HES: Highly Efficient and Scalable Technique for Matching Regex Patterns

Mohammad Hashem Haghighat, Zhe Fu
Department of Automation
Tsinghua University
Beijing, China
{l-a16, fu-z13}@mails.tsinghua.edu.cn

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

Abstract—**Several security devices use signature based detection engine to detect malicious activities through the internet. The main challenge of this scenario is to keep up with the increase of line speed. On one hand, regular expression (regex) patterns allow security analysts to express more complicated attacks. On the other hand, they make pattern matching procedure much more costly. Several finite automata based techniques have been proposed to speed up the matching procedure. However, they are still impractical in the real world, due to their high spatial or temporal complexity.**

In this paper, a novel technique, called HES, is proposed to handle tens of thousand regex patterns, with minimum space limitation. The experimental results over several rule sets including Snort and Bro, as two leading open source intrusion detection systems, as well as random regex patterns, reveals us HES matched patterns significantly faster than DFA, as one of the fastest state-of-the-art techniques. In addition, the HES storage requirement is close to NFA, which leads as one of the most compact method. These results proved that HES can be used in the real world, as a signature based matching engine, and gives us the power to use more regex patterns.

Keywords- Signature Matching; Regular Expression Patterns; Intrusion Detection System; Regex Parsing Rules.

I. INTRODUCTION

Several network security mechanisms like Intrusion Detection/Prevention Systems[1] and Deep Packet Inspections[2] use pattern matching techniques in order to detect malicious traffic. These devices should be highly effective to keep up with the increasing line speed (e.g. 40Gbps).

As the first and easiest way to match patterns, input string can be compared with all the patterns one by one, which is time consuming. Dozens of methods [1-6] have been proposed to speed up the matching procedure by creating different data structures.

Although the state-of-the-art techniques handle thousands of patterns quickly, matching more complex signatures is still an open problem. Simple malware patterns are usually explicit text strings (referred as simple patterns in this paper), like "*GetInfo\x0d*" which is a signature for a "back-door attack"" [7]. However, it is impossible to express signatures of complex malwares like polymorphic worms in such a way.

In contrast to simple patterns, Regular Expression[3] pattern is an alternative option to express complex signatures (e.g. *"^Entry/file/[0-9]{71,}//.*\x0Aannotate\x0A"*, for detecting a "Concurrent Versions System[4]" revision overflow attack [7]). In the real world, most of security devices such as Snort, Bro, Linux application protocol classifier (l7-filter), Cisco's security system, and matching accelerator on IBM PowerEN processor, use regex patterns in their rule set [8–12].

Hopcroft et al. in [13], proposed DFA5 method to match regex patterns. A DFA consists of a set of finite states, a transition function (δ) that maps any state and an input symbol to just one state, and a finite set of input symbols (∑).

In theory, DFA method is a good option for regex patterns since its matching time is O(1). However, its required large data structure raised a new concern. In other words, the DFA data structure for m regex patterns with the average length of n, needs $O(\sum^{nm})$ space [7], which causes stage explosion problem and makes the system unpractical.

NFA[6] is another well-known method proposed in [13]. It is similar to DFA except its transition function maps a pair of state and input symbol, to several states. Although NFA method solves the state explosion problem that requires *O(nm)* storage, its matching time is not reasonable. The NFA search complexity is $O(n^2m)$ [7].

In order to match regex patterns in a reasonable time, while minimizing the required storage, several variant finite automata based methods have been proposed [14-24]. Yu et al. in [14] proposed mDFA[7], based on grouping patterns and then, creating DFA for each group in a way that the size of each DFA does not exceed at a threshold.

In [15] Kumar et al. introduced D^2FA (Delayed input DFA), according to the fact that many states have the same set of outgoing transitions. So they all can be replaced with one link. As a result, the achieved DFA size will be reduced.

[1] IDS/IPS
[2] DPI
[3] Regex
[4] CVS
[5] Deterministic Finite Automata
[6] Non-deterministic Finite Automata
[7] multiple DFA

Since redundant transitions mapped to one, D^{2}FA requires creating a weighted tree based on similar links, which affects the system matching time.

Becchi and Crowley in [16] proposed HFA[8]. During the pre-processing phase, any nodes that would cause state explosion will be retain as NFA, while the rest are transformed to DFA. As a result, the size of data structure will be close to NFA, but the matching time is faster.

Smith et al. in [17] proposed a novel method named XFA[9], which tries to remove or at least mitigate ambiguity of states by adding some variables into DFA. XFA remembers the progress during the match of regex patterns. It uses some flags or counters as well as some programs to handle several regex operators. The main drawback of XFA is that within merging XFAs, defined variables and programs would be duplicated, which affects the system matching time. However, the authors provided some optimization techniques to remove duplication. XFA showed time complexity near to DFA, while its required storage is similar or even smaller than NFA.

In [18, 19] Kai et al. introduced PaCC[10] framework includes partition, compression, and matching procedures. The partition phase avoids DFA to be exploded, while the compression engine compresses the DFA. The authors argued that PaCC provides smaller data structure compared to NFA, while its matching time is even faster than DFA.

Based on the results provided by the authors, the finite automata based techniques can handle limited number of patterns. Several activated states at the same time are handled in NFA based solutions; while DFA based methods have to deal with storage problem. As a result, both types are not able to handle tens of thousands patterns in the real world.

As a practical option, we provide a highly efficient and scalable novel technique, named HES. The main idea behind HES is to first, extract simple patterns from regex signatures, then try to compare them with the input string, and finally, handle the rest of regex patterns. As an example assume "*S = abcd(\w+\s{6,9}|\d+)*efgh*" as a regex pattern. The extracted simple patterns would be "*abcd*" and "*efgh*". As a result, when the simple patterns are matched with the input string, "*\w+\s{6,9}|\d+)**" condition will be checked as the next step.

The main contribution of HES is summarized as below.

- Due to mapping regex matching problem to simple pattern matching, HES can match regex patterns significantly faster than DFA.
- HES is highly scalable, which means any simple pattern matching algorithm can be used as its matching engine.
- HES has minimum storage requirement due to dealing with simple patterns.
- HES can handle tens of thousands regex patterns at the same time.

The paper structure is described as follows. At first, in section II, regex patterns will be explained in more detail. In addition, regex parsing rules will be provided to generate parse tree. Then the HES technique including its architecture as well as pre-processing and matching phases will be described in section III. After that, HES will be evaluated in practice in section IV, and finally, in section V, the conclusion will be provided.

II. Regex Patterns

The aim of this section is to characterize regex patterns. In general, regex patterns consists of three main attributes as below.

1) ***Regex Operators:*** Regex Operators[11] denote operation over one (e.g. "*(RP$_1$)**") or two sub-patterns (e.g. "*RP$_1$|RP$_2$*"), where "*RP$_1$*" and "*RP$_2$*" can be any regex sub-patterns.

