Compressing Regular Expression Sets for Deep Packet Inspection

Alberto Bartoli, Simone Cumar, Andrea De Lorenzo, and Eric Medvet

Department of Engineering and Architecture, University of Trieste, Italy

Abstract. The ability to generate security-related alerts while analyzing network traffic in real time has become a key mechanism in many networking devices. This functionality relies on the application of large sets of regular expressions describing attack signatures to each individual packet. Implementing an engine of this form capable of operating at line speed is considerably difficult and the corresponding performance problems have been attacked from several points of view. In this work we propose a novel approach complementing earlier proposals: we suggest transforming the starting set of regular expressions to another set of expressions which is much smaller yet classifies network traffic in the same categories as the starting set. Key component of the transformation is an evolutionary search based on Genetic Programming: a large population of expressions represented as abstract syntax trees evolves by means of mutation and crossover, evolution being driven by fitness indexes tailored to the desired classification needs and which minimize the length of each expression. We assessed our proposals on real datasets composed of up to 474 expressions and the outcome has been very good, resulting in compressions in the order of 74%. Our results are highly encouraging and demonstrate the power of evolutionary techniques in an important application domain.

Keywords: Genetic programming, evolutionary optimization, intrusion detection, traffic classification.

1 Introduction

The ability to generate security-related alerts while analyzing network traffic in real time has become a key mechanism in many networking devices, ranging from intrusion detection systems to firewalls and switches. While early systems classified traffic based only on header-level packet information, modern systems are capable of detecting malicious patterns within the actual packet payload. This *deep packet inspection* capability is usually based on pattern descriptions expressed in the form of *regular expressions*, because fixed strings have become inadequate to describe attack signatures.

Implementing a regular expression evaluation engine capable of analyzing network traffic at line speed is considerably difficult, also because there are usually hundreds or thousands of regular expressions to be analyzed and this set needs to

T. Bartz-Beielstein et al. (Eds.): PPSN XIII 2014, LNCS 8672, pp. 394–403, 2014.

be periodically updated to address novel attacks. For this reason, there has been a considerable amount of recent proposals aimed at handling the corresponding performance problems. Such proposals have addressed different dimensions of the design space: optimization of the evaluation algorithm in representations of regular expressions based on Deterministic Finite Automata (DFA) [1–3]; DFA representations leading to faster hardware implementations and which require less memory [4–6]; optimization of the hardware implementation of DFA [7]; development of engines suitable for parallel hardware implementation [8, 9].

In this work, we address a different dimension of the design space and propose an approach which complements the existing proposals. Rather than optimizing the steps from the set of regular expressions to their run-time evaluation, we explore the possibility of greatly reducing the size of the set itself. To this end, we use an heuristic approach: rather than attempting to construct a new set of expressions formally equivalent to the original one (but simpler to evaluate at run-time) [10], we aim at constructing a set with the same detection behavior on the traffic of interest. As it turns out, this relaxed problem formulation allows a broad range of compressions and simplifications which would not be possible when insisting on having exactly the very same detection behavior on all possible strings.

Key component of our proposal is an evolutionary search phase based on *Genetic Programming* (GP). We create a population of regular expressions composed of the set of expressions to be simplified and further randomly-generated expressions. We then evolve this population by randomly combining expressions with genetic operators (crossover and mutation) for a predefined number of steps. The evolution is driven by a *multi-objective optimization* strategy aimed at minimizing two *fitness* indexes of each expression taken in isolation: classification errors on the traffic of interest and length of the expression. Finally, we construct a set of regular expressions meant to replace the original one by selecting a subset of the final population. We select this subset with a greedy procedure ensuring that the resulting subset tends to have the same detection behavior on the traffic of interest as the original set.

We assess our proposal on several real sets of regular expressions used in the Snort[1] intrusion detection system—one of the standard testbeds in this specific research field, e.g., [7, 10, 3]. We considered sets with a number of regular expressions ranging from 10 to 474 and an aggregate length ranging from 260 to about 59 742 The results are highly promising: we obtain a decrease in the number of regular expressions and a decrease in aggregate length in the order of 74%, on the average.

2 Our Approach

2.1 Problem Statement

We associate each set of regular expressions R with a numerical *cost* $c(R)$, which models the effort required for applying all expressions in R to a given string.

