Abstract—Performing Deep Packet Inspection at high speed is a fundamental task for network security and application-specific services. In state-of-the-art systems, sets of signatures to be searched are described by regular expressions, and finite automata (FAs) are employed for the search. In particular, deterministic FAs (DFAs) need a large amount of memory to represent current sets, therefore the target of many recent works has been the reduction of memory footprint of DFAs.

This paper, instead, focuses on speed multiplication by enlarging the amount of bytes observed in the text (i.e., searching for \(k\)-bytes per state-traversal). For this purpose, an interesting yet simple inverse homomorphism is employed to reduce the amount of transitions in the modified DFA. The algorithm results to be remarkably faster than standard DFAs, and provides also a good compression scheme that is orthogonal to other schemes.

I. INTRODUCTION

Many important services in current networks are based on payload inspection, in search of a predefined set of patterns which characterize specific classes of applications, viruses or protocol definitions. Nowadays, regular expressions are used to represent such signature sets, due to their increased expressiveness and powerfulness [1]. They are adopted by well known tools, such as Snort and Bro, and in firewalls and devices by different vendors such as Cisco.

Typically, FAs are employed to implement regular expression matching. Nondeterministic FAs (NFAs) are very space-efficient representations which require more state transitions per character, thus having a time complexity for lookup of \(O(m)\), where \(m\) is the number of states. Instead, DFAs require only one state traversal per character, but for the current expression sets they need a large amount of memory. For these reasons, many recent works have presented solutions to reduce DFAs, exploiting the redundancy of regular expressions.

This paper proposes a solution to increase the speed of regular expression searching techniques by multiplying the amount of bytes processed per cycle. Indeed, very few works have explored this possibility, since that, when processing \(k\) bytes per step, \(256^k\) transitions per state are needed, so even observing only 2 bytes per cycle would require 65536 transitions. Actually, the amount of states reachable in one-hop from a given state is limited and concentrated on its average. Such fact can be exploited in order to define a simple and effective way to build small-sized and fast DFAs that process \(k\) bytes per step. This involves the definition and application of a homomorphism [2], hence we name our DFA representation Homomorphic-DFA (h-DFA).

The remainder of the paper is organized as follows. In section II related works are discussed, while sec.III introduces the main idea of our algorithm elaborating on the definition of homomorphisms. Then, sec.IV describes the look for a good inverse homomorphism and in sec.V the scheme is optimized by means of permutations, char-stuffing and the enlargement of homomorphisms space. Finally, sec.VII presents the experimental results on real world regular expression sets, while sec.VIII concludes the paper.

II. RELATED WORKS

The current trend in research and industry is to use DFAs to represent regular expressions, in order to obtain higher performance, while trying to solve their problems in terms of memory requirements. The Delayed Input DFA (D\(^2\)FA) [3], for instance, is based on the observation that many states in DFAs have similar sets of outgoing transitions, which can be replaced with a single default transition. The drawback is the traversal of multiple states when processing a single input character, which entails a memory bandwidth increase. To limit such bandwidth requirements and to simplify the construction of D\(^2\)FA, in [4] an advanced algorithm (we will call it BEC-CRO) is introduced, which needs at most \(2N\) state traversals for processing a string of length \(N\). It exploits the remark that all regular expression evaluations begin at a single starting state, and the vast majority of transitions among states lead back either to the starting state or to its near neighbors.

In [5], the authors show that the memory compression of D\(^2\)FAs can be obtained also with a single memory access per character by assuming large memory accesses. The idea behind their \(\delta\)FAs exploits redundancy in standard DFAs, where many transitions for a given character are directed to a single state. They propose also to describe state numbers with a smaller number of bits by using the so-called Char-State compression.

The work in [6] tries to combine the advantages of DFAs (in terms of processing speed) with those of NFAs (in terms of low memory requirements, also for large sets of regular expressions). Therefore the authors propose a hybrid DFA-NFA solution: when constructing the automaton, any nodes that would contribute to state explosion retain an NFA encoding, while the others are transformed into DFA nodes.