2) ***Character Sets:*** Character Sets[12] describe the set of characters such as "*\d*" as all digits, "*\s*" as all space characters, "*\w*" as all alphanumeric characters, "*.*" as any arbitrary character, "*[c$_i$-c$_j$]*" as a range of characters, and "*[c$_1$c$_2$…c$_k$]*" as a set of "*k*" different characters.

3) ***Simple Patterns:*** Simple Patterns[13] represent the text string pattern without regex operator and character set.

In hierarchical view, each regex pattern is parsed by provided rules, as illustrated in Figure 1, where "*<RO>*", "*<CS>*", and "*<SP>*" denote the regex attributes (ruleset 4).

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 [$c_1 c_2 \cdots c_k$].}
4.3: <SP> = {Any simple pattern without regex operator and character set.}

Figure 1. Regex Pattern Parsing Rules.

[8]

[9] eXtensible FA

[10] Partition, Compression, and Combination

[11] RO

[12] CS

[13] SP

According to rule set 1, a regex pattern is parsed hierarchically as one of "Operator Free[14]", "One-operand Operator[15]", or "Two-operands Operator[16]" Regex Pattern. The rest of this section defines these categories in more detail.

A. Operator Free Regex Pattern

OFRP is the first type of regex pattern, which is created by the combination of both simple patterns and character sets. As an example "*S_1[a-z]\s\d.S_2\w*" is parsed as an OFRP, where S_1 and S_2 are two arbitrary simple patterns. Each OFRP is created by the recursive combination of *<OFRP-CS>* and *<OFRP-SP>* (rule number 2.1), where:

- *<OFRP-CS>*: denotes an OFRP, which is finished by a character set (rule number 2.2).
- *<OFRP-SP>*: denotes an OFRP, which is finished by a simple pattern (rule number 2.3).

B. One-operand Operator Regex Pattern

OORP is the second type of regex pattern (rule number 3.1). It starts and ends with "Nullable Regex Pattern" (NRP) (rule number 3.3). It also has a regex operator, which acts on a regex sub-pattern. It is important that the operand is not nullable. "*S_1(S_2)?S_3*", "*S_1\d**", and "*[b-k]+*" are different examples of one-operand operator regex pattern.

C. Two-operands Operator Regex Pattern

TORP is the third type of regex pattern (rule number 3.2). Like the previous item, its first and last patterns are nullable, while both operands are not (e.g "*S_1(S_2|S_3)S_4*", "*S_1(\d|S_2[a-k])*", and "*(S_1|\w)*").

D. Parsing Example

As an example consider "*S_1(S_2.{6}|(S_3\d)+)*S_4*" as a regex pattern, where S_1, S_2, S_3, and S_4 are four simple patterns. In the first level of hierarchy, the following rule is satisfied.

<RP> = <NRP>(<RP>)<RO><NRP>

Then:

- The first "*<NRP>*" leads to "*S_1*".
- "*<RP>*" leads to "*S_2.{6}|(S_3\d)+*".
- "*<RO>*" leads to "***".
- The last "*<NRP>*" leads to "*S_4*".

Similarly, the rest of parsing procedure is carried out. Figure 2 depicts achieved parse tree of the example.

HES will use the parsing rules of regex patterns in its pre-processing phase, in which in the next section the whole procedure will be discussed in detail.

III. HES Technique

This section describes HES in terms of its architecture, and pre-processing and matching phases in details.

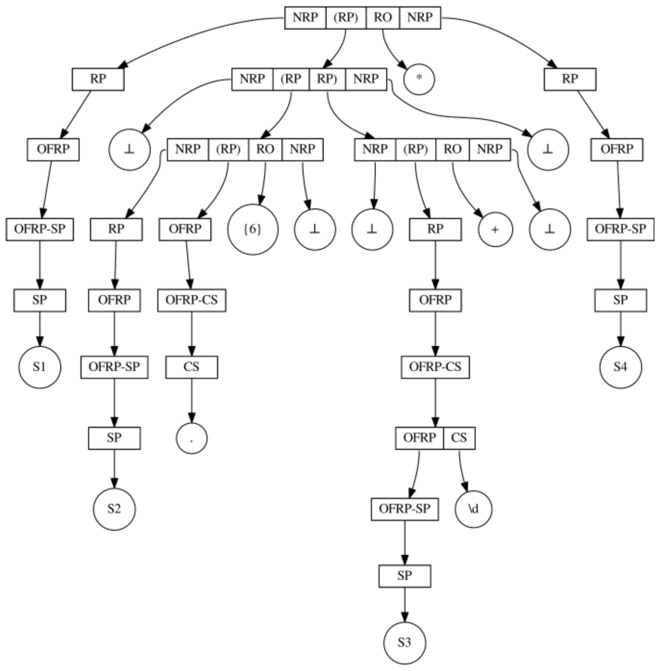


Figure 2. Parse Tree of "*S_1(S_2.{6}|(S_3\d)+)*S_4*".

A. Model Architecture

The overall procedure of HES is to map regex patterns to a set of simple patterns accompanying with metadata (details to be introduced in section III.C), so that the overall required matching time and storage size will be reduced to that of handling simple patterns. Figure 3 shows HES architecture.

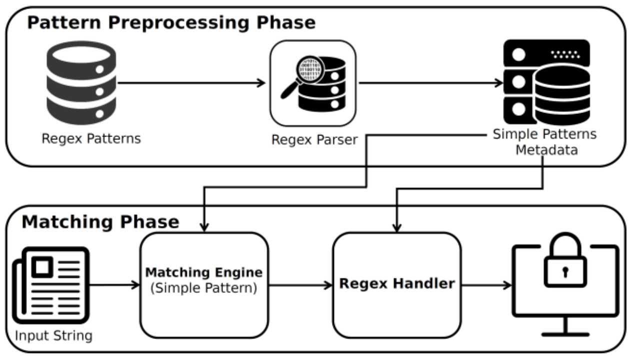


Figure 3. HES Architecture.

In the pre-processing phase, regex patterns are parsed using the parsing rules provided in section II to achieve simple patterns and metadata.

In the matching phase, any simple pattern matching method can be used as HES matching engine to match simple patterns. When a match is found, regex handler module tries to match the rest of regex pattern using corresponding metadata information.

B. Pre-processing Phase

In the pre-processing phase, all the regex patterns are parsed in order to extract simple patterns and metadata. This

[14] OFRP
[15] OORP
[16] TORP

extracted information is stored into data structure illustrated in Figure 4.

