Toward Fast Regex Pattern Matching using Simple Patterns

Mohammad Hashem Haghighat
Department of Automation
Tsinghua University
Beijing, China
l-a16@mails.tsinghua.edu.cn

Jun Li
Research Institute of Information Technology
Tsinghua University
Beijing, China
junl@tsinghua.edu.cn

***Abstract*— Nowadays network solutions employ pattern matching methods to apply protocol identification, traffic billing, load balancing, or detecting network malicious activities. Although regular expression (regex) patterns provide a powerful option to express signatures more effectively, they make the matching procedure more challenging. Dozens of finite automata based methods have been proposed to deal with regex patterns and boost the matching procedure. However, they suffer from high spatial or temporal complexity.**

Recently, we proposed HES as a practical novel method to match thousands of regex patterns in a reasonable time. Although HES was extremely faster than DFA, it was unable to support regex patterns without simple pattern (SP-Free regex patterns). In this paper we developed Enhanced HES (E-HES) to handle these kinds of patterns. Experimental results revealed that E-HES not only supported SP-Free regex patterns, but also it significantly optimized the regex handler check procedure. It leads us to match any kind of regex patterns in high bandwidth networks without spatial or temporal limitation.

***Index Terms*—Signature Matching, Regular Expression Patterns, Intrusion Detection Systems, Regex Parsing Rules, Network Security**

I. Introduction

Modern network services, consider packet payloads to perform load balancing, traffic billing, protocol identification, and intrusion detection. Pattern matching methods is the core feature of these systems, in which input traffic is observed by a set of pre-defined patterns.

Due to the vast improvement of networking area, any pattern matching method needs to keep up with high bandwidths. At first, patterns were described by exact strings[1]. However, they were getting more complicated using wildcard characters. Nowadays, Network security solutions use regular expression (regex) patterns to express their rules. Most of the rules sets of Snort [1], Bro [2], Linux application protocol classifier (l7-filter) [3], Ciscos security system [4], and matching accelerator on IBM PowerEN processor [5] are described by regex pattern.

Although, several simple pattern matching methods such as [6–12] were proposed for fast signature search, they were unable to handle regex patterns. To address this issue, finite automata as one of the most effective data structure was used. Hopcroft et al. first proposed two variants of finite automata based methods to support regex patterns: DFA[2] and NFA[3] [13].

A DFA composed by a set of finite states, a finite set of input symbols (Σ), and a transition function (δ) that maps any state and an input symbol to just one state. The key advantage of DFA is the fast matching time, since it is not related to the number of the patterns. However, DFA suffers from state explosion problem. As a result, it is not practical.

NFA defined exactly the same as DFA except it maps a pair of state and input symbol, to several states (instead of one). NFA solved the state explosion problem, but its matching time was a big concern. given m regex patterns having the length of n in the average, NFA required $O(n^2m)$ time to match patterns [14].

Different variant finite automata based solutions were then proposed to match regex patterns in a reasonable time, while minimizing the required storage. Yu et al. proposed $mDFA$, aiming at grouping all the patterns and creating DFA for each group, so that each DFA size does not exceed a threshold [15].

Kumar et al. in [16] compressed DFA and introduced D^2FA based on grouping the states having equivalent set of outgoing transitions. They also proposed a weighted tree to handle removed transition, in which the tree affected system matching time. Kumar et al. then proposed CD^2FA in [17] to improve D^2FA. According to D^2FA, Becchi and Crowley also proposed "backwards labeled transitions" and $A–DFA$ to increase the throughput of D^2FA [18], [19].

In [20] Becchi and Crowley also proposed a combination of DFA and NFA methods, in which, at the first step NFA was created. Then, only the nodes that avoid state explosion were transformed to DFA. The size of HFA was close to NFA, while the matching time was faster.

Smith et al. in [21], [22] proposed XFA to avoid state explosion by injecting some variables and programs into original DFA. The key point of XFA was to remember the progress during the match. The authors also provided some optimization techniques to remove program and variable

[1]We call the Simple Patterns (SP)

[2]Deterministic Finite Automata

[3]Non-deterministic Finite Automata

duplication. XFA matching time was close to DFA, and its storage requirement was smaller than NFA.

Based on HFA and XFA, Wang et al. in [23], [24] introduced PaCC framework containing these procedures: partition, compression, and matching. The first two procedures were responsible to avoid DFA being exploded and compress the final DFA. $PaCC$ achieved more compact data structure compared to NFA, and its matching time was even faster than DFA.

Recently Haghighat et. al. proposed a highly efficient and scalable technique called HES [25]. The key idea behind HES was to extract all the simple patterns from regex signatures, try to match them, and in case of finding a match, handle the rest of the regex pattern. HES was extremely faster than DFA, while its required storage was close to NFA. Although HES provided a finite automata independent framework, it still needs to be improved. In [25], we highlighted that HES only supports rexeg patterns containing at least one simple pattern, while the rest of regexes were left for the future (we called them as SP-Free regex patterns). Besides, the way of handling conditions after matching simple patterns was the same as brute force method, which, in the worst case, the input string was read for several times. For example, assume that $RP_1 = abc\backslash w * def$ and $RP_2 = ghi\backslash w * jkl$ be two different regex patterns. HES reads the following input string almost three times as described below.

$$input = abcghiaaa \cdots aa!defjkl$$

At first, "abc" and "ghi" are matched. The next match is "def", in which, the characters between "def" and "abc" ("$ghiaaa \cdots aa!$") are checked with "$\backslash w*$". All the characters satisfy "RHC" except the last one ("!"). In the next step, "jkl" is matched, which "$aaa \cdots aa!def$" is then checked with "$\backslash w*$", as "$aaa \cdots aa$" needs to be read one more time.

