SA2PX: A Tool to Translate SpamAssassin Regular Expression Rules to POSIX

Shi Pu, Cheng-Chung Tan, Jyh-Charn Liu
Computer Science and Engineering Department, Texas A&M University
{shipu, jonastan, liu}@cse.tamu.edu

ABSTRACT

This paper presents a software tool *sa2px* to translate regular expressions (regexps) in SpamAssassin (SA) rules into the POSIX format. The translated regexps can be implemented on different platforms, so that one could better separate the composition process of spam filtering rules from the on-line operations. *Sa2px* is consisted of three layers of functions. The first layer is responsible for translating plugins and special formats to their equivalent *basic* SA formats. The second layer uses a syntax conversion approach to translate *basic* SA rules to the POSIX format. The third layer uses a *backward grouping algorithm* to group multiple regexps together so that they can be packed into a DFA table using Flex or similar tools. Overall, *sa2px* can translate regexps in the whole rule set *(uri, body, header, rawbody* and *ReplaceTags plugin*), and the translation rate of 1115 SA regexp rules is 84.5%. In comparison, sa-compile can translate 296 rules of 453 *body* rules. The translated rules are then clustered into several main groups, except for some cases in which the regexp structures led to explosive state growth. Finally, DFA tables and (action number, *rule name*) pairs are generated. Experimental results show that the DFA table based implementation of these translated regexps cut down 66% of the execution time of the Perl (with sa-compile activated*)* based string scanning under process-level parallelization environment.

1. Introduction

SpamAssassin (SA) [6] is a sophisticated email content filtering system. Implemented in Perl, SA has an expandable and expressive representation system for composition of sophisticated spam detection rules (estimated around 1791 rules in SA release V.3.2.5). These rules are assigned scores by an off-line artificial neural net, and a Bayes trainer (sa-learn) assigns scores of (spamming) tokens using a set of user-defined *spam* (junk email) and *ham* (regular email). In online operations, an email is labeled as a *spam* by SA if its spam score is higher than 5 (by default). With the ever growing of the spam problem, i.e., 50% ~ 80% of email bandwidth waste [1, 2], it is important to expand the content filtering capabilities to more strategic locations in order to defeat those and future generation of spamming tools, e.g., Srizbi Trojan [5] with Reactor Mailer [9], Dark Mailer [9], Send Blaster [3], and Send-Safe Mailer [4, 9].

As of now, SA can run the entire rule set directly, or using the sa-compile utility tool to accelerate some *body rules*, but not any other rules, such as header, URI, or plugin rules. Sa-compile extracts 677 fixed strings from 296 out of 453 *body* rules to form a kind of hash table. That is, when a string match occurs, the corresponding SA rules of the string are triggered to perform NFA-based matching for that line. The 677 fixed strings can be further compiled into four deterministic finite automata (DFAs) using the lexer "re2c" [41], and the compiled codes can be invoked through Perl eXternal subroutine "Perl XS".

In this paper, we propose a regexp translation tool "*sa2px*" to translate regexps in SA rules into the POSIX format. *Sa2px* has a three-layer architecture. The first layer translates plugins and special formats to their equivalent *basic* SA formats. The second layer uses a syntax conversion approach to translate *basic* SA rules to the POSIX format. The third layer is designed to group multiple rules, based on a *backward grouping algorithm,* so that they can be implemented into a DFA table using Flex or similar tools. In comparison with the sa-compile, *sa2px* can convert 426 of the 453 *body* rules, and it can handle regexps in the whole SA rule set *(uri, body, header, rawbody*, and *ReplaceTags plugin* [35]). Results from experiments on public spam corpus Trec2007 [47] showed that DFAs generated by the *sa2px*-Flex tool chain correctly flag spamming contents. We further found that the hit distributions of the SA regexp rules and their translated versions for different detection goals, i.e., adult, medication, education etc., are consistent. The execution time experiment illustrates that the parallelized DFA-based binary scanners are 3-fold faster than SA NFA-based scanning, with sa-compile activated. For deployment consideration, *sa2px* can be either used as a dedicated regexp scanning engine for the SA, or its derived DFA tables could be deployed to a gateway packet level email contents filter [7].

The remainder of this paper is organized as follows: In section 2, the system framework of *sa2px* is introduced. Sections 3 and 4 discuss the Perl regexp syntax translation and the grouping algorithm, respectively. Experimental results and evaluation are illustrated in section 5. Section 6 discusses related work. Future work and conclusion are drawn in Section 7.

2. System Framework

SA rules can be roughly divided into three major types of operations: (1) list look up (DNSBL, URIBL, Local black/white list, reputation based), (2) content scanning and matching, and (3) miscellaneous operations (Bayesian filter, sender authentication, Perl functions). Except for a small number of cases, e.g., sender authentication, sending time, etc., the SA is largely based on string parsing, table lookup, and off-line statistical training for score assignments of patterns extracted from the parsing subsystem. After it is started, SA loads rules into several *baseline*

CEAS 2009 – Sixth Conference on Email and Anti-Spam, July 16-17, 2009, Mountain View, California USA

lists, according to the email fields that need to be tested and it invokes Perl to construct one NFA for each regexp. Some *baseline* rules are combined into *metarules* using Boolean operators. They use the *baseline* rules for detection, but have their own score assignments. Without considering *rule priority configuration, Bayes auto-learning module,* and *shortcircuit's cyclical matching*, SA rules and their relationship with the email scanning procedure is illustrated in Figure 1. After an email copy is decomposed into different constructs, each construct is scanned by its corresponding "test", e.g., the header is scanned by rules contained in "header_test", and a particular construct may be subject to multiple iterations of checking.

