective for small-scale literal rule sets. When faced with large-scale

ones, with the number of literals more than 60 and the literal length more than 8 bytes, Teddy's performance would degrade rapidly. Therefore Teddy cannot deal with large and complex rule sets, which are very common in today's DPI systems (e.g., the community rule set of Snort v3.0 [42] has 4024 rules and OWASP ModSecurity Core Rule Set v3.3.2 has 3725 rules).

In order to overcome FDR's issues and at the same time be scalable with rule sets, in this paper, we propose Harry, a scalable SIMD-based multi-literal matching engine for DPI. Harry adopts a column-vector-based matching algorithm to improve SIMD vector utilization and data-level parallelism. To support the algorithm, it takes two encoding methods to compress the mask table. Also, it utilizes VPERMB, the shuffle instruction, to implement shift operation so that it can be implemented with more advanced SIMD instruction sets. Harry can reach a throughput of 30∼70 Gbit/s, up to 52x that of AC and 2.09x of FDR, with the number of literal rules ranging from dozens, hundreds to thousands, on real network traffic workloads. It has been successfully deployed in Hyperscan [33]. Briefly speaking, this paper makes the following contributions:

- We propose a column-vector-based matching algorithm that improves SIMD vector utilization and data-level parallelism. Compared with FDR who needs 3 SIMD operations per input character, Harry needs only 0.43∼0.71 SIMD operation per input character. Also, the SIMD vector utilization has increased from 50% to 87.5%. Besides, to support the algorithm, we design two encoding methods to compress the mask table.
- We utilize VPERMB (the shuffle instruction) to implement shift operation to overcome the shortcomings of VPSLLDQ (the shift instruction) so that Harry can be implemented in AVX512, the most advanced SIMD instruction set.
- We implement Harry, FDR, and AC on commodity CPU with AVX512 SIMD support. We conduct experiments to compare Harry with FDR and AC.
- We integrate Harry into Hyperscan. Harry has been adopted by Hyperscan and is now working at the pre-filtering stage.

The rest of the paper is organized as follows: Section II introduces the background and our motivations. Section III gives the overview design of Harry. Section IV explains the column-vector-based matching algorithm. Section V introduces the two encoding methods. Section VI gives the implementation details of shift operation. Section VII evaluates Harry and analyzes the evaluation results. Finally, Section VIII concludes.

II. Background and Motivation

A. Shift-Or Algorithm

FDR is an extension of the classic Shift-Or algorithm. So before introducing FDR, we would like to explain the principle of Shift-Or [43], [44], which is critical for understanding the issues of FDR and the innovations of Harry.

Shift-Or is a bitwise-based literal matching algorithm. It converts literal matching into bitwise SHIFT and OR operations and supports matching multiple literals simultaneously. We would first explain the principle of single-literal Shift-Or and then multi-literal Shift-Or. We give the relevant notations in Table I.

TABLE I
Definition of Notations

Symbol	Description
Λ	ASCII character set
s	Input string
t	The t-th iteration
m	The number of characters processed in an iteration
T	Mask table
M	Match table of the t-th iteration
r	State mask of the t-th iteration
r'	State mask of the $(t-1)$-th iteration
w	The literal(single-literal matching)
w_i	The i-th literal(multi-literal matching)
n	The number of literals(multi-literal matching)
l	The literal length(single-literal matching) or maximum literal length(multi-literal matching)

1) Single-literal Shift-Or Matching: Suppose w is *'rry'*. Before matching, it needs to build a mask table, as shown in Fig. 1(A). The mask table consists of 256 rows and 3 columns. Each row vector is the mask of a certain ASCII character. We mark this table as T and $T[c]$ is the mask of c. Bit $T[c][i](i \in [0, 3))$ indicates whether or not the i-th character of *'rry'* is c. For example, bit 0 in the green cell ($T['r'][0]$) indicates that the first character of *'rry'* is *'r'*, bit 1 in the yellow cell ($T['y'][1]$) indicates that the second character of *'rry'* is not *'y'*. The mask table can be defined as:

$$T[c][i] = \begin{cases} 0, & \text{if } c = w[i] \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

where $c \in \Lambda, i \in [0, l)$.

After building the mask table, it begins to handle the input string. It processes a chunk of input characters at a time, which we call an iteration. For each input character c in an iteration, it loads $T[c]$ into the match table. Suppose it processes 4 characters in an iteration and the input string is *'rsyrry'*, with *'rs'* in previous iteration and *'yrry'* in current iteration, we show the match table in Fig. 1(B). We mark the match table of current iteration as M and the number of characters processed in an iteration as m.

In match table, 3 bits arranged on a diagonal line can tell us the match result of w. If they are all 0 bits, then it is a

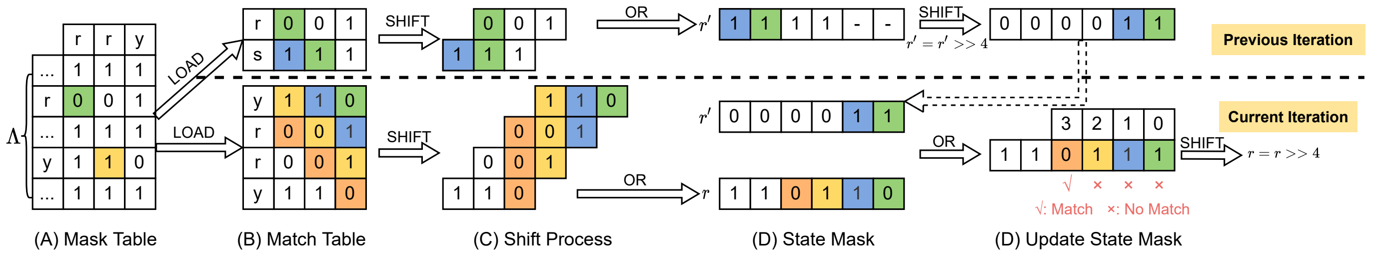


Fig. 1. Shift-Or Process of Single Literal Matching. In each iteration, the operation sequence is LOAD→SHIFT→OR→SHIFT.