[1] `http://www.snort.org`

This index should quantify the run-time cost of using R and its actual value depends on the specific technology used [7]. In this work we use the sum of the lengths of all expressions in R as a proxy for $c(R)$, i.e., we set $c(R) = \sum_{r \in R} \ell(r)$, where $\ell(r)$ is the length of the regular expression r represented as a string. It will be clear from the following sections that our approach may be applied with widely differing cost definitions: for example, one could consider the number of states of the Nondeterministic Finite Automata (NFA) implementing each expression [10] as well as the presence of specific hard-to-evaluate constructs [11].

We say that a regular expression r *matches* a string s, denoted $r \hookleftarrow s$, if r extracts at least one non-empty substring of s. We say that a set of regular expressions R matches a string s, denoted $R \hookleftarrow s$, if at least one of the regular expressions $r \in R$ matches s. We say that a set of regular expressions R_1 is *equivalent* to another set R_2 if the set of all the strings matched by R_1 is equal to the set of all the strings matched by R_2. Given a finite set S of sample strings, we say that R_1 is S*-equivalent* to R_2 if the set of all the strings in S matched by R_1 and R_2 is the same, i.e., $\{s \in S : R_1 \hookleftarrow s\} = \{s \in S : R_2 \hookleftarrow s\}$.

Given a starting set of regular expressions R_s, we generate synthetically from this set a *positive* set S^+ of matched strings and a *negative* set S^- of strings which are not matched. We aim at identifying a *different* set of regular expressions R_f such that:(i) R_s and R_f are $(S^+ \cup S^-)$-equivalent; and, (ii) $c(R_f) < c(R_s)$.

To solve this problem, we proceed as follows.

1. We generate S^+ and S^- from R_s with the same cardinality. We then randomly partition each set in three subsets to be used in the two next phases of the algorithm and for testing: i.e., we partition S^+ in $S^+_{\text{evolution}}$, $S^+_{\text{selection}}$ and S^+_{testing}, and the same for S^-. In this work we chose to use three equally-sized subsets, but different choices are possible.
2. In the *evolution* phase, we evolve the starting set of regular expressions R_s with a stochastic procedure based on GP. The evolution is driven by a multi-objective optimization strategy aimed at minimizing two performance indexes of each expression r taken in isolation: errors of r on $S^+_{\text{evolution}}$, $S^-_{\text{evolution}}$ and length of the expression $\ell(r)$. We execute n independent evolutions, each evolution producing a set of regular expressions R^i_e, with $i = 1, \ldots, n$.
3. In the *selection* phase, we construct a candidate target set R^i_f based on the set R^i_e generated in the previous phase, with $i = 1, \ldots, n$. The construction of each set R^i_f is made with a set coverage algorithm aimed at selecting a subset of R^i_e matching all examples in $S^+_{\text{selection}}$ and no example in $S^-_{\text{selection}}$. The coverage is driven by a greedy strategy aimed at minimizing the cost of R^i_f. We select as target set R_f the set R^i_f with smallest cost.

We emphasize that the evolution phase optimizes performance indexes of each regular expression taken in isolation, while the selection phase optimizes an index resulting from the coordinated effort of all the regular expressions.

We assessed our procedure on several sets of regular expressions used in Snort. For each set R_s, we assessed the generated target set R_f by comparing its cost $c(R_f)$ to the cost of the original set $c(R_s)$. Furthermore, we verified that R_f matches all strings in S^+_{testing} and does not match any string in S^-_{testing}.

The starting set of expressions R_s is obtained from detection rules generated by administrators, each expression in R_s being associated with exactly one detection rule. Transforming R_s to a different set R_f, much cheaper to evaluate at run-time, implies that when R_f matches a given string there is usually no immediate correspondence with detection rules. This issue is intrinsic to any approach aimed at optimizing R_s as a whole, e.g., [10], as opposed to optimizations where the original regular expressions are left unchanged. We remark, though, that identifying the detection rule in order to generate a meaningful alert description may be done rather simply: once R_f has classified a certain packet as a positive, it suffices to apply R_s on that packet. The key observation is that packet processing has to be performed at line speed, while alert description may proceed at a much slower pace. Indeed, this strategy also allows correcting any false positive misclassifications due to the tranformation from R_s to R_f—a packet classified as positive by R_f which is actually not matched by any expression in R_s would not generate any alert.

2.2 Representation

We represent each regular expression as an abstract syntax tree. A regular expression r is produced from a tree by concatenating node labels encountered in a depth-first post order visit of the tree. The label of each leaf node is an element from a predefined *terminal set* whereas the label of each branch node is an element from a predefined *function set*.