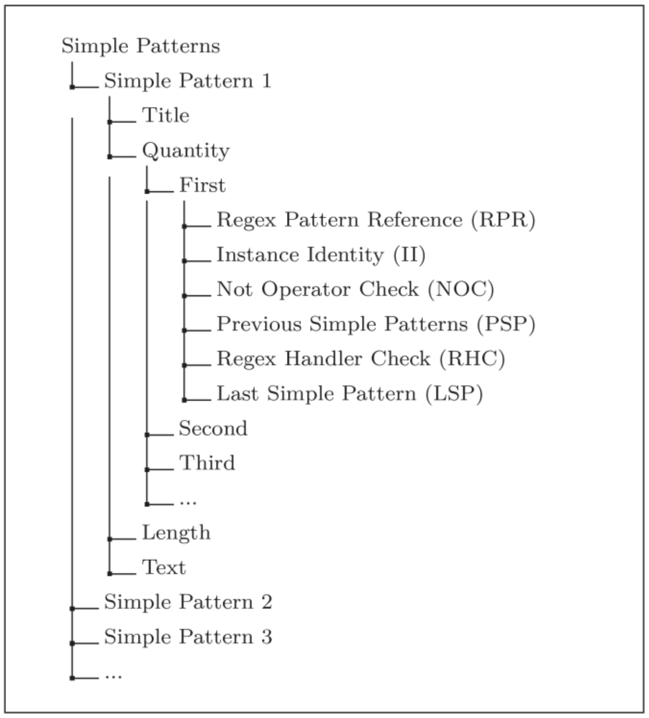


Figure 4. HES Data Structure Fields.

The HES data structure contains several attributes as metadata described below.

- **Title:** Title is a unique identifier of each extracted simple pattern.
- **Quantity:** Some simple patterns are seen in different regex patterns. Hence, the total number of instances is stored in “Quantity” field. Also for each pattern instance, the following attributes are preserved.
    - **Regex Pattern Reference (RPR):** Regex pattern reference is preserved here.
    - **Instance Identity (II):** A unique identifier of the pattern instance is stored in this field.
    - **Not Operator Check (NOC):** In regex patterns, “not” operator is different from other regex operators, since its corresponding pattern should not be matched. As a result, when “NOC” attribute is set to “True”, the matching condition will be reversed.
    - **Previous Simple Patterns (PSP):** Refers to previous instance/s that should be matched before the current pattern.
    - **Regex Handler Check (RHC):** RHC is a part of regex pattern including character sets and regex operators, which placed between the instance and its PSP.
    - **Last Simple Pattern (LSP):** Determines the last simple pattern of each regex pattern.
- **Length:** Points to the simple pattern length.
- **Text:** Contains the pattern text.

Definition 1. Let “RP”, “RO”, “SP”, and “CS”, be the regex pattern, regex operator, simple pattern, and character set, respectively. $find_{PSP}$ and $find_{RHC}$ functions are defined to find the *PSP* and *RHC* of each simple pattern instance, using $first_{SP}$, $last_{SP}$, $first_{RHC}$, and $last_{RHC}$.

- $find_{PSP}(SP,RO) \longrightarrow \{SP\}$: Returns the previous simple pattern of given *SP* based on *RO*.
- $find_{RHC}(SP,RO) \longrightarrow \{CS \times RO\}$: Returns the regex handler check of given *SP* based on *RO*.
- $first_{SP}(RP) \longrightarrow \{SP\}$: Returns the first simple pattern of given *RP*.
- $last_{SP}(RP) \longrightarrow \{SP\}$: Returns the last simple pattern of given *RP*.
- $first_{RHC}(RP) \longrightarrow \{CS \times RO\}$: Returns the first *RHC* of given *RP*.
- $lastRHC(RP) \longrightarrow \{CS\}$: Returns the last *RHC* of given *RP*.

Appendix A defines each function in detail. The HES pre-processing phase is expressed in Algorithm 1.

Algorithm 1 HES Pre-Processing Phase.

```
input: Regex Pattern $RP$
output: HES Data Structure
// Initialize HES data structure
metadata ⟵ NULL
// Finding the regex pattern type
if RP type is OFRP //$RP = CS_1 SP_1 CS_2 \cdots CS_n SP_n CS_{n+1}$
    SP ⟵ $last_{SP}$(RP)
    while SP exists do
        $\tau$ ⟵ set SP fields
        metadata.update($\tau$)
        SP ⟵ SP.PSP
    end
else if RP type is OORP //$RP = NRP_1(RP_2)RO\ NRP_3$
    $SP_3$ ⟵ $first_{SP}(NRP_3)$
    $SP_2$ ⟵ $first_{SP}(RP_2)$
    $\tau_3$ ⟵ set $SP_3$ fields based on RO operator
    $\tau_2$ ⟵ set $SP_2$ fields based on RO operator
    metadata.update($\tau_3$)
    metadata.update($\tau_2$)
else if RP type is OORP //$RP = NRP_1(RP_2|RP_3)NRP_4$
    $SP_4$ ⟵ $first_{SP}(NRP_4)$
    $SP_3$ ⟵ $first_{SP}(RP_3)$
    $SP_2$ ⟵ $first_{SP}(RP_2)$
    $\tau_4$ ⟵ set $SP_4$ fields based on “|” operator
    $\tau_3$ ⟵ set $SP_3$ fields based on “|” operator
    $\tau_2$ ⟵ set $SP_2$ fields based on “|” operator
    metadata.update($\tau_4$)
    metadata.update($\tau_3$)
    metadata.update($\tau_2$)
end
if(RP is the root of parse tree)
    $SP_1$ ⟵ $first_{SP}(NRP_1)$
    $\tau_1$ ⟵ set $SP_1$ fields based on RO operator
    metadata.update($\tau_1$)
end
return metadata
```

As described in Algorithm 1, the first step to hierarchically parse regex patterns is finding the pattern type. Then, the corresponding simple patterns are extracted, and finally, the HES metadata fields are computed. Algorithm 2 expresses setting the metadata fields of given simple pattern and regex operator.

Algorithm 2 **Setting SP Fields based on RO.**

```
input: Simple Pattern SP and Regex Operator RO
output: HES metadata node
if There is no instance of SP
    // Adding a new simple pattern node
    node ⟵ generate a new metadata node
    node.title ⟵ a unique name for SP;
    node.text ⟵ SP.text;
    node.length ⟵ SP.length;
    node.quantity ⟵ 1;
else
    // Finding the corresponding node
    node ⟵ find the metadata SP node
    node.quantity ⟵ increment the quantity
end
// Set the rest of the fields
node.II ⟵ a unique name for this instance
node.NOC ⟵ (RO = NOT : T, F);
node.RPR ⟵ regex pattern number;
node.PSP ⟵ $find_{PSP}$(SP, RO);
node.RHC ⟵ $find_{RHC}$(SP, RO);
return node
```

As an example, consider "*abcd(cmd|tty)*efgh*" and "*ijklm\d+(abcd)\xyz" as two different regex patterns. At first, these two patterns are parsed as depicted in Figure 5.

The next step is to traverse the trees in order to fill the HES data structure fields. At first, suppose traversing of the first regex pattern. As illustrated in Figure 5a, in the top most level of hierarchy, the pattern was parsed as a one-operand operator regex pattern, which includes three regex sub-patterns and a one-operand operator (see section II.B).

