Deep Packet Inspection with DFA-trees and Parametrized Language Overapproximation[*]

Daniel Luchaup[†] Lorenzo De Carli[†] Somesh Jha[†] Eric Bach[†]

***Abstract*— IPSs determine whether incoming traffic matches a database of vulnerability signatures defined as regular expressions. DFA representations are popular, but suffer from the state-explosion problem. We introduce a new matching structure: a tree of DFAs where the DFA associated with a node overapproximates those at its children, and the DFAs at the leaves represent the signature set. Matching works top-down, starting at the root of the tree and stopping at the first node whose DFA does not match. In the common case (benign traffic) matching does not reach the leaves. DFA-trees are built using Compact Overapproximate DFAs (CODFAs). A CODFA D' for D overapproximates the language accepted by D, has a smaller number of states than D, and has a low false-match rate. Although built from approximate DFAs, DFA-trees perform exact matching faster than a commonly used method, have a low memory overhead and a guaranteed good worst case performance.**

I. Introduction

Intrusion Prevention System (IPSs) scan network traffic for signs of suspicious activity. Many IPSs, such as Snort [20], rely on *attack signatures*, i.e. patterns strongly associated with malicious traffic. Deployed systems use hundreds or thousands of signatures, usually represented as regular expressions (REs) due to their flexibility. Matching each packet against each signature in such a large set is a significant challenge. A set of REs corresponding to signatures can be combined into into a single Nondeterministic Finite Automaton (NFA), but matching with NFAs is slow because a frontier of states has to be maintained. Deterministic Finite Automata (DFA) match quickly, but they suffer from the state-space explosion problem. When DFAs are combined, in the worst case the number of states can be the product of the number of states of the components, and the DFA that corresponds to all signatures becomes too large to fit in memory. Alternative compact representations for DFAs, such as XFAs [24], come at the cost of increased program complexity and processing time. A simple, practical approach [28] partitions a signature set such that signatures in each partition can be grouped into a single DFA of a bounded size. The processing time is proportional to the number of elements in the resulting DFA-set representation, which is significantly smaller than the size of the signature set, but it can still be large.

In this paper we introduce the *DFA-tree*, a novel data structure for matching REs. We use DFAs organized in a tree structure. The leaves of the tree are the DFAs obtained by grouping a signature set. The DFA at each node acts as a filter for the DFAs in the subtree rooted at that node. Payload scanning is performed top down: start with the root of the tree and, if it doesn't match we stop, otherwise we check whether it is a real match by recursively matching with the children. If a DFA at any of the leaves matches, then it is a real match (the traffic matches a real exploit or a vulnerability). In the common case of benign traffic, the payload does not match, and it is usually rejected before reaching the leaves.

A DFA-tree performs *exact* matching of a signature set, as if using a single DFA that combines all REs in the set. There are no false positives or negatives: *if a DFA-tree rejects a payload, then it doesn't match any RE in the signature set, and if it accepts a payload, then at least one RE matches.*

The main challenge in building a DFA-tree is generating the intermediate filter nodes. To do so, we introduce the concept of Compact Overapproximate DFA (CODFA) as the building block for the DFA-tree construction. An overapproximation for a DFA, $\mathcal{D}$, is another DFA $\mathcal{D}'$ which accepts a superset of $\mathcal{D}$'s language ($L(\mathcal{D}) \subseteq L(\mathcal{D}')$). A CODFA is an overapproximation which is more compact in terms of the number of states, and which has a low approximation error rate. We believe that finding the *best* CODFA, i.e. with the fewest number of states that does not exceed a given error rate, is a hard problem. Therefore we provide an approximation heuristic based on the following idea: a CODFA for $\mathcal{D}$ only keeps the most frequent or "hot" states of $\mathcal{D}$ and the transitions between them, and collapses the remaining states into a single state. We call this construction *shrinking*, because our algorithm collapses all states that are not "hot" into one.

Matching with DFA-trees achieved a factor of $4.7\times$ speedup in parsing HTTP packets using two published sets [11] of respectively 1.4K and 2.6K signatures extracted from the popular IPS Snort and also used by other authors (see [27] and [24]). The speedup is relative to the technique described in [28]. For an approximation error rate of 0.2% shrinking produced CODFAs that, on average, have 97% fewer states. Note that the DFA-tree intermediate nodes can potentially cause a slowdown as in the worst case the input needs to be matched against the entire tree. To quantify this effect we emulated the worst case by forcing the traversal of the entire tree, obtaining a slowdown of 26% (again with respect to [28]).

To summarize,this paper makes the following contributions:

1) We introduce the concept of DFA-trees to speedup the matching of traffic against a set of signatures.
2) We introduce the notion of Compact Overapproximate DFA. We provide a heuristic for constructing CODFAs.

Section II provides background on signature matching in

* Supported in part by NSF grants CNS-{1064944,1228620,1228782}

† U. of Wisconsin-Madison; emails: {luchaup,lorenzo,jha,bach}@cs.wisc.edu

IPSs. Section III introduce DFA-trees and CODFAs. Section IV describes shrinking. Section V presents experimental results. Sections VI and VII survey related work and conclude.

II. Background

Intrusion Prevention Systems (IPS) analyze network traffic searching for *attack signatures*, i.e. patterns strongly associated with malicious traffic. Timely detection of such patterns prevents or alleviates a wide range of network attacks. Regular Expressions (REs) are commonly used to describe attack signatures [26] and databases with thousands of REs are matched against each payload. Because attack traffic is a small percentage of the data seen on the wire, most network traffic is benign and, in the common case, no matches are found.

A. Signatures, FSMs and DFA-sets

Signatures represented as REs can easily be transformed into Finite Automata (FA). Ideally, multiple signatures $sig_1, sig_2, ..., sig_n$ would be merged into a single regular expression $sig_1|sig_2|...|sig_n$ and the resulting FA would match all signatures simultaneously. Nondeterministic Finite State Automata (*NFA*s) scale well space-wise, but at the cost of a slower and more complex matching procedure which explores multiple paths in parallel. Deterministic Finite Automata (*DFA*s) need more space (the minimal DFA for a given RE can be exponentially large [7]), but are popular due to their matching algorithm's simplicity and performance guarantees.

Definition 1: A *DFA* is a 5-tuple $\mathcal{D}$=$(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states; Σ is the alphabet, a finite set of input symbols (in our case a symbol is an 8-bit byte); $\delta : Q \times \Sigma \rightarrow Q$ is the transition function ($\delta(q, c)$ gives $\mathcal{D}$'s next state when its current state is q and the current input symbol is c); $q_0 \in Q$ is the start state; $F \subseteq Q$ is the set of final (or accepting) states.

Notations: The size of $\mathcal{D}$ is its number of states, $|\mathcal{D}| = |Q|$. The language accepted by $\mathcal{D}$ is $L(\mathcal{D})$. If e is a regular expression, $L(e)$ is the language that matches e, and $DFA(e)$ is the minimized *DFA* that accepts $L(e)$. Note that $e, L(e)$ and $DFA(e)$ are equivalent representations for the same language. If $\mathcal{D}_1, \mathcal{D}_2$ are two *DFA*s, then $\mathcal{D}_1 \oplus \mathcal{D}_2$ is the *NFA* for $L(\mathcal{D}_1) \cup L(\mathcal{D}_2)$ obtained by combining $\mathcal{D}_1$ and $\mathcal{D}_2$ as *NFA*s[1], and $\mathcal{D}_1 + \mathcal{D}_2$ is the *DFA* obtained by the determinization of $\mathcal{D}_1 \oplus \mathcal{D}_2$. `dfaMatch`$(w, \mathcal{D})$ performs traditional DFA scanning of word w using $\mathcal{D}$ by executing $q \leftarrow \delta(q, b)$ for every byte $b \in w$.

An IPS needs to know which signatures in the signature set match. If a payload prefix matches, the IPS may resume scanning, looking for other matches; i.e, it searches for all matches. For simplicity, we assume that we are only interested in the first match, and later when the distinction between these two behaviors becomes relevant, we will reconsider this assumption. This means that the REs are considered *suffix closed*, as if they ended in $\cdot^*$.

