Scalable Algorithms for NFA Multi-Striding and NFA-Based Deep Packet Inspection on GPUs

Scalable Algorithms for NFA Multi-striding and NFA-based Deep Packet Inspection on GPUs

Matteo Avalle, Fulvio Risso, *Member, IEEE,* and Riccardo Sisto, *Senior Member, ACM*

***Abstract*—Finite State Automata (FSA) are used by many network processing applications to match complex sets of regular expressions in network packets. In order to make FSA-based matching possible even at the ever increasing speed of modern networks, multi-striding has been introduced. This technique increases input parallelism by transforming the classical FSA that consumes input byte-by-byte into an equivalent one that consumes input in larger units. However, the algorithms used today for this transformation are so complex that they often result unfeasible for large and complex rule sets. This paper presents a set of new algorithms that extend the applicability of multi-striding to complex rule sets. These algorithms can transform Nondeterministic Finite Automata (NFA) into their multi-stride form with reduced memory and time requirements. Moreover, they exploit the massive parallelism of Graphical Processing Units for NFA-based matching. The final result is a boost of the overall processing speed on typical regex-based packet processing applications, with a speedup of almost one order of magnitude compared to the current state of the art algorithms.**

I. Introduction

Regular expression (regex) pattern matching is currently one of the most widely used techniques to inspect network traffic, also because of the possibility to be transformed into equivalent Finite State Automata (FSA), which provides a solid mathematical foundation and enables the use of well-known algorithms. For instance, simple algorithms exist to determine if a particular input matches a regular expression and for composing multiple FSAs together, such as when multiple rules (e.g., the several thousands patterns used by a NIDS) have to be checked within a single scan of the input.

FSAs come in two different forms with equivalent expressiveness, namely Deterministic FSA (DFA) and Nondeterministic FSA (NFA). A DFA guarantees a bounded execution time for the matching operation on any execution architecture, because for each input string a single path in the automaton has to be followed. However, the bounded execution time is achieved at the expense of a potentially huge memory consumption, particularly when complex regular expressions (e.g., with frequent use of repetition wildcards '`.*`') are used, or when many expressions are combined together in a single DFA. While this problem can be alleviated by using variations of the DFA, important limitations still exist on the number and the complexity of the rule sets that can be handled using this form. This may force application developers to use approximate (and simpler) regular expressions, which may either impair the capability to filter out exactly the desired traffic or potentially generate many false positives, which may be particularly critical in security applications.

Dipartimento di Automatica ed Informatica, Politecnico di Torino, Italy
Email: {matteo.avalle, fulvio.risso, riccardo.sisto}@polito.it
Manuscript received XXX; revised XXX.

The NFA form solves the previous problem because the number of states in the automaton is directly proportional to the length of the regular expressions used to create the FSA. However, this may lead to a non-predictable execution time, which may grow dramatically when strictly sequential matching algorithms are used. In fact, with a NFA, the matching of a given input may require to follow a number of paths in the automaton which is not predictable and depends on the complexity of the automaton and on the particular input data that are being processed. For this reason, software-based implementations with strict processing time constraints (such as packet processing applications running on general purpose CPUs) are usually based on DFA or on some of its variations.

Recently, we proposed iNFAnt [1], a new software solution for regex matching based on NFA that exploits the high parallelism of Graphical Processing Units (GPU) in order to reduce the variability of NFA processing times and make this approach usable in practice. This paper extends that work by exploring the possibility to improve the throughput by means of multi-striding, i.e. the transformation of the automaton into one that processes more than one byte of input at each step. Using this technique, if a DFA takes N steps to perform a matching, where N is the length of the input data (e.g., the size of a network packet), a 2-stride version of the DFA returns the same result in $N/2$ steps. With a NFA, similar reductions can be achieved, hence potentially doubling the throughput at each stride doubling.

Figure 1 presents an example of a simple regular expression with its corresponding standard and 2-stride NFA. In the figure, the states with the dashed border are the initial states while those with thick borders are the final (i.e. accepting) states. The notation '`0|255`' stands for "*any symbol in the range from 0 to 255*". This means that the transition with that label is in fact a set of transitions, one for each possible input symbol. In the 2-stride NFA each transition consumes two input symbols and corresponds to the combination of two adjacent transitions of the original NFA. For instance, in the original automaton the input string '`ac`' triggers two transitions: a transition from state q_0 to state q_1 that consumes '`a`', followed by a transition to state q_2 that consumes '`c`'. In the 2-stride automaton, instead, a single transition leads directly from q_0 to q_2 and consumes the pair of symbols '`ac`'.

Although, in theory, multi-striding can be applied by grouping together an arbitrary number of input symbols, in practice the use of too many symbols leads to so complex automata that the building process cannot be completed in reasonable time (and memory). Therefore, multi-striding is usually im-

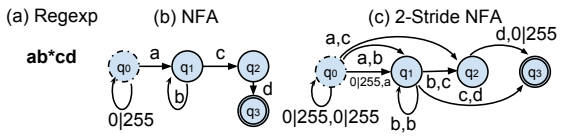


Fig. 1. A simple regular expression in its (a) "textual" representation, (b) NFA form and (c) 2-stride NFA form.

plemented by iteratively doubling the stride level until further doubling becomes unfeasible, i.e. first the NFA is transformed into a 2-stride NFA, then the latter is transformed into a 4-stride NFA, and so on. As shown intuitively in Figure 1, at each stride doubling the number of states does not change, but the number of transitions may increase quadratically in the worst case. For this reason, multi-striding is usually coupled with *Alphabet Compression* [2], [3], which reduces the number of transitions by performing a compression of its input alphabet. Unfortunately, current state-of-the-art algorithms for multi-striding and alphabet compression show poor scalability properties, hence supporting only very simple automata. Hence they are not compatible with the complex FSA that result from real-life rule sets for network packet inspection.

This paper addresses the above problem in two directions. First, it enhances in various ways the current multi-striding and alphabet compression techniques in order to cope with the complexity of the typical rule sets used in real applications. Second, it shows how the GPU-based algorithm of iNFAnt [1] can be adapted to efficiently execute the resulting multi-stride automata. The final result is the possibility to match complex rule sets in real time on high-speed networks, which was not possible with the previous software-based techniques.

This paper is structured as follows. Section II presents the state of the art, Section III provides background information about iNFAnt, multi-striding, and alphabet compression, while Sections IV to VII describe the new algorithms. Section VIII presents benchmark results of the new algorithms while, finally, Section IX draws conclusions.

II. Related work

Although the idea of processing more than one character at a time appeared in various previous papers, an algorithm for computing multi-stride DFAs by stride doubling was described for the first time in [2]. The same paper also proposed to combine multi-striding with alphabet compression, in order to reduce the quadratic increase of the number of transitions that occurs at each stride doubling step. Some improvements of these techniques, in terms of computational complexity, have been presented in [3] (later extended in [4]) and in [5] that provided, respectively, a new algorithm for alphabet compression and a new stride doubling algorithm, both with reduced computational complexities compared with the previous ones. [5] also presented the extension of these algorithms to NFA. Another algorithm for NFA stride doubling with similar complexity but without alphabet compression was presented in [6], while in [7] we showed our preliminary results toward more scalable algorithms for multi-striding.

An alternative NFA representation based on Ordered Binary Decision Diagrams (NFA-OBDD) is proposed in [8]. This approach is shown to outperform PCRE, a popular tool for regex matching based on NFA algorithms, when dealing with large rule sets. However, the throughput reported using the proposed approach is quite modest if compared to the one of [1], which exploits the parallelism of GPUs in order to counter the increase and variability of execution times. [8] also explores multi-striding with alphabet compression by applying the old algorithms proposed by [2] on the NFA before generating the OBDD representation. Of course, the introduction of multi-striding gives some improvement, but the scalability problems of the multi-striding and alphabet compression techniques proposed in [2] are confirmed here, as they cannot exceed the 2x stride level.

A common limitation of all the algorithms presented so far for computing the multi-stride versions of automata is their high CPU and memory consumption. As a consequence, when dealing with the complex rule sets found in real-life network applications, the multi-stride automata cannot be computed at all, or only very limited stride levels can be achieved with reasonable resources, as shown in [8].

These limitations are overcome by our improvements on the state of the art algorithms. Particularly, the alphabet compression algorithm has been redesigned to use multiple (but smaller) conversion tables at the same time, rather than just a single one, a technique that was originally proposed in [9] for standard (non multi-stride) automata. In addition, we also propose a more effective heuristic to distribute symbols across the multiple tables.

Other papers propose very efficient techniques for regular expression matching that are either not adequate for software implementation or not applicable to the general problem of regex matching with complex patterns. For instance, [2] proposes also a run-length encoding for automata that relies on priority encoders and TCAMs while [10] proposes another solution based on TCAMs, both best suited to implementations based on ASICs or FPGAs. Another example is the variable-stride technique presented in [11] which applies only to string matching and does not support regular expressions. Along this line, several papers [12] [13] [14] [15] propose fast GPU-oriented algorithms that operate on DFA-based representations. In order to deal with complex regex rule sets, the patterns that generate an excessive amount of states are kept in the NFA form and executed on the main CPU but this leads to poor performance.