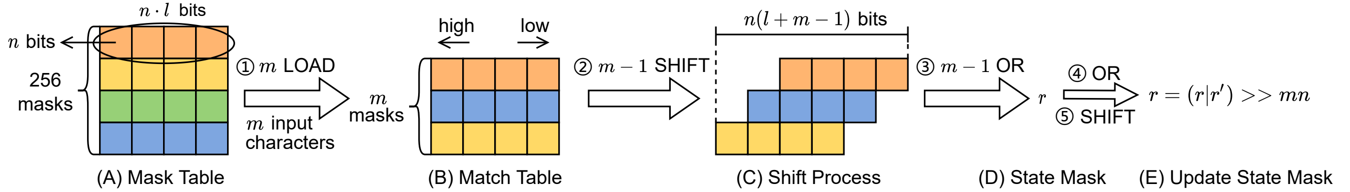


Fig. 2. Row-vector-based Shift-Or Model. Suppose there are n literals to be matched, whose maximum length is l, and m input characters to be processed in an iteration. Multi-literal shift-or matching has the same process as single-literal matching. Difference is that a cell in its tables contains n bits, each for a single literal. So its mask vector has nl bits and its shift step is n bits.

successful match, else no match. For example in Fig. 1(B), bits in the 3 orange cells indicate a match (*'rry'* is matched at index 3 of *'yrry'*), bits in the 3 yellow cells indicate no match (*'rry'* is not matched at index 2 of *'yrry'*).

To get the match results of w, Shift-Or algorithm shifts $M[i]$ left i bits to align the bits arranged on a diagonal line. As shown in Fig. 1(C), in current iteration, the 4 vectors of M are shifted left 0, 1, 2, 3 bits respectively, and in previous iteration, the vectors are shifted the same way. Then it performs OR operations on the shifted vectors to get the state mask. As shown in Fig. 1(D), the state mask of current iteration is r, whose bits are from the original match table and can reflect the match results of w. Besides, to guarantee a continuous matching at iteration boundaries (from previous iteration to current iteration), r should be updated with $r = r|r'$, where r' is the state mask of the previous iteration and has been shifted by $r' = r' >> 4$. After $r = r|r'$, $r[i]$ tells the match result of *'rry'* at index i, as shown in Fig. 1(D). Finally, r should also be shifted by $r = r >> 4$ and passed to the next iteration.

2) Multi-literal Shift-Or Matching: The multi-literal Shift-Or matching is much similar to single-literal matching except that a cell in mask table and match table has n bits, each for a certain literal. Suppose there are 2 literals, w_0 = *'rry'*, w_1 = *'yrr'*, and the input characters of current iteration is still *'yrry'*. The mask table and match table are shown in Fig. 3.

The mask table can be defined as:

$$T[c][i][j] = \begin{cases} 0, & \text{if } c = w_j[i] \\ 1, & \text{otherwise} \end{cases} \tag{2}$$

where $c \in \Lambda, i \in [0, l), j \in [0, n)$. $T[c][i][j]$ refers to the j-th bit in cell (c, i) and $T[c]$ still represents a one-dimensional mask vector. The definition of match table does not change, and neither does the shift-or process, except that the shift step

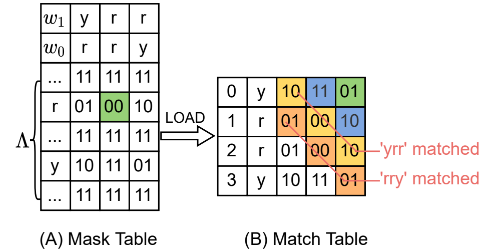


Fig. 3. Mask Table and Match Table of Two-literal Shift-Or. Each cell has two bits, the first for w_0 and the second for w_1. For example in the green cell of the mask table, the left bit 0 indicates that *'r'* is the second character of w_0 and the right bit 0 indicates that *'r'* is the second character of w_1. In match table, the 3 left bits in the orange cells indicate that w_0 is matched at index 3 of *'yrry'* and the 3 right bits in the yellow cells indicate that w_1 is matched at index 2 of *'yrry'*.

has changed from 1 bit to n bits. We abstract the shift-or model of multi-literal matching in Fig 2 and we call it row-vector-based shift-or model.

B. FDR

FDR takes the above row-vector-based shift-or model and utilizes SIMD to accelerate the matching process. Suppose the SIMD vector length is L. The shift-or process should be within an SIMD vector, so according to Fig. 2(C) we have:

$$n(m + l - 1) < L \tag{3}$$

Based on practical experience, FDR sets $n = 8$ and $l = 8$ to constraint a mask to be of 64 bits, so for FDR it is:

$$8(m + 7) < L \tag{4}$$

If there are more than 8 literals or if a literal has more than 8 characters, FDR will take the following strategies.

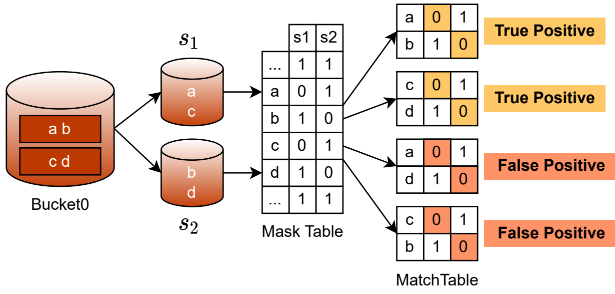


Fig. 4. Grouping. Group w_0 = *'ab'* and w_1 = *'cd'* into bucket 0. Then assemble 2 character sets $s_1 = \{w_0[0], w_1[0]\}$ and $s_2 = \{w_0[1], w_1[1]\}$. The mask table is defined as: If $c \in s_i$, then $T[c][i][0] = 0$, else $T[c][i][0] = 1$. We can see that each bucket occupies just 1 bit in a cell. The following shift-or process is the same as what we have illustrated. After grouping, a match will hit a certain bucket rather than an exact literal.