B. State Explosion and DFA-sets

Combining *DFA*s may result in a state space explosion. For instance, if $e_1 = \cdot^* str_1 \cdot^* str_2 \cdot^*$ and $e_2 = \cdot^* str_3 \cdot^* str_4 \cdot^*$ are regular expressions where str_1, str_2, str_3 and str_4 are strings without any repeating characters or characters in common, then the approximate sizes of the minimized *DFA*s are $|DFA(e_1)| \approx |str_1| + |str_2|$, $|DFA(e_2)| \approx |str_3| + |str_4|$. But $|DFA(e_1|e_2)| \approx |DFA(e_1)| \times |DFA(e_2)|$ [28].

[1]It has a new start state with an ϵ- transitions to $\mathcal{D}_1$ and $\mathcal{D}_2$'s start states

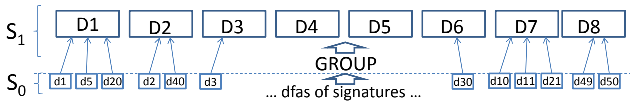

Fig. 1. The *GROUP* method partitions a set of *DFA*s, typically one per signature, into groups such that the *DFA*s in each group can be combined into a single *DFA* that fits in the memory. The result is a smaller set of *DFA*s. Here, *GROUP*$(\{d_1, .., d_{50}\}) = \{\mathcal{D}_1, .., \mathcal{D}_8\}$, and D_1=d_1+d_5+d_{20},..,D_8=d_{49}+d_{50}

When it is not practical to combine a large signature set into a single *DFA*, a common practice is to partition the signatures into multiple groups, and build a *DFA* for each group. The result is a *DFA-set* [28].

Definition 2: A DFA-set is a set of *DFA*s, $\mathcal{S}$=$\{\mathcal{D}_1, ..., \mathcal{D}_m\}$. The language accepted by $\mathcal{S}$ is $L(\mathcal{S})$=$L(\mathcal{D}_1) \cup ... \cup L(\mathcal{D}_m)$.

Algorithm 1: payload scanning using DFA-sets.

```
dfaSetMatch (payload $w$, DFA-set $\mathcal{S}$=$\{\mathcal{D}_1,.., \mathcal{D}_m\}$)
output: Is $w$ in $L(\mathcal{S})$?
  foreach $\mathcal{D} \in \mathcal{S}$ do
    if dfaMatch($w, \mathcal{D}$) /* normal DFA matching */ then
      return True;
  end
  return False;
```

In Algorithm 1, `dfaSetMatch` matches using DFA-sets. Figure 1 shows the transformation named *GROUP*, which partitions and groups signatures. *GROUP* transforms a large DFA-set to a smaller one, such that both sets accept the same language: *GROUP*$(\{d_1, d_2, ..., d_n\}) = \{\mathcal{D}_1, ..., \mathcal{D}_m\}$ where each $\mathcal{D}_i$ is the composition of some d_k's, $L(d_1) \cup ... \cup L(d_n) = L(\mathcal{D}_1) \cup ... \cup L(\mathcal{D}_m)$, and $m < n$.

Yu et al. [28] propose a simple practical heuristic to implement *GROUP*. It builds an interference graph with one node per signature and one edge between two nodes if the number of states for their combined *DFA* exceeds the sum for the two individual *DFA*s. The algorithm then greedily groups nodes that have few interferences into sets such that the number of states in the combined *DFA* for one partition does not exceed a heuristic parameter, MAX.

For instance, a set of over 1400 Snort HTTP signatures [11] could not be combined into a single *DFA* that fits in memory, but with Yu's method and MAX=50000, we grouped them into a group of 9 *DFA*s with sizes from 14399 to 49984 states.

C. Key insight: Low Match Frequency and High State Locality

There are two key insights: (1) matches occur rarely, and (2) the match process itself spends most of the time in a tiny fraction of the total number of states. Traditionally, information systems use such disparities in frequency (low entropy) for compression, and we use it to compress *DFA*s.

For instance, we scanned 650k HTTP packets with a 30k-state *DFA* which combines 247 signatures. Only 1789 packets (0.27%) matched. We also measured state occurrences,

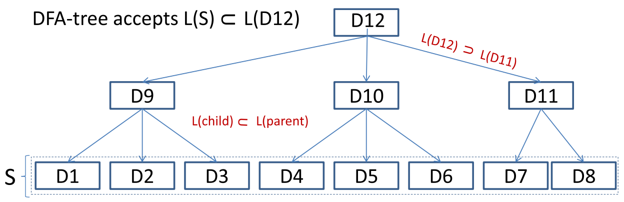

Fig. 2. A DFA-tree. The root is labeled with $\mathcal{D}_{12}$, and the leaves are labeled with $\mathcal{D}_1$,$\mathcal{D}_2$,... $\mathcal{D}_8$. This tree accepts the language $L(\mathcal{S})$ where $\mathcal{S} = \{\mathcal{D}_1,\mathcal{D}_2,\mathcal{D}_3,\mathcal{D}_4,\mathcal{D}_5,\mathcal{D}_6,\mathcal{D}_7,\mathcal{D}_8\}$. If a node $\mathcal{D}$ has a child $\mathcal{C}$ then $L(\mathcal{D}) \supseteq L(\mathcal{C})$. Overall, $L(\mathcal{S}) \subset L(\mathcal{D}_{12})$, i.e $\mathcal{D}_{12}$ is a filter for $L(\mathcal{S})$

by counting how often each state was encountered inside `dfaMatch`. The most frequent state occured 89% of the time, and the top 15 most frequent states occured 99% of the time.

III. DFA-trees

We introduce the *DFA-tree*, a new representation for DFA-set matching. If $\mathcal{S} = \{\mathcal{D}_1, ..., \mathcal{D}_m\}$, a DFA-tree for $\mathcal{S}$ accepts the language $L(\mathcal{S}) = L(\mathcal{D}_1) \cup ... \cup L(\mathcal{D}_m)$.

A DFA-tree is a tree in which each node is labeled with a *DFA*, and the *DFA*s satisfy certain language requirements.

Definition 3: A *labeled tree* over a set Λ is a directed tree where each vertex has a label, $\mathcal{T}$=(V, E, r_0, λ), where

- (V, E, r_0) is a tree
  - V is the set of vertexes
  - $E \subseteq V \times V$ is the set of directed edges
  - $r_0 \in V$ is the root of the tree
- Λ is the set of labels
- $\lambda : V \to \Lambda$ associates a label to each vertex

Definition 4: $\mathcal{D}'$ *overapproximates* $\mathcal{D}$ iff $L(\mathcal{D}) \subseteq L(\mathcal{D}')$.

Definition 5: A *DFA-tree* for an alphabet Σ is a labeled tree $\mathcal{T}$=$(V, E, r_0, \texttt{dfa})$ over $\mathbb{D}_\Sigma$, where

- $\mathbb{D}_\Sigma$, the label set, is the set of all *DFA*s with alphabet Σ.
- $\texttt{dfa} : V \to \mathbb{D}_\Sigma$ labels each node with a *DFA*.
- the *DFA* at a node overapproximates the *DFA* at any child: for all $(v_1, v_2) \in E : L(\texttt{dfa}(v_1)) \supseteq L(\texttt{dfa}(v_2))$.

Definition 6: $\mathcal{T}$ is a DFA-tree for a DFA-set $\mathcal{S}$ iff $\mathcal{S}$ is the set of *DFA*s stored at the leaves of $\mathcal{T}$.

The example in Fig. 2 is a DFA-tree for $\mathcal{S}$=$\{\mathcal{D}_1$,...,$\mathcal{D}_8\}$. The label of the root is $\mathcal{D}_{12}$. The subset requirement implies $L(\{D_1, \mathcal{D}_2, \mathcal{D}_3\}){\subseteq}L(\mathcal{D}_9)$, $L(\{D_4, \mathcal{D}_5, \mathcal{D}_6\}){\subseteq}L(\mathcal{D}_{10})$, $L(\mathcal{D}_7, \mathcal{D}_8){\subseteq}L(\mathcal{D}_{11})$, and $L(\{D_9, \mathcal{D}_{10}, \mathcal{D}_{11}\}){\subseteq}L(\mathcal{D}_{12})$.