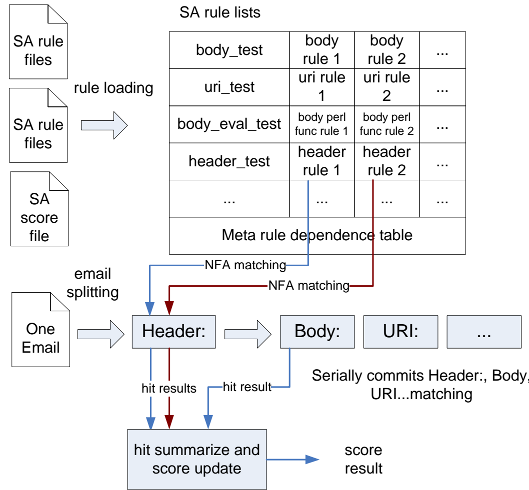

Figure 1: Existing SA regexp rules structures

As long as one can implement the semantics of checking rules, for on-line operations they can be implemented in the most cost-effective form so that the overall run-time costs, together with processing response time can be improved, especially for large volume enterprise email scans. Due to space limit and keep our discussion focused, we will use some rules to touch on this issue as a part of evaluation work. A preliminary analysis on the rule files of SA V. 3.2.5 shows that a total of 1115 regexp (62%) rules were counted from 1791 SA rules. Although not formally documented, we found that the general format of those regexps can be summarized as the one shown in Figure 2. *Rule type* assigns rule's test field, *rule name* is rule's unique ID, and a regexp starts and ends with '/' is the *rule body*.

(*Rule Type*) (header || body || rawbody || uri || full || …)	(*Rule Name*) (\s)+_{0,2}{\S}+	(*Rule Body*) if <rule type> == header: test field in email header =~ / + Perl RE + / + [ismx] else: / + Perl RE + / + [ismx]

Figure 2: The basic format of SA regexp rules

Most regexp rules in SA rule set follow this *basic* format, but some rules may use different symbols to separate its *rule body*. Furthermore, plugin rules may have their own formats and defined macros (for example, macro "<B>" represents "[bB]" in *ReplaceTag* plugin rules). All the *basic* format rules, and these exceptional formats that are not supported by sa-compile, will be translated by *sa2px*. Once in the POSIX format, these rules can be further used to produce DFA transition tables or input files of

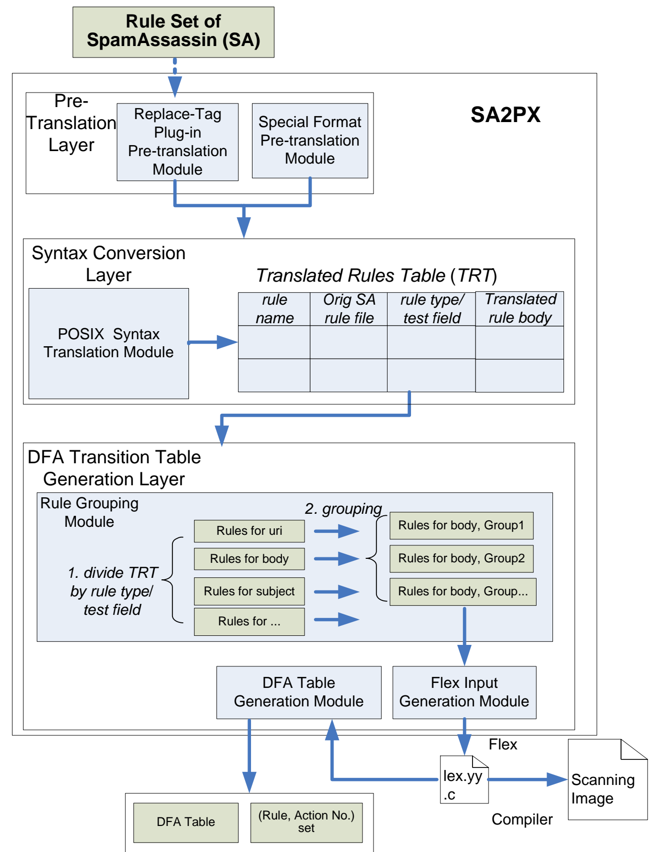

Figure 3: Three-layer sa2px framework

well-known lexers such as Flex and RE2C. Of course, the DFA table will become excessively large when too many regexps are merged, but clearly the system would have improved processing efficiency when multiple regexps into a few DFAs.

Referring to Figure 3, *sa2px* has a layered architecture: (1) plugin and special format pre-translation layer, (2) syntax conversion layer, and (3) DFA transition table generation.

The goal of the first layer is to translate plugin and special formats to their equivalent *basic* formats. Some third-party plugins use their own defined symbols to construct their regexp rules. A widely used plugin instance is *ReplaceTag*, which defines rules to detect the polymorphism of specific letters. For example, the class "*[wv]*" in Perl regexp syntax can be represented by macro symbol "*<W>*", then plugin *ReplaceTag* can use "*<W>*" to construct its rules. Plugin pre-translation transforms those plugin-defined patterns to their equivalent Perl regexp syntax. Furthermore, except for the most general *rule body* separator '/', SA is also compatible with various *rule body* separators, such as '’', '!', '%', '{'. The special format pre-translation is to detect those various *rule body* separators and normalize them to the most general one '/'. This step is necessary because *sa2px* use the general *rule body* separator to identify whether a SA rule is a regexp rule.

The syntax conversion layer uses a mapping table indexed by Perl regexp syntax patterns, such as "*\W*", "*\B*", "*(:pattern description)*" to translate rule patterns into their POSIX counterparts. For example, the entry of "*\w*" that means a "word"

character (alphanumeric plus "_") in Perl regexp syntax would be "*[a-zA-Z0-9_]*". Moreover, the POSIX syntax translation module constructs a *Translated Rule Table* (*TRT*) that saves *rule name*, the file location of rules in SA rule set, *rule type* or *target field* of header (if *rule type* is header), and the translated *rule body*.

The DFA transition table generation layer divides rules saved in *TRT* based on their *rule type* or *target field*, so that they can be grouped by the *Backward Grouping Algorithm* (*BGA*). *BGA* evaluates the interactive degree of rules to determine which rules to be grouped together. Then, the Flex input generation module creates a Flex input file for each cluster of grouped POSIX regexp rules. Flex allows one to define an *action* A_i for a regexp rule r_i and then creates its action number i. In the Flex generated lexer source file lex.yy.c, an action "switch" statement is used to make the one-to-one mapping between i and A_i. Since every action number is uniquely mapped to one accept state in the DFA transition table, pairs (i, v_{ri}) also represent a one-to-one mapping relationship between accept states and rule name. We utilize this feature to identify a matched translated SA rule as follows. Given a rule r_i, we declare a variable v_{ri} using the *rule name* of r_i, and then assign an (arbitrary) operation for v_{ri}, e.g., v_{ri} ++, as A_i. After Flex is invoked, it builds the mappings – $\{(i, A_i), \forall i\}$ and $\{(i, v_{ri}), \forall i\}$. Then DFA table generation module extracts the DFA transition table "*struct yy_trans_info yy_transition[ ]*" and $\{(i, v_{ri}), \forall i \in [1,N]\}$ from lex.yy.c as its output to flag the matched rules at the run time.