According to Algorithms 1 and 2, the following steps are taken.

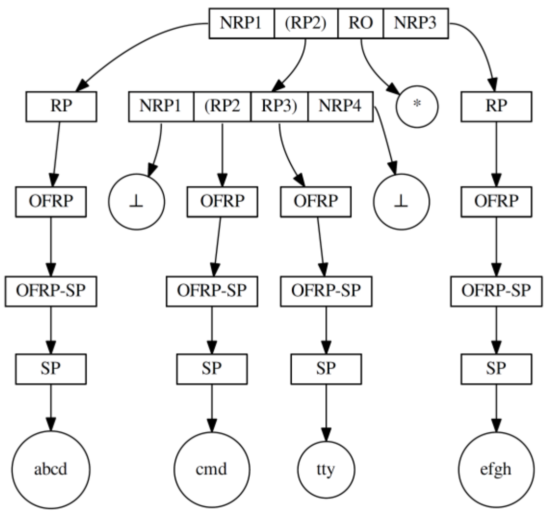
1) Call "$first_{SP}(NRP_3)$" function to find the first simple pattern of NRP_3, in which the result is "*efgh*".

2) It is the first instance of this simple pattern, so it is added to simple patterns metadata and the title, quantity, text, length, II, NOC, and RPR fields are set to their corresponding values.

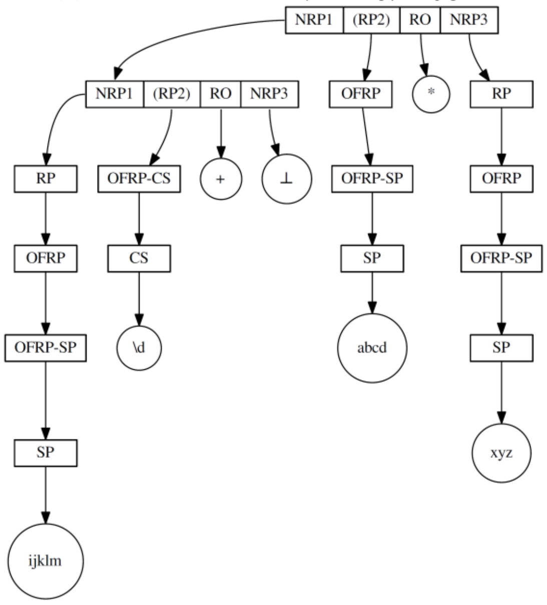
3) Call "$find_{PSP}(efgh, *)$" function to find the *PSP* of "*efgh*". According to equation 9 defined in Appendix A, the result is both "$last_{SP}(NRP_1)$" and "$last_{SP}(RP_2)$" together, due to *regex operator* "*".

a) As illustrated in Figure 5a, "NRP_1" is an operator free regex pattern. As a result "$last_{SP}(NRP_1)$" is computed according to equation 5, and its result is "*abcd*".

b) "RP_2" is a two-operands operator regex pattern. Therefore, "$lastSP(RP_2)$" is computed according to equation 25, which results is both "*cmd*" and "*tty*".



(a) Parse tree of "$abcd(cmd|tty) * efgh$".



(b) Parse tree of "$ijklm\backslash d + (abcd) * xyz$".

Figure 5. An example of the pre-processing phase.

Thus, $find_{PSP}(efgh, *) = \{abcd, cmd, tty\}$.

4) Call "$find_{RHC}(efgh, *)$" function to find the *RHC* of "*efgh*". According to equation 11, the result has two options as below:

a) "$last_{RHC}(NRP_1)first_{RHC}(NRP_3)$", if "$RP_2$" is not matched in the matching phase.

Both "NRP_1" and "NRP_3" are operator free regex patterns, which "$last_{RHC}(NRP_1)$" and "$first_{RHC}(NRP_3)$" equal to "⊥" according to equations 7 and 6, respectively. So their concatenation will be "⊥".

b) "$last_{RHC}(RP_2)first_{RHC}(NRP_3)$", if "$RP_2$" is matched in the matching phase.

As mentioned above, $first_{RHC}(NRP_3)$ = ⊥. But “RP_2” is a two-operands operator regex pattern, hence, $last_{RHC}(RP_2)$ is computed based on equation 27 as follows.

- “$last_{RHC}(RP_{22})NRP_{24}$”, if “$RP_{22}$” is matched in the matching phase.
- “$last_{RHC}(RP_{23})NRP_{24}$”, otherwise.

Based on Figure 5a, “NRP_{24}” equals to “⊥”. In addition, according to equation 7 both “$last_{RHC}(RP_{22})$” and “$last_{RHC}(RP_{23})$” equal to “⊥”. Therefore, the result of “$last_{RHC}(RP_2)$” in either cases is “⊥”.

As a result, $find_{RHC}(efgh, *)$ = ⊥.

5) The rest of the pre-processing phase is carried out accordingly.

After finishing the pre-processing phase, the HES data structure is filled as shown in Table 1. It is important to note that the RHC value of simple pattern “*xyz*” is “{⊥, \d+}”, as according to equation 11, when “*abcd*” is matched, the “*RHC*” value is “⊥”, and in case of matching “*ijklm*”, “*RHC*” is “\d”.

TABLE I. HES DATA STRUCTURE OF “*abcd(cmd|tty)*efgh*” AND “*ijklm\d+(abcd)\xyz”.

Title	1_1,2_2		1_2	1_3	1_4	2_1	2_3
Quantity	2		1	1	1	1	1
RPR[*]	1	2	1	1	1	2	2
II[§]	1	2	2	3	4	1	3
NOC[¶]	F	F	F	F	F	F	F
PSP[£]	⊥	1	1	1	{1,2,3}	⊥	{1,2}
RHC[¥]	⊥	\d+	⊥	⊥	⊥	⊥	{⊥ , \d+}
LSP[£]	F	F	F	F	T	F	T
Length	4		3	3	4	5	3
Text	*abcd*		*cmd*	*tty*	*efgh*	*ijklm*	*Xyz*

[*] Regex Pattern Reference
[§] Instance Identity
[¶] Not Operator Check
[£] Previous Simple Pattern
[¥] Regex Handler Check
[£] Last Simple Pattern

C. *Matching Phase*

After the pre-processing phase, the generated HES data structure is used to match the patterns. Any arbitrary simple pattern matching method such as [1-6] can be utilized to match simple patterns. In case of finding a match, its corresponding “PSP” and “RHC” values are considered to handle the rest of the regex pattern. The detailed procedure is described as below.

1) Consider an empty set named candidate, to store the matching regex patterns candidates during the matching phase.

2) Read the input string and call a simple pattern matching method.