Recently, [16] and [17] proposed new regex matching algorithms that are specifically designed for running NFA on GPUs and that improve the solution of iNFAnt [1], mainly by removing the useless processing of the transitions that start from inactive states. Our optimized algorithms for creating multi-stride automata provide a contribution that has more general validity, i.e. the multi-stride NFA created by our algorithms could be exploited, in principle, by any NFA-based regex processors, including the ones proposed in [16] and [17], after proper adaptation. In particular, adaptation is necessary if multimap alphabet compression (Section VI) is used. Then, the above papers can represent a valid alternative to our GPU-

based run-time regex engine presented in Section VII (based on [1]), but further work is necessary to adapt them to support our multimap algorithms.

III. Notation and Background

Table I lists the main symbols used in this paper along with a short description of their meaning.

TABLE I
COMMON SYMBOLS AND NOTATION USED IN THIS PAPER

Q	Set of states of an NFA
δ	Set of transitions of an NFA
Σ	Set of symbols of an NFA (alphabet)
A	Set of accepting states of an NFA
$\|X\|$	Number of elements of set X (cardinality)
$\overline{N}_{fo}$	Average number of states of an NFA that can be reached starting from a single state, by following all its outgoing transitions (average fan-out)
$\overline{L}_s$	Average number of transitions of an NFA connecting the same pair of states (average label size)
$\overline{R}$	Average number of transition ranges of an NFA connecting the same pair of states (similar to $\overline{L}_s$ but it counts *ranges* instead of single transitions)

A. NFA

A NFA can be formalized as a 5-tuple $\langle Q, \Sigma, q_0, \delta, A\rangle$ where Q is the set of states, Σ is the set of symbols (the alphabet), $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, i.e. a set of transitions, where each transition is a triple (q_1, s, q_2) with initial state $q_1 \in Q$, label $s \in \Sigma$ and final state $q_2 \in Q$, and $A \subseteq Q$ is the set of accepting states.

A NFA takes as input a sequence of symbols $s_1, \ldots, s_n$, with $s_i \in \Sigma$. A match is found at symbol s_k if there exists a sequence of $k+1$ states $q_0, \ldots, q_k$ starting with the initial state and terminating with an accepting state q_k, and there exists a sequence of transitions labeled $s_1, \ldots, s_k$ that bind each state of the sequence to the next state, i.e. $\forall i \in [1, k] : (q_{i-1}, s_i, q_i) \in \delta$.

The classical sequential NFA matching algorithm keeps track of all the states that can be reached after each input symbol and reports a match when an accepting state is reached. The algorithm is based on keeping a set of active states that, at the beginning, includes only the initial state. Input symbols are read sequentially and for each input symbol the next set of active states is computed by following all the enabled transitions, i.e., the ones that fire based on the given input symbol and that start from any of the current active states.

When the NFA represents a set R of regular expressions, it is possible to add the information about which regular expressions are matched in each acceptance state by means of a function $rules : A \rightarrow 2^R$ that maps each acceptance state onto the set of matched regular expressions.

B. Multi-striding

Algorithm 1 shows a simplified version of the state-of-the art stride doubling algorithm presented in [5] which builds the new set of transitions δ' by enumerating, for each reachable state q_0, all the possible combinations of two consecutive transitions (lines 4-6).

Algorithm 1 The stride doubling algorithm of [5].

```
1:  δ' = {}
2:  queue = {q0}; processed = {q0}
3:  while !queue.empty() do
4:      q0 = queue.pop()
5:      for all sA ∈ Σ, q1 ∈ Q | (q0, sA, q1) ∈ δ do
6:          for all sB ∈ Σ, q2 ∈ Q | (q1, sB, q2) ∈ δ do
7:              δ' = δ' ∪ {(q0, (sA, sB), q2)}
8:              if q2 ∉ processed then
9:                  queue.push(q2)
10:                 processed = processed ∪ {q2}
11:             end if
12:         end for
13:     end for
14: end while
```

For each of those combinations, the algorithm generates a transition in the new automaton that is equivalent to the two original transitions. For example, the new transition that brings directly from q_0 to q_2 will be labeled with the concatenation of the labels associated with the two original transitions (line 7). The algorithm starts by processing the initial state (line 2) and then it iterates through all the states that can be reached by using the generated compound transitions (lines 8-10). For simplicity, the presented pseudo-code does not include the particular case where the intermediate state q_1 is an accepting state while q_2 is not. This case is handled in [5] by also keeping the information about the rules matched in each acceptance state: for each rule r matched in q_1, an extra transition is added leading to a (possibly extra) accepting state q_a with no outgoing transitions such that $rules(q_a) = \{r\}$.

The asymptotic time complexity of the algorithm can be evaluated as $|Q| \cdot (\overline{N}_{fo} \cdot \overline{L}_s)^2$ where the meaning of $\overline{N}_{fo}$ and $\overline{L}_s$ is explained in Table I. This formula derives from the observation that, starting from each of the $|Q|$ states of the NFA, it is necessary to iterate through all its $\overline{N}_{fo} \cdot \overline{L}_s$ outgoing transitions and, then, for each reached state, $\overline{N}_{fo} \cdot \overline{L}_s$ compound transitions are added.

C. Alphabet compression

This step reduces the exponential growth of the alphabet size $|\Sigma|$ with the stride level (the size becomes $|\Sigma|^k$ for the k-stride NFA). This size is directly proportional to the $\overline{L}_s$ factor of the stride doubling asymptotic complexity formula. Hence, its growth impacts not only on the complexity of the resulting NFA but also on the possibility for the stride doubling algorithm to terminate in a reasonable time. As a consequence, alphabet compression is executed after each stride doubling step, in order to decrease the cardinality of the new alphabet.

The main idea behind any alphabet compression algorithm is that often there are symbols that are equivalent, i.e. they always trigger *exactly* the same set of transitions. This happens because often the patterns used to scan the network traffic are limited to alphanumeric characters (e.g., '`0-9A-Za-z`') and to a few other symbols, while the rest are ignored. If two (or more) input symbols (e.g., '`aa`' and '`bb`') always originate the same transitions, they are replaced with a single one (e.g., 'α'), thus decreasing the number of input symbols in the alphabet.

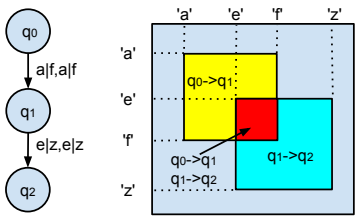


Fig. 2. An example of alphabet compression.

In essence, this process enables each new symbol of the new alphabet to represent an entire class of symbols of the original alphabet; as a consequence, the translated NFA that uses this dictionary has a smaller number of transitions than the original one although the two NFA are functionally equivalent.

Obviously, as alphabet compression changes the alphabet of the NFA, each input string (i.e., each packet) has to be translated to the new alphabet by substituting pairs of consecutive symbols with the new symbols assigned to their equivalence classes. However, as the time required to perform this operation is generally lower than the time required for matching, the overhead of this data translation can be considered negligible and it can be pipelined with the regular expression matching task.

Figure 2 shows the equivalence classes built for a simple 2-stride NFA. The map on the right hand side is a graphical representation of the space of symbol pairs to be partitioned into equivalence classes (each cell in the map represents a single symbol pair). Symbol pairs are partitioned into equivalence classes according to the transitions they can trigger. For example, the equivalence class of symbol pairs that can trigger only a transition from q_0 to q_1 (the area labeled with $q_0 \rightarrow q_1$) is made up of the symbol pairs '`a|f,a|f`' with the exclusion of '`e|f,e|f`'. In fact, the pairs '`e|f,e|f`' trigger two transitions, from q_0 to q_1 and from q_1 to q_2, so that they constitute another equivalence class. The total number of classes is 4: the other 2 classes are made up of the symbol pairs that can trigger only a transition from q_1 to q_2, and those that can trigger no transition (the area of the map not covered by rectangles). Consequently, the new alphabet is made up of only 4 symbols, each one assigned to one equivalence class, and the translation dictionary replaces all the symbols of each equivalence class with the new symbol assigned to that class.

Determining the equivalence classes by building the sets of transitions triggered by each symbol pair is unfeasible with large automata, mainly because of the huge amount of memory required: the size would be $O(|\Sigma|^2|Q|^2)$, as for each cell of the map a set of state pairs should be stored.

The algorithm proposed in [2] can perform the same operation using just one integer and one boolean for each symbol pair, but its time complexity is $O(|\Sigma|^4|Q|)$. The algorithm that can be considered the current state of the art [4] trades a slight increase in memory consumption (it requires 2 integers and 2 booleans per symbol pair) for a noticeable improvement in time efficiency. When combining this algorithm with stride doubling, [4] proposes to perform a preliminary compression step, followed by one or more stride-doubling steps, each one followed by a compression step.