1) Grouping: If there are more than 8 literals, FDR will group them into 8 buckets and try to match these buckets with the input string. The grouping principle is shown in Fig. 4. Formally, suppose each bucket has q_0, q_1, q_2, q_3,…,q_7 literals and w_0^v, w_1^v, …, $w_{q_v-1}^v$ are literals in bucket v. We define character set $\lambda_k^v = \bigcup_{i=0}^{q_v-1} w_i^v[k]$ and mask table T as:

$$T[c][k][v] = \begin{cases} 0, & \text{if } c \in \lambda_k^v \\ 1, & \text{otherwise} \end{cases} \tag{5}$$

where $c \in \Lambda, k \in [0, 8), v \in [0, 8)$. The grouping strategy is in fact a trade-off between space and accuracy. It achieves matching a large number of literals but drops the accuracy. For example, in Fig. 4, after grouping *'ab'* and *'cd'* into one bucket, all of *'ab'*, *'cd'*, *'ad'* and *'cb'* can be recognized as positive matches, among which *'ad'* and *'cb'* are actually false positives.

2) Truncation: If a literal is longer than 8 bytes, FDR will only match its 8-byte-long prefix. Of course, this can also introduce false positives.

3) Exact Matching: Based on what we have illustrated above, even if a certain bucket is matched, FDR still needs to match the literals in it one by one. On one hand, FDR needs to find the exact matched literal. On the other hand, it can distinguish the true positives from the false positives during this process. We call this step exact matching.

C. Problems of FDR & Motivations

FDR takes the row-vector-based shift-or model. In fact, this model has several issues.

1) Not Deeply Exploited Data-level Parallelism: According to condition 4, the larger L is, the larger m can be, which means that with wider SIMD instruction (larger L), FDR can process more characters in an iteration (larger m). Larger m can enhance the instruction-level parallelism, because multiple LOAD and SHIFT operations are independent of each other and can be performed in parallel, according to (A)→(B)→(C) in Fig. 2. However, the data-level parallelism cannot be improved with larger m, because no matter how many characters are processed, FDR always needs $3m$ SIMD operations to deal with m characters, i.e., 3 operations per character, as shown in Fig. 2. Due to the poor data-level parallelism, FDR can benefit little from advanced SIMD instruction sets that have wider SIMD instructions (larger L). Implementing FDR in AVX512 with 512-bit SIMD vectors cannot make much progress than implementing it in SSSE3 with 128-bit SIMD vectors.

2) Low SIMD Vector Utilization: On one hand, FDR's mask has fixedly 64 bits. On the other hand, it operates by masks (i.e., row vectors). Therefore with SSSE3, the SIMD vector utilization is only 50%, and it's even lower with other SIMD instruction sets.

3) Defects of the Shift Instruction: In SSSE3, the shift instruction is *PSLLDQ* and it works well. However, in AVX512, the shift instruction VPSLLDQ has a shortcoming that it cannot shift bits across 128-bit boundaries. This shortcoming has no effect if shift is not needed. Unfortunately, FDR needs a lot of shift operations, most of which have a long shift distance, which implies that FDR would lose plenty of bits if implemented with AVX512. These lost bits would lead to an unacceptable number of false positives that have a great influence on FDR's performance.

These issues motivate us to design Harry with higher SIMD vector utilization and data-level parallelism. More importantly, Harry should avoid the defects of VPSLLDQ and be compatible with AVX512, an advanced SIMD instruction set.

III. Harry Architecture

The overview design of Harry is shown in Fig. 5. Next, We would briefly introduce its core components.

A. Column-vector-based Matching Algorithm

In order to find the match positions, FDR performs LOAD, SHIFT and OR operations on masks, or called row vectors. Performing on row vectors has incurred that the number of SIMD operations relies on the number of input characters, as shown in Fig. 2. Actually, we can also perform on column vectors because shifting by column has the same effect to align the diagonal bits. More importantly, it provides us with the opportunity to decrease SIMD operations. So we have proposed a column-vector-based matching algorithm for Harry that needs only 0.43∼0.71 SIMD operation per input character, with much higher data-level parallelism than FDR who needs 3 operations per character.

B. The Encoding Methods

Although the column-vector-based matching algorithm is more efficient, implementing it on modern CPU is not that easy, because it needs 2048-bit-long SIMD vector to hold the column vector of the mask table, while the longest SIMD vector of modern CPU has only 512 bits (CPU with AVX512 SIMD support). The hardware limitation moves us to design new encoding methods to compress the mask table, so that we can use 512-bit-long SIMD vector to implement the matching algorithm.

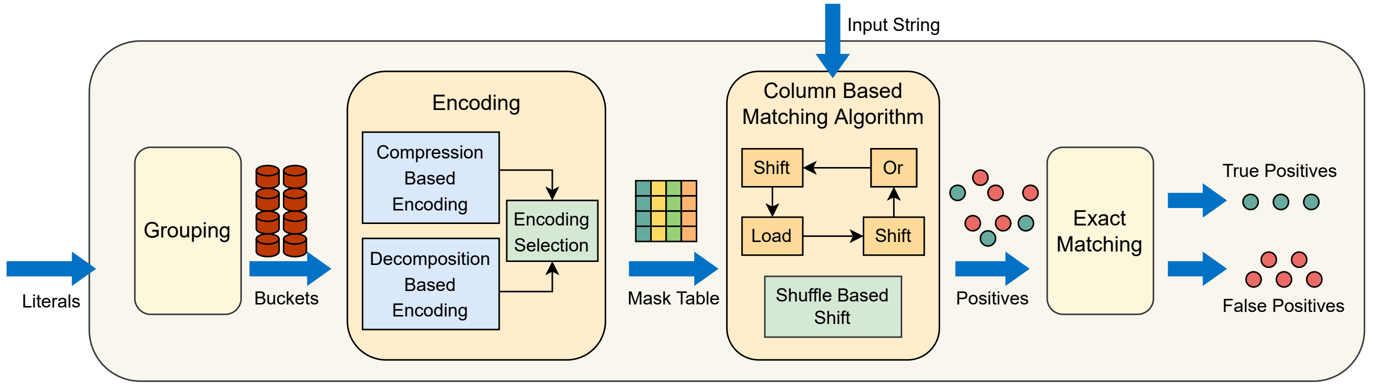


Fig. 5. The Architecture of Harry. Like FDR, Harry groups literals into 8 buckets and builds the mask table before matching. During the matching process, it can also produce false positives and needs exact matching to drop them. Different from FDR, it takes a completely new column-vector-based matching algorithm and designs two encoding methods to support the algorithm. Besides, in order to overcome the issues of VPSLLDQ (the shift instruction in AVX512), it implements shift operation by VPERMB, a shuffle instruction that is originally used to reorder the elements in a certain vector.