3) In case of finding a match, (due to the fact that a simple pattern can be seen in several regex patterns) do the following procedure for all its instances.

a) Check the instance's “PSP”. If it is set to “NULL”, means the instance is the first simple pattern of its corresponding regex. As a result it is added to the candidate set, if it satisfies the conditions provided by “RHC” and “NOC” attributes.

b) If the “PSP” value is not “NULL”, means the previous pattern is matched before. Then check the candidate set, find the corresponding entry, and check “RHC” and “NOC”.

- If the checks are passed, the instance is added to candidate set.
- Otherwise, the item referred by “PSP” is removed from the candidate set.

Algorithm 3 shows the whole procedure of the matching phase.

Algorithm 3 HES Matching Phase.

```
//System Initialization
candidate ⟵ NULL
foreach input characters as c
    //Calling a Simple Pattern Matching Method
    SP ⟵ call SP_matcth(c)
    foreach SP instances
        if SP instance has no PSP
            if SP.RHC is satisfied and SP.NOC is False
                candidate.add(instance, position)
            end
        else if PSP exists in candidate
            if SP.RHC is satisfied 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
    end
end
```

While the procedure of satisfying RHC is explained in Algorithm 4.

Algorithm 4 Satisfy RHC Procedure.

```
input: Instance
output: True or False
//Checking the previous match and current positions
if (instance.PSP.position + length of the match)
     is greater than the current position
    return False
end
// Check the characters in between
```

```
foreach character $c$ between instance.PSP.position
        and current position
    if $c$ does not satisfy instance.RHC
        return False
    end
//Returning True when everything were satisfied
return True
}
```

As an example, consider the following regex patterns,

$$
\begin{aligned}
RP_1 &= abcd(cmd|tty)*efgh \\
RP_2 &= ijklm\backslash d+(abcd)*xyz
\end{aligned}
$$

which their HES data structure is provided by Table 1. Now let assume that below input string is given to the system.

$$input = h53abcd\ fs\ efgh5ijklm23abcdabcdxyz$$

By reading the input, the following simple patterns are matched.

match $\longrightarrow$ *abcd*	position: 6
match $\longrightarrow$ *efgh*	position: 12
match $\longrightarrow$ *ijklm*	position: 18
match $\longrightarrow$ *abcd*	position: 24
match $\longrightarrow$ *abcd*	position: 28
match $\longrightarrow$ *xyz*	position: 31

The HES matching phase will be run as follows.

```
candidate: NULL
match: abcd
  title = 1_1,2_2
  Quantity = 2
    First instance:
      PSP=⊥, RHC=⊥, NOC=F, LSP=F
      Satisfied
      ⇒ add "RPR_II = 1_1" to candidate
    Second instance:
      PSP=1, RHC=\textbackslash d+, NOC=F, LSP=F
      Not Satisfied
      Due to no corresponding candidate
      ⇒ do nothing
----------------------------------------
candidate: {1_1}
match: efgh
  title = 1_4
  Quantity = 1
    First instance:
      PSP={1,2,3}, RHC=⊥, NOC=F, LSP=T
      Not Satisfied
      Due to "abcd" matching position
      ⇒ remove the instance
----------------------------------------
candidate: NULL
match: ijklm
  title = 2_1
  Quantity = 1
    First instance:
      Satisfied
      ⇒ add "RPR_II = 2_1" to candidate
----------------------------------------
candidate: {2_1}
match: abcd
  title = 1_1,2_2
  Quantity = 2
    First instance:
      PSP=⊥, RHC=⊥, NOC=F, LSP=F
      Satisfied
      ⇒ add "RPR_II = 1_1" to candidate
    Second instance:
      PSP=1, RHC=\textbackslash d+, NOC=F
      Satisfied
      ⇒ update the instance
----------------------------------------
candidate: {1_1, 2_2}
match: abcd
  title = 1_1,2_2
  Quantity = 2
    First instance:
      PSP=⊥, RHC=⊥, NOC=F, LSP=F
      Satisfied
      ⇒ add "RPR_II = 1_1" to candidate
    Second instance:
      PSP=1, RHC=\textbackslash d+, NOC=F, LSP=F
      Satisfied
      ⇒ update the instance
----------------------------------------
candidate: {1_1, 2_2}
match: xyz
  title = 2_3
  Quantity = 1
    First instance:
      PSP={1,2}, RHC=⊥, NOC=F, LSP=T
      Satisfied
      ⇒ report RP_2 as a match
```

IV. Experimental Evaluation

The aim of this section is to evaluate HES with the help of experimental tests. HES will be compared with previously proposed finite automata based methods including DFA, NFA, and HFA [16], in terms of matching speed as well as their storage costs. The source code of HFA is available in [25]. The overhead of “regex handler module” will also be investigated in detail. AC algorithm [1] was used as the HES matching module.

The evaluation setup is first discussed in section IV.A. Then the comparison between HES and other finite automata based methods in terms of their time and spatial performance are explained in sections IV.B and IV.C, respectively. As the final experiment, in section IV.D, the overhead of “regex handler module” is analyzed.

A. Evaluation Setup

In order to conduct a complete experimental analysis, we used two different pattern sets.

1) Regex patterns provided by Snort [8] and Bro [9] Intrusion Detection Systems.

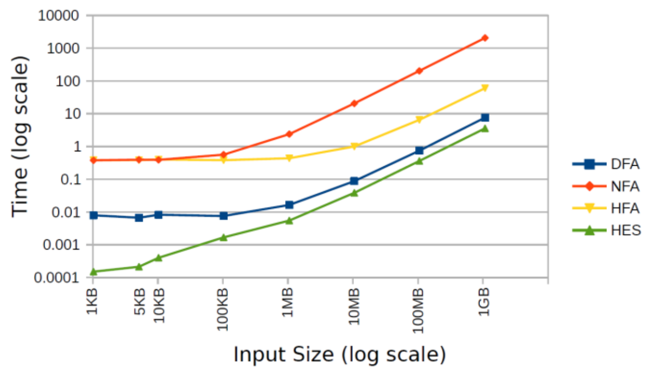
2) 10000 random regex patterns with different length, generated from public regex processor available in [25].

Several 1GB input data sets are generated, with different matching probability. The test system had 32GB Memory and its CPU was Core i7-6700HQ. The rest of this section explains about the evaluation results.

B. *Matching Time*

At first, HES was compared with the state of the art finite automata based methods including DFA, NFA, and HFA in terms of their matching time.

As discussed in section I, DFA and NFA provided the lowest and highest matching time in theory, while HFA took the advantages of both methods, in which its matching time was placed in between. Figure 6 shows the matching time of HES and the state-of-the-art techniques using Snort and Bro rules, while 10 percent of input data were matched by the patterns.



(a) Matching Time: Snort Rule set.