3. Perl Syntax Translation

Perl syntax translation consists of SA rule format extraction, and translation of a translatable rule to its new format using a mapping policy table. According to the *basic* format shown in Figure 2, for a Perl regexp rule, we extract its *rule type* and *test field* (if available), the *rule body* embedded by the *rule body* separators, and the modifier ("/ismx") at the end of the *rule body*. The process is depicted in Figure 4 without including an *error* state which indicates that the rule is not a regexp. Then the rule body is passed to the POSIX syntax translation routine.

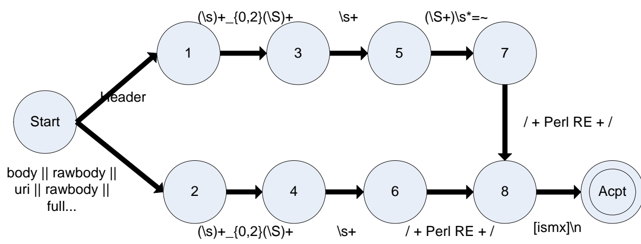

Figure 4: pseudo-DFA of format extraction process

First, the modifier pattern is extracted, with a rule case-sensitive flag (for "/i") and a single line flag (for "/s") are marked, so that it would not be lost in subsequent steps. Next, the syntax translation process is done through three phases of translation. At the first phase, translatable regexp patterns (in Perl format) are mapped to their corresponding POSIX patterns using the mapping policy table listed in Appendix Table 6, in which the left column lists all the Perl regexp syntax grammars [34]. The second phase translates case insensitive letters to their canonical forms. For instance, "*abc/i*" is exchanged to "*[aA][bB][cC]*", where "*i*" is the case insensitive modifier. In the third phase, nested classes are rewritten, so that the nesting symbols "[" and "]" are rewritten to "(" and ")", respectively, e.g., "*[[aA]bc[d-e]]*" is transformed to "*([aA]|[bc]|[d-e])*". Otherwise, when "[" and "]" are nested, these class symbols in the inner layer will be mistaken as quoted symbol "[" and "]". It is more complicated to rewrite nested positive-negative signs, e.g., "*[^[A-B]ab[^Cc][E-F]gh]*". In this case the nested negative class "*[^Cc]*" should become positive after rewriting because it also contains a negative sign in the outer class. The rewritten pattern for the mentioned example should be "*([^A-BabE-Fgh] | [Cc])*".

An example of syntax translation process is illustrated in Figure 5. The whole process involves modules in Layer 1 and Layer 2. First, the *rule body* starting symbol '*m*" and ending symbol '*i*' of *uri* rule FU_LONG_QUERY3 are substituted with "/" and "/i". And "<M><E><D><S>" defined by *ReplaceTags* plug-in are replaced by corresponding Perl regexp syntax strings. After processed by modules in Layer 1, all regexp rules satisfy standard Perl regexp syntax and follow the *basic* format. Then, Perl regexp syntax is translated to POSIX syntax. For example, "\b" in rule "*body VIA_GAP_GRA /\bvia.gra\b/i*" would be substituted by "*[\t\n\r\f\v\\\./\$!@#%&*\(\)\[\]\{\}=+- ,<>\?\"\'\/&\^~`]*", and every letter is translated to its canonical form. At the same time, non-Perl regexp rules, such as *meta* rules, Perl function rules, DKIM rules, and etc., are excluded.

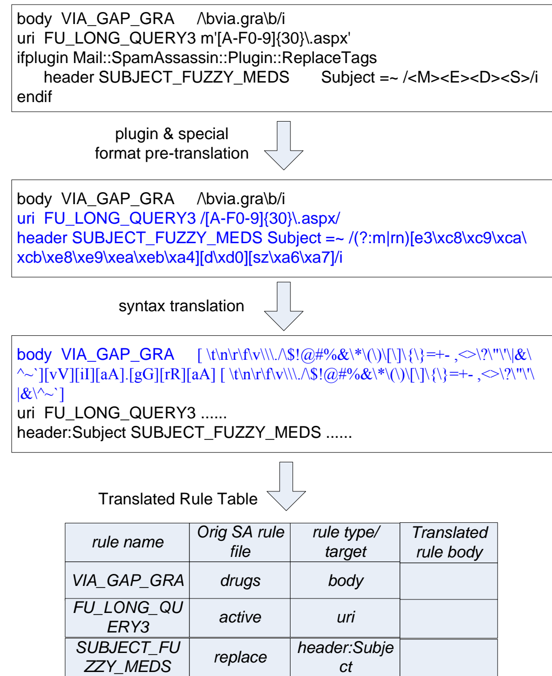

Figure 5: a syntax translation example

Referring to Appendix Table 6, the syntax translation technique would introduce some semantic differences between the original and translated versions. While most of the discrepancies are fairly benign, a particularly subtle issue related to the widely used *zero-width pattern* in SA regexp rules, i.e., "*(?:pattern)*", which does not extract a matched pattern, but it does not have a fully equivalent POSIX regexp pattern. We opted to translate it to a very similar POSIX pattern "*(pattern)*", which does extract the matched pattern. As a result, the translation may cause rule mismatching when the last input symbol set of a rule *A* overlaps

with first input symbol set of another rule *B*. To partially solve the rule overlapping problem we implemented a simple rewrite technique by inserting trailing context before the last symbol of rule *A*. For example, rule *A* is "*[ab]c(efg)*" and rule *B* is "*[ef]fgij*", so that "*efg*" in A and "*[ef]fg*" in B overlap; then rule A can be rewrite to "*[ab]c/(efg)*". Being limited by the definition of trailing context, this scheme can only be applied when "*(?:pattern)*", "*(?=pattern)*" or "*(?<=pattern)*" is in the tail of the regexp.

When the DFA table generation module is active, *sa2px* will discard regexp rules that are known to overflow Flex's buffer. Other rules also excluded are those that can be done using simple logic, e.g., checking if a specific email field is empty (*__HAS_ANY_URI*, *MISSING_DATE*, *__NONEMPTY_BODY*, *etc.*), or negative matching rules "!~" used in header checking ("=~" is considered a positive matching rule).