C. The Shift Implementation

VPSLLDQ, the shift instruction in AVX512, has a shortcoming that it cannot shift bits across 128-bit boundaries. This has a huge impact on Harry, because Harry's matching algorithm is based on plenty of shift operations. In order to deploy Harry on CPU with AVX512 SIMD, we replace VPSLLDQ with VPERMB, a shuffle instruction that is originally used to reorder the elements in a vector.

IV. Column-vector-based Matching Algorithm

The core of our column-vector-based matching algorithm is its shift-or model as shown in Fig. 6. It performs LOAD, SHIFT, and OR operations all on column vectors. We can see that this model has the following advantages:

- The number of SIMD operations (LOAD, SHIFT, and OR) relies on l, the literal length, rather than m, the number of characters processed in an iteration. This implies that no matter how many characters are processed in an iteration, Harry always needs $3l$ SIMD operations. In practice, Harry takes the same grouping and truncation strategies as FDR, with n=8 and l=8, so it needs only 24 operations to process m characters. Theoretically, the larger m is, the higher Harry's performance is.
- The column vector length of match table is mn and for Harry it is $8m$. Unlike FDR whose row vector (i.e., the mask) length is fixedly 64 bits, Harry can adjust its column vector length by changing m. The larger m is, the higher the SIMD vector utilization is and the more characters it can process per 24 SIMD operations. Suppose the SIMD vector has L bits, and from Fig. 6(C) we know that the $(l-1)$ SHIFT operations require a SIMD vector to be able to hold $n(m+l-1)$ bits, which is:

$$n(m+l-1) < L \tag{6}$$

And for Harry it is:

$$8(m+7) < L \tag{7}$$

Therefore in AVX512 where L = 512, Harry takes m = 56, which demonstrates that it needs only 24 SIMD operations per 56 input characters, i.e., 0.41 SIMD operation per input character, and the SIMD vector utilization is $\frac{mn}{L}$ = 87.5%, while FDR needs 3 SIMD operations per character with SIMD vector utilization being $\frac{nl}{L}$ = 12.5%.

V. The Encoding Method

A. Problem Analysis

We have analyzed that Harry takes $n = 8$ and $l = 8$, and with AVX512 SIMD support, it can process 56 characters in an iteration. As shown in Fig. 6(A)→(B), for Harry, given

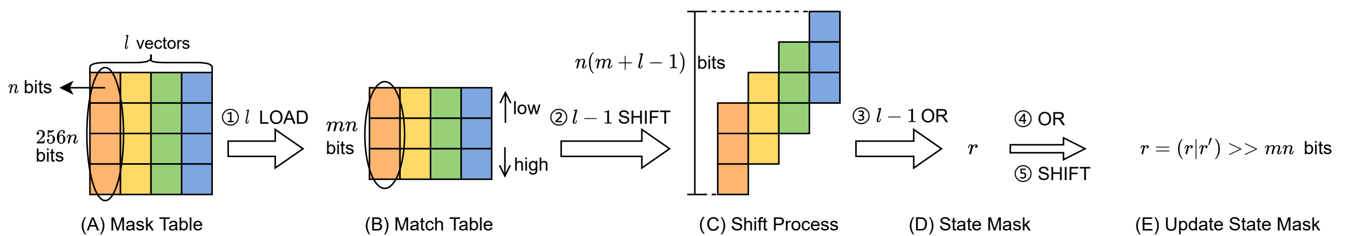


Fig. 6. Column-vector-based Shift-Or Model. Suppose there are n literals to be matched, whose maximum length is l, and m characters to be processed in an iteration. This model performs LOAD, SHIFT, and OR operations on column vectors rather than row vectors. It should be noticed that a column vector of the mask table contains $256n$ bits and that of match table contains mn bits. Besides, compared with the row-vector-based shift-or model, it needs $3l$, rather than $3m$, SIMD operations per m input characters.

56 characters, it loads 8 column vectors from the mask table, each containing 56 8-bit integers, and stores them in 8 SIMD vectors for shifting. This step is accomplished by VPERMB, the shuffle instruction in AVX512. We show VPERMB in Fig. 7(A) and the load step of Harry in Fig. 7(B). VPERMB is right the target SIMD instruction that is suitable for implementing Harry's load step. However, there is a problem that the source vector of VPERMB, which has 64 8-bit integers, cannot hold the column vector of mask table, which has 256 8-bit integers. This is the main difficulty to implement the column-vector-based algorithm. We need to find proper methods to compress the mask table.

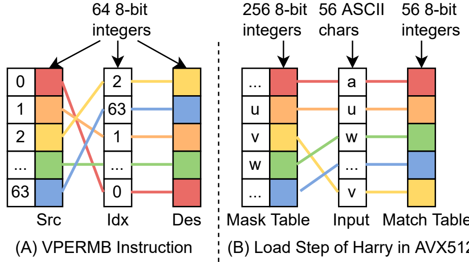


Fig. 7. VPERMB and Harry Load Step. In (A), VPERMB shuffles 8-bit integers in *Src* across lanes using the corresponding indices in *Idx*, and stores the result in *Des*. In (B), taking 56 input characters as indices, Harry picks 56 8-bit integers from a mask table column that includes 256 8-bit integers. In AVX512, Harry would store the 56 integers as the low 56 integers of an SIMD vector (an AVX512 SIMD vector can hold 64 integers), with the higher space left for shifting.