Kumar et al. showed in [7] how to increase the speed of D\(^2\)FAs by storing more information on the edges. This appears a general trend: in [7] transitions carry data on the
next reachable nodes, in [8] edges have different labels, and even in [9] and [10] transitions are a sort of “instructions”. In [9], some limits and potential improvements of standard DFAs are illustrated. First, normal data streams often match just few initial symbols of strings; thus, it can be convenient to split signatures, such that only the beginning needs to “be active”, while the remaining portions can be “put to sleep” (in an external memory). Then, the DFAs are not able to efficiently follow multiple partially matching signatures, this way yielding the so-called state blow-up. Therefore the authors propose the History-based Finite Automaton (H-FA) which remembers more information, such as encountering a closure, by storing them in a fast cache which represents a sort of history buffer. Finally, they address the issue of counting constraints by extending H-FA with counters, thus obtaining the History-based counting Finite Automata (H-cFA).

The same idea of using additional information to remember the transition history and reduce the number of required states is developed in another scheme, named extended finite automata (XFA) [10]. In brief, XFA uses a finite scratch memory to remember various types of information relevant to the progress of signature matching (e.g., counters of characters and instructions attached to edges and states).

This paper focuses on the second main issue for current regular expression searching techniques: speed. Very few works have explored the possibility to increase searching-speed by multiplying the amount of bytes processed per cycle. To the best of our knowledge, one single paper [11] proposed a solution to process \( k \) bytes per cycle. The main idea is based on the observation that, while for \( k \) bytes per state traversal potentially \( 256^k \) transitions must be stored, in actual DFAs the number of different transitions (even when \( k \) bytes are processed) is more limited. The scheme in [11] thus proposes the use of ECIs (Equivalent Character Identifiers), that define the set of input words (composed by \( k \) bytes) that produce transitions to the same next state. However, such observation is not enough to make DFAs feasible in real memories, then the authors use Run Length Encoding to encode the Transition Table. Finally the scheme is tested on a FPGA and compared to another scheme. Our approach is also more general, since we do not focus on a hardware, nor software implementation, but we propose a scheme that can be easily deployed in any form, as it is based on bitmask-like inverse homomorphisms.

**III. AN EFFICIENT REPRESENTATION FOR DFAS**

In the following we introduce the basics of our scheme. We want to succinctly describe the outgoing transitions for each state, so that, when computing the corresponding \( k \)-step DFA, we have to combine a small amount of one-step transitions.

Our main idea is to group all the symbols that produce a transition to a given node into a subset and find a series of functions that, only when applied to such a subset, provides a specific result or a set of results. When applied to all the other symbols, the result must be different. More formally, in each state, for each subset of symbols \( S_j \) that produces a transition to a node \( n_j \), we look for a function \( h_j(c) \) such that

\[
h_j(c) = x_j \in \begin{cases} X_j & \forall c \in S_j \\ U \setminus X_j & \forall c \notin S_j \end{cases}
\]

where \( U \) is the image of \( h_j(c) \) and \( X_j \) is the subset of the image of \( h_j(c) \) for \( c \in S_j \). By means of this series of functions \( h_j \), we can describe the transition set of each node as an array of tuples:

\[
(h_1 : x_{1,1}, \ldots, x_{1, N_1} : n_1) \ldots (h_d : x_{d,1}, \ldots, x_{d, N_d} : n_d)
\]

