Efficient Multistriding of Large Non-deterministic Finite State Automata for Deep Packet Inspection

Efficient Multistriding of Large Non-deterministic Finite State Automata for Deep Packet Inspection

Matteo Avalle, Fulvio Risso, Riccardo Sisto
Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italia
Email: {matteo.avalle, fulvio.risso, riccardo.sisto}@polito.it

***Abstract*—Multistride automata speed up input matching because each multistriding transformation halves the size of the input string, leading to a potential 2x speedup. However, up to now little effort has been spent in optimizing the building process of multistride automata, with the result that current algorithms cannot be applied to real-life, large automata such as the ones used in commercial IDSs, because the time and the memory space needed to create the new automaton quickly becomes unfeasible. In this paper, new algorithms for efficient building of multistride NFAs for packet inspection are presented, explaining how these new techniques can outperform the previous algorithms in terms of required time and memory usage.**

I. Introduction

Deep packet inspection is still at the foundation of many security tools, such as Intrusion Detection Systems (IDS), firewalls, spam filters and more. While string matching was the most common technique used in the past, the complexity of nowadays attacks requires the deployment of sophisticated tools based on regular expressions (regex).

One of the most memory-efficient ways to represent a regular expression (or a set of regular expressions) is based on *Nondeterministic Finite-state Automata* (NFA). An NFA can be seen as an oriented graph in which nodes represent states and arcs represent transitions, labeled with the input symbols (bytes). A simple example of this type of representation can be found in Figure 1; more details about NFA will be presented in Section II-A.

Since run-time throughput and memory consumption represent the key factors that characterize a deep packet inspection system, much effort has been dedicated to new optimization techniques that increase processing throughput and/or reduce memory consumption, thus enabling the matching of complex regular expression patterns. One of those techniques is *multistriding*, which creates a new NFA in which each transition consumes multiple bytes instead of just one, as shown in the example in Figure 1. This modification can potentially achieve an impressive performance boost, as it linearly reduces the number of steps (and memory accesses) required to process each input string. In practice, the length of the input string reduces to $1/n$, where n is the number of bytes grouped together. On the other side, it can increase the size of the NFA, as the space of symbols becomes much larger (256^n, where n is the number of bytes grouped together), potentially triggering a quick growth in the number of transitions.

This problem can be mitigated through an *alphabet compression* pass, which bases on the observation that many

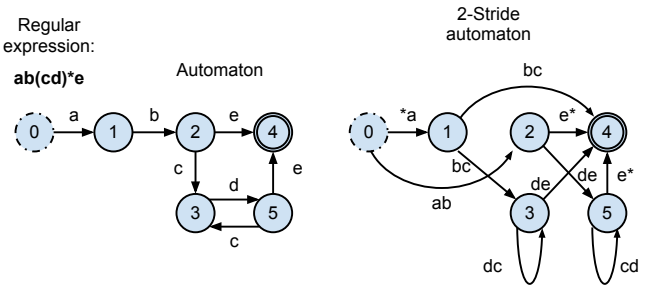

Fig. 1. A simple regular expression shown in its textual form, in NFA form and in 2-strided NFA form.

symbols are equivalent as they are always used together in every transition of the NFA. In case some symbols (e.g., the numbers from `'00'` to `'99'`) are equivalent, they can be replaced by a single one (e.g., `'x'`), reducing the cardinality of the symbol set and hence the complexity of transitions in the NFA. Although this technique requires that the input strings are translated into the new language (e.g., all the instances of the characters `'00-99'` in the input packet must be replaced by the symbol `'x'`), the impact on the run-time throughput is usually negligible on modern processors as the access to the input strings happens sequentially. This technique is discussed in more detail in Section II.

Unfortunately, the algorithms available in the literature for creating multistride automata are hardly suitable for real-world, complex patterns. Even if those algorithms can run "off line" and hence do not impact on the performance at run-time when network traffic is being filtered, we cannot accept that the computation takes several months on modern CPUs, or that a machine with 12GB of RAM is not sufficient, which sometimes happens when using the tools presented in [3].