B. The Compression-based Encoding Method

We have several ways to simplify the mask table. Usually, the input string contains only the commonly used ASCII characters, which are 0x00∼0x7f. Therefore, we can compress the mask table to be of 128 rows. Even more, if only consider the English characters which are 0x40∼0x7f, we can further compress the mask table to be of 64 rows. So a column vector has only 64×8 = 512 bits, which is right inside an AVX512 SIMD vector. Here, we compress the mask table by compressing the character set, from the whole ASCII character set to English character set. We call it compression-based encoding. And because the low 6 bits of English characters (0x40∼0x7f) are 000000∼111111, we can load elements from the mask table by the low 6 bits of the input characters, as shown in Fig. 8. We call Harry with this encoding method as Harry6b.

C. The Decomposition-based Encoding Method

1) Decompose FDR's Encoding Bits: Before introducing our decomposition-based encoding method, we would first explain FDR's encoding. To reduce the false positives caused by grouping, FDR encodes its mask table a with super character set. It uses 12 bits to represent a single character, with the lower 8 bits being the character's 8 ASCII bits and the higher 4 bits being the next character's low-level 4 bits. For example, as *'a'* = 01100001 and *'d'* = 01100100, if the input string is

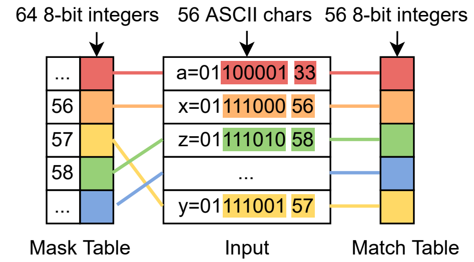


Fig. 8. Harry6b Load Step. After compressing, the mask table has only 64 rows and its column vector has 64 8-bit integers. For the 56 input characters *'axz...y'*, take their low 6 bits as indices to pick elements from the column vector of the mask table.

'ad', then the encoding of *'a'* would be 010001100001. Rather than compressing the mask table, FDR enlarges its mask table from 256 rows to 4096 rows. This can significantly reduce the false positives as it introduces more information to the mask table (the detailed reason can be found in the paper of Hyperscan [33]).

If Harry takes FDR's encoding, the column vector of mask table would contain 4096×8 bits, which is far beyond what a SIMD vector can hold. So we try to compress the mask table by decomposing the 12 bits into high 6 bits and low 6 bits. The mask table is changed as shown in Fig. 9. After decomposition, the column vector just contains 64×8 = 512 bits. We call Harry with this decomposition-based encoding

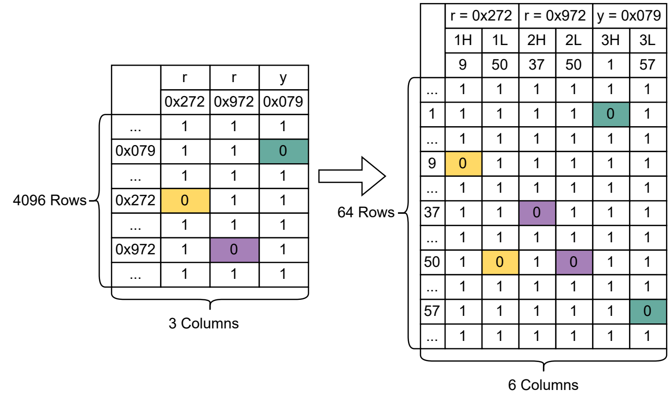


Fig. 9. Mask Table Change. Suppose the literal is *'rry'*. The left table is the mask table before decomposing. In the original ASCII character set, *'r'* is 0x72 and *'y'* is 0x79. According to FDR's super character set, *'r'*, *'r'*, and *'y'* of *'rry'* is encoded as 0x272, 0x972, and 0x079. After decomposing, each 12-bit character is regarded as two 6-bit parts, one high part (H) and one low part (L). We use a decimal value to represent a 6-bit part and it should be between 0 and 63. The dimension of mask table has changed from 4096×3 to 64×6 (for Harry, from 4096×8 to 64×16).

as Harry12b. For Harry12b, the step to load elements from column vectors of the mask table is also changed, as shown in Fig. 10.

2) Additional SIMD Operations: We can see from Fig. 9 and Fig. 10 that the number of LOAD operations, after decomposing the 12 bits, has doubled because the number of mask table columns doubled. Besides, before shifting, 3

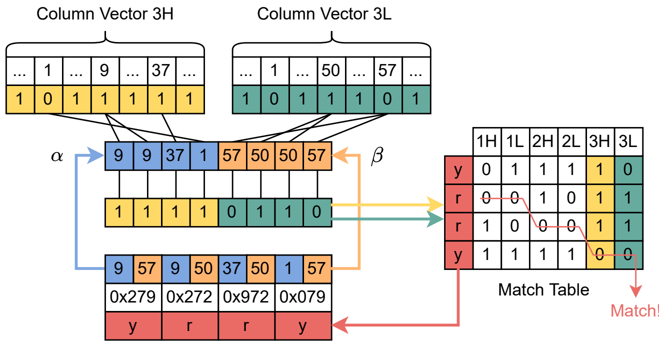


Fig. 10. Harry12b Load Step. For input characters {'y', 'r', 'r', 'y'}, decompose them into 2 vectors. The first vector(the blue vector in this figure) contains the high 6-bit decimal values of the 4 characters and the second vector(the orange one in this figure) contains the low 6-bit decimal values. We mark them as α and β. Then use VPERMB to pick elements from 3H and 3L (the last two column vectors of the new mask table in Fig. 9), taking α and β as index vectors respectively. Finally, put the picked elements (the yellow and green vectors) in match table.

additional OR operations should be performed on the 3 pairs of H&L vectors in match table. For Harry12b, it needs 16 additional SIMD operations(8 LOAD and 8 OR), so totally it needs 40 SIMD operations per 56 input characters, i.e., 0.71 SIMD operation per character.