where \( d \) is the state outdegree, \( n_1 \ldots n_d \) are the reachable states, \( x_{1,1} \ldots x_{k, N_k} \) are the different values that \( h_k \) takes in \( S_k \) or, in other words, a representation of \( X_k \) and \( N_k \) is the cardinality of \( X_k \). Such a representation helps reducing the redundancy of DFAs as regular Alphabet Compression Tables: it requires to store \( \sum N_k \) values, \( d \) functions and \( d \) pointers to next states. As an example of the compactness of such a representation, let us observe the transition set of the DFA state 1 in fig.1. In this case, the characters \( a,c,d \) and \( e \) all belong to a subset \( S_0 \) that produces a transition to state 0. Therefore we can describe the transition set with two tuples only:

\[
\{h_0; X_0; 0\}, \{h_1; X_1; 1\}
\]

where \( h_0 \) and \( h_1 \) are defined as in (1): when \( h_0 \) is applied to \( a,c,d \) and \( e \), the result is in \( X_0 \), while \( h_1(b) \in X_1 \).

By defining a set of functions to a DFA, we exploit the properties of inverse homomorphisms applied to DFAs. An homomorphism [2] is an application that maps symbols to strings belonging to a language \( L \). An inverse homomorphisms translate strings of a language \( L \) into symbols belonging to a given alphabet. From our point of view, by grouping all our functions \( h_j \) into a function \( H^{-1} \) such that

\[
H^{-1}(c) = h_j(c) \quad \forall c \in X_j
\]

define an inverse homomorphism (as emphasized by the exponent). On the other hand, by means of the representation in tuples we apply a homomorphism \( H \) to a DFA. The composition of the two is, of course, the original DFA.

**IV. THE LOOK FOR AN EFFECTIVE HOMOMORPHISM**

In order to find a description for \( H^{-1}(c) \), we test the following possible “bit-friendly” definition for \( h_j(x) \):

1. \( h_j(x) = (p_j \times x + q_j) \mod m_j \)
2. \( h_j(x) = (p_j \text{ AND } x) \mod m_j \)
3. \( h_j(x) = \text{popcount}(x \mod m_j) \)

These possible definitions are applied (with parameters \( p_j,q_j, m_j \) varying from 1 to 256 because \( x \) itself is a byte) to DFAs that recognize real data-sets (the ones shown in sec.VII).
In each test, we start by looking for a function that provides a single result in a given subdomain $S_j$; if none is found, we look for 2 results and so on. Once we find such a function, we follow the “definition” of $h_j(x)$: we check if it outputs the same value inside and outside a subset $S_j$, that is we check for the following condition:

$$\{x_i = h_j(c_i) : \forall c_i \notin S_j\} \cap \{x_j = h_j(c_j) : \forall c_j \in S_j\} = \emptyset$$

(3)

If the condition is not verified (the intersection is not empty), we drop the function and change the parameter again, as described by the pseudocode in algorithm 1. The algorithm can either finish the computation because it finds a good function (i.e.: the return value is FOUND) or fail. The failure happens if no combination of the parameters $\{p_j, q_j, m_j\}$ produces a function $h_j$ whose image set $X_j$ has less than $C_{\text{max}}$ elements.

```
Algorithm 1 Pseudocode for the search of function $h_j(x)$
1: for $\{p_j, q_j, m_j\} \rightarrow \{0, 0, 0\}, \{255, 255, 255\}$ do
2:     for $a \leftarrow 1, C_{\text{max}}$ do
3:         for all $c_j \in S_j$ do
4:             Compute the set $X_j = \{x_j = h_j(c_j)\}$.
5:         end for
6:         if $\text{Card}(X_j) > a$ then
7:             Try with a larger Cardinality $a$, goto 2
8:         end if
9:     end for
10:    if $h_j(c_i) \notin X_j$ then
11:        Try another function, goto 2
12:    end if
13: end for
14: return FOUND with parameters $\{p_j, q_j, m_j\}$.
15: end for
16: return FAIL.
```

The results show that, for practical values of the parameter $C_{\text{max}}$ (i.e.: $C_{\text{max}} \leq 64$), only $h_j(x) = (p_j \text{ AND } x) \mod m_j$ does not cause the algorithm to fail. Moreover, it turns out that $m_j = 255$ in all the tests. Therefore we can define $h_j(x)$ as a simple AND operation with a bitmask: $h_j(x) = x \text{ AND } p_j$