This paper addresses the problem of building large multistride NFAs efficiently, by proposing new algorithms and mixing them in a better building process, paying particular attention to computational complexity and memory consumption. For instance, to the best of the authors' knowledge, this paper studies for the first time the impact of the well-known technique of *NFA minimization* in the building process of multistride automata, in combination with improved multistriding and alphabet compression algorithms.

This paper is structured as follows: Section II recalls the basis of the NFA theory and of the underlying techniques;

Section III describes the state of the art algorithms, then Section IV enters into the details of the three new algorithms by explaining how they represent an improvement over the existing approaches both in computational complexity terms and in memory usage terms. Section V describes performance evaluation results while finally, in Section VI, some conclusions are drawn and the planned future works are presented.

II. Background

A. The NFA model

An NFA is a 5-tuple consisting of: *(i)* a finite set of states; *(ii)* a finite set of input symbols that represent the input alphabet (bytes in our case); *(iii)* a finite set of transitions, which are triples made of a current state, a non-empty set of input symbols (also called label) and a next state; *(iv)* a set of initial states and (v) a set of accepting states. An NFA will accept the packet $b_1...b_n$ if there is a path from an initial state to an accepting state, where the transitions on the path contain the symbols $b_1...b_n$ in their label.

Due to the nature of NFAs, multiple states can be reached at the same time by using the same string $b_1...b_n$: this feature distinguishes this family of automata from the Deterministic Finite-state Automata (DFA), in which only a single state path is possible for any input string.

B. Multistriding

The multistriding technique is based on building an equivalent NFA in which each transition consumes n input bytes instead of one (n is said the multistride level). The resulting NFA usually has the same states as the original NFA, while the number of transitions (due to the explosion of the cardinality of the input symbols) tends to grow exponentially. This growth is one of the biggest problems of the multistriding technique.

Most multistriding algorithms operate by iteratively merging pairs of input symbols; each iteration halves the input string length, and potentially squares the number of possible input symbols.

The multistriding algorithm presented in [2] (which represents the current state of the art) iterates through all the states of the NFA and searches, for each state, the states reached when following the pair of transitions associated to each combination of two symbols of the alphabet. For each such pair of transitions, a new transition is added to the new NFA. The computational complexity of the algorithm can be expressed as

$$T_m \propto N_s S^2$$

where T_m represents the time required to perform multistriding, N_s is the total number of states in the NFA while S is the number of symbols of the original NFA. When applying this algorithm iteratively, the set of symbols may square at each iteration, posing serious questions on the scalability of this technique with respect to S.

C. Alphabet compression

To reduce the problem of the exponential increase of the alphabet size with increasing multistride levels, [1] proposes a technique called "alphabet compression". This technique replaces the squared alphabet with a new, simpler one, in which the symbols that always appear together in the same transitions are "grouped" into an unique, equivalent symbol.

Groups are generated by sequentially analyzing all the transitions of an NFA, and must be numbered with unambiguous indexes. Then, a dictionary must be written to specify how to convert the old, 256^2-wide alphabet to the new one, which includes the minimum number of distinct symbols required to handle the NFA. Finally, the NFA transitions must be rewritten by using the new alphabet.

The new NFA will undoubtedly have better characteristics than the original one: the overall number of symbols will be reduced and, as a side effect, every transition will be labeled with a smaller number of symbols (e.g., the equivalent symbol A instead of the set of symbols $\{aa, ab, ac\}$), with consequent memory advantages. However, this modification requires the introduction of a new pre-processing step for the packet processor, which will need to translate the input data to the new alphabet before being fed to the NFA.

The most critical part in terms of performance, for what concerns alphabet compression, is the generation of the dictionary that maps symbols onto equivalence classes. Every other operation, defined in the rest of the paper as generic "post-processing" code, can be performed so quickly that its impact in terms of performance can be considered negligible.