4. Backward Grouping Algorithm

Assuming the string length denoted by *l* and regexp length *n*, the processing time complexity for DFA matching is O(*l*), but is O(n^2) for NFA matching in the worst case [45]. If the rule set contains *m* regexp rules, the time complexity of serially applying DFA matching for each rule separately is O(*m***l*), once all rules are combined together, the time complexity could be reduced to O(*l*). In terms of the memory overhead, when NFA is transformed to DFA, the states number is enlarged from O(n) to O(Σ^n). (Σ is the input symbols). Once the *m* regexp rules are combined together, the memory complexity grows from O(m* Σ^n) to O(Σ^{mn}) [37].

A key factor in combination of rules into compact DFA is the *interaction* [31] among rules. Consider a rule *IP_LINK_PLUS*:

[hH][tT][tT][pP][sS]?: \/\/[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+.{0,20}([cC][gG][iI]|[cC][lL][iI][cC][kK]|[aA][dD][sS]|[iI][dD]=).

Without the term "*.{0,20}*", the number of DFA states is 28. On the other hand, the DFA for this rule has 2317 states, most of which are allocated to the wildcard with length restriction "*.{0,20}*" because a state in this DFA needs to represent the combination of the current match of "*.{0,20}*" and any previous partial match [45]. There is a multiple effect when a large DFA like this one is combined with even a very small rule, such as the following *HTTP_EXCESSIVE_ESCAPES* rule:

[hH][tT][tT][pP][sS]?:\/\/[^ \t\n\r\f\v]*%(3[0-9]|[46][1-9a-fA-F]|[57]([0-9]|[aA]))

Although the DFA of *HTTP_EXCESSIVE_ESCAPES* only has 18 states, the total number of DFA states becomes 6034 when it is combined with *IP_LINK_PLUS.*

This example illustrates the strong interaction between two rules. Similar to the original *IP_LINK_PLUS*'s DFA, when state transiting to the one that starts to match "*.{0,20}*", following states need to track all possible previous partial matching and current matching for *IP_LINK_PLUS*, meanwhile keeping the tracking of whether *HTTP_EXCESSIVE_ESCAPES* rule is matched, which requires the replication of *HTTP_EXCESSIVE_ESCAPES*'s automata in each following state.

Next, we consider some basic statistics of different patterns. Table 1 illustrates the percentage of regexp rules containing wildcard ("*., +, ?, **"), wildcard with length restriction ("*.{n,m}*"), anything but class ("*[^…]*"), wildcard without length restriction ("*.**"), and class with length restriction ("*[…]{n,m}*") in the translated SA regexp rule set. As shown in Table 1, wildcards and class "*[…]*"are widely used in SA regexp rules, but ".*" is used only in few SA rules. Syntax structures "*.{n,m}*", "*[…]{n,m}*" and "*[^…]{n,m}*" are much more popular ways to implement ambiguous matching. Despite the high popularity of wildcards and class, this table indicates that the percentage of regexp rules that can cause DFA states exponential growth will not be large. This is because when the appended length restriction (*n-m*) is small or equals to 1, the state growth brought by wildcard or class in rule combination is polynomial [46].

Table 1: Translated regexps summary

	translated SA rules
number of regexp rule analyzed	942
percentage of rules with wildcards "*.+?**"	59.4%
percentage of rules with "*.{n,m}*"	5.5%
percentage of rules with "*[^…]*"	21.0%
percentage of rules with "*.**"	0.71%
percentage of rules with "*[…]{n,m}*"	13.3%
average number of "*.+?**" per rule	3.19
average number of "*.{n,m}*" per rule	1.48
average number of "*[^…]*" per rule	3.31
average number of "*.**" per rule	1.14
average number of "*[…]{n,m}*" per rule	2.46

An interaction graph-based grouping algorithm [31] has been proposed for grouping Linux L7- filter (70 regexp rules) and Bro's rules (222 rules), which is composed of an interaction graph construction phase and a regexp combination phase. Given *m* regexp rules, the time complexity for constructing an interaction graph is O(m^2*2^P) (*P* is the number of states in NFA which contains two rules, and O(2^P) is the time complexity of NFA to DFA conversion without optimization). And the time complexity for combination process is O(*m**2^Q) (*Q* >> *P* because the group candidate have far more than two regexp rules). Therefore, the time complexity of the whole algorithm is O(m^2*2^P) + O(*m**2^Q). When m is small, O(m^2*2^P) << O(*m**2^Q) and thus could be neglected. However, when *m* becomes larger, O(m^2*2^P) can not be ignored.

To take advantage of the fact that only few regexp rules cause DFA states exponential growth in the whole regexp rule set, we propose a *Backward Grouping Algorithm* (*BGA*) which removes regexp having the largest interaction to other members in each group. *BGA* reduces the whole compilation time from [O(m^2 * 2^P) + O(*m**2^Q)] to [O(*m**2^P) + O(*g**2^Q)].

The most popular rule types in SA regexp rule set are *body*, *uri*, *rawbody*, and *header*. Especially, *header* type rules can be further divided based on the target test fields, such as *Subject*, *From*, *Date*, *etc.*. According to [38], we select 12 heavily used test fields in email header, they are: *header:Subject*, *header:From*, *header:Date, header:X-Mailer, header:Content-Type, header: Return-Path, header:Message-ID, header:X-MIMEOLE,*

header:X-Spam-Relays-Untrusted, header:Received, header: User-Agent, and header:Auto-Submitted. Therefore, there are 15 groups (*body*, *uri*, *rawbody*, and 12 groups for *header*) before grouping.