In this paper we addressed these shortages and proposed "Enhanced HES (E-HES)". The paper structure is explained as follows. In section II a brief description of HES will be provided. Then, in section III E-HES will be discussed in detail including its pre-processing and matching phases. After that, in section IV E-HES will be evaluated and finally, in section V the paper will be concluded.

II. HES Method

This section introduced HES method proposed in [25] in brief. HES is a regex pattern matching method, provided a novel schema to reduce the regex pattern matching costs to match of simple patterns. In HES pre-processing phase all the regex patterns are parsed to extract simple patterns accompanying with metadata, and in its matching phase any arbitrary simple pattern matching method is called to match simple patterns and in case of finding a match, extracted metadata are used to handle the rest of corresponding regex pattern. Figure 1 illustrates the HES architecture.

The rest of this section explained the HES pre-processing and matching phases in brief.

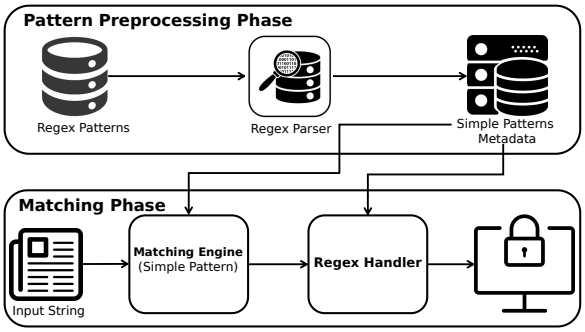


Fig. 1: HES Architecture [25].

A. HES Pre-processing Phase

The main goal of the HES pre-processing phase is to parse regex patterns in order to achieve simple patterns and metadata. All the regex patterns are categorized into "Operator Free (OFRP)", "One-operand Operator (OORP)" and "Two-operands Operator (TORP)" regex patterns and created by the combination of "Simple Patterns (SP)", "Character Sets (CS)", and "Regex Operators (RO)". In [25] We proposed four regex parsing rule sets as expressed in Figure 2.

Rule set 1:

1.1: <RP> = <OFRP> | <OORP> | <TORP>

Rule set 2:

2.1: <OFRP> = <OFRP-CS> | <OFRP-SP>

2.2: <OFRP-CS> = <OFRP><CS> | <CS>

2.3: <OFRP-SP> = <OFRP-CS><SP> | <SP>

Rule set 3:

3.1: <OORP> = <NRP>"("<RP>")"<RO><NRP>

3.2: <TORP> = <NRP>"("<RP>"|"<RP>")"<NRP>

3.3: <NRP> = <RP> | "⊥"

Rule set 4:

4.1: <RO> = {"*","+","?","^","{n}","{n,m}"}

4.2: <CS> = {Any character set such as "\d", "\s","\w","$[c_i - c_j]$", and so on.}

4.3: <SP> = {Any simple pattern without regex operator and character set.}

Fig. 2: Regex Pattern Parsing Rules [25].

All the regex patterns are parsed by the provided parsing rules, in which the extracted information are stored into the following data structure.

- **Title:** Determines a unique title of the extracted simple pattern.
- **Quantity:** Preserves the total number of instances of each simple pattern. Also for each simple pattern instance, the following attributes are stored.
  - **Regex Pattern Reference (RPR):** Specifies the regex pattern reference.
  - **Instance Identity:** Determines a unique identifier for the instance.

  - **Not Operator Check:** Reverses the matching condition, when this attribute was set to "True".
  - **Previous Simple Patterns:** Refers to previous instance/s of the current simple pattern, that should be matched before.
  - **Regex Handler Check:** Expresses the regex operators and/or character sets that placed between current instance and its PSP.
  - **Last Simple Pattern:** Denotes the last simple pattern of each regex pattern.
- **Length:** Stores the simple pattern length.
- **Text:** Preserves the pattern text.

B. HES Matching Phase

In the HES matching phase, the generated HES data structure is used to match the regex patterns as follows.

1) Consider an empty set top store already matched simple patterns.
2) Read input string and try to find a match using any simple pattern matching method.
3) In case of finding a match, find its entry in the HES data structure. For each instance of the match perform below procedure.
   a) Check its corresponding PSP which has been inserted into the candidate set before.
   b) Handle its corresponding RHC.
   c) If the RHC condition is satisfied and NOC flag was set to "False", add the simple pattern instance to the candidate set. Otherwise, remove the instance PSP from the candidate set.

C. Discussion

HES provided a highly scalable and flexible infrastructure to match regex patterns. It reduced the regex pattern matching problem to finding simple patterns. As a result, it is significantly faster than DFA. Also, it has minimum storage requirement which makes it practical to handle tens of thousands regex patterns.

Experimental results showed a great improvement of matching time, compared to the state-of-the-art techniques, while using almost the same storage as NFA. However, RHC conditions need to be handled in more optimum way. In other words, when a simple pattern is matched, the input characters between the matched pattern and its PSP were read to check whether the corresponding RHC wes satisfied or not, that affects the matching time. Also, HES was unable to support SP-Free regex patterns as explained in [25].

In this paper we focus on solving these problems, in which, in the next section, our novel method will be proposed in detail.

III. E-HES Method

In this section, E-HES is explained in detail. E-HES Uses the HES idea to match regex patterns, in which, all the simple patterns are first, extracted in the pre-processing phase, and then, a simple pattern matching method is used to find a match. E-HES provides a novel algorithm to handle RHCs which resolved the HES problem, discussed in Section II.

A. E-HES Pre-processing Phase

This section describes the E-HES pre-processing phase, which contains the following steps: 1) Parsing Regex Patterns, 2) Parsing RHCs, 3) Partitioning Character Sets, 4) Rewriting Character Sets, 5) Creating Character Map. The rest of this section describes above steps in more detail.