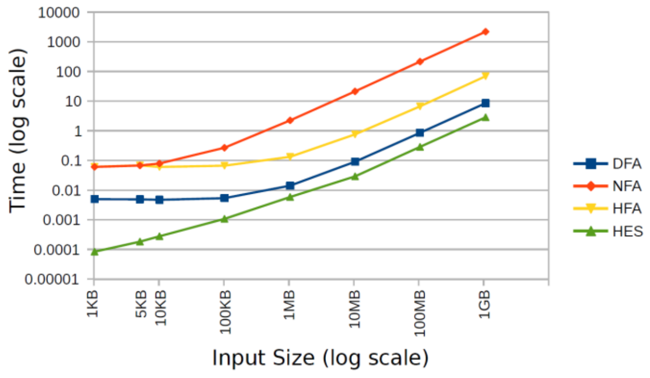


Figure 6. The comparison between HES versus DFA, NFA, and HFA methods in terms of their matching time.

As illustrated in Figure 6, increasing the input size, results in boosting the matching time. However, HES provides the best result compared to the state-of-the-art techniques.

It is important to note that the test was conducted over a limited number of regex patterns. DFA method suffers from state explosion problem, which forced us to use a limited number of patterns. Moreover, using more patterns affects the NFA and HFA matching time significantly, due to more states were activated at the same time.

Figure 7 illustrates increasing the matching time of DFA method in terms of growing the number of patterns, while HES matching time remained the same (around two seconds). In other words, due to state explosion problem, several page switches between disk and memory occurred in DFA method, that affected the matching time significantly.

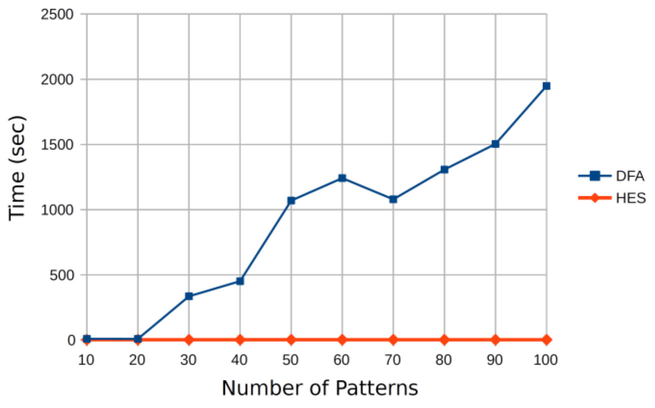


Figure 7. HES and DFA matching time during increasing the number of patterns.

It is noteworthy that the generated random patterns were more complicated than Snort and Bro rule sets, containing more regex operators. As a result both NFA and HFA methods did not respond any result in reasonable time. Thus, the test provided by Figure 7 was conducted over DFA and HES techniques.

C. *Memory Consumption*

Storage requirement plays an important role in any pattern matching system. Nowadays IDSes utilize tens of thousands patterns. As a result, we need an effective method to handle all of them. As described earlier, NFA provides one of the most compact data structures compared to other methods. The goal of this section is to make a comparison between NFA and HES in terms of spatial requirement.

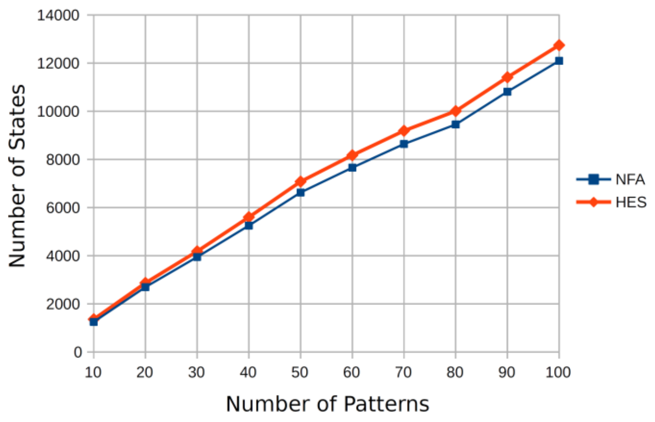


Figure 8. Spatial Requirements of HES and NFA.

As illustrated in Figure 8, the required storage of HES was close to NFA, since only simple patterns were used in the matching module. However, NFA was a bit smaller, because the HES regex handler module needed a small storage to handle the rest of the patterns.

D. Regex Handler Module Overhead

As previously mentioned, the main novelty of HES, is to map regex patterns to several simple ones. Then try to match simple patterns, and finally handling the regular part. In this section, we want to examine the regex handler module overhead.

In fact, when no simple pattern is matched, the matching time is the same as its used simple pattern matching method. In this situation, HES provides the best running time. On the other side, simpler pattern matches, leads to longer HES matching time. Figure 9 shows HES throughput for different hit percentage.

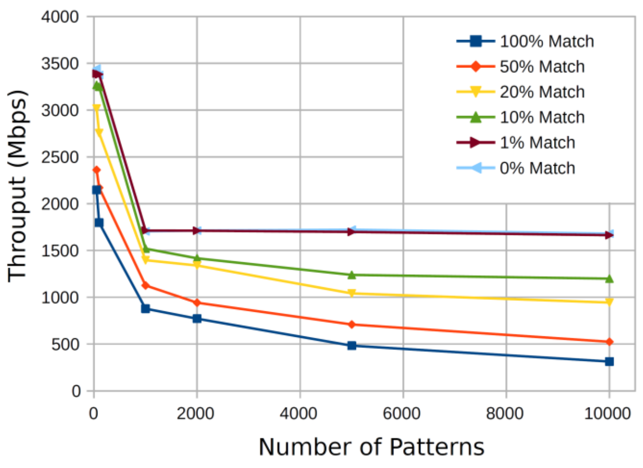


Figure 9. HES throughput for different hit percentage.

As depicted in Figure 9, increasing the match percentage resulted in decreasing the HES throughput. In other words, the worst case occurred when all input characters were matched with the patterns.

HES achieved around 1.7Gbps for 10000 patterns, which was considerable. It is important to note that in the real situation, the matching percentage should be a very small number. However, in the worst case, it achieved more than 300Mbps.

V. Conclusions

In this paper, we presented a highly scalable and efficient novel technique, called HES, for matching regex patterns. While the state-of-the-art techniques were suitable for limited number of patterns, the main contribution of HES is the ability to handle tens of thousands regex patterns with minimum storage requirements. In addition, HES is scalable in which, any simple pattern matching method can be used as its matching engine. Experimental results showed that HES matched patterns significantly faster than DFA, while its memory consumption was still comparable to NFA.

However, there is an exception to the HES technique. Consider a signature contains no simple pattern (for example "*\d+\s.*[t-w]*", which composed by character sets and regex operators). The main contribution of HES is to extract simple patterns and try to match them. After finding a match, handle the rest of the pattern. So, what should be done for this kind of patterns? In the future, we will address this situation.

In addition, more complicated regex operators that give security analysts to detect more attacks, like "Back-reference Operator", is another consideration for future study. Moreover, running the HES algorithm leverages parallel processing platforms, as FPGA will provide us higher throughout, which is another working area in the future.