Algorithm 2 shows how a DFA-tree for $\mathcal{S}$ performs fast matching on behalf of $\mathcal{S}$. If $\mathcal{S}$ is a DFA-set for the signatures of an IPS, `dfaSetMatch` most of the time rejects payloads. The DFA-tree can accelerate this decision by acting as a filter. Matching starts at the root of the tree and stops at the first node whose DFA does not match. The subset requirement of Definition 5 ensures the correctness of `dfaTreeMatch`:

Theorem 1: If $\mathcal{T}$=$(V, E, r_0, \texttt{dfa})$ is a DFA-tree for $\mathcal{S}$, then $\texttt{dfaTreeMatch}(w, r_0)\texttt{=True} \iff w \in L(\mathcal{S})$

The efficiency of `dfaTreeMatch` depends on how often the slow path is taken; that is, on how often the payload w is in $L(\texttt{dfa}(r_0))$. We know that if $\mathcal{S}$ is the DFA-set for the signatures of an IPS, then, in the common case, $w \notin L(\mathcal{S})$, unfortunately this does not imply that $w \notin L(\texttt{dfa}(r_0))$ (the

Algorithm 2: Matching with DFA-trees. If $\mathcal{T} = (V, E, r_0, \texttt{dfa})$ is a DFA-tree for a DFA-set $\mathcal{S}$, then $\texttt{dfaTreeMatch}(w, r_0)$ decides if $w \in L(\mathcal{S})$.

```
dfaTreeMatch (payload $w$, DFA-tree $v$)
    input : A payload: $w$
    input : A node $v$ in $\mathcal{T} = (V, E, r_0, \texttt{dfa})$
    output: Is any prefix of $w$ in $L(v)$?
    if dfaMatch($w$,dfa($v$)) then
        if $v$ is a leaf then return True; //real match
        // Slow path. Must not be often taken!
        foreach $(v, child) \in E$ do
            if dfaTreeMatch($w$,dfa($child$)) then
                return True;
            end
        end
    end
    return False;
```

implication goes the other way around according to $L(\mathcal{S}) \subseteq L(\texttt{dfa}(r_0))$. If we want $w \notin L(\texttt{dfa}(r_0))$ to happen frequently, we must carefully build the DFA-tree with this goal in mind.

Exact Matches - No false positives: A DFA-tree matches exactly the same language as the DFA-set set at its leaves. Although inner nodes are overapproximations, a match at an inner node does not imply a match by the DFA-tree. An inner node match is verified by matching against its children. An inner node match is declared a match if and only if a leaf under that node also declares it a match, thus it is a real match.

Worst Case Scenario & Resilience to Algorithmic Attacks: DFA-trees guarantee good worst case performance. There is no attack that can slow down matching beyond a tree-dependent limit. The best attack (i.e. our worst case scenario) forces the traversal of the entire tree. The slowdown compared to DFA-set matching is bounded by the ratio of number nodes to that of the leaves; i.e, a constant depending only on the shape of the tree. In Fig. 2 a full tree traversal matches the input against 12 DFAs, this is 1.5X (50%) slower than if DFA-sets were used and the input was matched against the 8 DFAs at the leaves. Such attacks or poor performance are easy to detect and, if persistent, the ISP can temporary switch to DFA-set matching.

Memory Overhead: The memory overhead of DFA-trees relative to DFA-sets is the extra space for the internal nodes. In Fig. 2, the overhead is 50%, assuming equal size DFAs. In our tests, the overhead is much smaller, because internal nodes tend to have many children.

A. DFA-tree Construction

We build the tree bottom up. We alternate *DFA* compression and grouping, until grouping fails to reduce the set size. If we obtain only one element then we have a DFA-tree, otherwise we obtain a set of trees, and matching uses all of them. Figure 3 shows an example of DFA-tree construction for the DFA-set $\mathcal{S}_1$ shown in Fig. 1.

CODFAs: The core concept that enables DFA-tree construction is that of *Compact Overapproximate DFA* (*CODFA*).

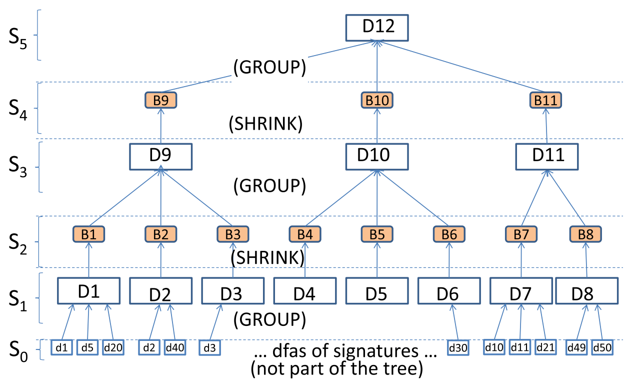

Fig. 3. Example of DFA-tree construction. The DFA-set at the leaves is S_1=*GROUP*(S_0), shown in Fig. 1 from S_0 = the set with one *DFA* per signature. S_2=*SHRINK*(S_1)=$\{$*SHRINK*($\mathcal{D}$)$|\mathcal{D}\in S_1\}$;S_3=*GROUP*(S_2); S_4=*SHRINK*(S_3);S_5=*GROUP*(S_4); If we ignore the *CODFA* results of *SHRINK* (shown as the small shaded *DFA*s), we obtain the tree in Fig. 2.

Definition 7: A *DFA* $\mathcal{D}'$, is a *Compact Overapproximate DFA* (*CODFA*) for another *DFA* $\mathcal{D}$, if:

1) $|\mathcal{D}| > |\mathcal{D}'|$
2) $L(\mathcal{D}) \subseteq L(D')$

The first condition enables combining small *CODFA*s, whereas the large original *DFA*s cannot be combined (see Figure 3). The second condition ensures the inclusion condition of the DFA-tree definition: $\mathcal{D}'$ must match every payload that $\mathcal{D}$ matches. $\mathcal{D}'$ is allowed to err only on the "positive" side i.e., to have false matches: strings that $\mathcal{D}'$ accepts but $\mathcal{D}$ rejects.

In practice, a *CODFA* $\mathcal{D}'$ for $\mathcal{D}$ must satisfy one more condition: $\mathcal{D}'$ must have a low false match rate, ϵ, compared to $\mathcal{D}$. This ensures that the slow path in Algorithm 2 is indeed taken infrequently, despite the approximation errors introduced at each level. The notion of *false-match rate* is defined in Section III-B as a function of the network traffic.

Assume for now that *SHRINK* is a procedure which returns a *CODFA*, $\mathcal{B} = SHRINK(\mathcal{D})$, for its argument $\mathcal{D}$. Section IV shows a heuristic for *SHRINK*, whose additional parameters are a maximum false match rate, ϵ, and a training set of payloads, $\mathcal{I}$. Here we are only concerned on how to use *SHRINK* and *GROUP* (from Section II-B) to construct a DFA-tree from a DFA-set. Instead of showing the code, we explain our method using the example in Fig. 3.

We start with $\mathcal{S}_0$, a set of *DFA*s, one for each signature, and group them as shown in Fig. 1 and the bottom of Fig. 3. The resulting DFA-set, $\mathcal{S}_1$ =*GROUP*($\mathcal{S}_0$), is the DFA-set that the method in [28] produces, and would use with `dfaSetMatch` to scan the payload. We build a DFA-tree for $\mathcal{S}_1$. Because of their large size, no two *DFA*s in $\mathcal{S}_1$ can be grouped within the memory limits. If we want to further group *DFA*s from $\mathcal{S}_1$, we must first reduce their sizes by overapproximation. We use *SHRINK* to compress each element in $\mathcal{S}_1$ and obtain $\mathcal{S}_2$, a set of *CODFA*s, then repeat grouping and compression steps until we can no longer reduce the size of the DFA-set. When this happens, we estimate if a higher ϵ would be acceptable. If yes, then we increase ϵ and resume compression and grouping; otherwise the algorithm terminates.