1) Parsing Regex Patterns: The first step of the pre-processing phase is to parse all the regex patterns in order to extract simple patterns and metadata, according to the HES method discussed in Section II.

Tables I expressed the metadata and simple patterns, extracted from the following regex patterns.

- $RP_1 = abcd\backslash d+\backslash w * cmd$
- $RP_2 = tty[a-z]\{4\}efgh$
- $RP_3 = abcd[3-7] + ijklm$
- $RP_4 = (\backslash s + [6-9]+)+$

TABLE I: An example of parsing different regex patterns.

Title	1_1,3_1		1_2	2_1	2_2	3_2	4_1
Quantity	2		1	1	1	1	1
RPR*	1	3	1	2	2	3	4
II†	1	1	2	1	2	2	1
NOC‡	F	F	F	F	F	F	F
PSP§	⊥	⊥	1	⊥	1	1	⊥
RHC¶	⊥	⊥	$\backslash d+ \backslash w*$	⊥	$[a-z]\{4\}$	[3-7]+	$(\backslash s$+[6-9]+)+
LSP**	F	F	T	F	T	T	T
Length	4		3	3	4	5	3
Text	$abcd$		cmd	tty	$efgh$	$ijklm$	null

* Regex Pattern Reference
† Instance Identity
‡ Not Operator Check
§ Previous Simple Patterns
¶ Regex Handler Check
** Last Simple Pattern

2) Parsing RHCs: The metadata extracted from the previous step contains RHCs. RHCs are regex patterns, composed from regex operators and character sets. As a result, they are parsed by provided regex parsing rules in Figure 2 to fill the data structure illustrated in Figure 3.

As depicted in Figure 3, the RHC data structure is similar to HES, while the character sets are stored with the same logic as preserving the simple patterns.

Definition 1. Let "RHC", "RO", and "CS" be the regex handler check, regex operator, and character set, respectively. The following functions are defined to compute the "Previous Character Set" of each RHC.

- $first_{CS}(RHC) \rightarrow \{CS\}$: Returns the first character set of given RHC.
- $last_{CS}(RHC) \rightarrow \{CS\}$: Returns the last character set of given RHC.
- $find_{PCS}(CS, RO) \rightarrow \{CS\}$: Returns the previous character set of given CS based on RO.

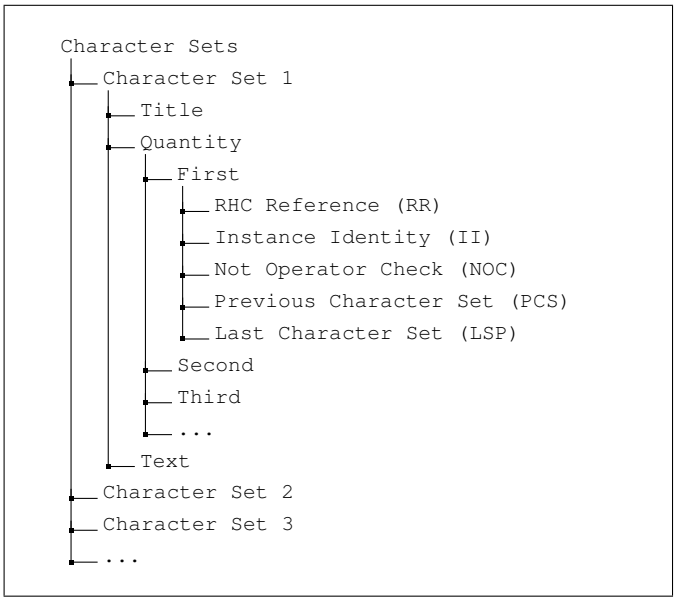


Fig. 3: HES Data Structure Fields.

For instance, $\backslash d+\backslash w*$, $[a-z]\{4\}$, $[3-7]+$, and $(\backslash s+[6-9]+)+$ are extracted RHCs from the example provided in Table I. Parsing RHCs results in the following RHC data structure shown in Table II.

TABLE II: An example of parsing different RHCs.

Title	1_2_1	1_2_2	2_2_1	3_2_1	4_1_1	4_1_2
Quantity	1	1	1	1	1	1
RR*	1_2	1_2	2_2	3_2	4_1	4_1
II†	1	2	1	1	1	2
NOC‡	F	F	F	F	F	F
PCS§	{⊥,1}	{1,2}	⊥	{⊥,1}	{⊥,1,2}	{1,2}
LCS¶	T	T	T	T	F	T
Text	$\backslash d$	$\backslash w$	$[a-z]$	$[3-7]$	$\backslash s$	$[6-9]$

* RHC Reference
† Instance Identity
‡ Not Operator Check
§ Previous Character Set
¶ Last Character Set

3) Partitioning Character Sets: The third step is to partition each two character sets in case that they have some elements in common to achieve "Disjoint Character Sets". Disjoint character sets of above example are defined as below.

$$\begin{aligned}
\backslash d, \backslash w &\longrightarrow \backslash d, [a-Z] \\
[a-Z], [a-z] &\longrightarrow [A-Z], [a-z] \\
\backslash d, [3-7] &\longrightarrow [01289], [3-7] \\
[3-7], [6-9] &\longrightarrow [3-5], [6-7], [8-9] \\
[01289], [8-9] &\longrightarrow [0-2], [8-9]
\end{aligned}$$

$\Rightarrow$ Disjoint Character Sets $=$
$\{[0-2], [3-5], [6-7], [8-9], [a-z], [A-Z], \backslash s\}$

4) Rewriting Character Sets: The next step is to rewrite all the extracted character sets according to the disjoint character sets. The result is stored into a matrix named Ψ, in which the following rewrote the character sets of the above example.

$$\Psi = \begin{array}{c} \begin{matrix} [a\text{-}z] & [A\text{-}Z] & \backslash s & [0\text{-}2] & [3\text{-}5] & [6\text{-}7] & [8\text{-}9] \end{matrix} \\ \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} \backslash d \\ \backslash w \\ [a\text{-}z] \\ [3\text{-}7] \\ [6\text{-}9] \\ \backslash s \end{matrix}$$

5) Creating Character Map: The last step in the E-HES pre-processing phase is to create a character map according to the disjoint character sets.