Such an outcome has a number of advantages: the number of parameters is limited to 1 (i.e.: small memory footprint), the operation is one of the most basic logic operation (i.e.: it costs a simple logic gate in a hardware implementation and it is very fast and parallelizable if our aim is a software engine) and the definition of $h_j(x)$ is amenable to be described by means of a tree, which means we can redefine each state transition set in a Longest-Prefix-Matching (LPM) description. Finally such a description is always achievable: even in the worst case (all characters produce a different transition and have a different tuple) the correctness of the scheme is not affected. In the following sections, we walk through the properties of such a representation and provide optimizations for our scheme. However, the main advantage is the possibility to concatenate two or more $h_j(x)$, such that we easily obtain a $k$-step DFA.

As an example, if $\{h_0 : X_0 : n_1\}$ describes a transition from state $n_0$ to $n_1$ and $\{h_1 : X_1 : n_2\}$ is a transition from $n_1$ to $n_2$, then it straightforward to verify that we the transition from $n_0$ to $n_2$ of the corresponding 2-step DFA is as simple as $\{h_0||h_1 : X_0||X_1 : n_2\}$, where $h_0||h_1$ indicates the concatenation of $h_0$ and $h_1$ and $X_1||X_2$ is defined as:

$$X_1||X_2 = \{a_1||a_2 : \forall a_1 \in X_1, \forall a_2 \in X_2\}$$

(4)

Therefore all we have to take care of is the cardinality of the image sets $X_j$, that determines the memory requirements.

V. OPTIMIZATIONS

In the following we describe the advantages and the properties of the bitmask definition for $h_j(x)$ and elaborate upon the problem of minimizing the cardinalities of the image sets.

A. Permutation for LPM

A first observation on subsets $S_j$ is that they may not be contiguous, i.e.: they may be the union of two or more non-contiguous subsets of symbols. Of course this is detrimental to our target to minimize the cardinality of $X_j$. We solve this problem by introducing a permutation of symbols: we define a translation table (it is a small 256-bytes table that does not increase the cost in terms of external memory accesses) that moves symbols in order to make all subsets as contiguous as possible. Finding the optimal translation table is a complex issue since subsets may vary from state to state. The good news is that in practical DFAs the number of subsets is very limited. The bad news is that, as subsets vary from state to state, it may happen that a certain symbol occurs in different subsets. Therefore finding the optimal translation table is an NP-complete problem as it is equivalent to the weighted maximum set packing problem: we want to find a set-packing (a collection of disjoint subsets) that maximize the total weight of its subsets. Such weight ($w$) must take into account the memory impact of the subsets (a large and frequent subset has high utility because, once we put it in the translation table, it is likely to be described by a single bitmask). Therefore $w$ is defined as the product of the cardinalities of subsets and the number of times those subsets appear in the whole DFA.

We attack the problem with the Co-occurrence Permutation algorithm, which is based on the co-occurrence of symbols in subsets. First, it computes the character co-occurrence matrix $A^{(0)}$, where an element $a_{i,j}^{(0)}$ represent the number of times characters $i$ and $j$ appears in the same subset multiplied by the cardinalities of the subsets they appear into (such that we replicate our weight metric). Then, the algorithm aggregates all 256 characters in 128 pairs, by grouping characters that present the largest co-occurrence, as depicted in the example of fig.2. A new co-occurrence matrix $A^{(1)}$ is computed for all the 128 pairs. Again, pairs are aggregated thus forming 4-characters groups, and so on. Therefore the algorithm recursively aggregates characters in a tree and the last matrix $A^{(8)}$ actually collapse into a scalar. Of course we have to define the co-occurrence of groups: given two symbols pairs $i,j$ and $l,m$, we can define the pairs co-occurrence in many ways. In our tests, we adopted $\min(a_{i,m}, a_{i,l}, a_{j,l}, a_{j,m})$ as the co-occurrence of pairs $i,j$ and $l,m$ as it proved best results on all datasets. Finally we put the symbols in the leaves of the
tree into a table and the translation table is simply the inverse permutation of such a table.