Each group is assigned to an *Initial Group* (*IG*). *BGA* is essentially a local greedy search heuristic, which iteratively removes regexps that leads to the greatest reduction on the number of DFA states for each *IG* until the final group size reaches a target size. Since the memory size of DFA transition table is proportional to the number of uncompressed DFA states due to the constant size of eligible input symbols, *BGA* uses the number of states in a regexp's DFA as the trigger for grouping.

```
Class regexp_group{
        regexp[] regexp_array;
        int regexp_numbers;
        int DFA_states_number;
        minus(regexp_group);
        add(regexp_group);
};

regexp_group[] FG_array;
Deque<regexp_group> GD;
typedef  rule regexp_group;
Table DSRT(string rulename,
        int reduced_states_number);

Class Initial_Group{
  regexp_group[] mains_array;
  regexp_group[] fragments_array;
  regexp_group orig;
};
```

Figure 6: The main data structures used in *BGA*

Let *fragment*s denote regexps being removed from a group, and *main (part)* the set of remaining regexps, the data structures used in this algorithm and their relationship are given in Figure 6. Class *regexp_group* is the basic structure to represent a group of regexp rules. Its operator *minus* (*add*) means removing (merging) the sub-group (another group) of regexp out of (into) the *regexp_group*. A single rule is an extreme case of *regexp_group* which has only one element in *regexp_array[]*.An array *FG_array* (file group array), a deque of regexp_group *GD*, and a table *DSRT* are used for grouping of original groups saved in *IG*.*orig*. *FG_array* stores structure information in original SA rule set. *GD* is designed as a temporary container in the grouping process. Each *DSRT* records the reduction of states number when a rule is removed from the *IG*.*orig* or one element in *FG_array*.

BGA is highlighted as follows. Before grouping, an initial step needs to decide whether or not a group is separated based on original SA rule set structure. This step is necessary for groups that have numerous regexp rules, e.g., the *body rule group* (426 rules). When too many regexps that have interactions are combined, the DFA generation process will slow down dramatically. To cut down the probability of such situation, the group whose DFA states number is higher than a threshold $Threshold_{split}$ will be split based on original SA rule set structure (.cf files) into *FG_array*.

If the number of DFA states of any elements in the *FG_array* is larger than a threshold $Threshold_{grouping}$, grouping starts with computing and sorting of a *DFA states reduction table* (*DSRT*). Let the element in *FG_array* be denoted as *fg*, the number of rules in *fg* as *N*, the ith regexp rule in *fg* as R_i ($1 \le i \le N$), and the regexp to DFA transformation function as *F*, whose output is the number of states in DFA. Then the process can be described as follows:

(1) $\forall R \in fg$, calculate $S_i = fg - R_i$ ($1 \le i \le N$);

(2) $\forall S_i$, compute $F(S_i)$ and insert $\{R_i, F(S_i)\}$ pair into $DSRT_{fg}$ in the descending order of $F(S_i)$.

The next step distinguishes BGA from the algorithm in [31]. Instead of merging regexps one by one, *BGA* aims to remove regexp rules which cause the largest state reduction.

(1). $\forall fg$, $\exists R_{max}$ in $DSRT_{fg}$, locate the entry $\{R_{max}, F(S_{Rmax})\}$ s.t. $F(S_{Rmax})$ = minimum of all $F(S_i)$ and remove R_{max} out from *fg*.

(2). Get two new groups $\{R_{max}\}$ and $\{fg - R_{max}\}$ and delete $\{R_{max}, F(S_{Rmax})\}$ from $DSRT_G$.

(3). Repeat the steps above until the DFA states nubmer of $\{fg - R_{max}\} < Threshold_{grouping}$.

(4). Merge split *fragment*s together from the smallest ones, until the number of DFA states of the merged fragments > $Threshold_{grouping}$.

(5) *IG*.*mains_array*[] and *IG.fragments_array*[] respectively keep track of *main part*s and *fragment*s of all *fg*s in FG_array_{IG}. Moreover, the *IG.fragments_array* is sorted in the ascending order of DFA states number.

For those *IG*s that have been split based on original SA rule set structure, some *fragment*s may not have interaction with others in different *fg*s. Finally, *fragment*s stored in *IG.fragments_array* need to be merged together. Let the length of fragments array be denoted as *L*, and a temporary *regexp_group* f is assigned to the first element of fragments array. This process can be described as:

$\forall i$ = 1,2⋯L-1, f += *IG.fragments_array*[i] if f's DFA states number < $Threshold_{grouping}$. *IG.fragments_array* [0] = f. Repeat the step iteratively until no element in *IG.fragments_array* can be merged together.

Compared to the forward grouping algorithm that uses the number of states to detect interactions between regexps, *BGA* may be less accurate but it is much more efficient because it does not require the grouping process to go through the whole interaction graph. As such, *BGA* is most suitable for the scenario in which only small proportions of regexps that have strong interaction with others. The threshold parameter $Threshold_{grouping}$ is reversely proportional to the number of *fragment*s. That said, if strong interaction widely exists among those *fragment* groups, it is practically impossible to combine them together.

Combining rule without detection rule subsumption may cause missing of rule hits. For example, rule "*FUZZY_CPILL*", "*FB_CIALIS_LEO3*", and "*__DRUGS_ERECTILE3*" all can catch input strings like "*Cialis*". If these three rules are combined together, only one of them can be matched at the same time. This issue is not considered in the current design, and it will be considered in the further refinement.

5. Evaluation

In total, 942 of 1115 SA regexp rules are translated to their POSIX patterns, or the translation rate of 84.5%. The grouping results under different granularities are summarized in Table 2.

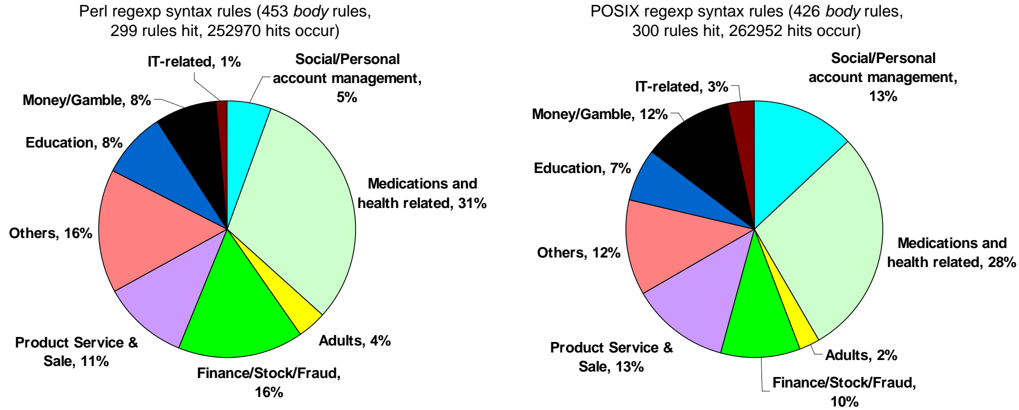

Figure 7: SA rules hits distribution diagram

Table 2: Summary of grouping results

$Thres_{grouping}$ of states number	group numbers				Total states number
	body	header (12 fields)	uri	rawbody	
No grouping	1	12	1	1	952772
10000	8	16	2	1	120324
8000	9	19	2	1	99148
6000	10	20	2	1	89583
4000	13	20	2	1	90610