Definition 2. Let Σ be the set of all possible input characters, and $\Omega = \{\omega_1, \omega_2, \cdots, \omega_k\}$ be the disjoint character sets. Then, Φ as the mapping function of input character to disjoint character set is defined as follows:

$$\forall c_i \in \Sigma : \Phi(c_i) = \begin{cases} \omega_j, & \text{if } \exists j \cdot c_i \in \omega_j \\ \backslash o, & \text{else} \end{cases} \tag{1}$$

where $\backslash o$ is defined as below:

$$\backslash o = \Big(\bigcup_{j=1}^{k} \omega_j \Big)' \tag{2}$$

Considering one byte characters, there are 256 different possible values. The mapping table based on below disjoint character sets is explained in Table III.
$\Omega = \{[0-2], [3-5], [6-7], [8-9], [a-z], [A-Z], \backslash s\}$.

TABLE III: Mapping all the characters according to the disjoint character sets.

ASCII Code	Mapping Function (Φ)	Description
9	$\backslash s$	Horizontal Tab
10		Line Feed
11		Vertical Tab
12		Form Feed
13		Carriage return
32		Space
$[48 \sim 50]$	$[0-2]$	Numbers 0 to 2
$[51 \sim 53]$	$[3-5]$	Numbers 3 to 5
$[54 \sim 55]$	$[6-7]$	Numbers 6 and 7
$[56 \sim 57]$	$[8-9]$	Numbers 8 and 9
$[65 \sim 90]$	$[A-Z]$	English upper case letters
$[97 \sim 122]$	$[a-z]$	English lower case letters
$[0 \sim 8]$	$\backslash o$	The rest of the characters
$[14 \sim 31]$		
$[33 \sim 47]$		
$[58 \sim 64]$		
$[91 \sim 96]$		
$[123 \sim 255]$		

B. E-HES Matching Phase

The matching phase of E-HES contains six steps: 1) System Initialization, 2) Buffering Input Characters, 3) Calling a

Simple Pattern Matching Method, 4) Handling Corresponding RHCs, 5) Checking for the whole Regex Pattern Match, 6) and Checking all the SP-Free regex patterns, as described in Algorithm 1.

Algorithm 1 E-HES Matching Phase

```
//System Initialization
candidate ⟵ NULL
buffer ⟵ initialize buffer
foreach input characters as c
  //Buffering Input Character
  buffer ⟵ buffer character c
  //Calling a Simple Pattern Matching Method
  SP ⟵ call SP_matcth(c)
  //Handling corrensponding RHCs
  foreach SP instances
    if SP instance has no PSP
      if SP.RHC is satisfied in buffer and
         SP.NOC is False
        candidate.add(instance,position)
      end
    elseif PSP exists in candidate
      if SP.RHC is satisfied in buffer and
         SP.NOC is False
        candidate.update(instance, position)
      else
        candidate.remove(PSP)
      end
    end
    //Checking for the whole Regex Pattern match
    if SP.LSP is True
      publish SP as a regex pattern match
    end
    //Checking all the SP-Free Regex Patterns
    foreach SP-Free regex patterns as SPF-RP
      if SPF-RP is satisfied in buffer
        publish SPF-RP as a regex pattern match
      end
    end
  end
end
```

At first, the system is initialized by considering an empty set to preserve matching candidates. Also the procedure expressed in Algorithm 2 is performed in order to provide a data structure to buffer input characters.

Algorithm 2 Buffer Initialization

```
input: Nothing
output: Buffer
//Creating the buffer data structure
foreach ω ∈ (Ω ∪ {\o})
  buffer[ω].next ⟵ NULL
  buffer[ω].last_node ⟵ buffer[ω]
end
//Storing the most recent buffered item
buffer[current] ⟵ NULL
```

The second step in the matching phase is to map and buffer incoming characters according to Equation 1 as below.

1) Compute $\Phi(c)$ for each input character c.
2) Generate and initialize a new buffer node.
3) Link the generated node to the buffer.
4) Update buffer's most recent node.

The whole procedure is defined in Algorithm 3.

Algorithm 3 Buffering input characters

```
input: Character and its position
output: Buffer
//Find the corresponding disjoint character set
ω ⟵ Φ(c)
if the most recent buffered item is not ω
  //Generate and initialize a new buffer node
  n: generate new buffer node
    n.previous ⟵ buffer[ω].last_node
    n.position ⟵ position
    n.next ⟵ NULL
  //Link the node to the buffer
  buffer[ω].last_node.next ⟵ n
  //Update the buffer's most recent node
  buffer[current] ⟵ ω
end
```

In the third step of the matching phase, a simple pattern matching method is called. In case of finding a match, all its instances are extracted and their RHCs are handled in the forth step of the E-HES matching phase. In other words, a simple pattern may be existed in different regex patterns, in which the RHC procedure is performed for each simple pattern instance individually. The RHC procedure is explained as follows.

