Bypassing Space Explosion in High-Speed Regular Expression Matching

Jignesh Patel, Alex X. Liu, and Eric Torng

***Abstract*—Network intrusion detection and prevention systems commonly use regular expression (RE) signatures to represent individual security threats. While the corresponding deterministic finite state automata (DFA) for any one RE is typically small, the DFA that corresponds to the entire set of REs is usually too large to be constructed or deployed. To address this issue, a variety of alternative automata implementations that compress the size of the final automaton have been proposed such as extended finite automata (XFA) and delayed input DFA (D^2FA). The resulting final automata are typically much smaller than the corresponding DFA. However, the previously proposed automata construction algorithms do suffer from some drawbacks. First, most employ a "Union then Minimize" framework where the automata for each RE are first joined before minimization occurs. This leads to an expensive nondeterministic finite automata (NFA) to DFA subset construction on a relatively large NFA. Second, most construct the corresponding large DFA as an intermediate step. In some cases, this DFA is so large that the final automaton cannot be constructed even though the final automaton is small enough to be deployed. In this paper, we propose a "Minimize then Union" framework for constructing compact alternative automata focusing on the D^2FA. We show that we can construct an almost optimal final D^2FA with small intermediate parsers. The key to our approach is a space- and time-efficient routine for merging two compact D^2FA into a compact D^2FA. In our experiments, our algorithm runs on average 155 times faster and uses 1500 times less memory than previous algorithms. For example, we are able to construct a D^2FA with over 80 000 000 states using only 1 GB of main memory in only 77 min.**

***Index Terms*— Deep packet inspection, information security, intrusion detection and prevention, network security, regular expression matching.**

I. Introduction

A. Background and Problem Statement

THE CORE component of today's network security devices such as network intrusion detection and prevention systems is signature-based deep packet inspection. The contents of every packet need to be compared against a set of signatures. Application-level signature analysis can also be used for detecting peer-to-peer traffic, providing advanced QoS mechanisms. In the past, the signatures were specified as simple strings. Today, most deep packet inspection engines such as Snort [2], [3], Bro [4], TippingPoint X505, and Cisco security appliances use *regular expressions* (REs) to define the signatures. REs are used instead of simple string patterns because REs are fundamentally more expressive and thus are able to describe a wider variety of attack signatures [5]. As a result, there has been a lot of recent work on implementing high-speed RE parsers for network applications.

Most RE parsers use some variant of the deterministic finite state automata (DFA) representation of REs. A DFA is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, A)$, where Q is a set of states, Σ is an alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the start, and $A \subseteq Q$ is the set of accepting states. Any set of REs can be converted into an equivalent DFA with the minimum number of states [6], [7]. DFAs have the property of needing constant memory access per input symbol, and hence result in predictable and fast bandwidth. The main problem with DFAs is space explosion: A huge amount of memory is needed to store the transition function, which has $|Q| \times |\Sigma|$ entries. Specifically, the number of states can be very large (state explosion), and the number of transitions per state is large ($|\Sigma|$).

To address the DFA space explosion problem, a variety of DFA variants have been proposed that require much less memory than DFAs to store. For example, there is the *Delayed Input DFA* (D^2FA) proposed by Kumar *et al.* [8]. The basic idea of D^2FA is that in a typical DFA for real-world RE set, given two states u and v, $\delta(u, c) = \delta(v, c)$ for many symbols $c \in \Sigma$. We can remove all the transitions for v from δ for which $\delta(u, c) = \delta(v, c)$ and make a note that v's transitions were removed based on u's transitions. When the D^2FA is later processing input and is in state v and encounters input symbol x, if $\delta(v, x)$ is missing, the D^2FA can use $\delta(u, x)$ to determine the next state. We can do the same thing for most states in the DFA, and it results in tremendous transition compression. Kumar *et al.* observe an average decrease of 97.6% in the amount of memory required to store a D^2FA when compared to its corresponding DFA.

In more detail, to build a D^2FA from a DFA, just do the following two steps. First, for each state $u \in Q$, pick a *deferred* state, denoted by $F(u)$. (We can have $F(u) = u$.) Second, for each state $u \in Q$ for which $F(u) \neq u$, remove all the transitions for u for which $\delta(u, x) = \delta(F(u), x)$.