The most efficient lossless alphabet compression algorithm published in literature is described in [1]. Initially, the entire $(S \times S)$ alphabet is grouped into a single class. Then the algorithm iterates through every transition and, at each iteration, it manipulates the equivalence classes so that the symbol set in the transition label corresponds exactly to a set of classes. This operation is performed just by splitting equivalence classes into smaller ones: this ensures that all the classes have disjoint sets of symbols and that, after each split, the equivalence relation between the previous transition labels and sets of classes remains valid.

In order to efficiently perform this operation, this algorithm exploits an helper array whose size is equal to the alphabet size. By using this technique it is possible to perform alphabet compression with the following computational complexity:

$$T_c \propto S' N'_s \overline{N}'_{fo}$$

where T_c is the time required for alphabet compression, S' and N'_s are the number of symbols and states of the input (already multistrided) NFA, while $\overline{N}'_{fo}$ is the average number of transitions exiting from each state (i.e., the average fan-out of the NFA). As alphabet compression is performed after a multistriding iteration, S' is usually equal to S^2, N'_s normally coincides with N_s while $\overline{N}'_{fo}$ is slightly larger than $\overline{N}_{fo}$.

D. State minimization

The NFA state minimization problem, i.e. the search for an equivalent NFA with the minimum number of states, can be solved by the "subset construction" technique [7] which is based on applying a set of equivalence rules. A first rule states that *two states sharing exactly the same set of outgoing transitions can be considered equivalent*, where transitions are considered equal if they fire upon the same symbol and land in the same state. A second rule is the dual of this one: *two states that have exactly the same set of ingoing transitions can be merged together*; in this case two transitions are considered equal if they are associated to the same symbol and originate from the same state.

Applying these rules for reducing the size of the NFA means finding a pair of equivalent states and then joining them into a single state that groups together all the ingoing and outgoing transition sets of the two original states.

The application of these rules alone is not enough to transform the automaton into its minimal form. Further rules are needed for this purpose. However the two rules explained here are enough for the objectives of this paper.

III. Related work

State of the art algorithms for multistriding and associated alphabet compression are proposed in [2] and [1]: these two algorithms (discussed in Section II) have scalability problems. For what concerns multistriding, the main problem is due to the S^2 component in the computational complexity formula. For what concerns instead alphabet compression, memory occupancy explodes mainly because of the necessity of a huge support array.

An alternative set of algorithms was developed in [3]. The main difference against the ones presented in [2], [1] is the lack of the helper array, which reduces memory requirements to a minimum. Even if [3] presents results for up to 16x multistrided NFAs for large real-world rulesets, unfortunately a bug has later been discovered in the implementation of these algorithms. After fixing this bug, the running time of the program became so large that it was practically impossible to go beyond the 2x multistride NFA for the Snort and L7 rulesets presented in [3].

An alternative method to perform alphabet compression is presented in [5]; however, the computational complexity of the described technique is worse than the already presented algorithms, hence it has not been taken into consideration in this paper. Another approach can be found in [6], which proposes a slightly different version of multistride called variable-stride. This represents one among the many papers that exploit particular features available only in special-purpose devices such as FPGAs: for example, the usage of *do-not-care* bits in look-up tables and similar features cannot be implemented for a general purpose device without having huge performance losses, and for this reason these have not been considered in this paper.

For what concerns automata minimization, many different papers can be found in literature: for example, [7], [4], [2] apply this technique to automata for packet inspection. However, all these papers just consider performing minimization on the "original" automaton, while the effects of minimization applied after each multistride step has not yet been studied. This has instead become one of the aims of our work, i.e. to find out a new algorithm for multistriding that, by combining multistriding, alphabet compression, and minimization in a proper way, reaches the performance needed for computing multistride NFAs for large rule sets in reasonable time.

IV. The new algorithm

A. Multistriding

As the main factor that limits the performance of the multistriding algorithm is the S^2 component in the time complexity, our algorithm iterates through all the states of the NFA and it follows all the reachable pairs of *transitions* in order to combine them together in a single, compound transition. This is a great improvement as iterating on every possible combination of symbols results in a much slower algorithm, as transitions are usually one order of magnitude less.