Algorithm 2 shows the alphabet compression algorithm proposed in [4], adapted to the case of an input NFA having an alphabet made of pairs of symbols (the alphabet is $\Sigma \times \Sigma$), like the one obtained from a stride doubling step. With this algorithm the translation dictionary (an array of integers called `map`) is built in an iterative way that the authors call *cluster division*: initially all the elements of the dictionary are filled with the same value (0), meaning that all the possible symbol pairs are translated to the same equivalence class, and then it iteratively divides the classes by considering separately each possible combination of states $(q_1, q_2) \in Q \times Q$ (lines 2-3). The main idea of each division step is that a symbol pair '`ab`' has to be remapped to a new class if from q_1 to q_2 there is no transition labeled with it but there are transitions labeled with other symbol pairs that previously belonged to the same class as '`ab`'. The algorithm uses the two arrays of booleans named `char` and `class` to record respectively the set of symbol pairs that label the transitions from q_1 to q_2 and the classes covered by these symbol pairs. A first iteration computes these arrays and a second iteration does the necessary remapping, using a support array of integers named `remap`, which records the already performed remapping operations.

Algorithm 2 The alphabet compression algorithm presented in [4], rewritten for a stride-2 NFA

```
1:  map[|Σ|][|Σ|] = 0; size = 0
2:  for all q1 ∈ Q do
3:      for all q2 ∈ Q do
4:          char[|Σ|][|Σ|] = false
5:          class[|Σ| × |Σ|] = false
6:          remap[|Σ| × |Σ|] = 0
7:          for all (a, b) ∈ Σ × Σ | (q1, (a, b), q2) ∈ δ do
8:              char[a][b] = true
9:              class[map[a][b]] = true
10:         end for
11:         for all (a, b) ∈ Σ × Σ do
12:             if !char[a][b] & class[map[a][b]] then
13:                 if remap[map[a][b]] = 0 then
14:                     remap[map[a][b]] = ++size
15:                 end if
16:                 map[a][b] = remap[map[a][b]]
17:             end if
18:         end for
19:     end for
20: end for
```

The asymptotic time complexity of this algorithm can be evaluated as $|Q|^2 \cdot |\Sigma|^2$. The $|\Sigma|^2$ factor is due to the iteration through the $|\Sigma|^2$ elements of the alphabet, while the $|Q|^2$ factor comes from the repetition of these iterations for every pair of states. $|\Sigma|^2$ represents the most critical factor, as $|\Sigma|$ rapidly increases with the stride level while $|Q|$ is almost constant. Also memory consumption is a concern, as the size of the data structures used by the algorithm are proportional to $|\Sigma|^2$.

Due to these limiting factors, reaching high stride levels with big rule sets by using the current state-of-the-art algorithms results unfeasible.

D. iNFAnt

iNFAnt [1] is an efficient NFA-based regex matching processor that runs on GPUs. Its algorithm consists of iteratively

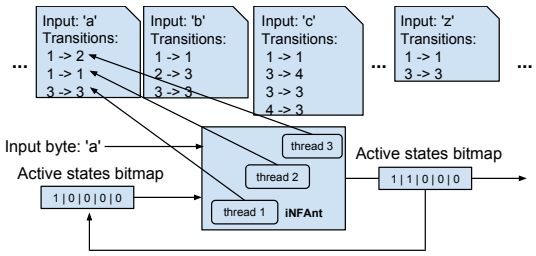


Fig. 3. Overview of the iNFAnt processing structure.

reading input symbols and updating the set of active states, represented by a bit vector where each bit corresponds to a state of the NFA. After reading a new input symbol, the algorithm looks for the transitions that can be triggered by that symbol. This operation is made simple by storing transitions already grouped by input symbol. These transitions are then used to update a bit vector that keeps the active states. As shown in Figure 3, several transitions are processed in parallel by assigning them to different threads, which contributes to alleviate the dependency of the update time on the number of transitions triggered by the input symbol. This is made possible by exploiting the high number of threads that can run on a GPU and the high memory bandwidth that can be obtained by adopting a clever memory access policy. Thread divergence is almost non-existent, as (by design) all threads perform exactly the same operations, although on different data (i.e., different transitions). However, some useless processing may occur: some threads analyze transitions that are not active in the current starting state and may become idle.

In addition to this form of parallelism, iNFAnt can also process several input strings in parallel, by assigning each string to a different group of threads. The scheduler can then exploit the large number of threads to hide memory latency, by scheduling new groups of threads when the current ones are waiting for data from main memory (memory access time is far beyond the processor cycle time), hence keeping the processor busy almost all the time.

Thanks to the two forms of parallelism, iNFAnt has a reasonably stable throughput, which depends on the average number of transitions per symbol. When the number of transitions triggered by a symbol exceeds the maximum number of threads supported by the GPU, the algorithm has to iterate transition processing, hence decreasing throughput.

IV. Multi-striding with Range Notation

The asymptotic time complexity of the state of the art stride-doubling algorithm depends mostly on the number of compound transitions to be generated, which tends to increase rapidly at each stride step. Hence, the time required for stride doubling becomes quite large even after the first doubling.

In order to reduce the impact of this problem, we change the way transitions are represented: transitions are grouped according to their source and destination states and the labels of the transitions of each group are stored by using *ranges* instead of individual symbols. For example, if we consider a NFA with the transitions that link q_0 to q_1 labeled by all the symbols between '97' and '122', these transitions could be replaced by a single transition labeled with the range '97|122'. When the symbols are not completely contiguous, more ranges are used.

The range notation is exploited in the stride doubling algorithm by treating ranges as atomic entities during the creation of compound transitions: rather than iterating on symbols, the algorithm just iterates on ranges and it creates compound transitions by combining ranges together. For example, a transition set labeled by the ranges '97|122' and '200|200', when combined with another set labeled only by the range '50|100, generates a combined set of transitions labeled with just two pairs of ranges: '97|122,50|100' and '200|200,50|100'. Moreover, the generation of these compound transitions requires only two iterations of the algorithm, compared to the huge number of iterations necessary if working with individual symbols[1]. With range notation, the asymptotic time complexity of stride doubling changes from $|Q|(\overline{N}_{fo} \cdot \overline{L}_s)^2$ to $|Q|(\overline{N}_{fo} \cdot \overline{R})^2$, i.e. the $\overline{L}_s$ factor, which counts the average number of transitions that link two states, is replaced by $\overline{R}$, which is the average number of *ranges* that are necessary to represent these transitions.

In the worst case, if a set of transitions does not include any contiguous symbols, a number of ranges equal to the number of symbols labeling the transition is generated. However, as regular expressions usually handle human-readable alphanumeric characters and the ASCII code represents them with contiguous values, the vast majority of NFA are expected to benefit from the range notation. Moreover, the advantages of this notation tend to be more visible when the NFA is more complex. In fact, complex NFA have more states and more transitions, hence higher probability to have transitions with contiguous symbols.

Finally, even with "unluckily incompressible" rule sets, it is still possible to greatly exploit the range notation starting from the second stride doubling iteration: as it will be shown in Section V, our alphabet compression algorithm generates equivalence classes in a way that increases (if possible) the amount of contiguous labels in each transition set.

V. Improving Alphabet Compression

The state-of-the-art algorithm for alphabet compression [4] is appropriate with small to medium sized NFA but still requires prohibitive resources with multiple-stride NFA resulting from rule sets of realistic sizes. Our new algorithm (shown in Algorithm 3) is more efficient in terms of both memory and processing requirements which allows it to handle larger automata. Furthermore, the algorithm also improves the effectiveness of the range notation. From the memory standpoint, the algorithm requires approximately four times less memory than the one in [4], as it needs only an integer per symbol pair (i.e., only the transition map itself) compared

[1]For the sake of precision, with individual symbols, the Cartesian product between symbols would have generated $(122-96+200-199)\cdot(100-49) = 1377$ pairs of symbols.

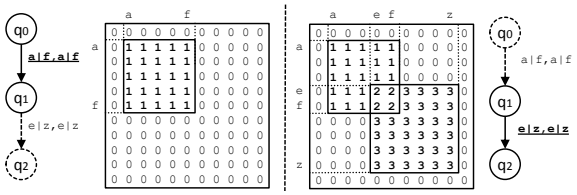


Fig. 4. Example of the improved alphabet compression algorithm.

to the two integers and two booleans[2] required by [4][3]. From the processing standpoint, our algorithm can potentially run at twice the speed of [4] as it performs a single iteration through the main data structure instead of two.

Algorithm 3 Improved alphabet compression algorithm.

```
1:  map[|Σ|][|Σ|] = 0
2:  size = 0
3:  for all q1 ∈ Q do
4:      for all q2 ∈ Q do
5:          buffer = {}
6:          for all (a, b) ∈ Σ × Σ | (q1, (a, b), q2) ∈ δ do
7:              if map[a][b] ∉ keys(buffer) then
8:                  buffer = buffer ∪ {map[a][b], ++size}
9:              end if
10:             map[a][b] = buffer[map[a][b]]
11:         end for
12:     end for
13: end for
```

The main idea of the new algorithm is to use a different criterion for the cluster division step (proposed in [4]): *all* the symbol pairs that can lead from q_1 to q_2 are remapped onto new equivalence classes, taking care of using the same new class for symbol pairs that previously belonged to the same class. For example, if pairs ('aa'), ('bb'), and ('cc') can lead from q_1 to q_2 and pairs ('aa') and ('bb') were previously mapped to the equivalence class 1 while ('cc') was mapped to 0, then ('aa') and ('bb') will be mapped to the same new symbol (e.g. 2) and ('cc') to a new other symbol (e.g. 3). In this way, correctness is still guaranteed, i.e. if two symbol pairs lead to the same destination states from each state, they are mapped to the same class.