Acknowledgment

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] 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.

[2] 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.

[3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol. 20, no. 10, pp. 762–772, 1977.

[4] 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.

[5] 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.

[6] 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.

[7] F. Yu, Y. Diao, R. H. Katz, and T. Lakshman, "Fast packet patternmatching algorithms," in Algorithms for Next Generation Networks. Springer, 2010, pp. 219–238.

[8] SNORT, "Snort: Network intrusion detection and prevention system," https://www.snort.org/downloads#rules, 2017.

[9] Bro, "The bro network security monitor," https://www.bro.org/download/index.html, 2017.

[10] l7 filter, "Application layer packet classifier for linux," http://l7-filter.sourceforge.net/, 2009.

[11] Cisco, "Cisco ios ips deployment guide," https://www.cisco.com, 2015.

[12] IBM, "Poweren pme public pattern sets," https://www.ibm.com/developerworks/community/wikis/home?lang =en#!/wiki/PowerEN+PME+Public+Pattern+Sets, 2012.

[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, 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.

[15] 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.

[16] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in Proceedings of the 2007 ACM CoNEXT conference. ACM, 2007, p. 1.

[17] 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.

[18] 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.

[19] 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.

[20] Y.-K. Chang and C.-H. Shih, "A memory efficient pattern matching scheme for regular expressions," Procedia Computer Science, vol. 110, pp. 250–257, 2017.

[21] J. van Lunteren, "Scalable dfa compilation for high-performance regular-expression matching," in Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems. ACM, 2016, pp. 10–19.

[22] X. Chen, B. Jones, M. Becchi, and T. Wolf, "Picking pesky parameters: Optimizing regular expression matching in practice," IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 5, pp. 1430–1442, 2016.

[23] M. Najam, U. Younis, and R. ur Rasool, "Speculative parallel pattern matching using stride-k dfa for deep packet inspection," Journal of Network and Computer Applications, vol. 54, pp. 78–87, 2015.

[24] M. Becchi and P. Crowley, "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.

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

APPENDIX

A. HES Pre-Processing Phase Relations

1) Operator Free Regex Pattern Relations

Definition A1. Let "*RP*" be an operator free regex pattern, which is described by equation 1.

$$RP = \{CS_1\}SP_1\{CS_2\}SP_2\{CS_3\}\cdots\{CS_n\}SP_n\{CS_{n+1}\} \quad (1)$$

where $\{CS_i\} = \{CS_{i1}CS_{i2} \ldots CS_{in_i}\}$ for $(n_i \geq 1)$, then:

$$\begin{aligned} find_{PSP}(SP_i, NULL) &= SP_{i-1} \quad (1 < i \leq n) & (2)\\ find_{RHC}(SP_i, NULL) &= CS_i \quad (1 < i \leq n) & (3)\\ first_{SP}(RP) &= SP_1 & (4)\\ last_{SP}(RP) &= SP_n & (5)\\ first_{RHC}(RP) &= CS_1 & (6)\\ last_{RHC}(RP) &= CS_{n+1} & (7) \end{aligned}$$

2) One-operand Operator Regex Pattern Relations

Definition A2. Let "*RP*" be a one-operand operator regex pattern, which is described by equation 8.

$$RP = NRP_1(RP_2)RO\ NRP_3 \quad (8)$$

then:

$$find_{PSP}\Big(first_{SP}(NRP_3), RO\Big) = \quad (9)$$

$$\begin{cases} \{last_{SP}(NRP_1), last_{SP}(RP_2)\}, & RO \in \{*, ?, \hat{}\,\} \\ last_{SP}(RP_2), & RO \in \{+, \{n\}, \{n,m\}\} \\ last_{SP}(NRP_1), & last_{SP}(RP_2) = \perp \end{cases}$$

$$find_{PSP}\Big(first_{SP}(RP_2), RO\Big) = \quad (10)$$

$$\begin{cases} \{last_{SP}(NRP_1), last_{SP}(RP_2)\}, & \begin{matrix} RO \in \{*, +, \{\{n,m\} \\ \wedge (n \leq i < m)\}\} \end{matrix} \\ last_{SP}(RP_2), & RO \in \{\{n\}, \{n,m\}\} \wedge (i < n) \\ last_{SP}(NRP_1), & \begin{matrix} RO \in \{?, \hat{}\,, \{\{n\} \wedge (i = n)\}, \\ \{\{n,m\} \wedge (i = m)\}\} \\ \vee\ last_{SP}(RP_2) = \perp \end{matrix} \end{cases}$$

$$find_{RHC}\Big(first_{SP}(NRP_3), RO\Big) = \quad (11)$$

$$\begin{cases} last_{RHC}(RP_2).first_{RHC}(NRP_3), & RP_2 \text{ is matched} \\ last_{RHC}(NRP_1).(RP_2)RO.first_{RHC}(NRP_3), & \begin{matrix} RP_2 \text{ is not matched} \\ \wedge\ last_{RHC}(RP_2) = RP_2 \end{matrix} \\ last_{RHC}(NRP_1).first_{RHC}(NRP_3), & \text{else} \end{cases}$$

$$find_{RHC}\Big(first_{SP}(RP_2), RO\Big) = \quad (12)$$

$$\begin{cases} last_{RHC}(RP_2).first_{RHC}(RP_2), & RP_2 \text{ was matched} \\ last_{RHC}(NRP_1).first_{RHC}(RP_2), & RP_2 \text{ was not matched} \end{cases}$$

$$first_{SP}(RP) = \quad (13)$$

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

$$last_{SP}(RP) = \tag{14}$$
$$\begin{cases} last_{SP}(NRP_3), & last_{SP}(NRP_3) \neq \perp \\ \{last_{SP}(NRP_1), last_{SP}(RP_2)\}, & \begin{array}{l} last_{SP}(NRP_3) = \perp \\ \wedge RO \in \{*, ?, ^\wedge\} \end{array} \\ last_{SP}(RP_2), & \begin{array}{l} last_{SP}(NRP_3) = \perp \\ \wedge last_{SP}(RP_2) \neq \perp \\ \wedge RO \in \{+, \{n\}, \{n, m\}\} \end{array} \\ last_{SP}(NRP_1), & \text{else} \end{cases}$$

$$first_{RHC}(RP) = \tag{15}$$
$$\begin{cases} first_{RHC}(NRP_1), & first_{RHC}(NRP_1) \neq NRP_1 \\ NRP_1.first_{RHC}(RP_2), & \begin{array}{l} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_2 \text{ is matched} \end{array} \\ NRP_1.(RP_2)RO.first_{RHC}(NRP_3), & \begin{array}{l} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_2 \text{ is not matched} \\ \wedge first_{RHC}(RP_2) = RP_2 \end{array} \\ NRP_1.first_{RHC}(NRP_3), & \text{else} \end{cases}$$

$$last_{RHC}(RP) = \tag{16}$$
$$\begin{cases} last_{RHC}(NRP_3), & first_{RHC}(NRP_3) \neq NRP_3 \\ last_{RHC}(RP_2).NRP_3, & \begin{array}{l} last_{RHC}(NRP_3) = NRP_3 \\ \wedge RP_2 \text{ is matched} \end{array} \\ last_{RHC}(NRP_1).(RP_2)RO.NRP_3, & \begin{array}{l} last_{RHC}(NRP_3) = NRP_3 \\ \wedge RP_2 \text{ is not matched} \\ \wedge last_{RHC}(RP_2) = RP_2 \end{array} \\ last_{RHC}(NRP_1).NRP_3, & \text{else} \end{cases}$$