If we ignore the *CODFA*s in Fig. 3 (the small shaded *DFA*s at the levels of $\mathcal{S}_2$ and $\mathcal{S}_4$), and the *DFA*s for the signatures ($\mathcal{S}_0$, at the bottom) then the relationship between the rest of the *DFA*s forms the *DFA-tree* shown in Fig. 2. It contains a vertex for each *DFA* at the leaves, and for each *DFA* created by *GROUP*. The edges show the language inclusion relationship due to overapproximation and grouping. We do not add vertexes for the intermediate *CODFA*s because we use the more precise *DFA*s that they approximate. We do not create vertexes for the signature *DFA*s in $\mathcal{S}_0$, because $L(\mathcal{S}_0) = L(\mathcal{S}_1)$ and a match for any *DFA* at the leaves, $\mathcal{S}_1$, is a real match. The *DFA*s at the leaves store additional information for accepting states, so that an IPS can identify the exact set of signatures that match, and possibly resume scanning.

B. Measuring Approximation Errors

Inferring the distribution of the payloads scanned by an ISP is outside the scope of our work. Instead, we assume that we have a collection of payloads, $\mathcal{I}$, which is a finite multiset (i.e. repetitions allowed)[2] with $|\mathcal{I}|$ elements representative of the payload distribution. By abuse of notation we call $\mathcal{I}$ a *training set*. We define a number of parameters based on $\mathcal{I}$.

Definition 8: Let $\mathcal{D}, \mathcal{D}'$ be *DFA*s over the same alphabet Σ, and $\mathcal{I} \neq \emptyset$ a training set with elements in Σ^+.

- A *false match* for $\mathcal{D}'$ relative to $\mathcal{D}$ is any $w \in L(\mathcal{D}')$-$L(\mathcal{D})$.
- $N^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) = |(L(\mathcal{D}') - L(\mathcal{D})) \cap \mathcal{I}|$ is the number of false matches for $\mathcal{D}'$ relative to $\mathcal{D}$, as measured on $\mathcal{I}$.
- $F^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) = N^+_\mathcal{I}(\mathcal{D}', \mathcal{D})/|\mathcal{I}|$ is the probability of a false match for $\mathcal{D}'$ relative to $\mathcal{D}$, as measured on $\mathcal{I}$.
- $P_\mathcal{I}(L) = |L \cap \mathcal{I}|/|\mathcal{I}|$ the probability that an arbitrary string in $\mathcal{I}$ belongs to the language L.

We quantify the false-match rate measured on a training set $\mathcal{I}$ using the function $F^+_\mathcal{I}$. The definitions immediately imply that $F^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) = P_\mathcal{I}(L(\mathcal{D}') - L(\mathcal{D}))$.

Lemma 1: If $\mathcal{D}$, $\mathcal{D}'$ are two *DFA*s such that $L(\mathcal{D}) \subseteq L(\mathcal{D}')$, then for every training set $\mathcal{I}$ and $\epsilon \in [0..1]$, we have:

$$F^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) \leq \epsilon \iff |\mathcal{I} - L(\mathcal{D}')| \geq |\mathcal{I} - L(\mathcal{D})| - \epsilon * |\mathcal{I}|$$

Note that $|\mathcal{I} - L(\mathcal{D})|$ is the number of inputs in $\mathcal{I}$ that $\mathcal{D}$ rejects. To attain $F^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) \leq \epsilon$, any $\mathcal{D}'$ must have at most $\epsilon * |\mathcal{I}|$ false matches and reject at least $|\mathcal{I} - L(\mathcal{D})| - \epsilon * |\mathcal{I}|$ inputs in $\mathcal{I}$. These limits do not depend on $\mathcal{D}'$.

Conflicting Constraints

Having a small *DFA* size and having a low false match rate are conflicting constraints for a *CODFA*. Given a training set $\mathcal{I}$, a *DFA* $\mathcal{D}$, and a number k, then in the finite set of *DFA*s with at most k states, there is one that has the fewest false matches relative to $\mathcal{D}$, when measured on $\mathcal{I}$. We can define a function, using the false match rate of this best approximation:

$$minN^+_{\mathcal{I},\mathcal{D}}(k) = min\{N^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) | L(\mathcal{D}) \subseteq L(\mathcal{D}') \wedge |\mathcal{D}'| \leq k\}$$

Lemma 2: For every *DFA* $\mathcal{D}$, and every training set $\mathcal{I}$, for all $k_1, k_2 : 1 \leq k_1 \leq k_2 \Longrightarrow minN^+_{\mathcal{I},\mathcal{D}}(k_1) \geq minN^+_{\mathcal{I},\mathcal{D}}(k_2)$.

If $|\mathcal{D}'| = |\mathcal{D}|$ then there is no need to approximate; we can simply pick $\mathcal{D}' = \mathcal{D}$ and we have 0 false matches. At the other extreme, if $|\mathcal{D}'| = 1$ then $\mathcal{D}'$ must be the one state *DFA* that accepts everything and $F^+_\mathcal{I}(\mathcal{D}', \mathcal{D}) = 1 - P_\mathcal{I}(L(\mathcal{D}))$.

[2]such repetitions allow modeling of non-uniform distributions

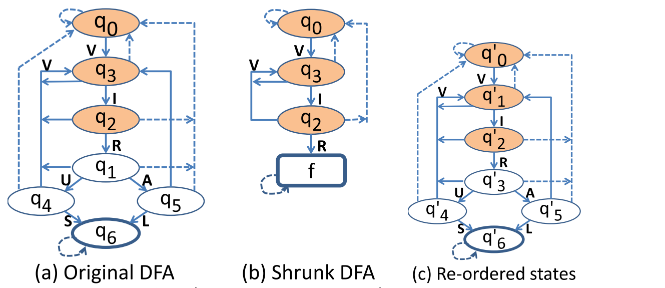

Fig. 4. (a)DFA for $\cdot^{*}$`(VIRUS|VIRAL)`$\cdot^{*}$. The light-dotted lines are the default transitions. Assume that the states ordered in decreasing order of occurrences are q_0,q_3,q_2,q_1,q_4,q_5,q_6. H=$\{q_0,q_3,q_2\}$ are the "hot" states. (b) The corresponding shrunk *DFA*.(c) If states are re-numbered in decreasing order of occurrence, then H=$\{q'_0,q'_1,q'_2\}$ is the valid candidate H_2.

Finding the best CODFA is NP-hard: Finding a *CODFA* within specific limits is a problem with potential applications outside IPSs, but exact, efficient, solutions are unlikely.

Problem 1: Given $\mathcal{D}$, $\mathcal{I}$, and $\epsilon \in [0..1]$, find $\mathcal{D}'$ such that $L(\mathcal{D}) \subseteq L(\mathcal{D}')$, $F^{+}_{\mathcal{I}}(\mathcal{D}',\mathcal{D}) \leq \epsilon$ and $\forall \mathcal{D}'' : F^{+}_{\mathcal{I}}(\mathcal{D}'',\mathcal{D}) \leq \epsilon \implies |\mathcal{D}''| \geq |\mathcal{D}'|$

Problem 1 searches a smallest *CODFA* for a given ϵ. It is NP-hard. This can be shown by picking $\epsilon = 0$, and a reduction from the NP-hard problem *Minimum Inferred Finite State Automaton [AL8]* from [8], which requires finding the minimal DFA consistent with a finite set of positive and a finite set of negative examples.

In practice, it may be useful to solve a weaker problem, which looks for any solution with fewer elements than D. This is the SHRINK problem.

Problem 2: Given $\mathcal{D}$, $\mathcal{I}$, and $\epsilon \in [0..1]$, find $\mathcal{D}'$ such that $L(\mathcal{D}) \subseteq L(\mathcal{D}')$, $F^{+}_{\mathcal{I}}(\mathcal{D}',\mathcal{D}) \leq \epsilon$ and $|\mathcal{D}'| < |\mathcal{D}|$.

Solution quality for SHRINK is judged by $|D'|$: the smaller $|D'|$ the better. For good solutions we need a heuristic.

IV. Practical Heuristic For SHRINK

We present a SHRINK heuristic that works very well in practice. For instance, our algorithm compressed a *DFA* with 100k states to one with 619 states, whose false match rate is $\epsilon \leq 0.8\%$. The heuristic is sound; i.e, if it finds a solution, that solution satisfies the constraints of Problem 2. The heuristic is not complete; i.e, a solution may exist, but we may fail to find it. However, we have not encountered this in practice.