The buffer hash-map is used to store the re-mapping operations performed at each cluster division step (it maps classes onto new classes). Initially there are no re-mapping operations (line 5). For each symbol pair that can lead from q_1 to q_2, if it was previously mapped to a class for which a re-mapping does not already exist in the buffer (line 7) a new class is created and a new re-mapping from the previous class to the new class is added to the buffer (line 8). Vice versa, if a re-mapping already exists in the buffer, this is simply used to remap the symbol pair.

Figure 4 shows how the algorithm works for the same FSA presented in Figure 2. The left-hand side of the figure shows the translation map resulting after processing the transitions from state q_0 to state q_1. All the symbol pairs that can lead from q_0 to q_1, corresponding to the square area with coordinates from 'a,a' to 'f,f', have been re-mapped to the same new class 1. In the second step, the transitions from state q_1 to q_2 are examined. They are labeled by symbol pairs corresponding to the square area with coordinates from 'e,e' to 'z,z'. This area can be divided into two parts: the former that overlaps with the area previously filled with class 1 and the latter that does not overlap with it and is still filled with the original class 0. These two parts are re-filled with two different new classes, namely 2 and 3, thus leading to the situation shown in the rightmost part of the figure.

Hence, as expected the algorithm has generated four different classes: class 0, corresponding to unused symbol pairs, class 1 corresponding to the symbol pairs that can trigger only the $q_0 \rightarrow q_1$ transitions, class 3 corresponding to the symbol pairs that can trigger only the $q_1 \rightarrow q_2$ transitions, and class 2, corresponding to the symbols that can trigger both.

The only difference in terms of output between the new algorithm and [4] is that we perform a remapping even when it is not strictly necessary, possibly leading to non-contiguous class numbers. For example, if the algorithm generates 4 classes it is possible that those are numbered 0, 2, 3, and 4, as the class 1 was overwritten in some intermediate steps of the algorithm. Even if the way integer values are assigned to classes is irrelevant from a theoretical point of view, the assignment of non-contiguous values may negatively affect stride doubling, because the effectiveness of the range notation decreases. For this reason, an additional phase is added to the algorithm where the generated classes are re-numbered in order to guarantee that they are identified by contiguous integers.This step is performed inside the algorithm that translates the NFA to the new alphabet (which has to be executed anyway after alphabet compression), hence adding a very small overhead to the overall process. The translation algorithm is very simple: it iterates through each transition of the NFA and substitutes the old symbol pair with the corresponding class number. When doing so, the class is also renumbered (if needed). This adds a couple of memory accesses at each iteration and it requires keeping a dictionary to remember which classes have already been renumbered. Clearly, the additional cost of class renumbering is negligible compared to the rest of the algorithm. The same applies for memory requirements, as the size of the additional dictionary is equal to the cardinality of the compressed alphabet, which is way less than the size of the support map.

Finally, as an added value, class renumbering chooses new class indexes with an heuristic that increases the amount of contiguous values in each set of transitions that connect two states: this is done in order to increase the efficiency of the range notation (i.e., to reduce $\overline{R}$). As the asymptotic time complexity of the stride doubling algorithm is proportional to $\overline{R}^2$, reducing $\overline{R}$ speeds up the next stride doubling iteration. The heuristic is based on the following idea. Let us consider the first state pair that is processed. If, for instance, the transitions that link these states are labeled by the 3 equivalence classes 563, 236 and 1243, the algorithm renames class 563 to 1, class 236 to 2 and class 1243 to 3, thus guaranteeing

[2]The vast majority of the compilers use an integer to store a boolean for performance reasons.

[3]For the sake of precision, our algorithm uses also an hashed map whose number of entries is equal to the average number of equivalence classes the transitions connecting two states belong to; however the size of this data structure can be considered negligible compared to the main data structure.

the contiguousness of their symbols. For the next state pairs, only the classes that occur for the first time are re-mapped to new (consecutive) integers. Hence, the labels of the transitions connecting the state pairs that are processed first will more likely have more contiguous symbols, while the ones of the state pairs processed last will probably have more sparse sets. By processing state pairs with more complex transition sets first, the overall number of ranges is reduced. This can be achieved in practice by sorting the state pairs according to the complexity of their transition sets before performing renumbering.

The time complexity of the new alphabet compression algorithm can be expressed as $|Q|^2 \cdot \overline{L_s}^2$. Compared to the complexity of the algorithm in [4] and reported in Section III-C, it can be noted that the $\overline{L_s}^2$ factor replaces $|\Sigma|^2$. This is an important improvement because usually $\overline{L_s}$ is lower than $|\Sigma|$, as typically in an NFA only few states are connected to each other with all the possible symbols of the alphabet.

A. Optimized remapping

In our experiments we found that often different state pairs are connected by transitions with the same symbols, i.e. the symbols that can lead from q_x to q_y are the same that can lead from q_z to q_t. In this case, the remapping done by the algorithm for the first pair of states would be completely overwritten when processing the second pair of states, as they refer exactly to the same area in the map, which means that the processing of the second pair of states could be safely omitted. As this case is frequent, this optimization leads to an important reduction of the processing time.

The high frequency of this case in real situations depends on several different causes. One is the already mentioned alphanumeric nature of most regular expressions. Another one is the way the NFA is typically generated from a set of regular expressions. For instance, the combination of two sub-expressions by the AND operator results in a duplication of the initial set of transitions. However, most frequently the presence of state pairs with exactly the same transition sets is caused by the '$.*$' pattern, included in many regular expressions, which causes the generation of state pairs q_x, q_y such that any symbol in the alphabet can trigger a transition from q_x to q_y. A state pair with this set of transitions forces the algorithm to update the *entire* support map, and, being these state pairs frequent, they are also responsible for a considerable portion of the time required to process the entire NFA. For this reason, omitting the processing of transition sets that have already been processed for previous state pairs represents a simple but very effective optimization.

In order to recognize and skip state pairs whose transition sets have already been processed, the proposed algorithm exploits the above mentioned sorting of state pairs according to the complexity of their transition sets. However, in order to guarantee that state pairs with identical transition sets are adjacent, this sorting has to be refined by introducing a secondary sorting criterion. The criterion used is not relevant, provided that a precedence relationship is defined for any two different transition sets. A simple way is to represent transition sets as strings of alphabetically ordered lists of state pair ranges (e.g. '{(a|f,a|f),(a|f,i|k)}'), and to use the alphabetic order of transition sets as the secondary sorting criterion. With state pairs sorted in this way, the algorithm compares the transition set of the n-th state pair to the one of the (n-1)-th state pair and, if they are found to be the same, it skips the remapping step.

Using a $O(n \cdot log(n))$ sorting algorithm, the asymptotic time complexity of the sorting phase is $|Q|^2 \cdot \overline{R} \cdot log(|Q|^2 \cdot \overline{R})$, which is negligible compared to the asymptotic time complexity of the algorithm that generates the equivalence classes. This stems from the fact that $\overline{L_s}$, which is the most critical factor in the formula, cannot be smaller than $\overline{R}$.

VI. Multi-Map Alphabet Compression

Although alphabet compression is usually very efficient in terms of symbol compression ratio, it is not always sufficient to enable further stride doubling steps with the current hardware. The inability to further reduce the alphabet size mainly depends on the necessity to create additional equivalence classes when transition sets overlap: for instance, in Figure 2 *three* equivalence classes are needed to represent the symbol pairs of *two* transition sets due to the overlap of their labels

As it would be difficult to further increase the compression ratio of the current alphabet compression technique, another way to enable further stride doubling is to increase the degree of parallelism of the packet processing problem. Let us denote N the number of input symbols to be processed and T the time taken to process them with a single processor. The idea is that if we are unable to process *one* string of $N/2$ symbols in a time $T/2$ with *one* processor because stride doubling is unfeasible, we may manage to process *two* different strings of length $N/2$ in $T/2$ by using *two* processors in parallel.

In order to split the matching problem into two separate problems, the transition sets of the NFA are partitioned into two disjoint groups in such a way that the amount of *overlapping* transition sets is minimized in each group. Then, the alphabet compression algorithm is applied separately on each group of transitions, generating two translation dictionaries, i.e. two target alphabets and two translation maps, instead of a single one. This operation is likely to be feasible even if alphabet compression is unfeasible on the whole transition sets. The two dictionaries are then used to build a NFA in which the two different alphabets coexist: some transitions are labeled by the symbols of the first dictionary, while the others are labeled by the symbols of the second one. If the transition sets are partitioned into the two groups by minimizing the number of overlaps, each group can be compressed more efficiently than the original set, hence reducing the overall number of generated symbols. This more efficient compression originates a simpler NFA (i.e. with fewer transitions) than the one generated by using the normal alphabet compression technique. As finding the optimum partitioning is NP-complete, a heuristic is used.

Because the new NFA contains symbols belonging to two different alphabets at the same time, both the input translation and the matching algorithm change: input translation generates

two strings, one for each dictionary, and the two resulting strings are then processed in parallel: a first process considers only symbols belonging to the first alphabet while the second one considers only symbols belonging to the second alphabet, but both processes update the same set of active states. In this way, the set is updated correctly, since in each state each possible transition of the original NFA is either applied by the first or by the second process.