In this table, comparing the clusters without grouping, it is found that even when all of the translated rules are grouped under the largest granularity ($Threshold_{grouping}$ = 10000), the number of states is reduced sharply. When the threshold $Threshold_{grouping}$ is gradually reduced from 10000 to 6000, the number of groups grows smaller than that when $Threshold_{grouping}$ drops from 6000 to 4000. It suggests that 6000~8000 should be a good range for $Threshold_{grouping}$. When $Threshold_{grouping}$'s value drops to 4000, the fragments become more, and the total DFA states number slightly grows. The growth is brought by fragment merging whose purpose is to reduce the number of groups as many as possible, meanwhile control the number of states within the limitation of $Threshold_{grouping}$.

Table 3: Group results comparison (no grouping and $Threshold_{grouping}$ = 8000)

Test area/ Target field	Rules	DFA states/(before grouping)	DFA states/(after grouping)	group numbers
body	425	420594	48547	9
uri	31	209547	5509	2
header/subject	158	25288	12135	3
header/X-Spam-RelaysUntrusted	54	282642	18256	6

The comparison between no grouping and grouping when $Threshold_{grouping}$ = 8000 and $Threshold_{split}$ = 10000 is shown in Table 3, in which only Initial Groups that trigger grouping are illustrated. To evaluate the effect of *BGA*, Table 4 lists the grouping results of *body* rule group.

Table 4: Grouping results of *body* rule group

Test area/ Target field		Rules	DFA states
body	fraud-main	55	5215
	drug-main	66	5143
	misc-main	100	7352
	active-main	185	7483
	fragment-1	4	872
	fragment-2	1	1169
	fragment-3	2	6560
	fragment-4	6	7330
	fragment-5	6	7423

In Table 3 and Table 4, it is shown that most *fragment*s containing only few rules. This fact suggests that only few rules have strong interaction with other rules in the large rule set, making *BGA* suitable for grouping SA regexp rule set. We also evaluate the memory overhead of the generated DFA transition table. Furthermore, after applying the compression technique introduced in [40], the memory consuming of one DFA transition table is relatively small. For example, the memory size of *fraud-main* group's DFA transition table that has 5215 states is 213KB (estimated) on 32-bit platform.

5.1 Hit Distribution Comparison

The hits frequency comparison experiment is to test if there is significant difference between the original SA and the binary images generated by *sa2px*, based on the TREC2007 corpus. Only *body* type regexp rules scanning are activated in SA, and the number of activated rules is 453.

When $Threshold_{grouping}$ = 8000, a total of 9 binary images cover all the rules in *body* type IG, and the number of effective rules is 426. The same as SA's rule hitting policy, one rule in the binary images' scanning process for the same email sample will not be hit twice. Hit distributions for SA and the *sa2px* generated binary images are displayed in Figure 7. Despite the differences in their total number of hits, the results shown in the figure illustrate that the DFAs generated by sa2px capture a great amount of spamming contents. The similarity between two hit distributions

shows that the translated DFAs have consistent detection performance as the original rule set. The discrepancy is likely contributed by the rule overlapping and subsumption problem.

5.2 Execution Time Comparison

In this experiment, we compare the execution time of DFAs generated by *sa2px* and the SA with the sa-compile activated. The experiment process is displayed in Figure 8 and the *Linux Redhat* based experiment platform is equipped with two Intel Xeon quad-cores 2.66GHZ and 6GB Memory. We use the first 1000 emails in TREC2007 as the input, and repeat this experiment 20 times. For simplicity, we only use *body* rules in this experiment, and the results are given in Table 5**.**

Table 5: Execution Time Comparison

Scanner	Rules Covered	Aver. exec time (s)
SA do_body_test (sa-compile applied)	*453*	*0.0325*
1 binary images generated by sa2px	*185*	*0.0031*
4 binary images generated by sa2px	*405*	*0.0107*

In Table 5, it is shown that the DFA based execution time (≈ 0.0031s) is 10 times shorter than SA email body scanning procedure – *do_body_test* function (≈ 0.03s), even sa-compile technique has already been applied. Here, the binary image used is *active_main* which contains 185 regexp rules which are originally saved in *72_active.cf*. The third row evaluates the system overhead when 4 binary scanner images are process-level parallel deployed. The execution procedure diagram is given in Figure 8. In this experiment, the four images are *active_main*, *fraud_main*, *drug_main*, *misc_replace_main* which contains 406 *body* rules (89.62% *body* rules). When the four images are parallelized, the email body scanning procedure's execution time is reduced by 66%.

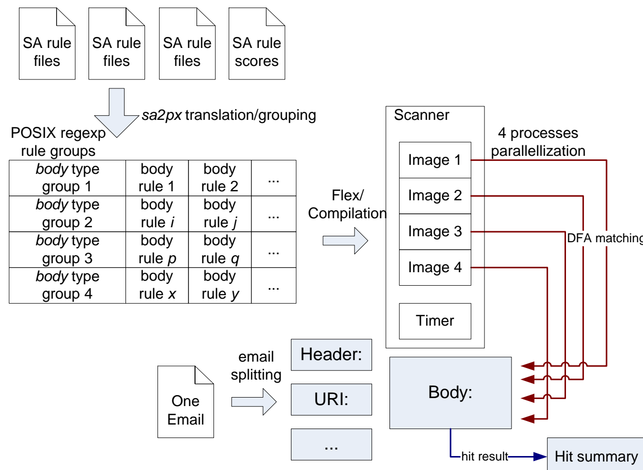

Figure 8: 4 binary images parallelized execution

6. Related Work

Some of the earliest type of content-based filters used keywords for filtering [8]. In addition to content screening, SA also employed additional techniques, such as sender authentication, IP/DNS blacklist, reputation query, and etc., based on third party sources. Many spam filtering techniques have been developed in the literature. Spamato [10] is another spam filter that queries other specialized filters, e.g., Razor [11]. Other spam filtering schemes include remote reputation system, for example Spamhaus [17], SORBS [18], Secure Computing [19], Pyzor [20]; remote DNS/IP/URI black/whitelist, such as DNSBL [21] and URIBL [22], IADB [23]; sender authentication, for example, DKIM [24], HashCash [25], Sender Policy Framework [26]. Some early research on worm fingerprint [12] that collects TCP SYN, suggested that spam traffic delivered by botnets shares similar characteristics on Internet and Transport Layers. Machine learning techniques such as Naïve Bayes [13], Random Forest (PILFER) [14], SVM [15], [16] have also been studied for spam filtering. The detection model proposed in [7] tracks ages and frequency of occurrences of parsed patterns at the ingress of an enterprise network.