Unfortunately, transitions may fire upon ranges of symbols (e.g., the pair of symbols $[a*]$), which requires us to perform the cartesian product of the corresponding symbol sets in order to determine the exact symbols used by that transition. This means replacing the S^2 component of the asymptotic computational complexity formula with $\overline{N}_{fo}^2 S^2$, that may be a problem in the worst case but it is usually much more efficient in the normal case. Additionally, iterating on transitions offers the possibility to replace symbols with *ranges* (e.g., $\{aa-az\}$ instead of $\{aa, ab, ..., az\}$), which enables the usage of a compressed memory representation for contiguous series of symbols and speeds up operations. This exploits a common characteristics of regular expression patterns that tend to express ranges very frequently, hence providing huge advantages in real operating conditions. Moreover, combining together symbol sets is simple, as it consists of writing every possible combination of "compressed" ranges.

When this feature is introduced, the time complexity of the algorithm for multistriding becomes:

$$T_m \propto N_s \overline{N}_{fo}^2 \overline{R}^2$$

where $\overline{R}$ is the average number of "ranges" per transition. In the worst case, this value equals half of the alphabet size (this occurs, for example, if every transition is labeled with all the odd symbols of the alphabet). Usually, however, $\overline{R}$ is far lower: in our experience even complex rule sets have values for $\overline{R}^2$ that hardly exceed few hundreds, while at the same time S^2 reaches peaks in the order of hundred of thousands for the same 2x multistride NFA. This phenomenon is not just due to statistical reasons (even if, in any case, the probability to have just odd symbols in a transition is very low), but it is also caused by a particular feature of the alphabet compression algorithm, which pays particular attention to the symbol numbering phase in order to maximize the number of contiguous

symbols in a transition (more details in Section IV-B). In fact, from our observations, $\overline{N}_{fo}^2\overline{R}^2$ grows at a slower rate than S^2 in all the analyzed rule sets. For this reason, although the result based on asymptotic complexity looks cumbersome, we expect that our algorithm outperforms the existing state-of-the-art. At the same time, memory occupancy should not increase significantly.

B. Alphabet compression

State of the art alphabet compression algorithms [2] are very efficient in terms of computing time. In fact, they are most memory intensive, as they exploit a big support array that stores, for each symbol of the alphabet, its equivalence class. The algorithm needs only to scan every transition of the automaton and to update this array according to a set of rules that guarantee the correctness of the equivalence classes. This technique is very efficient but, unfortunately, at high levels of multistriding this huge support array may reach prohibitive sizes. Moreover, the algorithm proposed in [2] requires the storage of four different values for each symbol, thus further increasing the memory occupancy.

As the computational complexity is not the main limitation here, our alphabet compression algorithm focuses on memory consumption and it requires to store a single value per each symbol, namely the equivalence class number. This is possible as the new algorithm creates an unchecked number of equivalence classes, i.e. it does not perform any consistency check on the support array. Even with this simplification, the generated equivalence classes are exactly the same as the ones generated by the original algorithm. The only downside is that a "class renumbering" step is now required, as it is necessary to remove all the generated empty classes.

However, the class renumbering procedure can be easily implemented through very limited changes in the post-processing code, without impacting on the algorithm complexity: for this reason, it is possible to state that this additional procedure does not have any impact to the overall performance. Moreover, this operation is also very useful because a clever class renumbering can also help the multistriding algorithm to run even faster, as by maximizing the amount of contiguous class indexes for each transition it is possible to keep the value of $\overline{R}$ low.

The new algorithm, shown in Figure 2, consists in a main iteration over all transitions (line 3). Then, for every element of the symbol set (line 5), it updates the corresponding area in the support vector (line 10) by assigning it to a new class. The only performed check (lines 6 to 9) is needed to ensure that all the elements of a symbol set initially belonging to the same equivalence class are mapped to the same new equivalence class.