1) Check the instance length with the instance's PSP and current positions.
2) For each character set of RHC, try to find at least one buffer entry (ω) that violates the RHC rule (i.e. $\Psi(\omega) = 0$), then return "False" value.
   a) Set current position as pivot.
   b) Define $\Xi = instance.RHC.LCS$ which is the last character set of the RHC as the acceptable character sets.
   c) Compute bitwise-or of $\Psi[\Xi]$ as acceptable disjoint character sets.
   d) Find the most recent nodes of the buffer (for each disjoint character set) based on the pivot.
   e) Determine the most recent character set of the buffer as ω_{recent}.
   f) Return "False" when ω_{recent} is not satisfied, according to the acceptable disjoint character sets.
   g) Update pivot according to condition that ω_{recent} is not satisfied with the acceptable character set.
   h) Update Ξ based on previous character set of ω_{recent}.
   i) Perform items 2c to 2h until the pivot value becomes less than instance's PSP position or $\perp$ is added to the acceptable disjoint character set.
3) If all the buffered data obey the whole RHC rule, return "True" value.

The pseudo-code of handling RHC is expressed by Algorithm 4

Algorithm 4 Handling the RHC of an Instance

```
input: Instance and Buffer
output: True or False
//Initializing required variables
Ξ ⟵ instance.RHC.LCS
β_start ⟵ instance.PSP.position + instance.PSP.length
β_end ⟵ current position - 1
```

```
5  //Checking the previous match and current positions
6  if (instance.PSP.position + length of the match)
7  |   is greater than the current position
8  | return False
9  end
10 foreach ω ∈ (Ω ∪ {\o})
11 | Δ[ω] ⟵ buf[ω].last_node
12 end
13 while β_end is greater than β_start or ⊥ ∈ Ξ do
14 | /* Finding acceptable ω according to
15 |    acceptable Character Sets */
16 | ψ ⟵ bitwise_or(Ψ[Ξ]);
17 | foreach ω ∈ (Ω ∪ {\o})
18 | | //Finding the most recent nodes of all ω
19 | | while Δ[ω] is greater than β_end do
20 | | | Δ[ω] ⟵ Δ[ω].previous
21 | | end
22 | end
23 | //Finding the ω of the most recent node
24 | ω_recent ⟵ index of max(δ.position) ∀δ ∈ Δ
25 | //Checking the most recent ω is not acceptable
26 | if ψ[ω_recent] equals to 0
27 | | return False
28 | end
29 | //Finding the previous check position
30 | β_end ⟵ max(δ.position) ∀δ ∈ Δ, Ψ[index_δ][ω_recent] = 0
31 | Ξ ⟵ ω_recent.PCS
32 end
33 //Returning True when everything were satisfied
34 return True
```

The fifth step in the matching phase is to publish the regex match result, which is done when all the SPs of a regex pattern are matched (see Algorithm 1 Lines 28∼33).

Finally, in the sixth step, all the SP-Free regex patterns are checked in the buffer, acording to algorithm 4, in order to publish the regex patterns match, which contains no simple pattern.

To clarify the matching procedure, suppose the regex patterns, provided in Section III-A. Hence, input characters are mapped based on Table III. Now, assume that string "$abcd543\ cmd5433\ 789\ 898989898tty$" is taken as input. As a result, Algorithm 3 generates the E-HES buffer as shown in Figure 4, where the left array represents the disjoint character set and the right boxes are buffer nodes including string position.

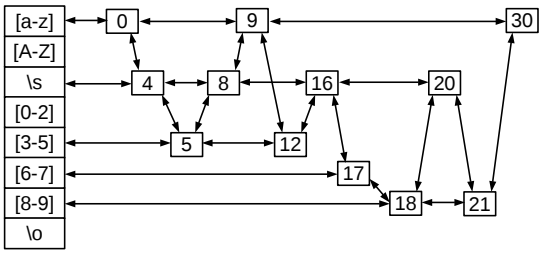


Fig. 4: Buffering input string "$abcd543\ cmd5433\ 789\ 898989898efgh$" based on disjoint character sets $\Omega = \{[0-2], [3-5], [6-7], [8-9], [a-z], [A-Z], \backslash s\}$.

Reading the input string results in matching the following simple patterns.

match ⟶ $abcd$	position: 0
match ⟶ cmd	position: 8
match ⟶ $efgh$	position: 29

```
candidate: NULL
match: abcd
  title = 1_1, 3_1
  Quantity = 2
    First instance:
      PSP=⊥, RHC=⊥, NOC=F, LSP=F
      Handling RHC
        Satisfied due to RHC=⊥
      ⇒ add RPR_II = 1_1 to candidate
    Second instance:
      PSP=⊥, RHC=⊥, NOC=F, LSP=F
      Handling RHC
        Satisfied due to RHC=⊥
      ⇒ add RPR_II = 3_1 to candidate
Checking SP-Free Regex Patterns
  First instance: 4_1
    PSP=⊥, RHC=(\s+[6 − 9]+)+, NOC=F, LSP=T
    Handling RHC
    β_start = 0
      Round 1:
        Ξ : {[6 − 9]}
        β_end : 0
        Ψ : {0000011}
        Δ : {0 null null null null null null}
        max_node: [a − z]
      Not Satisfied due to Ψ[[a − z]] = 0
----------------------------------------
candidate: {1_1, 3_1}
match: cmd
  title = 1_2
  Quantity = 1
    First instance:
      PSP=1, RHC=\d+\w*, NOC=F, LSP=T
      Handling RHC
        β_start = 4
        Round 1:
          Ξ : {\d, \w}
          β_end = 8
          Ψ : {0001111} ∨ {1101111} = {1101111}
          Δ : {0 null 7 null 4 null null}
          max_node: \s
        Not Satisfied due to Ψ[\s] = 0
      ⇒ remove the instance
Checking SP-Free Regex Patterns
  First instance: 4_1
    PSP=⊥, RHC=(\s+[6 − 9]+)+, NOC=F, LSP=T
    Handling RHC
    β_start = 0
      Round 1:
        Ξ : {[6 − 9]}
        β_end : 0
        Ψ : {0000011}
        Δ : {0 null 7 null 4 null null}
        max_node: \s
      Not Satisfied due to Ψ[\s] = 0
----------------------------------------
candidate: {3_1}
match: efgh
  title = 2_2
  Quantity = 1
    First instance:
      PSP=tty, RHC=[a − z]{4}, NOC=F, LSP=T
        Not Satisfied due to tty in not in
candidate
Checking SP-Free Regex Patterns
  First instance: 4_1
    PSP=⊥, RHC=(\s+[6 − 9]+)+, NOC=F, LSP=T
    Handling RHC
      β_start = 0
      Round 1:
        Ξ : {[6 − 9]}
        β_end : 30
        Ψ : {0000011}
```