String matching is also widely used for Intrusion Detection Systems (IDS), such as Snort [27]. Bro [28] and L-7 filter [29]. Given n regexp rules, Snort and L-7 filter process them individually in n automata. Bro implements a lazy-DFA [36] which maintains a subset of the DFAs that matches the most common strings in memory, and for uncommon strings, it extends the subset from the corresponding NFA. Besides the lazy-DFA grouping scheme, a complexity $O(n^2)$ approach proposed in [31] used an $n \times n$ matrix to cluster DFAs that have the least interactions. Another grouping approach proposed in [32] stores the core part of DFA in memory statically and other additional DFA transitions in cache to increase cache hits. The "Snort2Bro" program [30] converts Snort signatures to Bro signatures. Snort2Bro is designed to translate Snort signature structures and payload signatures into the Bro format, and it does not support PCRE signatures translation. Moreover, only 6% rules in the Snort (v2.2) rule set handled by Snort2Bro are Perl regexps and the average length of these Perl regexp rules is 23. (Snort v2.8 has 36% Perl regexp rules of 16462 rules in total, and their average length is 59) The PCRE2POSIX in the PCRE library [39] (Perl Compatible Regexp library) provides a POSIX-style interface, but all regexp rules are still based on the Perl grammar.

An important research field on deploying regexp to computational resource limited platform is DFA transition table memory optimization. Yu *et al*. [31] proposed several rewriting principals to mitigate states explosion. D^2FA [44] utilizes states that have the same transition trigger input to reduce redundant; other approaches [45, 46] use auxiliary variables to remember previous transition information, or eliminate path ambiguity. The *sa2px* proposed in this paper is orthogonal to these optimization schemes. Moreover, the translated regexp rules can apply these optimization approaches to further reduce the states number of their DFA transition tables.

7. Future Work and Conclusion

This paper presents the design of *sa2px* to translate SA regexp rules into the POSIX format and further generate their DFA transition tables, so that the rich collection of content scanning rules can be ported to different platforms. 84.5% of SA regexp rules can be translate from Perl regexp syntax to its corresponding POSIX format via *sa2px*. In addition to the basic syntax translation of over 900 regexp rules, we present a *Backward*

Grouping Algorithm to group regexps for creation of compact DFA tables. Experimental results show that it is feasible to convert a large portion of complex SA regexps into high speed, and compact scanners for high speed content inspection. An unsolved problem is subsumption of redundant rules. We can solve this problem (in the future) using the redundancy check of Flex to identify them, and remove and rewrite subsumed rules.

8. ACKNOWLEDGMENTS

This research is supported in part by the Army Research Office under the contract number C07-00485, and by National Science Foundation under the contract numbers CNS- 0530210 and DUE-0516825. This research is granted from MS Research, and CISCO University Research Program.

9. REFERENCES

[1] Messaging Anti-Abuse Working Group (2007). Email metrics report. http://www.maawg.org/about/EMR

[2] MX Logic Threat Center, http://www.mxlogic.com/threat_center

[3] SendBlaster, http://www.sendblaster.com

[4] Send-Safe Mailer, http://www.send-safe.com

[5] Srizbi botnet, http://en.wikipedia.org/wiki/Srizbi_botnet

[6] SpamAssassin, http://spamassassin.apache.org

[7] S. Lin, C. Tan, J. Liu and M. Oehler. “High-Speed Detection of Unsolicited Bulk Emails”. In Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2007. Dec. 2007. page 175 ~ 184.

[8] W. Cohen. “Learning Rules That Classify E-mail”. In Proceedings of the AAAI Spring Symposium on Machine Learning and Information Access. Mar. 1996. page 18 ~ 25

[9] H. Stern . “A Survey of Modern Spam Tools”. In Proceedings of Fourth Conference on Email and Anti-Spam. Aug. 2007.

[10] K. Albrecht, N. Burri and R. Wattenhofer. “Spamato – An Extendable Spam Filter System”. In Proceedings of Second Conference on Email and Anti-Spam. July 2005.

[11] V. Prakash. Vipul’s Razor. http://razor.sourceforget.net

[12] S Singh, C. Estan, G. Varghese and S. Savage. “Automated Worm Fingerprinting”. In Proceedings of Sixth Symposium on Operating System Design and Implementation (OSDI’04). Dec. 2004. Page 45 ~ 60

[13] V. Metis, I. Androutsopoulos and G. Paliouras. “Spam Filtering with Naïve Bayes – Which Naïve Bayes?”. In Proceedings of Third Conference on Email and Anti-Spam. July 2006. Page 125 ~ 133.

[14] I. Fette, N. Sadeh and A. Tomasic. “Learning to Detect Phishing Emails”. In Proceedings of Sixteenth International World Wide Web Conference. May 2007. Page 649 ~ 656.

[15] D. Sculley and G. Wachman. “Relaxed Online Support Vector Machines for Spam Filtering”. In Proceedings of Thirtieth Annual ACM SIGIR Conference. July 2007.

[16] A. Bergholz, G. Paab, F. Reichartz, S. Strobel, M. Moens, B. Witten. “Detecting Known and New Salting Tricks in Unwanted Emails”. In Proceedings of Fourth Conference on Email and Anti-Spam. Aug. 2007.