In addition to the reduced memory usage, this algorithm avoids to fully scan the support array at each iteration, as it only needs to update the cells corresponding to each transition symbol set. This fact greatly helps to improve performance in many different conditions, as transitions labeled with few symbols can be processed several order of degrees more quickly than transitions labeled with the entire alphabet. In order to be as general as possible, these situations have not been taken in consideration during the analysis of the asympthotic complexity of the algorithm, but in many real cases it is possible to appreciate serious performance improvements thanks to this fact.

```
Require: transitions
 1: map = {0}
 2: classcount = 1
 3: for all t ∈ transitions do
 4:     buffer = {0, 0}
 5:     for all s ∈ symbols(t) do
 6:         if buffer[map[s]] == 0 then
 7:             newclass = classcount++
 8:             buffer.insert(map[s], newclass)
 9:         end if
10:         map[s] = buffer[map[s]]
11:     end for
12: end for
```

Fig. 2. Alphabet compression algorithm.

The overall complexity of the algorithm is given by

$$T_c \propto S' N'_s N'_{fo}$$

It is possible to note how the post-processing overhead is so low that it is not even visible in the computational complexity formula.

C. Automaton compression

Due to the complexity of the minimization algorithm, previous works used it only on the initial automaton, avoiding any pass on multistride automata. However, as we feel that minimization could be extremely effective also on multistride automata, our idea was to perform a lightweight minimization, but at each multistriding step. More precisely, we propose to perform a state reduction step based on the two equivalence rules presented in Section II-D, which is not too complex, but can significantly reduce the number of states. In fact, we observed that often there are large portions of the NFA that can be further compressed after each multistriding step using only those simple equivalence rules. The reduction algorithm, even if simple, has proven to be particularly efficient in spotting and optimizing equivalent states that are close to the terminal states of the NFA, which occur frequently.

The new algorithm basically performs three operations. *(i)* it merges the terminal states in order to guarantee that after each optimization the automaton has a single terminal state for each regular expression; this is necessary as sometimes the multistride algorithm tends to duplicate or merge together accepting states. *(ii)* once the terminal states are fixed, the automaton is scanned from the initial states to the accepting ones in order to find states with an equivalent set of incoming transitions (second minimization rule). Finally *(iii)* the same routine is performed backward, from the accepting states to the initial ones in order to compare outgoing transitions. If some states have been merged, a new iteration of the steps

(ii) and *(iii)* is performed, in order to discover if the previous merge created some new possible equivalent states.

The algorithm performs its task very quickly, as it just needs to navigate the NFA. The time complexity is

$$T_{am} \propto N_s \overline{N}_{fo}^4 K$$

where K represents the number of needed iterations. This is usually a very low value, as in our datasets it is between two to ten. The $\overline{N}_{fo}^4$ factor can be considered not problematic as it is usually a small value, even for very complex datasets.

V. Performance comparisons

In order to analyze the capabilities of the proposed algorithm combination several tests have been performed. We used a blade server with an Intel i7-960 processor (quad core + HT) and 12GB of RAM (triple channel). We used only one core of the processor as all the tools are currently single-threaded.

In order to check also the correctness of the results, a "validator" has been developed: this tool generates a set of packets shaped in order to trigger a precise pattern of regular expressions, and then it compares the expected pattern to the results obtained by the multistride NFA.

Our results have been compared to the algorithm presented in [3], after fixing the bug mentioned in Section III. As said, this implementation looks very similar to the algorithm described in [2], [1], whose implementation instead is not publicly available; the most notable difference consists in the absence of the support array.

A. The comparison data

Four different regular expression sets have been used for the experiments.

The first ruleset has been chosen in order to enable comparisons with the results reported in previous papers, as it was already used in [3], [1]. This ruleset includes 534 regular expressions extracted from the rulesets of Snort, a well-known commercial IDS. This is referred to as Snort534.

The second ruleset was selected to have a simple test case and it includes only the first 50 regular expressions of Snort534; for this reason it has been named "Snort50".