$\Delta : \{9\ null\ 20\ null\ 12\ 17\ 21\}$
max_node: $[8-9]$
$\Psi[[8-9]] = 1$
Round 2:
$\Xi : \{\backslash s, [6-9]\}$
$\beta_{end} : 20$
$\Psi : \{0010011\}$
$\Delta : \{9\ null\ 20\ null\ 12\ 17\ 18\}$
max_node: $\backslash s$
$\Psi[\backslash s] = 1$
Round 3:
$\Xi : \{\perp, \backslash s, [6-9]\}$
$\beta_{end} : 12$
Satisfied due to $\perp \in \Xi$
LSP is True
$\Rightarrow$ report RP_4 as a match

IV. Experimental Evaluation

HES handled RHCs in a very simple way (just checking all the characters between matched pattern and its PSP). Although this procedure is enough for most cases, in some situations, dealing with RHCs needs reading the input for several times.

In this paper, E-HES was proposed to deal with RHCs in optimum way. The aim of this section is to observe the E-HES method using experimental tests. E-HES is compared with the state-of-the-art finite automata based techniques as well as HES, in terms of matching time and storage requirement. The source codes of DFA, NFA, and HFA are available in [26]. The overhead of "regex handler module" is then investigated.

A. Evaluation Setup

We used AC algorithm [6] as the HES and E-HES matching modules. Also, we utilized two different pattern sets, to serve our observation.

1) Snort [1] regex pattern sets.
2) Bro [2] regex pattern sets.

Several input data sets ranging from 1KB to 1GB are randomly generated, with different matching probabilities. The memory and CPU of the test system was 32GB and Core i7-6700HQ, respectively.

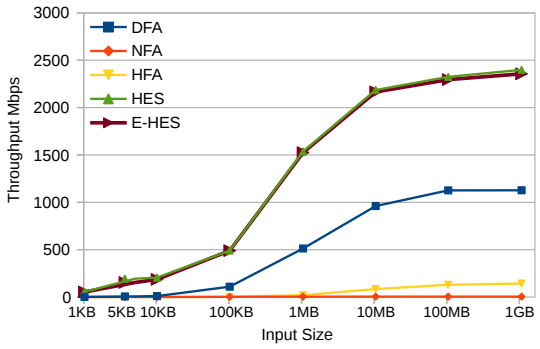
B. Matching Time

The main concern of any signature matching method is to keep up with the increase of line speed. In other words, using thousands of regex patterns, a signature matching method has to check the input string with minimum overhead. In this section, E-HES was compared with both HES and finite automata based techniques, in which their throughput are illustrated in Figure 5.

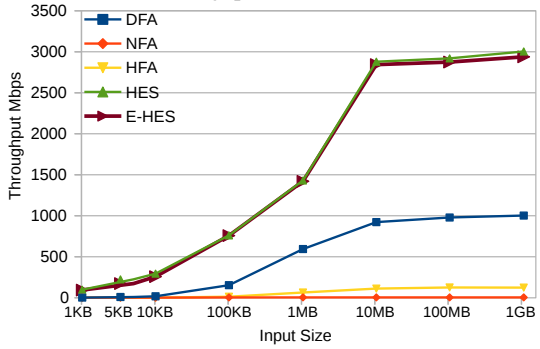
As illustrated in Figure 5, E-HES was slower than HES due to buffering input string. However, it achieved better results compared to DFA, as one the fastest finite automata based methods, while the throughput was a bit smaller than 1.3 Gbps.

C. RHC Overhead

E-HES buffered input string to satisfy RHCs in optimum way. However, it decreased the system throughput compared to HES. The main goal of this section is to highlight the importance of buffering input strings, compared to handling RHCc as simple as HES did.



(a) Throughput: Snort Rule set.



(b) Throughput: Bro Rule set.

Fig. 5: The comparison between E-HES and HES versus DFA, NFA, and HFA methods in terms of their throughput.

To achieve our goal, we fed both HES and E-HES by three sets of input strings, in which each set contained several strings ranging from 1KB to 1GB. The first set was generated in a way that the strings needed to be read once, in order to handle the RHC of each simple pattern match, using HES. Respectively, the second and third sets was generated in a way that the whole input strings had to be read ten and a hundred times. Figure 6 shows the matching time of the test.

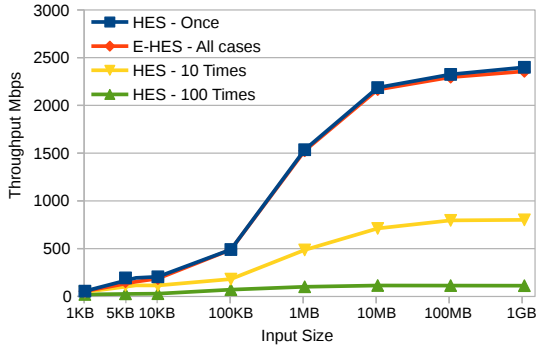


Fig. 6: The matching time of E-HES and HES when the input strings need to be read several times.