D. False Positives

We have mentioned that grouping and truncation may introduce false positives. We call them GPF (Grouping False Positive) and TFP (Truncation False Positive). Here the new encoding methods also introduce false positives as some information will be lost after compressing the mask table. Take compression-based encoding as an example, we explain why false positives may appear in Fig. 11. We call these

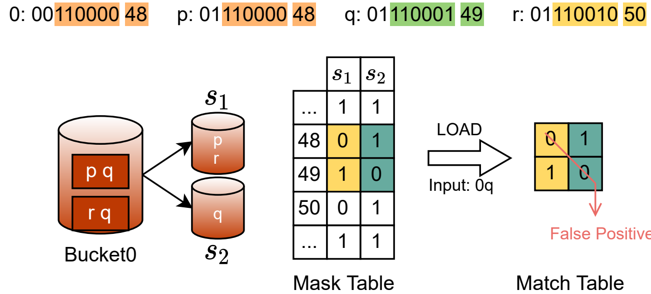


Fig. 11. Encoding False Positives. Suppose bucket 0 has 2 literals, *'rq'* and *'pq'*, and the input string is *'0q'*. Because *'0'* has the same lower 6 bits as *'p'*, so it matches with this bucket. Obviously, it is a false positive match.

false positives as EFP(Encoding False Positive), which are produced because some information is lost after compressing the mask table. Theoretically speaking, though the two encoding methods generate two types of mask table with the same size, the decomposition-based encoding introduces much fewer EFPs than the compression-based encoding, because Harry12b (Harry with decomposition-based encoding) would load double vectors from the mask table. Besides, although they introduce EFPs that may degrade the performance, Harry still benefits a lot because they compress the mask table and make it possible to implement the efficient column-vector-based matching algorithm.

E. Select Between the Two Encoding Methods

We have introduced two encoding methods, both of which have their advantages and disadvantages. The compression-based encoding needs no more SIMD operations but it introduces more EFPs. The decomposition-based encoding introduces much fewer EFPs but needs more SIMD operations. How to select a proper encoding is a topic worth discussing. In Fig. 11, if the bucket contains only *'rq'*, then bit $T[48][0]$ will change from 0 to 1 and the false positive will not appear. Generally speaking, the more zero bits there are in mask table, the more false positives there will appear during matching. In practice, Harry decides on which method to take by comparing the zero bit rates of Harry6b and Harry12b, which we mark as zr_{6b} and zr_{12b}. It executes a heuristic selection algorithm. When zr_{6b} is relatively low, which means that the compression-based encoding wouldn't introduce too many false positives, Harry will select the compression-based encoding (Harry6b) because it needs less SIMD operations. When zr_{6b} is relatively high, which means that the compression-based encoding would produce much more false positives than the decomposition-based encoding, Harry would select the decomposition-based encoding (Harry12b).

VI. The SHUFFLE-based Shift Implementation

A. Problem Analysis

As shown in Fig. 6, with the column-vector-based matching algorithm, the number of SIMD operations is independent of m, the number of input characters processed in an iteration. Theoretically for Harry, the larger m is, the higher the matching performance is, as long as condition (7) is guaranteed. For AVX512 Harry where L = 512, m can be 56. So in Fig. 6(C), Harry would shift 56×8 = 448 bits in a 512-bit-long SIMD vector for 7 times, among which the shift distances are 8, 16, 24, 32, 40, 48, and 56 bits respectively. Unfortunately, VPSLLDQ (the shift instruction in AVX512) cannot shift bits across 128-bit boundaries and lots of bits will be lost after shifting. Take the example of the first time shift whose shift distance is just 8 bits, the shift result is shown in Fig. 12.

As for the other shifts, there are even more lost bits because their shift distances are longer than the first time shift. Too many lost bits would cause unacceptable false positives, making it impossible to implement the matching algorithm in AVX512, the most advanced SIMD instruction set.

B. Use Shuffle Instruction to Shift

We have mentioned VPERMB, the shuffle instruction of AVX512, in Fig. 7(A). It loads specified elements from a source vector into a destination vector, according to an index vector. Actually, we can leverage this instruction to implement SHIFT. Take AVX512 Harry as an example, we operate as shown in Fig. 13.

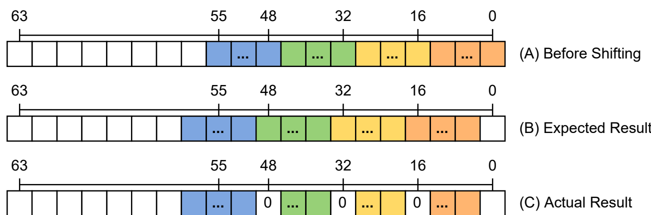


Fig. 12. Shift Result in AVX512. Shift 56 bytes (448 bits) in a 64-byte (512-bit) SIMD vector. The expected shift result is like (B) with no bits lost. But the actual shift result is like (C) with 3 bytes at 16-byte (128-bit) boundaries (16, 32, 48) lost.

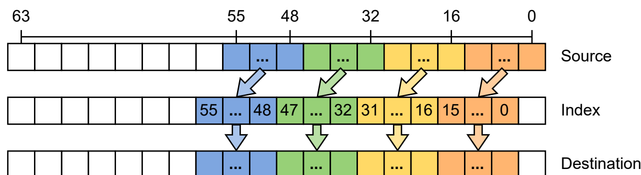


Fig. 13. Shuffle Shift. The data bytes are located in source vector as the low 56 bytes, whose indices are 0∼55. By putting the index sequence 0∼55 at indices 1∼56 of the index vector and calling VPERMB, we take 0∼55 elements of the source vector and put them at indices 1∼56 in destination vector. This plays the same role as shift.