3) Two-operands Operator Regex Pattern Relations

Definition A3. Let "*RP*" be a two-operand operator regex pattern, which is described by equation 17.

$$RP = NRP_1(RP_2|RP_3)NRP_4 \tag{17}$$

then:

$$find_{PSP}\Big(first_{SP}(NRP_4), |\Big) = \tag{18}$$
$$\begin{cases} \{last_{SP}(RP_2), last_{SP}(RP_3)\}, & \begin{array}{l} last_{SP}(RP_2) \neq \perp \\ \vee last_{SP}(RP_3) \neq \perp \end{array} \\ last_{SP}(NRP_1), & \text{else} \end{cases}$$

$$find_{PSP}\Big(first_{SP}(RP_3), |\Big) = last_{SP}(NRP_1) \tag{19}$$

$$find_{PSP}\Big(first_{SP}(RP_2), |\Big) = last_{SP}(NRP_1) \tag{20}$$

$$find_{RHC}\Big(first_{SP}(NRP_4), |\Big) = \tag{21}$$
$$\begin{cases} last_{RHC}(RP_3).first_{RHC}(NRP_4), & RP_3 \text{ is matched} \\ last_{RHC}(RP_2).first_{RHC}(NRP_4), & RP_2 \text{ is matched} \\ last_{RHC}(NRP_1).RP_3.first_{RHC}(NRP_4), & \begin{array}{r} RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge last_{RHC}(RP_2) \neq RP_2 \\ \wedge last_{RHC}(RP_3) = RP_3 \end{array} \\ last_{RHC}(NRP_1).RP_2.first_{RHC}(NRP_4), & \begin{array}{r} RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge last_{RHC}(RP_2) = RP_2 \\ \wedge last_{RHC}(RP_3) \neq RP_3 \end{array} \\ \begin{array}{l} \{last_{RHC}(NRP_1).RP_2.first_{RHC}(NRP_4) \\ last_{RHC}(NRP_1).RP_3.first_{RHC}(NRP_4)\} \end{array}, & \text{else} \end{cases}$$

$$find_{RHC}\Big(first_{SP}(RP_3), |\Big) = last_{RHC}(NRP_1).first_{RHC}(RP_3) \tag{22}$$

$$find_{RHC}\Big(first_{SP}(RP_2), |\Big) = last_{RHC}(NRP_1).first_{RHC}(RP_2) \tag{23}$$

$$first_{SP}(RP) = \tag{24}$$
$$\begin{cases} first_{SP}(NRP_1), & first_{SP}(NRP_1) \neq \perp \\ \{first_{SP}(RP_2), first_{SP}(RP_3)\}, & \begin{array}{l} first_{SP}(NRP_1) = \perp \\ \wedge(first_{SP}(RP_2) \neq \perp \\ \vee first_{SP}(RP_3) \neq \perp) \end{array} \\ first_{SP}(NRP_4), & \text{else} \end{cases}$$

$$last_{SP}(RP) = \tag{25}$$
$$\begin{cases} last_{SP}(NRP_4), & last_{SP}(NRP_4) \neq \perp \\ \{last_{SP}(RP_2), last_{SP}(RP_3)\}, & \begin{array}{l} last_{SP}(NRP_4) = \perp \\ \wedge(last_{SP}(RP_2) \neq \perp \\ \vee last_{SP}(RP_3) \neq \perp) \end{array} \\ last_{SP}(NRP_1), & \text{else} \end{cases}$$

$$first_{RHC}(RP) = \begin{cases} first_{RHC}(NRP_1), & first_{RHC}(NRP_1) \neq NRP_1 \\ NRP_1.first_{RHC}(RP_2), & \begin{array}{c} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_2 \text{ is matched} \end{array} \\ NRP_1.first_{RHC}(RP_3), & \begin{array}{c} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_3 \text{ is matched} \end{array} \\ NRP_1.RP_2.first_{RHC}(NRP_4), & \begin{array}{c} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge first_{RHC}(RP_2) = RP_2 \\ \wedge first_{RHC}(RP_3) \neq RP_3 \end{array} \\ NRP_1.RP_3.first_{RHC}(NRP_4), & \begin{array}{c} first_{RHC}(NRP_1) = NRP_1 \\ \wedge RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge first_{RHC}(RP_2) \neq RP_2 \\ \wedge first_{RHC}(RP_3) = RP_3 \end{array} \\ \begin{Bmatrix} NRP_1.RP_2.first_{RHC}(NRP_4) \\ NRP_1.RP_3.first_{RHC}(NRP_4) \end{Bmatrix}, & \text{else} \end{cases} \tag{26}$$

$$last_{RHC}(RP) = \begin{cases} last_{RHC}(NRP_4), & last_{RHC}(NRP_4) \neq NRP_4 \\ last_{RHC}(RP_2).NRP_4, & \begin{array}{c} last_{RHC}(NRP_4) = NRP_4 \\ \wedge RP_2 \text{ is matched} \end{array} \\ last_{RHC}(RP_3).NRP_4, & \begin{array}{c} last_{RHC}(NRP_4) = NRP_4 \\ \wedge RP_3 \text{ is matched} \end{array} \\ last_{RHC}(NRP_1).RP_2.NRP_4, & \begin{array}{c} last_{RHC}(NRP_4) = NRP_4 \\ \wedge RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge last_{RHC}(RP_2) = RP_2 \\ \wedge last_{RHC}(RP_3) \neq RP_3 \end{array} \\ last_{RHC}(NRP_1).RP_3.NRP_4, & \begin{array}{c} last_{RHC}(NRP_4) = NRP_4 \\ \wedge RP_2 \text{ and } RP_3 \\ \text{are not matched} \\ \wedge last_{RHC}(RP_2) \neq RP_2 \\ \wedge last_{RHC}(RP_3) = RP_3 \end{array} \\ \begin{Bmatrix} last_{RHC}(NRP_1).RP_2.NRP_4 \\ last_{RHC}(NRP_1).RP_3.NRP_4 \end{Bmatrix}, & \text{else} \end{cases} \tag{27}$$