Our insight comes from an observation made in Section II-C that, in practice, the large majority of the scanning time is spent in a small set of *hot* states. Our method does not build a *CODFA* from scratch, it starts with the original *DFA* and uses a greedy approach which preserves the *hot* states and the transitions among them, and which merges all other states into a single new accepting state. This process is called *shrinking*.

In the rest of the section, we formally define shrinking, show that it is an overapproximation, and finally show how to pick a set of *hot* states to obtain a given false match limit.

A. Shrunk DFAs

Informally, if $H \subseteq Q$, the shrunk *DFA* corresponding to $\mathcal{D}$=$(Q, \Sigma, \delta, q_0, F)$ and H is the *restriction* of $\mathcal{D}$ to H, i.e. the *DFA* obtained from $\mathcal{D}$ by keeping the states in H and the transitions between them, and by collapsing the remaining states into a new final state. Figure 4(a) shows the *DFA* corresponding to the regular expression $\cdot^{*}$`(VIRUS|VIRAL)`$\cdot^{*}$. The light-dotted lines are the default transitions. Figure 4(b) shows the shrunk *DFA* obtained by grouping in a new state $f \notin Q$ all states that are not in the hot set H=$\{q_0, q_3, q_2\}$.

Definition 9: The shrunk *DFA* for $\mathcal{D}$=$(Q, \Sigma, \delta, q_0, F)$ and the "hot" set $H \subseteq Q$ is $\mathcal{D}_H$=$(Q', \Sigma, \delta_H, q_0, F_H)$, where

- $Q' = H \cup \{q_0, f\}$; f is a new final state.
- $\mathcal{D}_H$ and $\mathcal{D}$ have the same alphabet.
- $\delta_H : Q' \times \Sigma \to Q'$: is the transition function
$$\delta_H(s,c) = \begin{cases} \delta(s,c), & \text{if } s \neq f \wedge \delta(s,c) \in H \cup \{q_0\} \\ f, & \text{if } s = f \vee \delta(s,c) \notin H \cup \{q_0\} \end{cases}$$
- $q_0 \in Q$ is the same start state as for $\mathcal{D}$.
- $F_H = (F \cap H) \cup \{f\}$

Lemma 3: If all states are reachable from q_0 in $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, then $\forall H : H \subseteq Q \implies L(\mathcal{D}) \subseteq L(\mathcal{D}_H)$, and the inequality is strict if $((Q - H) - F) \neq \emptyset$

Therefore a shrunk *DFA* overapproximates the original *DFA*.

Corollary 1: If $\mathcal{D}$=$(Q, \Sigma, \delta, q_0, F)$ and $H' \subseteq H \subseteq Q$, then $L(\mathcal{D}_H) \subseteq L(\mathcal{D}_{H'})$ and $F^{+}_{\mathcal{I}}(\mathcal{D}_H, \mathcal{D}) \leq F^{+}_{\mathcal{I}}(\mathcal{D}_{H'}, \mathcal{D})$.

Proof: Follows from Lemma 3 and $\mathcal{D}_{H'} = (\mathcal{D}_H)_{H'}$.

Notations: $\texttt{states}_{\mathcal{D}}\texttt{(}w\texttt{)}$ is the set of states encountered while performing regular DFA scanning of w with $\mathcal{D}$. If Q is ordered as $\{q_0, q_1, ..., q_{|Q|-1}\}$, then $\texttt{smax}_{\mathcal{D}}\texttt{(}w\texttt{)}$ is the largest index k of a state $q_k \in \texttt{states}_{\mathcal{D}}\texttt{(}w\texttt{)}$.

Example: the sequence of states for the *DFA* in Fig. 4(a) while scanning w=`avian` is q_0, q_3, q_2, q_0, q_0; $\texttt{states}_{\mathcal{D}}\texttt{(}w\texttt{)}$ is $\{q_0, q_2, q_3\}$; and $\texttt{smax}_{\mathcal{D}}\texttt{(}w\texttt{)}$ is 3.

Lemma 4: $\forall \mathcal{D}, H' \subset H \subseteq Q : L(\mathcal{D}_{H'}) - L(\mathcal{D}_H) = \{w | w \notin L(\mathcal{D}) \wedge (\texttt{states}_{\mathcal{D}}(\texttt{w}) \subseteq H) \wedge (\texttt{states}_{\mathcal{D}}(\texttt{w}) \not\subseteq H')\}$

Note that if $\texttt{states}_{\mathcal{D}}(\texttt{w}) \not\subseteq H'$, then $w \in L(\mathcal{D}_{H'})$.

B. Using Shrunk DFAs for SHRINK

Given $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, a training set $\mathcal{I}$, and $\epsilon \in (0..1)$, we want to find a small set of hot states H such that $F^{+}_{\mathcal{I}}(\mathcal{D}_H, \mathcal{D}) \leq \epsilon$. The idea is simple, since the majority of the time is spent in a small set of states, we select the hot set to be the first few states in decreasing order of frequency. The complexity is in deciding how many states to select, since the state frequency cannot be directly related to the desired ϵ.

If Q is ordered as $\{q_0, q_1, ..., q_{|Q|-1}\}$, we restrict our search for H to the $|Q|$ sets of *valid candidates* of the form H_k=$\{q_0, q_1, ..., q_k\}$, with $\{q_0\}$=$H_0 \subset H_1 \subset ... H_k ... \subset H_{|Q|-1}$=$Q$. We aim to pick the smallest k such that $F^{+}_{\mathcal{I}}(\mathcal{D}_{H_k}, \mathcal{D}) \leq \epsilon$. For that, we use the intermediary sets $T_k = \{w | w \notin L(\mathcal{D}) \wedge \texttt{smax}_{\mathcal{D}}(\texttt{w}) = k\}$; T_k is the set of inputs correctly rejected by $\mathcal{D}_{H_k}$, but incorrectly accepted by $\mathcal{D}_{H_{k-1}}$.

Lemma 5: For $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ and $k \in \{1, ..., |Q| - 1\}$:

1) $\overline{L(\mathcal{D}_{H_k})} - \overline{L(\mathcal{D}_{H_{k-1}})} = T_k$, where $\overline{L}$=$\{w \in \Sigma^{*} | w \notin L\}$
2) $\overline{L(\mathcal{D}_{H_k})} = \cup_{i=0}^{k} T_i$, and T_i are disjoint sets
3) $|\mathcal{I} - L(\mathcal{D}_{H_k})| = \Sigma_{i=0}^{k} |\mathcal{I} \cap T_i|$

Corollary 2: If $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ and $\epsilon \in (0..1)$, then the minimum k for which $F^{+}_{\mathcal{I}}(\mathcal{D}_{H_k}, \mathcal{D}) \leq \epsilon$ is the minimum k for which $\Sigma_{i=0}^{k} |\mathcal{I} \cap T_i| \geq |\mathcal{I} - L(\mathcal{D})| - \epsilon * |\mathcal{I}|$.

The quantity on the right side of the inequality is a constant that does not depend on k and can be evaluated in a single scan of the training set. All elements $|\mathcal{I} \cap T_i|$ on the left side of the inequality can also be computed in a single scan. This is important because scanning a good training set is expensive: on our data, a single scan takes between 3.5s and 16s.

Our heuristic for *SHRINK* is shown in Algorithm 3. First we scan the training set and determine $|\mathcal{I}|, |L(\mathcal{D}) \cap \mathcal{I}|$, and how often each state in $\mathcal{D}$ occurs. Then we order $\{q_1, ..., q_{|Q|-1}\}$ in decreasing order of occurrences (line 2), to help reduce the size of the candidate H_k. For instance, if $\mathcal{D}$ is the *DFA* in Fig. 4(a), then H=$\{q_0, q_3, q_2\}$ yields $\mathcal{D}_H$ in Fig.4(b). But H is not a valid candidate for $\mathcal{D}$. The smallest candidate with consecutive states that contains H is $H_3 = \{q_0,q_1,q_2,q_3\}$; the solution in Fig.4(b) cannot be found using valid candidates. But if we reorder the states of $\mathcal{D}$ from Fig. 4(a) as in Fig. 4(c), then H becomes the valid candidate $H_2 = \{q'_0, q'_1, q'_2\}$, and the solution in Fig. 4(b) can be found.