From a practical point of view, using more processing units to perform the matching is not a problem as the target hardware is a GPU with hundreds of available processing elements. Moreover, each processing unit works on a NFA with, in average, half of the transitions per symbol, thus hopefully halving the per-symbol processing cost of each unit. At the same time, having simpler NFA makes it more likely to be able to complete an additional stride doubling step.

The modified compression algorithm (Algorithm 4) is similar to the one presented in Section V, but uses two maps, $map1$ and $map2$. For each state pair, the algorithm uses the map on which the update causes the generation of the smallest number of new equivalence classes. This is achieved by first updating both maps ($ApplySymbols$ corresponds to lines 5-11 of Algorithm 3, with the addition of the recording of the old state of the modified cells of the map) and then keeping only the one that has reached the smallest size ($Undo$ restores the map to its previous state).

Algorithm 4 The multimap alphabet compression algorithm.

```
1:  map1[|Σ|][|Σ|] = 0; map2[|Σ|][|Σ|] = 0
2:  size1 = 0; size2 = 0
3:  for all q1 ∈ Q do
4:      for all q2 ∈ Q do
5:          ApplySymbols(map1, size1, q1, q2)
6:          ApplySymbols(map2, size2, q1, q2)
7:          if size1 < size2 then
8:              Undo(map2, size2, q1, q2)}
9:          else
10:             Undo(map1, size1, q1, q2)
11:         end if
12:     end for
13: end for
```

The greedy algorithm used for the selection of the map has been chosen after having considered different possible alternatives:

- GRASP (Greedy Randomized Adaptive Search Procedure), which alternates greedy optimizations to randomizations in order to overcome local minimums;
- Top-Down, similar to the algorithm proposed in [9], which creates a different map per each transition and then it works by merging the maps and minimizing the amount of additional generated symbols at each pass;
- Top-Down2, which adds a randomization pass to the previous technique to avoid local minimums.

The considered algorithms have been evaluated in terms of both time and compression efficiency, by applying them to the NFA obtained using the same real rule sets used for our experiments (see section VIII for details). As the greedy algorithm is the simplest one, it is also the most time efficient. For what concerns compression efficiency, the results in Table II show that no heuristic outperforms the others in all cases. The greedy algorithm represents the best choice because it combines best time efficiency with good compression results on average.

TABLE II
ALPHABET SIZE WITH DIFFERENT MULTIMAP HEURISTICS.

Rule set	No compress.	Greedy	GRASP	Top-down	Top-down2
Snort FTP	463	313	310	313	310
Snort SMTP	1078	721	1050	695	695
Snort 534	4842	2094	3632	2094	2094
Snort HTTP	8272	6097	7568	10251	9420
ET HTTP	4353	2805	3256	3195	2934
Snort Full	16644	24123	18349	21020	21020

The multi-map algorithm requires approximately twice the time and memory compared to the original algorithm, because of the necessity to update two maps instead of a single one. However, thanks to the better compression provided, subsequent stride doubling steps will be performed faster and with less memory compared to the original technique.

A. Run-time packet processing

Even if, in theory, multi-map alphabet compression should be able to greatly improve the maximum achievable stride level and the processing throughput, applying further stride doubling and alphabet compression iterations to an NFA that has been compressed with the multi-map algorithm causes additional issues that must be properly handled, and that can be explained by taking into account the example in Figure 5. When a 2-map alphabet compression is performed on the uncompressed 2-stride NFA of Figure 5a, the transition from q_1 to q_2 and those from q_2 to q_4 are assigned to the first support map, while the transition from q_2 to q_3 is assigned to the second support map. The support maps generated in this way are shown in Figure 5e, while the compressed NFA is shown in Figure 5b.

If the input string is '`a,b,c,d`', the following two strings are generated by applying the two translation dictionaries: `1,2` (by using the first map), and `0,4` (by using the second map). The packet processor has to concurrently process symbols `1` and `0` at the first iteration, and symbols `2` and `4` at the next one. At the first iteration, if the active state set includes only state q_1, the transition from q_1 to q_2 is triggered by symbol `1`, thus adding q_2 to the new active state set. Then, at the next iteration, the active state set includes only q_2 and both the transitions from q_2 to q_3 and from q_2 to q_4 are triggered by symbols `4` and `2` respectively. Thus, the final active state set includes exactly states q_3 and q_4, as expected.

If a new stride doubling step is performed, e.g., to get the 4-stride NFA of Figure 5c, the presence of two translation maps complicates the input string translation and the matching algorithm. In fact, let us assume we apply exactly the same algorithms explained above. The input string '`a,b,c,d`' is translated into the two strings '`1,2`' and '`0,4`', using the two maps. Then, the matching algorithm consumes the pairs of symbols '`1,2`' and '`0,4`' concurrently. If initially the only active state is q_1, the pair '`1,2`' triggers the transition from q_1 to q_4, while the pair '`0,4`' does not trigger any transition. Thus, the final active state set includes only state q_4, which

is wrong, the reason being because the transition ‘1,4’ has not been fired while it should have been. The particularity of the symbol pair 1,4 is that its two components belong to different translation maps while our translation algorithm only generates two translations: one made only of symbols of the first alphabet, another made only of symbols of the second alphabet. In order to make sure that we get the right result of matching at stride level 4, it is then necessary to generate more translations of the input string, including all the possible combinations of the two translation maps. In our example this implies generating 4 translations, i.e. the strings ‘1,2’, ‘1,4’, ‘0,2’ and ‘0,4’, where the first string is obtained by using the first map for both symbols in each pair, the second one is obtained by using the first map for the first symbol and the second map for the second symbol in each pair, and so on.

Applying again the 2-map alphabet compression algorithm on the 4-stride NFA, we get two new translation maps, shown in Figure 5f, and the new compressed NFA shown in Figure 5d. Having introduced two new alphabet translations, the 4 input strings we had for the uncompressed 4-stride NFA have to be translated into 8 strings: 4 obtained by applying the first new translation map and 4 obtained by applying the second one. Taking again the above example, the string 1,4 is translated by the leftmost map of Figure 5f into y while the remaining strings are translated into 0. Using the rightmost map, the same four strings are translated into ‘z’ (generated from string 1,2), and three equal symbols ‘0’, ‘0’, ‘0’.

In general, our approach increases the compression rate but the number of strings to be processed in parallel is multiplied by the number of maps (with 2 maps it is doubled) at each alphabet compression step. At each stride doubling step, instead, the number of strings to be processed in parallel is squared, and the length of the input strings is halved. Then, after n combined stride doubling and alphabet compression steps, the number of strings that we must process in parallel becomes $S_n = m^{2^n-1}$ where m is the number of maps used for alphabet compression, and the length of the input strings becomes N/n, where N is the original length of the input string. However, increasing the number of input strings to be processed in parallel is not necessarily an issue: it just makes the problem more parallel. Furthermore, even if some threads are “wasted”, the GPU code can minimize this overhead by greatly reducing the total amount of memory accesses, which is the actual main bottleneck of iNFAnt. In fact, we can note that multiple occurrences of the same symbol in different strings at the same offset are redundant because they lead to the same target states. Detecting redundant symbols and avoiding their processing is a way to reduce memory accesses.

Redundancies happen frequently and can be observed even in our simple example: of the 8 1-symbol strings generated from the translation of the original 4-symbol input, 6 strings are identical (‘0’). Hence, five of them could be ignored.

B. Redundancy in input string translations

Experimentally, it is easy to show that the phenomenon of symbol redundancy occurs with any rule set, and that, after a certain stride level, redundant symbols tend to increase. The

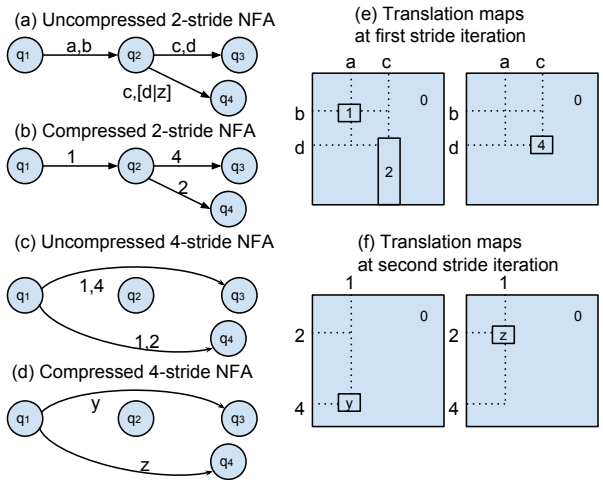


Fig. 5. Multi-map compression of a sample NFA at 2x and 4x stride levels

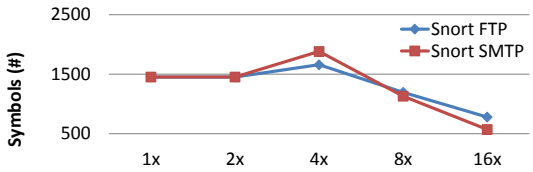


Fig. 6. Amount of non-repeated symbols present in random input data streams when translated to be processed with 2-map multistrided NFA

experiment consists of generating the NFA at different stride levels for the considered rule set, using the stride doubling and the 2-map alphabet compression algorithms proposed here. Then, various pseudo-random input packets are generated, and, for each stride level they are translated by using the dictionaries originated by the alphabet compression, as previously explained. Then, the input strings so generated for each stride level are reduced by removing redundant symbols (i.e., symbols that are equal to other symbols occurring in other strings at the same position). For example, if we have the two strings ‘1,2,3,4’, and ‘**1,**0,**3,**3’, we can remove the symbols in bold as they are the repetition of symbols occurring at the same position in the other string, hence leaving only the meaningful symbols. Finally, the total length of all the reduced strings, which represents the total number of non-redundant symbols, is computed.