As depicted in Figure 6, in case of reading the read input string only once, the HES method provided the best

throughput, while it achieved around 2.4 Gbps for 1GB input data. However, for the inputs that required HES to read the whole string for several times, the throughput was decreased significantly. We achieved 801 Mbps and 112 Mbps when the input was read ten and a hundred times, respectively. On the other side, E-HES provided steady results, in which its matching time was not related to the input string. As a result, it achieved 1.3 Gbps for all cases.

D. Memory Consumption

The main concern of the E-HES method is its storage requirement, as it buffers the input string. Table IV provides the required buffer size for different inputs ranging from 1KB to 1GB.

TABLE IV: E-HES Storage Requirement for Different Input Size

Input	1KB	5KB	10KB	100KB	1MB	10MB	100MB	1GB
Buffer	1.26KB	6.48KB	13KB	162KB	1.89MB	19MB	243MB	2.43GB

As explained in Table IV, E-HES required around 2.5 times input size to buffer the data. This storage requirement was reasonable. Because, it was increasing linearly (not exponentially). Also, by reading new input (new flow data), the whole buffer is flushed and an empty buffer is chosen. Besides, we defined "$flush$" function to flush the buffer, when the number of inserted data exceeds a threshold value. The whole procedure of the "$flush$" function is expressed by Algorithm 5.

Algorithm 5 E-HES Flushing Buffer

```
input: Candidate and Buffer
output: Empty Buffer
//Initializing required variables
foreach instance ∈ Candidate
    χ ⟵ Find the next SP of instance
    Handle the RHC of χ
end
return null
```

Figure 7 shows the flushing overhead in case of setting the threshold value to 100MB, in which the overhead was 281 micro second in the worst case.

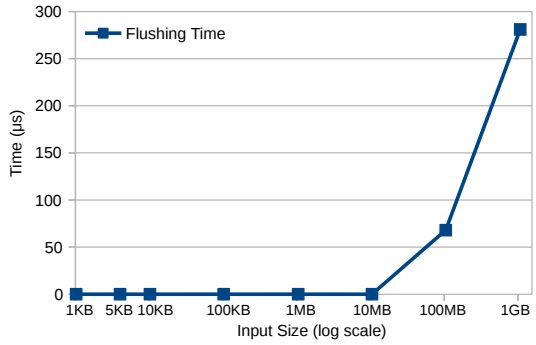


Fig. 7: The overhead of flushing function for different input sizes.

V. Conclusions

In this paper we developed an enhanced version of HES method, called E-HES to support SP-Free regex patterns as well as, making the RHC procedure faster. While finite automata based techniques were suitable for limited number of patterns, and HES method handled specific type of regexes, E-HES provides a powerful framework to support all kind of patterns with minimum spatial and temporal requirements. Although experimental results showed that E-HES was slower than HES, it still matched patterns faster than DFA. Also, E-HES resisted against bad input data, compared to HES, which provided worse throughput.

In the future we plan to continue improving E-HES to support modern regex operators like "Back-reference", which enable more advanced features for network solutions. Besides, deploying E-HES on parallel processing platforms like FPGA will provide us higher throughput, is another area for the future study.

VI. Acknowledgments

This work was supported by the National Key Technology R&D Program of China under Grant No. 2015BAK34B00 and the National Key Research and Development Program of China under Grant No. 2016YFB1000102.

References

[1] SNORT, "Snort: Network intrusion detection and prevention system," https://www.snort.org/downloads#rules, 2017.
[2] Bro, "The bro network security monitor," https://www.bro.org/download/index.html, 2017.
[3] l7 filter, "Application layer packet classifier for linux," http://l7-filter.sourceforge.net/, 2009.
[4] Cisco, "Cisco ios ips deployment guide," https://www.cisco.com, 2015.
[5] IBM, "Poweren pme public pattern sets," https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/PowerEN+PME+Public+Pattern+Sets, 2012.
[6] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.
[7] M. Aldwairi and K. Al-Khamaiseh, "Exhaust: optimizing wu-manber pattern matching for intrusion detection using bloom filters," in *Web Applications and Networking (WSWAN), 2015 2nd World Symposium on*. IEEE, 2015, pp. 1–6.
[8] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.
[9] B. Choi, J. Chae, M. Jamshed, K. Park, and D. Han, "Dfc: Accelerating string pattern matching for network applications." in *NSDI*, 2016, pp. 551–565.
[10] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.
[11] C.-H. Lin, J.-C. Li, C.-H. Liu, and S.-C. Chang, "Perfect hashing based parallel algorithms for multiple string matching on graphic processing units," *IEEE Transactions on Parallel and Distributed Systems*, 2017.
[12] C. Allauzen and M. Raffinot, "Factor oracle of a set of words," *Technical report 99-11*, 1999.
[13] J. E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to automata theory, languages, and computation," *ACM SIGACT News*, vol. 32, no. 1, pp. 60–65, 2001.
[14] F. Yu, Y. Diao, R. H. Katz, and T. Lakshman, "Fast packet pattern-matching algorithms," in *Algorithms for Next Generation Networks*. Springer, 2010, pp. 219–238.
[15] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 2006, pp. 93–102.

[16] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 4. ACM, 2006, pp. 339–350.

[17] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*. ACM, 2006, pp. 81–92.

[18] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM, 2007, pp. 145–154.

[19] ——, "A-dfa: A time-and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 1, p. 4, 2013.

[20] ——, "A hybrid finite automaton for practical deep packet inspection," in *Proceedings of the 2007 ACM CoNEXT conference*. ACM, 2007, p. 1.

[21] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. ACM, 2008, pp. 207–218.

[22] R. Smith, C. Estan, and S. Jha, "Xfa: Faster signature matching with extended automata," in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*. IEEE, 2008, pp. 187–201.

[23] K. Wang, Z. Fu, X. Hu, and J. Li, "Practical regular expression matching free of scalability and performance barriers," *Computer Communications*, vol. 54, pp. 97–119, 2014.