The third ruleset is a superset of Snort534, built by selecting all the rules having a syntax compatible to the NFA generator[1], hence resulting in 1514 regular expressions.

The last ruleset completely differs from the previous ones as it is composed of all the protocol signatures extracted from the L7 traffic classifier[2]. This particolar ruleset is composed of just 115 regular expressions, but each one is very complex; in terms of overall complexity, it can be considered an intermediate point between the Snort534 and Snort1514 rulesets.

In order to properly perform the conversion between regular expression sets and NFA, a slightly modified version of the NFA generator software provided by Michela Becchi [2], available online, has been used. An important property of this tool is the capability to generate automata that have the minimum number of states as possible, which represents an excellent starting point for our experiments.

[1]Some rules of Snort use non standard syntax features that the NFA converter cannot handle.

[2]Available at http://l7-filter.sf.net/.

	Snort50	Snort534	Snort1514	L7	Stride
# of states (N_s)	1,023	9,538	47,168	3,402	1x
# of transitions (N_t)	1,072	10,401	52,515	11,342	1x
avg fanout ($\overline{N}_{fo}$)	1.1	1.16	1.16	3.55	1x
# of symbols (S)	255	255	255	255	1x
# of states (N_s)	1,023	9,449	38,918	2,939	2x
# of transitions (N_t)	1,227	12,088	60,348	37,936	2x
avg fanout ($\overline{N}_{fo}$)	1.26	1.35	1.61	13.37	2x
# of symbols (S)	1,029	4,784	14,517	13,582	2x
# of states (N_s)	1,023	9,349			4x
# of transitions (N_t)	1,729	17,252			4x
avg fanout ($\overline{N}_{fo}$)	1.77	1.93			4x
# of symbols (S)	8,594	398,159			4x
# of states (N_s)	1,023				8x
# of transitions (N_t)	3,575				8x
avg fanout ($\overline{N}_{fo}$)	3.62				8x
# of symbols (S)	91,546				8x

TABLE I
Quantitative analysis of the automata at different multistride levels.

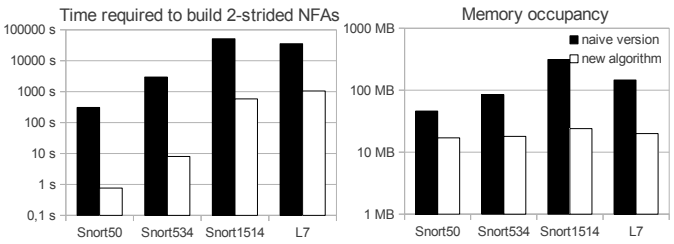

Fig. 3. Performance comparison between the old toolset used in [3] and the new algorithms in term of timing and memory occupation required to compute the 2-strided version of four different rulesets (this comprises all the three operations: multistride, alphabet compression and NFA minimization)

B. Results

The performance of the new algorithms, used all together one after the other, compared to the one of the algorithm used in [3], has been tested by applying successive multistriding steps on the selected rulesets, thus first computing the 2x multistride NFA, then the 4x and so on. The experiment was stopped, for each ruleset, when the time needed for one multistriding step exceeded 24 hours. This value has been chosen as a reasonable upper bound for considering optimization times feasible.

The results of this test on the new algorithm chain are reported in Table I, where the main characteristics of each multistride NFA are presented. Where no results are given, the 24-hours limit has been exceeded. As it is possible to observe, the 4x multistride level was reached for the Snort534 ruleset, while for the smaller Snort50 ruleset even the 8x level was reached. For instance, the algorithm used in [3] was able to reach 2x multistride for all the considered rulesets, but it exceeded the 24-hours limits for all the datasets when trying to compute the 4x level.

Unfortunately, even with the new algorithm the 4x multistride level could not be reached for the two biggest automata.

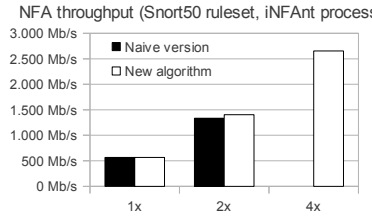

Fig. 4. Benchmark test performed to measure processing throughput changes by using the new algorithms with respect to the vanilla implementations. Tests have been performed by using the iNFAnt packet processor at different multistride levels.