Using the new order, we re-scan the input (line 3 of Alg. 3) and measure all sizes $T[i] = |\mathcal{I} \cap T_i|$ = the number of times i=$\mathtt{smax}_{\mathcal{D}}(w) \wedge w \in \mathcal{I} - L(\mathcal{D})$. At line 5 we use Corollary 2 to find the best k. If we find such a k, then the result is $\mathcal{D}_{H_k}$.

SHRINK **Complexity:** If the total length of the payloads in $\mathcal{I}$ is $L(\mathcal{I}) = \Sigma_{w \in \mathcal{I}}|w|$, then scanning the training set at lines 1 and 3 takes $O(L(\mathcal{I}))$ time. Sorting at line 2 takes $O(|Q| \times log(|Q|))$. Finding k at line 5 takes $O(|Q|)$ time, and building the shrunk DFA at line 6 is dominated by minimization in $O(|Q| \times log(|Q|))$. *SHRINK*'s time complexity is $O(L(\mathcal{I}) + |Q| \times log(|Q|))$, and memory overhead is linear in $|Q|$.

Algorithm 3: SHRINK heuristic

```
SHRINK ($\mathcal{I}, \mathcal{D}, \epsilon$): heuristic for SHRINK problem
  input : a training set $\mathcal{I}$
  input : a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$
  input : a false match rate $\epsilon \in [0, 1]$
  output: a shrunk DFA, $\mathcal{D}'$, such that $F^+_{\mathcal{I}}(\mathcal{D}', \mathcal{D}) \leq \epsilon$
1 Scan all $w \in \mathcal{I}$ once with dfaMatch and measure:
    1) $|\mathcal{I}|$ = the number of elements in $\mathcal{I}$
    2) $|L(\mathcal{D}) \cap \mathcal{I}|$ = the number of matches in $\mathcal{I}$
    3) $\forall q_k \in Q$: $n_k$ = the number of times $q_k$
       occured at transition $q \leftarrow \delta(q, b)$ of dfaMatch
2 Reorder the states $\{q_1, q_2, ..., q_{|Q|-1}\}$ in decreasing order
  of occurences: $n_1 \geq n_2 \geq ... \geq n_{|Q|-1}$;
3 Scan all $w \in \mathcal{I}$ once with dfaMatch:
    for each $w \notin L(\mathcal{D})$ increment $T[\mathtt{smax}_{\mathcal{D}}(w)]$.
4 $reject \leftarrow |\mathcal{I} - L(\mathcal{D})| - \epsilon * |\mathcal{I}|$;
5 if $\exists k < |Q| : \Sigma_{i=0}^{k-1} T[i] < reject \leq \Sigma_{i=0}^{k} T[i]$ then
6 |   return $\mathcal{D}_{\{q_0, q_1, ..., q_{k-1}\}}$;            // A SOLUTION
  end
7 return HEURISTIC_FAILED ;           // NO SOLUTION
```

V. Experimental Evaluation

Experimental Highlights: We used DFA-trees for published sets of HTTP SNORT signatures [11] and scanned traces from three sources. We obtained a speedup of $4.7\times$ relative to the method from [28]. The average space overhead was 15%. Worst-case attacks can only achieve a 26% slowdown on average. Shrinking is effective: for an approximation error rate of 0.2% the average compression is 97%.

Name	#Signatures	Origin
snort-small	1376	Snort March 2007 ruleset
snort-large	2592	Snort October 2009 ruleset

TABLE I
Description of signatures

Name	Size/#packets	Origin
Department	3.34GB/2.6M	Departmental webserver
Campus	18.6GB/13M	Campus upstream link
DARPA	1.78GB/1.5M	Lincoln lab DARPA set

TABLE II
Description of traces

A. Comparison metrics

In order to compare matching using DFA-sets versus matching using DFA-trees, we ran two experiments. We used $\mathcal{I}$, a payload set, $\mathcal{S} = \{\mathcal{D}_1, ..., \mathcal{D}_m\}$, a DFA-set which groups a set of signatures (such as $\mathcal{S}_1$ in Fig. 3), and $\mathcal{T}$, a DFA-tree for $\mathcal{S}$.

In the first experiment (the baseline) we scanned every element in $\mathcal{I}$ using $\mathcal{S}$ and `dfaSetMatch`. We obtained t_1, the wall-clock time to scan $\mathcal{I}$. In the second experiment we scanned every element in $\mathcal{I}$ using $\mathcal{T}$ and `dfaTreeMatch`. We obtained t_2, the wall-clock time to scan $\mathcal{I}$.

We define the following metrics for `dfaTreeMatch` relative to `dfaSetMatch`, as measured on $\mathcal{I}$:

Speedup: The speedup is t_1/t_2. The best achievable speedup is equal to the number of leaves of $\mathcal{T}$, $|\mathcal{S}| = m$. This corresponds to the ideal case of scanning with a single *DFA*.

Efficiency: Because the speedup is upper bounded by the number of leaves, it is interesting to know how close it gets to this theoretical limit. The *efficiency* is $\frac{t_1}{t_2 \times |\mathcal{S}|}$, i.e. the ratio between the speedup and the ideal speedup. Efficiency is inspired by the *work-efficiency* metric in parallel algorithms.

Memory overhead: The memory overhead of $\mathcal{T}$ relative to $\mathcal{S}$ is $\frac{\text{\# of DFA states at internal nodes}}{\text{\# of DFA states at leaves}}$

Worst-case guarantees: The effectiveness of our method is based on the assumption that most input will not match neither the original language, nor its approximation. An attack aimed at decreasing performance would feed malicious payloads that always require a visit of all the nodes in the DFA-tree. In this case, performance becomes worse than the baseline because `dfaTreeMatch` visits every *DFA* in the tree, whereas `dfaSetMatch` would only visit the leaves. For a given DFA-tree the slowdown can be estimated to be at most $\frac{\text{\# of nodes in } \mathcal{T}}{\text{\# of nodes in } \mathcal{S}}$.

For the example in Fig. 2, in the worst case DFA-tree matching can cause a slowdown of at most $12/8 = 1.5\times$, i.e. 50% slower than DFA-set matching. If a particular payload is scanned in 2 seconds with `dfaTreeMatch`, and in 10s with `dfaSetMatch`, then the speedup is $10/2 = 5$, and the efficiency is $5/8 = 0.625$, or 62.5%. If all the *DFA*s have the same size, then the memory overhead is about $4/8 = 50\%$.

Compression: We measured how effective the algorithm *SHRINK* is in reducing the DFA size, by showing the relative number of states removed. If $\mathcal{D}'$ =*SHRINK*$(\mathcal{I}, \mathcal{D}, \epsilon)$ we define the compression to be $\frac{|D|-|D'|}{|\mathcal{D}|}$.

B. Methodology

We used four parameters to measure how matching with DFA-trees compares versus matching with DFA-sets: (1) a set of signatures, sigs=$\{s_1, s_2, ..., s_n\}$; (2) a maximum number of allowed states per *DFA*, MAX; (3) a training set of payloads, $\mathcal{I}_t$; and (4) an evaluation set of payloads, $\mathcal{I}_e$.

We built a *DFA* for each signature, and obtained a set of *DFA*s, $\mathcal{S}_0$, then we used *GROUP* to group them in into a DFA-set $\mathcal{S}_1$ =*GROUP*($\mathcal{S}_0$,MAX), where $\mathcal{S}_1$ is the DFA-set corresponding to the method in [28] (see Fig. 1). We used the training set $\mathcal{I}_t$ to build $\mathcal{T}$, a DFA-tree for $\mathcal{S}_1$; i.e. *SHRINK* in Algorithm 3 uses $\mathcal{I}_t$ as the training set (see Fig. 3). We start building the DFA-tree using $\epsilon = 0.002$ as the false match rate for *SHRINK*, and increase it, if needed and possible. For space reasons we don't report results for other starting values of ϵ.