Now, by means of Co-occurrence Permutation, subsets $S_j$ can be described with single bitmasks. Then we can use a LPM description of transition-set, thus taking efficient ideas from the widely studied field of IP lookup.

### B. Bitmap trees

As described above, thanks to the permutation, we can define each state by means of LPM structures, such as trees. The adoption of trees is twofold useful: it reduces the memory footprint of the bitmask description and it provides us with another faster way to compute the bitmask parameter in the definition. As for the latter issue, it is straightforward to see that computing $p_j$ and $X_j$ (the result of $h_j$ on subset $S_j$) can be now simply demanded to the creation of a tree of all the 256 possible values of the symbol (where last level leaves point to next state) and its subsequent pruning.

Therefore, to store our tuples representation (2), we can simply use a bitmap tree. However, this does not preclude permutation; on the contrary, it takes advantages of the use of a permutation algorithm, because if the characters of same subsets are close to each other, they most likely produce short branches in the tree. In order to construct a bitmap tree representation of a state, for each character $c$ we get the next state $s_{next}$ and add $c$ in a tree, such that the leaf points to $s_{next}$ as shown in fig.3 (where next states are 1, 2 and 3). Once we observed all the symbols, we prune the tree: if both children of a node $x$ point to the same next state, $x$ inherits children’s pointer and children are removed. Finally, we can also remove from the tree the subset described with the largest number of leaves, as it can be stored as a “default transition” to be taken when no match is obtained. In the example of fig.3, we remove the leaves pointing to state 1 as they are the most frequent.

Therefore, in order to resume the overall algorithm, the first step is the computation of subsets and functions $h_j(x)$ that can define an inverse homomorphism. Then we add a translation table by adopting Co-occurrence Permutation, and finally we simply compute an LPM description of each state of our DFA.

### VI. THE K-STEP DFA

As earlier described, the homomorphic (or LPM) description allows for a simple yet memory-efficient computation of $k$-step DFAs. The algorithm for the creation of a $k$-step h-DFA is shown in alg.2: it is based on a recursive procedure `compute_1-step` that takes a $k$-step h-DFA $D'$ and a 1-step h-DFA $D$ and computes the $(k+1)$-step h-DFA $D''$. As shown in the pseudocode, we add transitions defined by the concatenation of functions $h_1$$h_2$ as defined in IV. Such a concatenation may as well be seen as a concatenation of trees.

![Figure 2. An example of Co-occurrence Permutation for 3-bit characters.](image)

![Figure 3. An example of state construction in h-DFA for 3-bit characters. The numbers on the leaves are pointers to next states.](image)

### Algorithm 2 Pseudocode for the creation of a $k$-step DFA

```plaintext
procedure compute_1-step(D, D')
1: for all state $s$ in DFA $D$ do
2:   for all next state $s_1$ of $s$ do
3:     for all next state $s_2$ of $s_1$ in $D'$ do
4:       Add_transition($D''$, $s$, $s_2$, $h_1$$h_2$);
5:   end for
6: end for
7: return $D''$

procedure compute_k-step(D)
1: for $i ← 1, k$ do
2:   $D' ←$ compute_1-step($D$, $D'$)
3: end for
4: return $D'$
```

### VII. RESULTS

The experimental runs have been performed on data sets of the Snort and Bro IDSs and Cisco security appliances. Such data sets, presenting hundreds of regexes, have been randomly reduced for obtaining fair comparisons to other sets used in literature ([8][3][6][4]). The characteristics of the data sets are summarized in table II, where we list the number of rules, the ascii length range and the percentage of rules including “wildcards symbols” (i.e. *, +, ?). Moreover, the table shows the number of states and transitions for a standard DFA which recognizes such data sets. The regular expressions in the data sets are given as input to the regex-tool [12], that produces the corresponding standard DFAs. Such DFAs are, in turn, used as start-point for our algorithms.