[24] K. Wang and J. Li, "Freme: A pattern partition based engine for fast and scalable regular expression matching in practice," *Journal of Network and Computer Applications*, vol. 55, pp. 154–169, 2015.

[25] M. H. Haghighat, Z. fu, and J. Li, "Hes: Highly efficient and scalable technique for matching regex patterns." in *Information and Network Technologies (ICINT 2018), 2018 3rd International Conference on*, 2018.

[26] M. Becchi, "Regular expression processor," http://regex.wustl.edu/index.php/Main_Page, 2011.

APPENDIX

A. Operator Free RHC Relations

Definition A1. Let "RHC" be an operator free regex pattern, which is described by equation 3.

$$RHC = CS_1 CS_2 \cdots CS_n \quad \text{for } (n \geq 1) \tag{3}$$

where "$CS_i\ (1 \leq i \leq n)$" are different character sets. then:

$$find_{PCS}(CS_i, NULL) = CS_{i-1} \quad (1 < i \leq n) \tag{4}$$

$$first_{CS}(RHC) = CS_1 \tag{5}$$

$$last_{CS}(RHC) = CS_n \tag{6}$$

B. One-operand Operator RHC Relations

Definition A2. Let "RHC" be a one-operand operator regex pattern, which is described by equation 7.

$$RHC = NRP_1(RP_2)RO\ NRP_3 \tag{7}$$

then:

$$find_{PCS}\Big(first_{CS}(NRP_3), RO\Big) = \tag{8}$$
$$\begin{cases} \{last_{CS}(NRP_1), last_{CS}(RP_2)\}, & RO \in \{*, ?, \hat{}\} \\ last_{CS}(RP_2), & RO \in \{+, \{n\}, \{n, m\}\} \\ last_{CS}(NRP_1), & last_{CS}(RP_2) = \perp \end{cases}$$

$$find_{PCS}\Big(first_{CS}(RP_2), RO\Big) = \tag{9}$$
$$\begin{cases} \{last_{CS}(NRP_1), last_{CS}(RP_2)\}, & RO \in \{*, +, \{\{n, m\} \\ & \wedge(n \leq i < m)\}\} \\ last_{CS}(RP_2), & RO \in \{\{n\}, \{n, m\}\} \wedge (i < n) \\ last_{CS}(NRP_1), & RO \in \{?, \hat{}, \{\{n\} \wedge (i = n)\}, \\ & \{\{n, m\} \wedge (i = m)\}\} \\ & \vee\ last_{CS}(RP_2) = \perp \end{cases}$$

$$first_{CS}(RHC) = \tag{10}$$
$$\begin{cases} first_{CS}(NRP_1), & first_{CS}(NRP_1) \neq \perp \\ \{first_{CS}(RP_2), first_{CS}(NRP_3)\}, & first_{CS}(NRP_1) = \perp \\ & \wedge RO \in \{*, ?, \hat{}\} \\ first_{CS}(RP_2), & first_{CS}(NRP_1) = \perp \\ & \wedge RO \in \{+, \{n\}, \{n, m\}\} \\ & \wedge first_{CS}(RP_2) \neq \perp \\ first_{CS}(NRP_3), & \text{else} \end{cases}$$

$$last_{CS}(RHC) = \tag{11}$$
$$\begin{cases} last_{CS}(NRP_3), & last_{CS}(NRP_3) \neq \perp \\ \{last_{CS}(NRP_1), last_{CS}(RP_2)\}, & last_{CS}(NRP_3) = \perp \\ & \wedge RO \in \{*, ?, \hat{}\} \\ last_{CS}(RP_2), & last_{CS}(NRP_3) = \perp \\ & \wedge last_{CS}(RP_2) \neq \perp \\ & \wedge RO \in \{+, \{n\}, \{n, m\}\} \\ last_{CS}(NRP_1), & \text{else} \end{cases}$$

C. Two-operands Operator RHC Relations

Definition A3. Let "RHC" be a two-operand operator regex pattern, which is described by equation 12.

$$RP = NRP_1(RP_2|RP_3)NRP_4 \tag{12}$$

then:

$$find_{PCS}\Big(first_{CS}(NRP_4), |\Big) = \tag{13}$$
$$\begin{cases} \{last_{CS}(RP_2), last_{CS}(RP_3)\}, & last_{CS}(RP_2) \neq \perp \\ & \vee last_{CS}(RP_3) \neq \perp \\ last_{CS}(NRP_1), & \text{else} \end{cases}$$

$$find_{PCS}\Big(first_{CS}(RP_3), |\Big) = last_{CS}(NRP_1) \tag{14}$$

$$find_{PCS}\Big(first_{CS}(RP_2), |\Big) = last_{CS}(NRP_1) \tag{15}$$

$$first_{CS}(RHC) = \tag{16}$$
$$\begin{cases} first_{CS}(NRP_1), & first_{CS}(NRP_1) \neq \perp \\ \{first_{CS}(RP_2), first_{CS}(RP_3)\}, & first_{CS}(NRP_1) = \perp \\ & \wedge(first_{CS}(RP_2) \neq \perp \\ & \vee first_{CS}(RP_3) \neq \perp) \\ first_{CS}(NRP_4), & \text{else} \end{cases}$$

$$last_{CS}(RHC) = \tag{17}$$
$$\begin{cases} last_{CS}(NRP_4), & last_{CS}(NRP_4) \neq \perp \\ \{last_{CS}(RP_2), last_{CS}(RP_3)\}, & last_{CS}(NRP_4) = \perp \\ & \wedge(last_{CS}(RP_2) \neq \perp \\ & \vee last_{CS}(RP_3) \neq \perp) \\ last_{CS}(NRP_1), & \text{else} \end{cases}$$