The terminal set is composed of constants (`a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`, `\x00`, ..., `\x07`, `-`, `?`, `(`, `)`, {, }, `.`, `@`, `#`, _, ␣, ...) and character classes (`\w`, `\W`, `\d`, `\D`, `\s`, `\S`, `a-z` and `A-Z`). The function set is composed of the following operators: the concatenator $\cdot\cdot$, which concatenates its two children (the dot character $\cdot$ represents a placeholder for the children nodes of the corresponding node); the character class operators `[`$\cdot$`]` and `[^`$\cdot$`]`, the non-capturing group `(?:`$\cdot$`)` operator, the capturing group `(`$\cdot$`)` operator, the disjunction $\cdot$`|`$\cdot$ operator and the greedy quantifiers ($\cdot$`*`, $\cdot$`+`, $\cdot$`?`, $\cdot${$\cdot$`,`$\cdot$}).

2.3 Set Equivalence by Sample Strings

An essential component of our heuristic approach is the choice of the sets S^+, S^- of sample strings to be used for checking the (relaxed) equivalence of the starting set and final set of regular expressions. These samples may be chosen in several ways, for example by using a synthetic *traffic generator* specialized for evaluating deep packet inspection architectures [12]. Another possibility consists in using samples of real traffic explicitly collected for assessing intrusion detection systems [13, 14]. In this paper, we chose to use a simpler approach in which we generate traffic synthetically based solely on the structure of the regular expressions in the starting set R_s, as described below. Further experimentation with traffic generation strategies like those of the cited works is certainly required in order to better validate our results.

For each regular expression $r \in R_s$, we generate k positive strings s such $r(s) = s$—where $r(s)$ denotes the leftmost non-empty substring of s extracted by r. Then, we generate $k|R_s|$ random strings such that R does not match any of these negative strings. The outcome of the procedure consists of the sets S^+, S^-, such that(i) $\forall s \in S^+, R_s \hookleftarrow s$, (ii) $\forall s \in S^-, R_s \not\hookleftarrow s$, and (iii) $|S^+| = |S^-| = k|R_s|$.

We generate each positive string s from a $r \in R_s$ as follows. We traverse the tree representation of r (see previous section) in depth-first: each function node generates a string which depends on the node and its children; each terminal node generates a string which depends on the node only. For example, the terminal node `\d` generates a digit with uniform probability; the disjuction node $\cdot|\cdot$ generates the first child or the second child string, with equal probability.

We generate each negative string $s \in S^-$ at random. If $R_s \hookleftarrow s$, we drop s and randomly generate a new one. Negative strings have a maximum length of 120 characters.

2.4 Evolution Phase

In this phase, we evolve the starting set of regular expressions R_s with a procedure based on GP and produce a set of regular expressions R_e^i which will be used in the next phase: the whole procedure described in this section is repeated for $i = 1, \ldots, n$ and different random seeds. We use an approach which follows closely a proposal for generating automatically regular expressions for *text extraction* based on labelled examples [15]. We summarize the approach in order to provide sufficient background for this work and outline at the end of this section the changes which we applied to the original approach.

The evolutionary search, described below, is based on the NSGA-II [16] multi-objective optimization algorithm. Each candidate solution r has two *fitness* indexes to be minimized: the length $\ell(r)$ of the regular expression and an index $e(r)$ quantifying the classification errors of r on $S^+_{\text{evolution}}$, $S^-_{\text{evolution}}$. In detail, the index $e(r)$ is defined as:

$$e(r) = \sum_{s \in S^+_{\text{evolution}}} d\left(s, r\left(s\right)\right) + \sum_{s \in S^-_{\text{evolution}}} d\left(\varnothing, r\left(s\right)\right) \tag{1}$$

where $d(s_1, s_2)$ is the Levenshtein distance (edit distance) [17] between strings s_1 and s_2—note that $d(\varnothing, s) = \ell(s)$. In other words, $e(r)$ is the sum of two components: sum of distances between positive strings and what was actually extracted from the positive string; and, sum of distances between the empty string and what was actually extracted from the negative strings. The rationale is that a perfect r should extract exactly s from each $s \in S^+_{\text{evolution}}$—since positives s have been generated such that $r(s) = s$, with $r \in R_s$—and should not extract any string from each $s \in S^-_{\text{evolution}}$. We remark that $e(r)$ quantifies *extraction* errors rather classification errors, that is, the desired behavior is described in terms of (possibly empty) substrings to be extracted from sample strings, rather than in terms of two categories of strings. We chose to not deviate from such

formulation because the cited paper argues that fitness definitions based on mere classification could not be adequate to drive the evolutionary search toward the generation of regular expressions with the desired behavior—different fitness indexes could be explored in future work, though.