VII. Evaluation

A. Environment

We evaluate Harry on a commodity CPU. We conduct experiments on a Linux server (Ubuntu 20.04) with Intel Xeon Gold 6348 (2.60GHz, 2 sockets, 28 cores per socket) with 32GB DDR4 memory. The processor has the support of AVX512. The encoding methods are implemented in C++. The column-vector-based matching algorithm is implemented in C. We use GCC 9.4.0 to compile them without any optimization.

B. Dataset

To evaluate the efficiency of Harry, we conduct experiments on a variety of real-world literals and input strings. Literals are collected from OWASP ModSecurity Core Rule Set v3.3.2 [45], and Snort Emerging Threats Rules v2.9.0 [46]. Both ModSecurity and Snort are DPI-based applications. Input strings are collected from IXIA HTTP raw packets and Alexa Non-HTTP raw packets. Also, we have generated a random input string for evaluation. We present them in Table II.

TABLE II
Input String

Type	Size
Random	763 KB
IXIA HTTP Messages	121 MB
Alexa non-HTTP Messages	4MB

C. Performance and Scalability

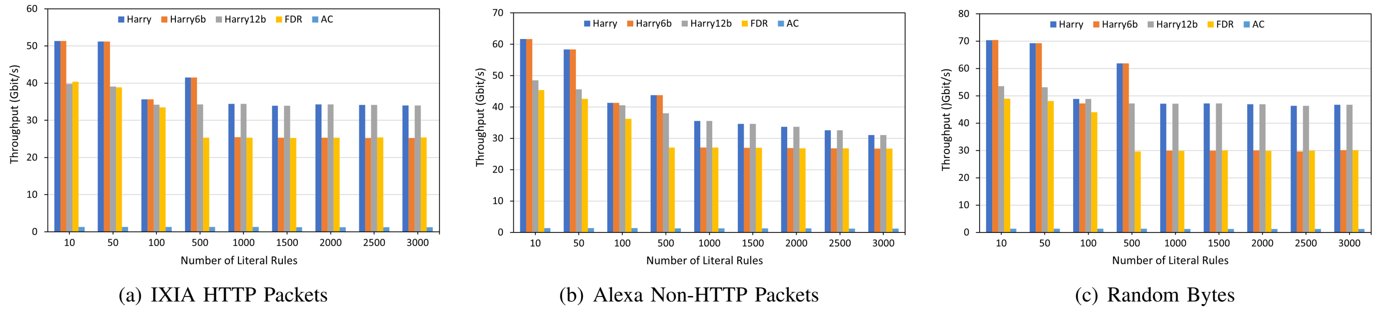
To demonstrate the performance and scalability of Harry, we conduct 6 groups of experiments. We classify the input strings into 3 types, which are http packets, non-http packets and random bytes. For each type of input string, we match it with different kinds of literal rule sets, including rule sets collected from ModSecurity and Snort. And for a certain kind of rule set, we extract several sub-sets, each containing different numbers of rules, ranging from dozens (small-scale), hundreds (medium-scale) to thousands (large-scale). Apart from Harry, we have also tested Harry6b and Harry12b to verify the correctness of our heuristic encoding selection algorithm. We conduct each experiment 1000 times to get the average throughput. The experimental result is shown in Fig. 14 (ModSecurity Rule Set) and Fig. 15 (Snort Rule Set), from which we can see that Harry is faster than FDR and AC in all cases.

In Fig. 14(a) with HTTP packets as the input, Harry achieves 1.06x∼1.63x performance of FDR and 26x∼38x that of AC. In Fig. 14(b) with Non-HTTP packets as the input, Harry achieves 1.14x∼1.62x performance of FDR and 25x∼45x that of AC. In Fig. 14(c) with random bytes as the input, Harry achieves 1.11x∼2.09x performance of FDR and 36x∼52x that of AC. Fig. 15 presents a similar result. Also, we see that Harry can always accord with the better one between Harry6b and Harry12b, which demonstrates that our encoding selection algorithm works correctly and effectively.

Besides, observing the performance of Harry6b and Harry12b, we can find that in most cases, Harry6b is faster than Harry12b when there are less than 1000 rules, but much slower than Harry12b if rules keep increasing. This matches with what we have discussed about the advantages and disadvantages of two encoding methods. Compared to Harry12b (decomposition-based encoding), Harry6b (compression-based encoding) introduces more EFPs but needs no additional SIMD operations, so it outperforms Harry12b in small-scale literal rule sets because dozens of literals won't introduce many EFPs. But as the rule set gets larger, the overhead caused by more and more EFPs finally defeat the benefit brought by its fewer SIMD operations.

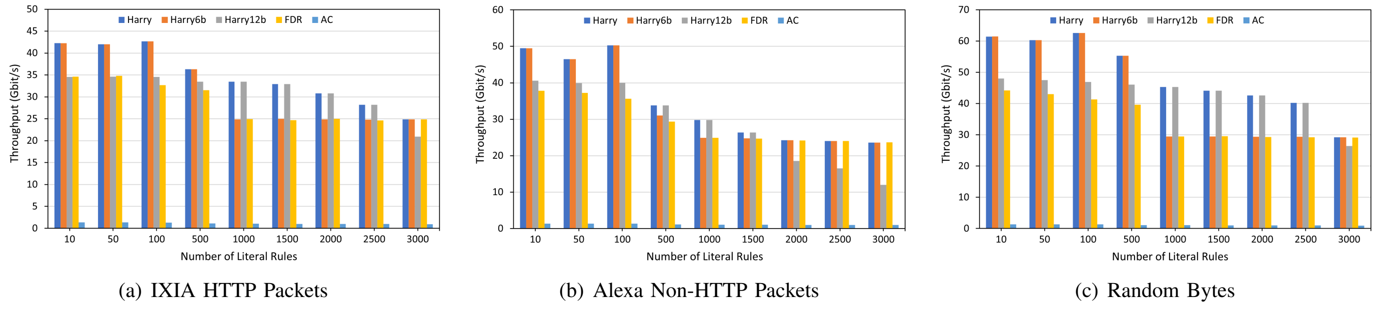
D. False Positives

We have analyzed that both Harry6b and Harry12b introduce additional false positives because they compress the mask table and some information is lost. We also conduct experiments to compare the number of false positives of Harry6b, Harry12b, and Harry. We still use Snort and ModSecurity rule sets and take Alexa Non-HTTP packets as the input string, the experimental result is shown in Fig. 16. It can be seen that as the number of literals increases, the number of false positives of both Harry6b and Harry12b increases, but the former increases much faster. With the literal rules increasing, the false positives of Harry6b would increase to be unacceptable. So Harry will select Harry6b when there are not so many literal rules because Harry6b needs less SIMD operations, but it selects Harry12b when literals get more, because Harry12b has much fewer false positives.