Then we compared matching with dfaTreeMatch using $\mathcal{T}$, relative to dfaSetMatch using $\mathcal{S}$, measured on the evaluation set $\mathcal{I}_e$. We report the metrics from Sec. V-A.

C. Datasets and Testbed

We used two signature sets, and three payload sources. All six combinations of signature sets and payloads were tested.
Signatures: We used two sets of signatures created from rules for the Snort IDS. These sets were published [11] by the authors of [27]. The sets are summarized in Table I.
Traces: We used three distinct sets of traces, summarized in Table II. Each set was partitioned into a *training set*, used for DFA-tree construction, and a *data set* used for evaluation (i.e. we used a 2-way cross validation; we also used a 4-way cross validation with similar results, omitted here for simplicity). The *Department* set consists of two traces of traffic from our department; we used the first for training and the second for evaluation. The *Campus* set consists of two traces, each including six 2GB samples from different times of the day. We used the first capture for training, and the second for evaluation. The *DARPA* set consists of two traces from the well-known DARPA intrusion detection evaluation set provided by Lincoln Lab [13]. In particular, we used week #2 for training and week #4 for evaluation.
Testbed: We ran all the experiments on a Linux server with 8 Xeon cores running at 2.5GHz and 16GB RAM. Note that our implementation is single-threaded, thus the number of cores does not affect results.

D. Results

Speedup, efficiency: Fig. 5, 6 show respectively the speedup and efficiency for all combinations of signatures/trace sets, as a function of MAX, the maximum number of states in a *DFA*. The same information is summarized in Table III. In many cases, our approach leads to speedups well above 4×, and efficiency is between 40% and 60% of the theoretical optimum (we remind that such optimum is unattainable in practice because it requires that all the signatures be merged into a single DFA which retains the same cache locality of the leaf nodes in the DFA-set $\mathcal{S}$). However, performance - in particular efficiency - for the snort-large signature set are consistently worse than for snort-small. We determined that (1) the large size of the set causes the instantiation of 32-60 leaves, making the impact of each matching payload significant, and (2) this set matches a consistent percentage of payloads, above 5% in all traces. In particular, 33% of the payloads in the DARPA trace match some signature, causing the low speedup observable in Fig. 5. We plan to investigate the issue further, but we speculate that such unusually high matching rates are due to signatures that are obsolete or have excessive generality.

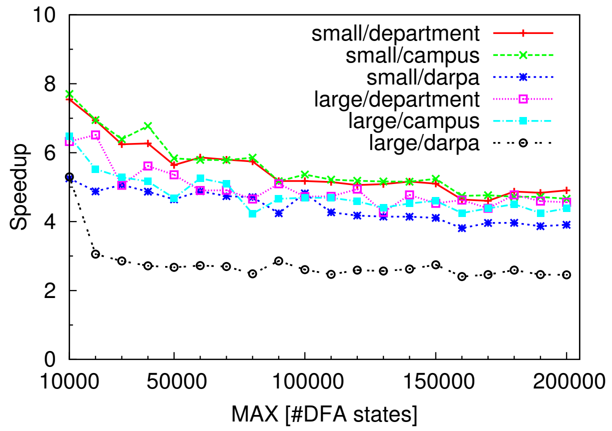

Fig. 5. Speedups

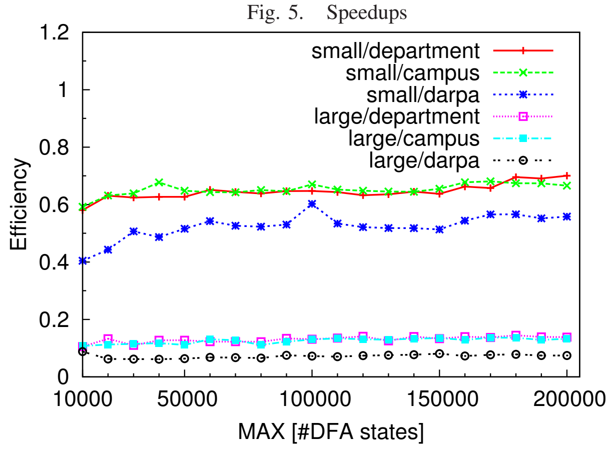

Fig. 6. Efficiency

Worst-case performance: We estimated the worst-case slowdown (see Sections III and V-A) by computing the ratio between the number of *DFA*s in the DFA-tree, and the number of *DFA*s in the baseline DFA-set. The results are shown in table III; additional experiments (omitted) involving worst-case matching times confirmed this prediction.

Memory overhead: Figure 7 and columns 9-11 of Table III summarize the memory overhead due to the additional nodes in the DFA-tree. The overhead is limited, averaging 15% and not exceeding 30% in any of the tests. The memory used during matching was between 100MB and 5GB, depending on the data structure (DFA-set or DFA-tree), signature set and the upper bound for DFA size, MAX. For instance, if MAX=200000, a single DFA could use up to 200MB.

Compression: Columns 15-17 of Table III show the average reduction of *DFA* sizes resulted from shrinking. For most DFAs the set of hot states is small, allowing an average size reduction of 97% for an approximation error rate of 0.2%.

Trace set	Sig. set	Speedup			Efficiency			Mem. overhead			Worst-case			Compression		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Depart.	snort-small	4.60	5.49	7.55	58%	65%	70%	12%	18%	28%	25%	29%	38%	33%	97%	99%
	snort-large	4.27	4.97	6.52	11%	13%	14%	11%	15%	19%	18%	21%	28%	11%	98%	99%
Campus	snort-small	4.66	5.56	7.70	59%	65%	68%	6.9%	12%	19%	25%	29%	38%	44%	96%	99%
	snort-large	4.23	4.78	6.47	11%	13%	14%	9.6%	14%	18%	18%	21%	25%	15%	99%	99%
DARPA	snort-small	3.81	4.42	5.26	40%	52%	60%	13%	20%	30%	25%	35%	43%	45%	96%	99%
	snort-large	2.41	2.77	5.30	6%	7%	9%	9.6%	13%	17%	16%	20%	25%	10%	99%	99%
Overall		2.41	4.67	7.70	6%	36%	70%	6.9%	15%	30%	16%	26%	43%	10%	97%	99%

TABLE III
SUMMARY OF RESULTS

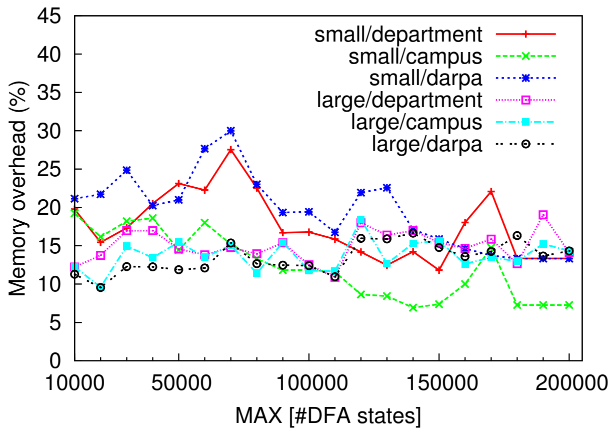

Fig. 7. Memory overhead

E. Discussion

Results show that (1) shrinking is effective, and (2) our approach leads to a significant speedup over the baseline matching, with comparable memory usage. We did not evaluate the performance within a full-fledged intrusion detection system, for two main reasons. The first is that our goal is to provide a useful primitive - fast DFA-based matching - not a complete NIDS solution. The second is that NIDS analyzes are built around and optimized for a specific set of primitives; it is unclear how to estimate the impact of a new primitive without restructuring the architecture of the system. The relationship – and possibly the integration – of shrinking with other work in the area is qualitatively discussed in Section VI.

VI. Related Work

The use of REs to describe attack signatures is well established in the literature [3], [23], [26], [24]. DFA representations for REs [10] offer deterministic matching time and memory bandwidth, but scan only one byte at a time. This problem is addressed by multi-stride automata [4] which consume k symbols at a time, leading to a k-fold decrease in the number of processing steps. Speculative Parallel Pattern Matching [15] partitions the payload in n chunks and speculatively matches them in parallel, later validating the result. Both are compatible with our work as they can be adapted to use DFA-trees.