Each evolutionary search is made on a population of 500 candidate solutions. The initial population consists of all the regular expressions in the starting set R_s and $500 - |R_s|$ regular expressions generated at random. The population evolves for 500 generations, as follows (recall that we execute n independent searches, each producing a set of 500 candidate solutions). Let P be the current population. We generate an evolved population P' as follows: 20% of the regular expressions are generated at random, 20% of the regular expressions are generated by applying the genetic operator "mutation" to regular expressions of P, and 60% of the regular expressions are generated by applying the genetic operator "crossover" to a pair of individuals of P. We select regular expressions for mutation and crossover with a *tournament* of size 7, i.e., we pick 7 regular expressions at random from P and then select the best regular expressions in this set, according to NSGA-II. Finally, we generate the next population by choosing the regular expressions with highest fitness among those in P and P'. The size of the population is kept constant during the evolution. Upon generation of a new regular expression, we check its syntactic correctness: if the check fails, we discard the regular expression and generate a new one. The outcome set R_e^i is set to the final population.

The approach described in this paper differs from the original proposal in [15] in the following points.

1. The initial population is not generated completely at random: it includes all the expressions in the starting set.
2. The terminal set includes more constants: enlarging the cardinality of the terminal set, as well as of the function set, greatly enlarges the size of the solution space.
3. The function set includes the disjunction operator: it is disadvantageous to use in text extraction because it tends to promote overfitting of the labelled examples. Furthermore, the function set includes the greedy quantifiers (`·*`, `·+`, `·?`, `·{·,·}`) and does not include possessive quantifiers (`·*+`, `·++`, `·?+`, `·{·,·}+`). The former are included because largely used in the starting set R_s, the latter are not included because they are often not supported in deep packet inspection tools.

Inclusion of greedy quantifiers with the standard Java engine for processing regular expressions often results in unacceptably long execution times for this form of evolutionary search [15]. For this reason, we used a different engine, internally built with NFA[2], where the processing cost depends only on the length of the inputs rather than also on the structure of the expression.

[2] RE2: `https://code.google.com/p/re2`

2.5 Selection Phase

In this phase we construct a candidate target set R_f^i based on the set R_e^i of regular expressions resulting from the ith evolution ($i = 1, \ldots, n$) and then select as target set R_f the set R_f^i with smallest cost.

To construct each R_f^i, we consider $S^+_{\text{selection}}$ as a set to be covered by regular expressions in R_f^i (an element of $S^+_{\text{selection}}$ being covered if it is matched by a regular expression in R_f^i). We then execute a *set coverage* procedure aimed at selecting a subset of R_e^i matching all examples in $S^+_{\text{selection}}$ and no example in $S^-_{\text{selection}}$, as follows.

We define the *score* $\mathcal{S}(r, S', S'')$ of a regular expression r on the sets S', S'' as the number of examples in S', S'' which r handles correctly:

$$\mathcal{S}(r, S', S'') = |\{s \in S' : r \hookleftarrow s\}| + |\{s \in S'' : r \not\hookleftarrow s\}| \tag{2}$$

Similarly, we define the score $\mathcal{S}(R, S', S'')$ of a set R on the sets S', S'' of regular expressions the number of examples in S', S'' which R as a whole handles correctly:

$$\mathcal{S}(R, S', S'') = |\{s \in S' : R \hookleftarrow s\}| + |\{s \in S'' : R \not\hookleftarrow s\}| \tag{3}$$

The greedy set coverage algorithm starts with $R_f^i := \emptyset$, $S' := S^+_{\text{selection}}$ and consists of the following steps:

1. select $r \in R_e^i \setminus R_f^i$ with highest score $\mathcal{S}(r, S', S^-_{\text{selection}})$;
2. if $\mathcal{S}(R_f^i \cup \{r\}, S^+_{\text{selection}}, S^-_{\text{selection}}) \leq \mathcal{S}(R_f^i, S^+_{\text{selection}}, S^-_{\text{selection}})$ then terminate;
3. $R_f^i := R_f^i \cup \{r\}$;
4. $S' := S' \setminus \{s \in S^+_{\text{selection}} : R_f^i \hookleftarrow s\}$
5. if $S' = \emptyset$ or $R_f^i = R_e^i$ then terminate, otherwise go to step 1.