Figure 6 plots the results of this experiment using 1500 bytes packets and the rule sets used by the popular Snort NIDS to analyze FTP and SMTP packets (each plot refers to a single rule set). These rule sets have been chosen because of the possibility to reach high stride levels thanks to their reduced size, consisting in about 20 simple regular expressions (more details on the rule sets will be presented in Section VIII).

In absence of symbol duplications there would be no length change when moving from 1x to 2x (because 2 strings of length $N/2$ each are generated), but there would be a length increase of two times when moving from 2x to 4x (because 8 strings of length $N/4$ each are generated), an increase of 16 times when moving from 4x to 8x (because 128 strings of length $N/8$ each are generated) and so on. In Figure 6, instead, we can see that from 2x to 4x the data increase is

way less than the doubling that would occur in the absence of symbol duplications, while at further stride levels there is even a linear reduction of the total number of symbols rather than the exponential growth that would occur in the absence of symbol duplications.

Intuitively, as the stride level increases, overlapping transition sets that need to be separated in order to avoid generating extra equivalence classes are less likely to occur. This happens because at each stride level transition sets are compressed into fewer single classes and, at the same time, the size of the support maps increases. For this reason, support maps become increasingly less populated (particularly the secondary ones), and the amount of input string patterns with several different possible translations dramatically decreases at each stride level. This trend can also be understood considering the asymptotic behavior: in the hypothesis that we can compute the 2^k-stride NFA for arbitrarily large values of k using 2-map compression, we would reach a limit situation where the input string is translated into strings of length 1. In this extreme case, if N is the number of accepting states (which typically corresponds to the number of regular expressions in the rule set), we would necessarily end up with $N + 1$ meaningful symbols, each one leading from the initial state to one of the accepting states, plus an extra symbol representing all the other cases. Hence, asymptotically, the number of meaningful symbols tends to $N + 1$, where N is the number of accepting states of the NFA. Consequently, as the rule sets used in the experiments presented in Figure 6 are composed of about 20 regular expressions each, it can be expected that the number of meaningful symbols to be processed will tend to about 20, as well as the alphabet size. This explains the reduction that we can already observe starting from the 8-stride NFA (where 128 strings are generated when translating each input string).

The behavior in the "transient phase", represented by the first stride levels, as well as the maximum number of useful symbols, depends on the complexity of the rule set taken in consideration. However, in our test cases, all the rule sets for which it was possible to go beyond the "4x barrier" have generated a huge amount of redundant, useless symbols.

Based on these considerations, iNFAnt has been optimized to take advantage from this phenomenon and avoid to process useless symbols, with the consequence that it can achieve high performance at any stride level, as shown in the next section.

Finally, it is worth noting that the maximum number of useful symbols to be processed does not depend on the input string: there is, in fact, an "upper bound" that depends only on the rule sets, thus making it impossible to forge a malicious incompressible input data string.

VII. iNFAnt Improvements

After presenting the algorithms that improve the *generation* of multi-stride NFA, which are independent from the actual regex processor, this section presents the adaptations of iNFAnt to the above mentioned algorithms. The modifications were designed to maintain the excellent properties of the original implementation in terms of (limited) code divergence and coalesced memory access patterns.

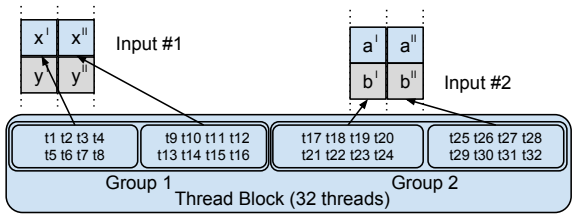


Fig. 7. Thread task scheduling during the processing of 2 input messages translated in 2 different strings each

nVidia GPUs are composed of multiple processing units, each one able to execute one or more thread blocks in parallel, with 32 to 1024 threads per block. The original version of iNFAnt followed this architecture closely, by assigning a different network packet to each block and by using one thread per transition in the computation of the next active states. However, when operating at high stride levels this choice becomes no longer adequate because the average number of transitions per symbol tends to decrease, often becoming much lower than 32. In this case, the architectural constraint of having a minimum of 32 threads per block implies that some threads will remain idle, thus wasting resources.

For this reason, our new thread hierarchy defines a "thread group" that does no longer coincide with the hardware thread block. Particularly, each thread block is logically partitioned into several thread groups, each one processing a different input string, which comes from either different network packets, different maps, or both. This re-organization of threads not only enables a reduction of the number of inactive threads (achieved also in [16] and [17]), but also helps to better manage the parallel processing of the several string translations that originate from multi-map compression. In the new algorithm, the threads of each group cooperatively process all the translations of a single input string.

Figure 7 shows an example where a 32-wide thread block has been logically partitioned to process two packets, each one translated into two different input strings because of the 2-stride NFA with 2-map compression, hence resulting in 4 distinct thread groups (8 threads wide) running in parallel. In that picture, each byte from each input packet (i.e., packets *Input #1* and *Input #2*) is in fact translated into two different symbols (e.g., *x'* and *x"* for the first packet, *a'* and *a"* for the second), which represent the actual input for a different thread group. In this way we maintain the parallel processing of the transitions associated to each input symbol (in the example, each symbol is processed by eight threads), as in the original iNFAnt version, but reducing the number of useless threads.

Thanks to the nVidia GPU hardware architecture, processing different strings in the same thread block also enables a reduction of memory accesses. In fact, as discussed in Section VI, the probability that two or more threads have to process the same symbol at the same time in the same thread block is very high, due to the symbol redundancy phenomenon discussed in Section VI-B. When this happens, the memory management unit of the GPU can recognize the presence of memory requests for the same location and join them together, thus reducing the real amount of accesses to the main memory.

TABLE III
THE RULE SETS USED FOR TESTING.

Type	Name	Rules (#)	Size (KB)	States (#)	TpS (#)	Thr (Mpps)
Small	Snort FTP	17	4	132	15	951.55
	Snort SMTP	26	12	433	30	809.48
Medium	Snort 534	534	208	9538	127	289.89
	Snort HTTP	189	92	3538	295	273.71
Large	ET HTTP	457	428	18425	525	500.68
	Snort Full	1514	1100	47168	1696	25.62

Thanks to this feature, as far as we manage to process all the translations of an input string by threads belonging to the same block, the amount of actual memory accesses corresponds to the numbers plotted in Figure 6. This means that even if the number of symbols that must be processed increases at each stride doubling, the code can actually get rid of this (theoretical) additional load.

However, this is possible as far as the number of translations is small enough to fit into a single block. In order to make this possible in a broader range of cases, an additional optimization has been added to the code that performs input translation: the data translator "compresses" its output by completely dropping the strings that contain only redundant symbols. This trivial compression mechanism dramatically reduces the number of translations that must be processed for each input string. This was confirmed by our experiments: even at the highest stride levels and with the most complex rule sets, we never had to process more than 16 translations per input string.

This reduction of the number of translations also reduces the total number of threads required, which contributes to increase the scalability of the solution, because the maximum number of threads is an hardware constraint. Moreover, if the number of strings to process increases too much it is even possible that the data transfer between the CPU and the GPU becomes the main bottleneck of the system, thus limiting the maximum achievable throughput.

VIII. EXPERIMENTAL EVALUATION

The experimental evaluation of our algorithms has been carried out on an Intel i7-960 quad core workstation running at 3.20Ghz, 12GB RAM with an nVidia Tesla c2050 GPU.

Table III lists all the rule sets used in our tests along with their size, average number of transitions per symbol (TpS), and average throughput as obtained by vanilla iNFAnt, without multi-striding. The throughput has been measured by taking into account only the GPU kernel execution time; more details about the overall processing costs will be shown in Section VIII-D. Rule sets have been classified as *small*, *medium* or *large*, depending on the complexity of the generated NFA; some have been extracted from the Snort and EmergingThreats (ET) commercial ruleset databases. *Snort Full* includes the full Snort rule set, with the only exclusion of the rules that have no standard PERL syntax. *Snort 534* is a selection of 534 rules used as benchmark in [3]. The other Snort and ET rule sets have been built by selecting only the rules of a single protocol, specified in the name of each rule set.

A. Multi-stride NFA building process

This section compares the performance of our algorithms for building multi-stride NFA (named "*new*" in the single-map form and "*2-new*" in the 2-map form) with the state-of-the-art algorithms by Becchi and Crowley [4], named "*TACO13*", implemented based on the pseudo-code presented in [4]. Other algorithms presented in Section II, such as [2], [8], [1], are so inefficient that often they cannot even complete our tests and hence they have not been considered in our analysis. In order to evaluate the contribution of the state pair sorting and the consequent elimination of the redundant transitions (Section V-A), we present also the results obtained with a version of our single-map algorithms (named "*unopt*") that does not include this optimization.