The state space explosion of *DFA* was addressed in [28], [24], [23], [1], [27]. Common patterns that cause the explosion are analyzed in [28], [1]. Yu et al. [28] introduces the notion that two *DFA*s *interact* with each other if their combined *DFA* has more states than the two of them together. These interactions are used to greedily combine signatures into a DFA-set; this is the *GROUP* procedure used in this paper. Our work builds on top of *GROUP*; we enable more aggressive merging by compressing the *DFA*s that can no longer be grouped, and then grouping the compressed versions. Our method can work with any other grouping algorithm.

Sommer and Paxson [26] observe that for most packets the matching process visits a limited number of states. They incrementally build the *DFA*, by adding states only when they are needed. However, such incremental construction involves a certain overhead, and an attacker can still force the instantiation of the full *DFA*. Smith et al.[24] use EXtended Finite Automata (XFAs) to address the *DFA* state explosion problem. They generate a compact representation similar to a *DFA*, but using additional memory and code. They however assume REs consisting of non-overlapping patterns interleaved by $\cdot^*$ terms, an assumption that may not always be valid in practice [3]. Other techniques decrease the memory footprint of a *DFA* by performing state [12] or alphabet [2] compression, or more sophisticated forms of encoding [14]. Such techniques can be used with our work, as they work on the *DFA* representation.

NFAs [10] are usually more compact than *DFA*s, but have the drawback that multiple states may be active at the same time and they do not exhibit deterministic computation and memory bandwidth guarantees, desirable properties that *DFA*s have. Hardware implementations [22] alleviate *NFA*s disadvantages. Another solution [1] uses a *hybrid-FA* which is mostly a *DFA*, but retain parts that would cause a state-space explosion in NFA form. Shrinking can be seen as complementary, as it can be applied to the *DFA* part of the automaton.

Other researchers considered the idea of pre-filtering traffic. Snort [25] defines for each rule a *fast pattern* - the longest fixed string appearing in the rule [5], which sifts traffic before rule matching; an approach that may fail if such string is very short or appears frequently. The work closest to ours is perhaps protomatching by Rubin et al.[21]. Protomatching detects, by manual inspection, a number of meta patterns and transforms each signature that fits one of the patterns into a smaller RE. It is purely syntactic, it assumes that the smaller expressions match infrequently, and it does not scale: if the smaller expressions do not fit into a single DFA, this method does not allow for further reductions. Our method can be considered a generalization of these approaches, with the advantage of being fully automated, and based on the characteristics of the traffic. We make no assumptions about the input, other than that we have a representative sample.

Finding the best CODFA and DFA shrinking can be considered variants of automata (or grammatical) inference [18] which searches for DFA consistent with a set of positive and a set of negative examples. Exact learning of such a minimal DFA is NP-hard [8], [9]. The concept of *quotient automata* is often used in automata inference work [18], [6], [17]. A *quotient automaton* is obtained by partitioning the states of an automaton, and merging the states in the same partition. The result may not be deterministic. A shrunk DFA is a special case of *quotient automaton*, guaranteed to be deterministic and smaller than the original. SHRINK's result are CODFA with bounded error rate. We are not aware of other work that investigates the probability that a word is in the language difference between a DFA and its *quotient automaton*.

The "best matching method" cannot be specified. There is a huge volume of related work, a variety of matching methods (both in software and in hardware, using specialized devices, or generic ones such as TCAM[16][19] or GPU[29][30]), and a diversity of testing conditions (signature sets, set sizes, payloads, architectures, measured results or estimated based on back-of-the-envelope speedup calculations [19]). The choice of a method also depends on the budget available for the hardware, thus we cannot offer a direct comparison with the "best method". Instead, we offer a method that can enhance other DFA-based software or hardware approaches. We used a commodity computer and published signature sets[11].

VII. Conclusions and future work

We propose the DFA-tree structure for arbitrary REs matching in IPS. It has high matching speed, low memory overhead, and guaranteed worst case performance. In our tests on thousands of signatures, compared to previous representations, DFA-trees match $4.7\times$ faster, require only 15% extra space, and the best attack could slow them down by only 26%.

We introduce the concept of Compact Overapproximate DFA, i.e. a smaller *DFA* that accepts a superset of language of the DFA which is approximated. *CODFA*s with certain accuracy and space parameters allow for a hierarchical representation of large signature sets as DFA-trees. Defining the NP-hard problem of finding the smallest *CODFA* for a given accuracy is a main theoretical contribution of the paper. The main practical contribution of this paper is the use of shrinking as a heuristic. Shrinking builds a *CODFA* by maintaining the states and the transitions between states in a set of high frequency, "hot" states, and by merging the rest of the states into a single new final state. This greedy method averages a 97% size reduction, due to the state locality of DFA matching.

Future Work: Frequency based language approximation and DFA compression could perhaps be applied in other CS areas.

We aimed to introduce the DFA-trees and shrunk-DFAs as useful primitives for IPSs, and our tests show that they are very promising, but we have not fully explored various design choices and possible optimizations. For instance, we obtained *CODFA*s by shrinking, but we believe that there are better methods that take into account the structure of the directed graph induced by a *DFA*'s states and transitions. Also, we attempted a fast and simple DFA-tree construction, but optimizations are possible to improve their efficiency. Matching with DFA-trees can easily be parallelized, or perhaps used with hardware methods for signature matching.

References

[1] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. CoNEXT, 2007.

[2] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. ANCS, 2007.

[3] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating regular expression matching engines on network and general purpose processors. ANCS, 2009.

[4] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. ISCA, 2006.

[5] Brian Caswell, Jay Beale, and Andrew Baker. *Snort IDS and IPS Toolkit*. Syngress, Waltham, MA, USA, 2007.

[6] P. Dupont, L. Miclet, and E. Vidal. What is the search space of the regular inference? ICGI, 1994.

[7] Andrzej Ehrenfeucht and Paul Zeiger. Complexity measures for regular expressions. STOC, 1974.

[8] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[9] E Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302 – 320, 1978.

[10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, Boston, MA, USA, 2006.

[11] http://www.cs.rutgers.edu/∼vinodg/papers/raid2010/data/.

[12] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. SIGCOMM, 2006.

[13] Richard Lippmann, Joshua W. Haines, David J. Fried, Jonathan Korba, and Kumar Das. Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. RAID, 2000.

[14] C. Liu and J. Wu. Fast deep packet inspection with a dual finite automata. *IEEE Transactions on Computers*, 2011.

[15] Daniel Luchaup, Randy Smith, Cristian Estan, and Somesh Jha. Multi-byte regular expression matching with speculation. RAID, 2009.

[16] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. USENIX Security, 2010.

[17] Rajesh Parekh and Vasant Honavar. Efficient learning of regular languages using teacher-supplied positive samples and learner-generated queries. 1993.

[18] Rajesh Parekh and Vasant G. Honavar. Learning DFA from simple examples. *Mach. Learn.*, 44(1-2), July 2001.

[19] Kunyang Peng, Siyuan Tang, Min Chen, and Qunfeng Dong. Chain-based dfa deflation for fast and scalable regular expression matching using tcam. ANCS, 2011.

[20] Martin Roesch. Snort-lightweight intrusion detection for networks. 1999.

[21] Shai Rubin, Somesh Jha, and Barton P. Miller. Protomatching network traffic for high throughput network intrusion detection. CCS, 2006.

[22] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. FCCM, 2001.

[23] Randy Smith, Cristian Estan, and Somesh Jha. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. SIGCOMM, 2008.

[24] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. IEEE SSP, Oakland, CA, 2008.

[25] Snort website. `http://www.snort.org/`.

[26] Robin Sommer and Vern Paxson. Enhancing byte-level network intrusion detection signatures with context. CCS, 2003.

[27] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving nfa-based signature matching using ordered binary decision diagrams. RAID, 2010.

[28] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. ANCS, 2006.

[29] Xiaodong Yu and Michela Becchi. Gpu acceleration of regular expression matching for large datasets: Exploring the implementation space. CF, 2013.

[30] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. Gpu-based nfa implementation for memory efficient high speed regular expression matching. PPoPP, 2012.