In other words, candidates for inclusion in R_f^i are taken from R_e^i and the choice is driven by the score of candidates on S', $S^-_{\text{selection}}$. The strategy is greedy in the sense that once a candidate is chosen it cannot be removed by a later choice.

These steps are followed by further *completion* steps, to be executed in case of termination with $S' \neq \emptyset$. The completion steps consist in a further execution of the above algorithm, this time starting from the R_f^i obtained at the end of the former algorithm (rather than from $R_f^i := \emptyset$) and by selecting candidates from the original expressions R_s—i.e., in step 1, r is chosen in $R_s \setminus R_f^i$ rather than in $R_e^i \setminus R_f^i$. The rationale is that if elements from R_e^i fail to detect some positives, then the missing positives can be detected by some of the original expressions in R_s.

3 Experimental Evaluation

3.1 Datasets

We used several real sets of regular expressions used in the Snort intrusion detection system, which have been collected by the Netbench project [18]. Table 1

Table 1. Datasets

R_s	$\lvert R_s\rvert$	$c(R_s)$	k	$\lvert S^+ \cup S^-\rvert$
chat.rules.pcre	14	307	105	2940
pop3.rules.pcre	16	265	105	3360
policy.rules.pcre	10	260	105	2100
web-php.rules.pcre	16	400	105	3360
ftp.rules.pcre	35	645	60	4200
spyware-put.rules.pcre	460	16 277	60	55 200
web-activex.rules.pcre	474	59 742	60	56 880

lists these sets, along with their cardinality and their cost (i.e., aggregate length of all the regular expressions in the set). The table also shows the value k we used in the procedure for generating S^+, S^- for each set R_s and the resulting number $|S^+ \cup S^-| = 2k|R_s|$ of sample strings.

3.2 Results and Discussion

We applied our approach to each dataset R_s and assessed, in each case, the quality of the resulting set R_f with the following indexes. We quantified the cost reduction by computing the *compression* ratio defined as $1 - \frac{c(R_f)}{c(R_s)}$. Concerning the detection behavior, we computed False Positive Rate (FPR, i.e., percentage of strings in S^-_{testing} which are matched by R_f) and False Negative Rate (FNR, i.e., percentage of strings in S^+_{testing} which are not matched by R_f). We also computed *accuracy* as $1 - \frac{1}{2}(\text{FPR} + \text{FNR})$. Of course, R_s exhibits $\text{FPR} = \text{FNR} = 0$ by construction of sets S^+, S^-. Thus, R_f should also exhibit $\text{FPR} = \text{FNR} = 0$ but coupled with a compression rate close to 100%.

Table 2 shows the results of our experimental evaluation. The table also shows the performance indexes without the completion steps in the selection phase, in order to highlight to which extent these steps improve results.

Table 2. Results

	Without completion steps				With completion steps			
R_s	FPR	FNR	Acc.	$1 - \frac{c(R_f)}{c(R_s)}$	FPR	FNR	Acc.	$1 - \frac{c(R_f)}{c(R_s)}$
chat.rules.pcre	0.0	50.0	75.0	96.10	0.0	0.0	100.0	70.66
pop3.rules.pcre	2.7	0.0	98.7	91.33	2.7	0.0	98.7	91.33
policy.rules.pcre	88.5	0.0	55.9	8.62	88.5	0.0	55.9	8.62
web-php.rules.pcre	24.5	6.3	84.6	67.00	24.5	0.0	87.8	66.50
ftp.rules.pcre	15.9	7.4	88.4	53.96	15.9	0.0	92.2	48.99
spyware-put.rules.pcre	3.3	9.5	93.6	99.01	1.6	0.0	98.3	91.26
web-activex.rules.pcre	0.0	0.0	100.0	99.97	0.0	0.0	100.0	99.97

It can be seen that the average compression ratio amongst the datasets is 74%, but the key result is that the two largest datasets (spyware-put.rules.pcre and web-activex.rules.pcre) can be compressed to less than 1% of the original size—without affecting accuracy significantly.

We also remark that FNR is zero for all the datasets (thanks to the completion steps) and that FPR is very low for 4 on 7 datasets, but can be reduced to zero on all the datasets as discussed in Section 2.1 (it suffices to apply the original R_s only on those strings which are matched by R_f, which still allows exploiting the advantages of compressions because only R_f has to be applied at line speed).

We performed our experiments on an Intel i5-3470 3.20GHz machine with 8 GB RAM: the time required to process a single dataset was of 4 h on the average.