The performance of building the multi-stride NFA has been evaluated by measuring the total time and the maximum amount of memory taken by each algorithm on our rule sets. The time taken for 2- and 4-stride NFA have been measured separately. Measurements have been repeated in order to obtain statistical significance. Results, plotted in Figure 8, have been normalized taking the performance of the *TACO13* algorithm (executed with a preliminary compression of the initial alphabet, as proposed in [4]) as reference. Vertical lines represent the min and max obtained values while rectangles represent the average value plus and minus standard deviation.

Figure 8 shows that the various algorithms behave nearly the same when considering the 2-stride level, particularly with respect to the memory requirements (Figure 8b). The reason is that the size of the most critical memory structure (the support map of size $|\Sigma|^2$) is negligible compared to the memory required by the software to load (e.g., run-time libraries).

The situation changes when moving to stride level 4 (Figures 8c,d), where *unopt* is, on average, 10 times faster than *TACO13*. Adding state pair sorting and redundant string elimination further reduces the time by another order of magnitude, while the 2-map compression is more than three orders of magnitude faster than *TACO13*. Memory consumption improves considerably as well: the new algorithm halves the requirements compared to *TACO13*; the theoretical *4x* improvement (our algorithm uses one fourth of the memory used by *TACO13* for each item in the map) cannot be reached because of the additional data structures needed by the algorithms, whose occupancy is not negligible in the case of simple NFA. Instead, multi-map alphabet compression greatly reduces the alphabet size, thus decreasing the size of the support structures as well.

Figure 9a shows the results of another experiment that has been performed to evaluate the maximum stride level that can be obtained in a 24-hours time span. For a given algorithm and rule set, the experiment consists of applying the algorithm to iteratively double the stride level of the NFA until the 24 hours limit is reached. The maximum obtained stride level is shown in the graph. In four cases out of six *new* outperforms *TACO13*, while *2-new* does even better because it iterates at least one time more than *TACO13* with all the rule sets, and even more times in the simplest cases.

One of the motivations for the run-time processing improvement when using multi-map alphabet compression is the

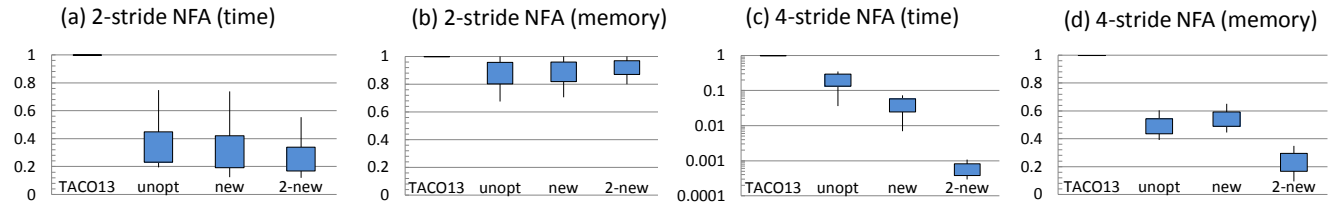


Fig. 8. Comparison of total required time and maximum required memory when generating 2-stride and 4-stride NFA for all rule sets

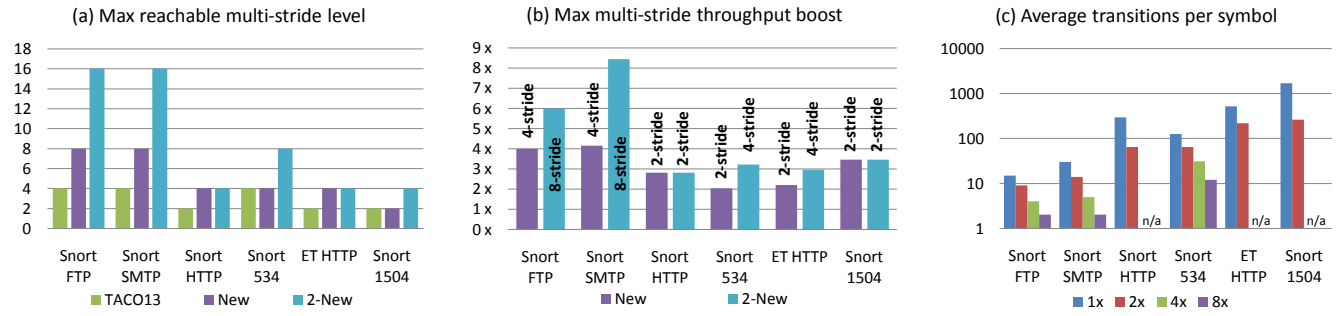


Fig. 9. Per-ruleset and per-algorithm performance comparison: (a) maximum stride level reachable in a 24-hour timespan, (b) maximum throughput boost of iNFAnt and (c) average amount of transitions per symbol

expected reduction of the average amount of transitions per symbol when the stride level increases. This value greatly influences the average processing cost per symbol in iNFAnt and, as described in Section VI, it is expected to be at least halved at every iteration of the multi-striding algorithm. This reduction has been experimentally observed on the selected rule sets, as shown in Figure 9c, which reports this number at different stride levels for various rule sets. This reduction is always present at all stride levels and with any rule set; even, this effect seems to be amplified with very large rule sets. For example, *Snort 1504* achieves a reduction rate >80%.

B. Run-time traffic processing

This section analyzes the run-time performance in regex matching that can be achieved by using the new algorithms. The throughput of the new iNFAnt was measured for each rule set with the NFA obtained by the *new* and *2-new* algorithms, using a set of real traffic captures taken from our University network. TACO13 has been omitted as its NFA coincides with *new*. Each result has been normalized with respect to the throughput obtained by using iNFAnt with the 1-stride NFA for the same rule set, which was reported in the last column of Table III; the resulting speedups are shown in Figure 9b. The tests were repeated also with two types of synthetic traffic, the first one made of packets filled with randomly generated payloads, and the second one built as a worst case, by maximizing the number of active states generated during processing. The measured throughput was rather stable, resulting (in the worst case) in a 20% worsening compared to the real traffic. For the sake of brevity, we reported in Figure 9b only the results obtained with real traces.

In this experiment, the NFA with the largest stride level that could be computed and loaded in the GPU has been used for each ruleset. These stride levels are indicated in the columns of Figure 9b. By comparing them with the ones reported in Figure 9a, it is possible to notice that often the NFA with the maximum stride level that can be computed in a 24 hours time span is too large to be loaded in the GPU.

As already noted in [1], a 2x throughput boost can be expected at each stride doubling. Figure 9b substantially confirms the above findings, showing that there are cases in which the boost is even higher and cases for which it is slightly lower. The results also show that, although with *2-new* the throughput boost at each stride doubling is usually slightly less than 2x, the overall speedup is almost always greater than that achievable with *1-new*. The only exceptions are *SnortHTTP* and *Snort1504*, for which the same boost is achieved with *1-new* and *2-new*, which is due to the memory limitations of the GPU which prevent the largest NFA from being loaded. However, this problem should disappear with future generations hardware, as GPU vendors are constantly increasing the amount of memory on their video boards.

Divergence, as measured by the nVidia profiler, is usually below 5% and never exceeds 10%.

C. Multi-map alphabet compression efficiency

As the NFA generated by the multi-map algorithm are completely different from the ones obtained with the other techniques, additional tests have been performed in order to show that the benefits of multi-map do not depend on the rule sets. This is important in order to guarantee that the efficiency of alphabet compression cannot be compromised by malicious rule sets, carefully created in order to decrease the throughput of the pattern matching algorithm.

These additional tests exploit randomly generated rule sets, obtained by using a generator that extends the algorithm described in [18] with the possibility to generate "nested" regular expressions such as $ab(c.*d)+$, thus yielding more realistic

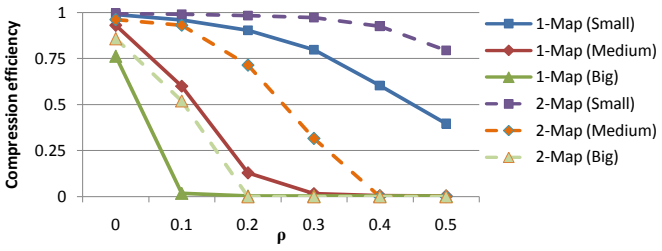
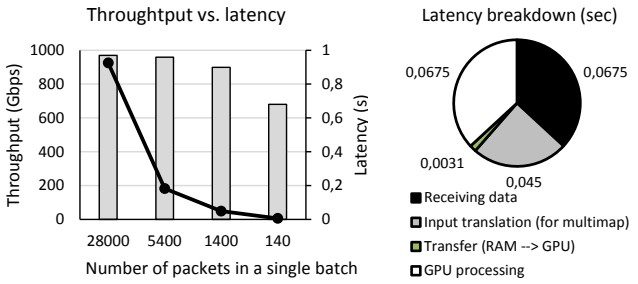


Fig. 10. Compression efficiency of 1-map and 2-map alphabet compression



(a) (b)

Fig. 11. (a): Throughput (bars) and latency (line) with different sizes of the processing batch; (b): latency breakdown in case of a batch of 5400 packets.

rule sets. Moreover, this generator includes a parameter (ρ) that controls the percentage of wildcards that occur in the generated expressions. With $\rho = 0$, pure plain sequences of characters (or sets of characters) are generated, while with $\rho = 0.5$ each generated character (or set of characters) is associated to one of the possible wildcards, like repetition operators such as $+$, $*$, $\{a, b\}$ or the optionality operator $?$.