[17] Spamhaus. http://www.spamhaus.org/sbl/

[18] Spam and open-relay blocking sytem (SORBS). http://www.sorbs Presence of Noisy User Feedback”. In Proceedings of Fourth Conference on Email and Anti-Spam. Aug. 2007.net

[19] Secure Computing. Ironmail. http://www.securecomputing.com

[20] Pyzor. http://pyzor.sourceforge.net/

[21] DNSBL. http://www.dnsbl.com

[22] URIBL. http://www.uribl.com

[23] Institute for Spam and Internet Public Policy (ISIPP) IADB program. http://www.isipp.com/iadb.php

[24] DomainKeys Identified Mail (DKIM). http://www.dkim.org

[25] HashCash. http://www.hashcash.org

[26] M. Wong and W. Schlitt. “Sender Policy Framekwork (SPF) for Authorizing Use of Domains in E-Mail”. http://www.ietf.org/rfc/rfc4408.txt

[27] Snort Network Intrusion Detection System. http://www.snort.org

[28] Bro Intrusion Detection System. http://bro-ids.org/

[29] Application Layer Packet Classifier for Linux. http://l7-filter.sourceforge.net

[30] Snort2bro, http://svn.icir.org/bro/trunk/bro/scripts/snort2bro/snort2bro

[31] F. Yu, Z. Chen, Y. Diao, T.V. Lakshman, R.H. Katz. “Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection”. In Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems 2006. Dec. 2006. Page 93 ~ 102.

[32] A. Majumder, R. Rastogi, S. Vanama. “Scalable Regular Expression Matching on Data Streams”. In Proceedings of ACM SIGMOD/PODS Conference 2008. June, 2008. Page 161 ~ 172.

[33] Flex. http://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html

[34] Perl Regular Expressions. http://www.perl.com/doc/manual/html/pod/perlre.html.

[35] Spam Assassin ReplaceTags Plug-in. http://wiki.apache.org/spamassassin/ReplaceTags?highlight=(ReplaceTag)

[36] R. Sommer and V. Paxson, “Enabling Byte-Level Network Intrusion Detection Signatures with Context”. In Proceedings of ACM Conference on Computer and Communications Security, Oct. 2003. Page 262 ~ 271.

[37] J.E. Hopcroft, R. Motwani and J.D. Ullman. “Introduction to Automata Theory, Languages, and Computation”. Addison Wesley, 2001.

[38] RFC 821. Simple Mail Transfer Protocol http://www.ietf.org/rfc/rfc2821.txt

[39] PCRE – Perl Compatible Regular Expression library. http://www.pcre.org/

[40] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. “Compilers-Principles, Techniques, & Tools” 2nd Edition. Pearson Addison-Wesley.

[41] RE2C. http://re2c.org

[42] R. Sidhu, V. K. Prasanna, “Fast Regular Expression Matching using FPGAs”. In Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) ‘01. Apr., 2001

[43] Z. K. Baker, H. J. Jung, and V. K. Prasanna. “Regular Expression Software Deceleration for Intrusion Dection System”. In Proceedings of Field Programmable Logic and Applications ‘06. Aug. 2006. Page 1 ~ 8.

[44] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. “Algorithm to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection”. In Proceedings of ACM Special Interest Group on Data Communication, 2006. Aug. 2006. Page 339 ~ 350.

[45] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese. “Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia”. In Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2007. Dec. 2007. Page 155 ~ 164.

[46] R. Smith, C. Estan, S. Jha, and S. Kong. “Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata”. In Proceedings of ACM Special Interest Group on Data Communication, 2008. Aug. 2008. Page 207 ~ 218.

[47] TREC 2007. http://trec.nist.gov/data/spam.html

10. Appendix

Appendix Table 6: Complete mapping policy table

<table>
<tr><th colspan="2">Perl RE</th><th>Translation Actions</th></tr>
<tr><td rowspan="3">Modifier</td><td>i</td><td>Change every character ‘c’ from ‘c’ or ‘C’ to ‘[cC]’</td></tr>
<tr><td>m, x</td><td>Discard, we temporarily ignore multiple lines mode</td></tr>
<tr><td>s</td><td>Change every ‘.’ to “(.|\n)”</td></tr>
<tr><td>Meta-Chars</td><td>\, ., |, (), [], ^, $</td><td>Unchanged</td></tr>
<tr><td>Quantifiers</td><td>*, +, ?, {n,m}</td><td>Unchanged</td></tr>
<tr><td rowspan="9">Quote String</td><td>\t, \n, \r, \f, \a, \e, \0, \x</td><td>Unchanged</td></tr>
<tr><td>\c[, \l, \u, \L, \U, \E, \Q</td><td>Ignored, not used in SA</td></tr>
<tr><td>\w</td><td>[a-zA-Z0-9_]</td></tr>
<tr><td>\W</td><td>[^a-zA-Z0-9_]</td></tr>
<tr><td>\s</td><td>[\t\n\r\f\v]</td></tr>
<tr><td>\S</td><td>[^ \t\n\r\f\v]</td></tr>
<tr><td>\d</td><td>[0-9]</td></tr>
<tr><td>\D</td><td>[^0-9]</td></tr>
<tr><td>\b</td><td>[\t\n\r\f\v\\\./\$!@#%&*\(\)\[\]\{\}=+-,<>\?\"\'\|&\^~`]</td></tr>
<tr><td rowspan="3">Zero-width Assertion</td><td>\B</td><td>[^ \t\n\r\f\v\\\./\$!@#%&*\(\)\[\]\{\}=+-,<>\?\"\'\|&\^~`]</td></tr>
<tr><td>\A, \Z</td><td>Discard, used once in pattern. Translation to “^” or “$” is not compatible with POSIX when they are in the middle of a regexp.</td></tr>
<tr><td>\z, \G</td><td>Ignored, not used in SA</td></tr>
<tr><td rowspan="5">Patterns</td><td>(?#text), (?>pattern)</td><td>Ignored, not used in SA</td></tr>
<tr><td>(?:pattern) or (?:imsx-imsx:pattern)</td><td>If it is in the tail of regexp, change to /(pattern), otherwise change to (pattern) Then change every character ‘c’ from ‘c’ or ‘C’ to ‘[cC]’</td></tr>
<tr><td>(?=pattern), (?<=pattern)</td><td>If it is in the tail of regexp, change to /(pattern), otherwise change to (pattern)</td></tr>
<tr><td>(?!pattern), (?<!pattern),</td><td>Discard, not compatible with POSIX</td></tr>
<tr><td>(?imsx-imsx)</td><td>change every character ‘c’ in pattern from ‘c’ or ‘C’ to ‘[cC]’</td></tr>
</table>