4 Concluding Remarks

Applying large sets of regular expressions to network traffic while operating at line speed is a challenging problem which has been attacked from several perspectives. In this work, we propose a novel approach complementing earlier proposals and assessed its feasibility. We considered the possibility of transforming the starting set of regular expressions to another set of expressions which is much smaller yet classifies network traffic in the same categories as the starting set. Key component of the transformation is an evolutionary search based on GP: a large population of regular expressions represented as abstract syntax trees evolves by means of mutation and crossover, evolution being driven by fitness indexes tailored to the desired classification needs and which minimize the length of each expression. The desired set of expressions is then built with a greedy algorithm which selects from the available expressions a small set set matching all positive samples and not matching any negative. We remark that the evolutionary search optimizes each expression taken in isolation, while the selection phase optimizes the performance of the target population.

We experimented with real datasets and the outcome has been very good, resulting in compressions in the order of 74% across all datasets but well above 90% on the bigger datasets composed of hundreds of expressions. Such compressions could be even improved further by applying other proposals to the final result, e.g., by minimizing the number of states of the NFA representing the final set of expressions [10]. While our proposal certainly needs further investigation, in particular, concerning its performance on real network traffic (see Section 2.3), we do believe that our results are highly encouraging and demonstrate the power of evolutionary techniques in an important application domain.

References

1. Yu, F., Chen, Z., Diao, Y., Lakshman, T., Katz, R.H.: 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, pp. 93–102. ACM (2006)
2. Kumar, S., Dharmapurikar, S., Yu, F., Crowley, P., Turner, J.: Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM Computer Communication Review 36(4), 339–350 (2006)

3. Becchi, M., Crowley, P.: An improved algorithm to accelerate regular expression evaluation. In: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems, pp. 145–154. ACM (2007)
4. Brodie, B.C., Taylor, D.E., Cytron, R.K.: A scalable architecture for high-throughput regular-expression pattern matching. In: ACM SIGARCH Computer Architecture News, vol. 34, pp. 191–202. IEEE Computer Society (2006)
5. Kong, S., Smith, R., Estan, C.: Efficient signature matching with multiple alphabet compression tables. In: Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks, vol. 1. ACM (2008)
6. Becchi, M., Cadambi, S.: Memory-efficient regular expression search using state merging. In: INFOCOM 2007, 26th IEEE International Conference on Computer Communications, pp. 1064–1072. IEEE (2007)
7. Meiners, C., Patel, J., Norige, E., Liu, A., Torng, E.: Fast regular expression matching using small TCAM. IEEE/ACM Transactions on Networking 22(1), 94–109 (2014)
8. Becchi, M., Crowley, P.: A hybrid finite automaton for practical deep packet inspection. In: Proceedings of the 2007 ACM CoNEXT Conference, p. 1. ACM (2007)
9. Becchi, M., Crowley, P.: Efficient regular expression evaluation: theory to practice. In: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, pp. 50–59. ACM (2008)
10. Kosar, V., Korenek, J.: Reduction of fpga resources for regular expression matching by relation similarity. In: 2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), pp. 401–402. IEEE (2011)
11. Bispo, J., Sourdis, I., Cardoso, J.M.P., Vassiliadis, S.: Synthesis of regular expressions targeting fPGAs: Current status and open issues. In: Diniz, P.C., Marques, E., Bertels, K., Fernandes, M.M., Cardoso, J.M.P. (eds.) ARCS 2007. LNCS, vol. 4419, pp. 179–190. Springer, Heidelberg (2007)
12. Becchi, M., Franklin, M., Crowley, P.: A workload for evaluating deep packet inspection architectures. In: IEEE International Symposium on Workload Characterization, IISWC 2008, pp. 79–89. IEEE (2008)
13. Shiravi, A., Shiravi, H., Tavallaee, M., Ghorbani, A.A.: Toward developing a systematic approach to generate benchmark datasets for intrusion detection. Computers & Security 31(3), 357–374 (2012)
14. Black hat USA 2010: SprayPAL: how capturing and replaying attack traffic can save your IDS 1/2 (September 2010)
15. Bartoli, A., Davanzo, G., De Lorenzo, A., Medvet, E., Sorio, E.: Automatic synthesis of regular expressions from examples. IEEE Computer (2013) (Early Access Online)
16. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: Nsga-ii. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
17. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10, 707 (1966)
18. Pus, V., Tobola, J., Kosar, V., Kastil, J., Korenek, J.: Netbench: Framework for evaluation of packet processing algorithms. In: Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems, pp. 95–96. IEEE Computer Society (2011)