Three different classes of rule sets have been generated: "small" rule sets, made by 20 regular expressions of 20 characters each, "medium" rule sets, made by 50 regexps of 50 characters each, and "large" rule sets, made by 100 regexps of 100 characters each. For each class, rule sets with different values of ρ have been generated in order to study the compression efficiency when varying this parameter.

For each generated rule set, the 2-stride NFA has been built by using both *new* and *2-new*. Then, for each NFA a *compression efficiency ratio* has been calculated as $C_{eff} = (|\Sigma| - |\Sigma'|)/|\Sigma|$, where $|\Sigma|$ and $|\Sigma|'$ are the alphabet sizes of the NFA measured before and after compression (*1* means that the alphabet size has been reduced to a negligible value, while *0* means that the compressor was unable to reduce the alphabet size). For each value ρ, 10 rule sets have been generated for each class and the average compression efficiency has been plotted in Figure 10. The graph shows that the efficiency of the 2-map algorithms (dashed lines) is always better than that of the 1-map algorithms (continuous lines). Moreover, it clearly shows that, even if with medium or large rule sets the efficiency drops to 0 with complex rules (i.e., when ρ increases), multi-map compression can support more complex rules than its single-map counterpart.

D. System-wide run-time processing

The last tests aim at assessing run-time performance from a system-wide perspective, hence considering that packets coming from the network are (*i*) buffered, (*ii*) translated according to the multiple maps in use, (*iii*) transferred to the GPU through the PCI Express bus, and finally (*iv*) processed by the NFA algorithm. Figure 11(a) shows the system throughput (bars) and the maximum latency experienced by each packet (line) with batches of different sizes. Experiments use the real traffic traces and the 4-stride Snort_534 dataset, with 2 maps. The used network is a 10 Gigabit Ethernet. Although the throughput is maximum with the largest batch, it decreases slowly with smaller batches, while at the same time latency improves considerably. For instance, a batch of 1400 packets achieves 92% of the throughput of the largest batch (28000 packets), but latency drops from 926 to 49 ms. This confirms that our system does not trade throughput for latency and it can achieve high throughput while keeping latency small.

Figure 11(b) explores the contribution of the above steps to the overall latency. While, by definition, buffering and GPU processing take the same time because we feed our system with the maximum amount of data that can be processed, the CPU-to-GPU transferring time is negligible in all conditions. Instead, translation time, which increases with higher stride levels, can contribute significantly to the overall latency and it can also potentially limit the throughput when its duration exceeds the GPU processing time (as in some of our 8x tests), as the usual parallelization strategy (i.e., the CPU translates the n-th packet while the GPU analyzes the (n-1)-th packet) is no longer enough. However, the symbol translation code used in the experiment could be greatly improved: features like the data compression presented in Section VII can reduce the global amount of memory accesses, thus increasing the translation throughput; furthermore several CPU cores can be used to translate (*i*) multiple packets in parallel, (*ii*) multiple maps in parallel, and (*iii*) even partition the same input packet on different cores, as the translation process is stateless.

Finally, it can be noted that the throughput of the translation algorithm depends mainly on the number of memory lookups, which can be safely assumed having a fixed cost because dictionaries are so large to prevent the CPU to cache them. Since the amount of memory accesses is only affected by the number of used dictionaries and not by their size, the cost of the translation depends only on the stride level. Consequently, the NFA (hence the rule set) has almost no influence on the translation process. In conclusion, the translation process does not represent a problem for the time being, while several possible improvement strategies are available for the future.

IX. Conclusion

Multi-striding, a well-known technique to improve the efficiency of regex matching, presents some critical issues in the process of building the multi-stride automaton. This paper has presented new algorithms to overcome these limits, focusing on NFA-based regex matching. First, a more efficient set of algorithms (in terms of processing time and memory) to perform stride doubling and single-map alphabet compression has been proposed. Second, a new algorithm for alphabet compression

based on the multi-map technique has been presented, along with a companion GPU-based matching algorithm that makes it particularly effective because of the capability to better exploit the parallelism allowed by multi-map compression in the NFA matching at run-time.

The experimental results show that the new algorithms for building multi-stride NFA outperform the existing ones in terms of both time and memory, hence enabling either to handle more complex rule sets or to reach higher stride levels. This result, coupled with the more efficient run-time matching algorithm on GPUs, yields faster processing of network traffic. Furthermore, tests with randomly generated and worst case patterns show that the results do not depend on the chosen rule sets nor are sensible to properly-crafted malicious traffic.

Two future research directions are envisionable. First, a more efficient memory management in iNFAnt, mimicking the technique proposed in [19], which enables the processing of larger NFA. This could broaden the applicability of our algorithms because we are able to build very efficient multi-stride automata but we may be unable to use them with the current hardware as they do not fit in the (limited) memory size of current GPU cards. Second, the translation process can represent another area of research, although with the current hardware it is not yet a problem.

REFERENCES

[1] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "infant, nfa pattern matching on gpgpu devices," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 20–26, Oct. 2010.

[2] B. Brodie, D. Taylor, and R. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," in *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, May. 2006, pp. 191–202.

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

[4] ——, "A-dfa, a time- and space-efficient dfa compression algorithm for fast regular expression evaluation," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 4:1–4:26, 2013.

[5] ——, "Efficient regular expression evaluation, theory to practice," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 50–59.

[6] N. Yamagaki, R. Sidhu, and S. Kamiya, "High-speed regular expression matching engine using multi-character nfa," in *International Conference on Field Programmable Logic and Applications*, 2008, pp. 131–136.

[7] M. Avalle, F. Risso, and R. Sisto, "Efficient multistriding of large non-deterministic finite state automata for deep packet inspection," in *IEEE International Conference on Communications*, 2012, pp. 1094—-1099.

[8] L. Yang, R. Karim, V. Ganapathy, and R. Smith, "Fast, memory-efficient regular expression matching with nfa-obdds," *Computer Networks*, vol. 55, no. 15, pp. 3376–3393, 2011.

[9] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," *Proceedings of the 4th international conference on Security and privacy in communication networks - SecureComm '08*, p. 1, 2008.

[10] C. Meiners, J. Patel, E. Norige, E. Torng, and A. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010, pp. 8–8.

[11] N. Hua, H. Song, and T. V. Lakshman, "Variable-stride multi-pattern matching for scalable deep packet inspection," *IEEE INFOCOM 2009 - The 28th Conference on Computer Communications*, pp. 415–423, 2009.

[12] R. Smith, N. Goyal, J. Ormont, K. Sankaralingam, and C. Estan, "Evaluating GPUs for network packet signature matching," in *Proceedings of ISPASS '09*, 2009, pp. 175–184.

[13] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, "Regular expression matching on graphics hardware for intrusion detection," in *proceedings of RAID '09*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 265–283.

[14] L. Wang, S. Chen, Y. Tang, and J. Su, "Gregex: Gpu based high speed regular expression matching engine," in *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, ser. IMIS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 366–370.

[15] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus, a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 317–328.

[16] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, "Gpu-based nfa implementation for memory efficient high speed regular expression matching," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 129–140.

[17] X. Yu and M. Becchi, "Gpu acceleration of regular expression matching for large datasets, exploring the implementation space," in *Proceedings of the ACM International Conference on Computing Frontiers*, ser. CF '13. New York, NY, USA: ACM, 2013, pp. 18:1–18:10.

[18] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures." in *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008.

[19] X. Liu, X. Liu, and N. Sun, "Fast and compact regular expression matching using character substitution," in *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2011.

Matteo Avalle (matteo.avalle@polito.it) graduated in Computer Engineering in 2010 from Politecnico di Torino, Turin, Italy, and received a Ph.D. degree in Computer Engineering from the same university in 2014. His research interests include formal methods applied to security protocols and parallel/distributed computing applied to network packet processing and traffic analysis. He is now Chief Technical Officer at Fluentify LTD.

Fulvio Risso (fulvio.risso@polito.it) graduated in Computer Engineering in 1995 and received the Ph.D. degree in Computer Engineering in 2000, both from Politecnico di Torino, Torino, Italy. He is currently assistant professor with the Department of Control and Computer Engineering of Politecnico di Torino. His current research activities focus on efficient data processing applied to traffic analysis, data plane packet processing and software-defined networks.

Riccardo Sisto (riccardo.sisto@polito.it) received the M.S. degree in Electronic Engineering in 1987, and the Ph.D. degree in Computer Engineering in 1992, both from Politecnico di Torino, Torino, Italy. Since 1991 he has been working at Politecnico di Torino, where he is currently Full Professor of Computer Engineering. His main research interests are in the area of formal methods applied to software and communication protocols, distributed systems, and computer security.