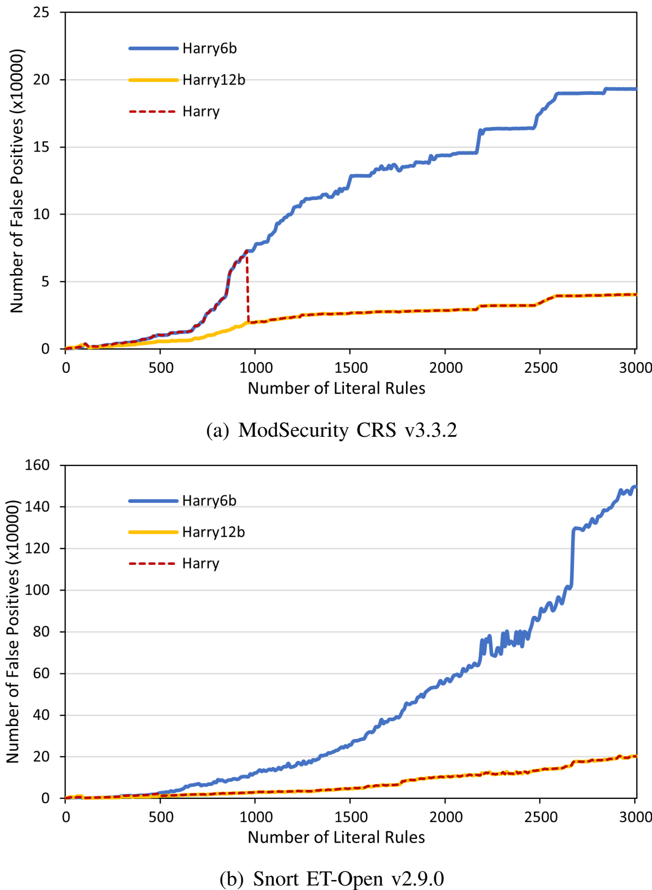
(a) IXIA HTTP Packets (b) Alexa Non-HTTP Packets (c) Random Bytes

Fig. 14. ModSecurity Core Rule Set v3.3.2



(a) IXIA HTTP Packets (b) Alexa Non-HTTP Packets (c) Random Bytes

Fig. 15. Snort ET-Open Rule Set v2.9.0



(a) ModSecurity CRS v3.3.2

(b) Snort ET-Open v2.9.0

Fig. 16. False Positives of Harry6b, Harry12b and Harry.

VIII. Conclusion

In this paper, we propose Harry, a highly-optimized and scalable SIMD-based multi-literal matching engine. We design a column-vector-based shift-or matching algorithm to deeply exploit the data-level parallelism of SIMD and improve SIMD vector utilization. Harry needs only 0.43∼0.71 SIMD operations per input character, much more efficient than FDR that needs 3 SIMD operations per character. Also, the SIMD vector utilization increases from 50% to 87.5%. To support the matching algorithm, we propose two encoding methods, suitable for small-scale and large-scale rule sets respectively. Besides, we have overcome the issues of VPSLLDQ, the SIMD shift instruction, by introducing a novel way to implement shift with VPERMB, the SIMD shuffle instruction that is originally used to reorder elements. Finally, we implement Harry, FDR, and AC on commodity CPU with AVX512 SIMD support. We compare Harry to FDR, the state-of-the-art multi-literal matching engine integrated in Hyperscan and AC, the classic widely-used multi-literal matching algorithm. The evaluation shows that Harry can reach a throughput of 30∼70 Gbit/s, up to 52x that of AC and 2.09x of FDR. It has been successfully deployed in Hyperscan.

Acknowledgment

The work was supported by the National Natural Science Foundation of China under Grant No. 61972101, and was performed when Hao Xu was an intern at Intel. Jin Zhao is the corresponding author.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] Snort. [Online]. Available: https://snort.org/

[5] Suricata. [Online]. Available: https://suricata.io/

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

[7] 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.

[8] Modsecurity. [Online]. Available: https://github.com/SpiderLabs/ModSecurity/

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

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

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

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

[13] "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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] S. Wang, M. Zhang, G. Li, C. Liu, Y. Liu, X. Jia, and M. Xu, "Making multi-string pattern matching scalable and cost-efficient with programmable switching asics," in *Proc. IEEE INFOCOM*, 2021, pp. 1–10.

[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] 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.

[31] 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.

[32] Scaling cloudflare's massive waf. [Online]. Available: http://www.scalescale.com/scaling-cloudflaresmassive-waf/

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] X. Wang, B. Liu, J. Jiang, Y. Xu, Y. Wang, and X. Wang, "Kangaroo: Accelerating string matching by running multiple collaborative finite state machines," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 10, pp. 1784–1796, 2014.

[42] Snort community rules. [Online]. Available: https://www.snort.org/downloads/#rule-downloads/

[43] K. Fredriksson, "Shift-or string matching with super-alphabets," *Information Processing Letters*, vol. 87, no. 4, pp. 201–204, 2003.

[44] 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.

[45] Owasp modsecurity core rule set. [Online]. Available: https://coreruleset.org/installation/

[46] Snort emerging threats rules. [Online]. Available: https://rules.emergingthreats.net/open/snort-2.9.0/rules/