However, overcoming the 2x limit for medium-sized automata like Snort50 and Snort534 can still be considered a great result, considering that the results reported in [2] demonstrated the capability of 4x multistride only for a simple ruleset made up of 20 regular expressions.

In our experiments we measured and compared to [3] also the time and memory occupancy required to perform a multistride optimization. Figure 3 shows these results for the 2x multistride computation step, for which both implementations terminate within the 24-hours limit. It is possible to notice how the new algorithm chain outperforms the previous one, especially in terms of computation time.

Since the new algorithm chain produces multistride NFAs that in general are different from the ones produced by the previous algorithms, one may ask how these NFAs behave compared to the ones generated by the previous algorithms in terms of achieved throughput. Indeed, the expectation is that the runtime throughput is at least not adversely affected by using the new algorithm chain. In order to test this aspect, some comparisons have been performed by using the GPU-accelerated regex processor presented in [3]. Measures show that, as expected, the processing throughput nearly doubles after each optimization step. Moreover, NFAs obtained by using the new tool chain are never worse and up to 5% faster than the others, as it can be seen in Figure 4. For this reason, it is possible to state that the new algorithms produce multistride NFAs in a more efficient way, without introducing runtime performance penalties but even giving a small throughput enhancement.

VI. Conclusion

This paper has proposed an improved algorithm chain for computing multistrided NFAs, where new, more memory-aware, versions of the algorithms originally developed in [2] have been introduced, and a lightweight state minimization step is applied after each multistriding step.

Using previous algorithms, only the NFAs of very small rulesets could be handled. As an example, the best results presented in literature ([2]) report the computation of 4x multistride NFAs only for rulesets composed of up to 20 regular expressions extracted from the Snort ruleset, and the computation of 2x multistride NFAs only for rulesets composed of 90 such expressions. Slightly larger NFAs could be handled by a naive variation of the algorithms presented in [2] but at the cost of computation times of hours.

Using the proposed algorithm chain, 4x multistride NFAs can now be computed on quite larger rulesets, composed of more than 500 complex regular expressions taken from the Snort ruleset, in reasonable time and with affordable memory consumption. The computation time for 2x multistride NFAs has been greatly reduced compared to the time taken by the above mentioned naive variation of the algorithm presented in [2]. In this way, higher multistride levels are becoming feasible for more complex rulesets at reasonable time and memory costs. Even for rulesets close to the ones used in commercial network applications, at least 2x multistriding is now practical.

The factors that still limit further optimization steps mainly regard the exponential growth of the number of possible symbols. Alphabet compression is a very good technique to counter this problem, but it is not enough to completely eliminate it. Performing a slight form of NFA minimization after every multistriding iteration proved to further reduce this problem but not to the extent of going beyond the 2x multistride level for the most complex rulesets. As a future work, it could be interesting to deepen the studies in this field in order to understand which are the best results achievable by using other minimization algorithms or other algorithm mixings.

References

[1] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 2007 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, Orlando, FL, December 2007. ACM.

[2] M. Becchi and P. Crowley. Efficient regular expression evaluation: theory to practice. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59, New York, NY, USA, 2008. ACM.

[3] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40:20–26, 2010.

[4] T. Kameda and P. Weiner. On the state minimization of nondeterministic finite automata. *Computers, IEEE Transactions on*, C-19(7):617 – 627, july 1970.

[5] X. Liu, X. Liu, and N. Sun. Fast and compact regular expression matching using character substitution (to appear). In *Proceedings of ACM/IEE Symposium on Architectures for Networking and Communications Systems*, Brooklyn, NY, USA, Oct. 2011.

[6] 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 *USENIX Security'10 Proceedings of the 19th USENIX conference on Security*, Berkeley, CA, USA, 2010.

[7] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114 –125, april 1959.