Table VII displays the percentages of transitions and memory reduction for the different 1-step algorithms with respect to data sets representations through a standard DFA. h-DFA achieves a compression degree which is comparable to the other referred algorithms while requiring a single memory access per state. Because of its orthogonality with other schemes, this is a very appealing result. Table VII also shows the mean number of memory accesses per character. $D^{3FA}$
always requires the largest amount of memory accesses, while h-DFA requires always a single access (as δFA).

Instead, tab.III shows the memory reduction obtained by computing 2 and 3-step h-DFA according to what described in sec.VI. In addition, Char-State compression [5] is adopted to reduce the number of bits required to store transitions. The memory reduction is computed with respect to a standard representation of 2 and 3-step DFA (respectively with 256^2 and 256^3 transitions per state). The compression percentages are very high. Certainly, 3-step h-DFAs still require large amounts of memory (in the order of tens or even hundreds of megabytes). However, considering the consumptions of standard 2 and 3-step DFAs, which reach even hundreds of gigabytes, our solution is a big step forward and can easily fit into DRAM. In fact, h-DFA requires at most 10 megabytes to represent 2-step DFA and (in some cases) less than 100MB for a 3-step, thus offering a great speed-up without the unfeasible memory requirements of standard DFAs. Moreover, as our technique is orthogonal to other schemes, we believe that a combination of different compression schemes can reach higher speed-ups requiring a smaller amount of memory.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>D^2FA</th>
<th>BEC-CRO</th>
<th>δFA + C-S</th>
<th>h-DFA + C-S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>trans</td>
<td>mem</td>
<td>acc</td>
<td>trans</td>
</tr>
<tr>
<td>Snort24</td>
<td>98.9</td>
<td>95.9</td>
<td>4.12</td>
<td>89.5</td>
</tr>
<tr>
<td>Cisco30</td>
<td>98.8</td>
<td>97.2</td>
<td>3.18</td>
<td>97.3</td>
</tr>
<tr>
<td>Cisco50</td>
<td>98.7</td>
<td>97.1</td>
<td>5</td>
<td>72.2</td>
</tr>
<tr>
<td>Cisco100</td>
<td>99.1</td>
<td>97.9</td>
<td>5.3</td>
<td>74.6</td>
</tr>
<tr>
<td>Bro217</td>
<td>99.4</td>
<td>98.3</td>
<td>2.35</td>
<td>76.4</td>
</tr>
</tbody>
</table>

Table I COMPRESSION IN TERMS OF TRANSITIONS AND MEMORY AND NUMBER OF ACCESSES.

VIII. CONCLUSIONS

In this paper we presented a technique for a simple and effective definition of k-step DFAs, i.e. DFAs that consume k symbols per state-traversals. The paper exploit the fact that real DFAs have a limited average out-degree. The proposed technique is based on bitmasks that define an inverse homomorphism which can be, in turn, concatenated in order to obtain a k-step DFA. A number of optimizations that allows for the application of LPM techniques are also described. The results show that 2-step and 3-step DFAs can be easily obtained with a limited amount of memory for many real-world regular expressions datasets. An interesting evolution would be to explore the possibility of fractional-steps DFA such as 1.5-step DFA where each state-traversals consume one and a half byte (12 bits).

ACKNOWLEDGEMENTS

This work has been partially sponsored by the European Project FP7-ICT PRISM, contract number 215350. We would like to thank Sailesh Kumar, William Eatherton and John Williams for providing the regular expression set used in Cisco devices.

REFERENCES