When traversing the D^2FA, if on current state u and current input symbol x, $\delta(u, x)$ is missing (i.e., has been removed), we can use $\delta(F(u), x)$ to get the next state. Of course, $\delta(F(u), x)$ might be missing too, in which case we then use $\delta(F(F(u)), x)$ to get the next state, and so on. The only restriction on selecting

Manuscript received September 16, 2012; accepted September 11, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor I. Keslassy. Date of publication April 29, 2014; date of current version December 15, 2014. This work was supported in part by the National Science Foundation under Grants No. CNS-1347953, No. CNS-1017588, and No.CNS-1017598. Some preliminary results of this paper were published in the Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, February 5–8, 2012. *(Corresponding author: Alex X. Liu.)*

The authors are with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824 USA (e-mail: patelji1@cse.msu.edu; alexliu@cse.msu.edu; torng@cse.msu.edu).

Digital Object Identifier 10.1109/TNET.2014.2309014

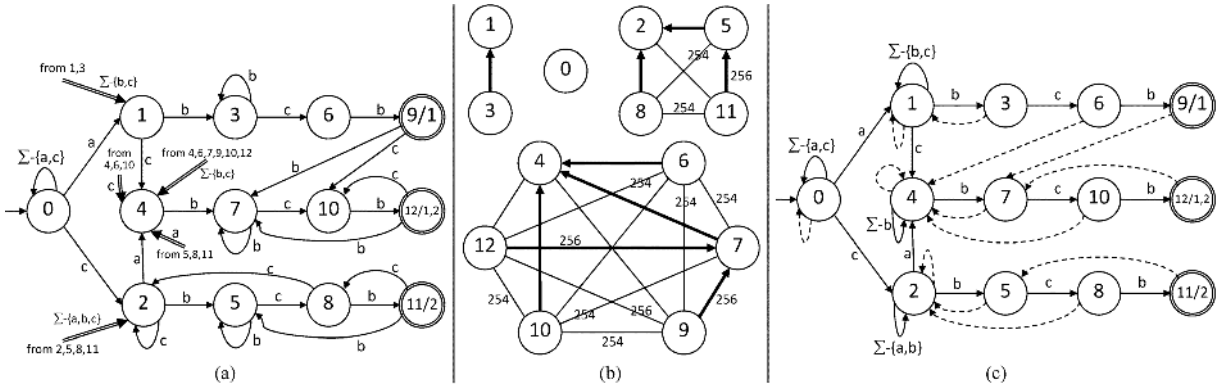


Fig. 1. (a) DFA for $\{.*\texttt{a}.*\texttt{bcb}, .*\texttt{c}.*\texttt{bcb}\}$. (b) Corresponding SRG. Edges of weight ≤ 1 not shown. Unlabeled edges have weight 255. (c) The D^{2}FA.

deferred states is that the function F cannot create a cycle other than a self-loop on the states; otherwise all states on that cycle might have their transitions on some $x \in \Sigma$ removed and there would be no way of finding the next state.

Fig. 1(a) shows a DFA for the REs set $\{.*\texttt{a}.*\texttt{bcb}, .*\texttt{c}.*\texttt{bcb}\}$, and Fig. 1(c) shows the D^{2}FA built from the DFA. The dashed lines represent deferred states. The DFA has $13 \times 256 = 3328$ transitions, whereas the D^{2}FA only has 1030 actual transitions and 9 deferred transitions.

D^{2}FA are very effective at dealing with the DFA space explosion problem. In particular, D^{2}FA exhibit tremendous transition compression reducing the size of the DFA by a huge factor; this makes D^{2}FA much more practical for a software implementation of RE matching than DFAs. D^{2}FAs are also used as a starting point for advanced techniques like those in [9] and [10].

This leads us to the fundamental problem we address in this paper. Given as input a set of REs $\mathcal{R}$, build a compact D^{2}FA as efficiently as possible that also supports frequent updates. Efficiency is important as current methods for constructing D^{2}FA may be so expensive in both time and space that they may not be able to construct the final D^{2}FA even if the D^{2}FA is small enough to be deployed in networking devices that have limited computing resources. Such issues become doubly important when we consider the issue of the frequent updates (typically additions) to $\mathcal{R}$ that occur as new security threats are identified. The limited resource networking device must be able to efficiently compute the new D^{2}FA. One subtle but important point about this problem is that the resulting D^{2}FA must report which RE (or REs) from $\mathcal{R}$ matched a given input; this applies because each RE typically corresponds to a unique security threat. Finally, while we focus on D^{2}FA in this paper, we believe that our techniques can be generalized to other compact RE matching automata solutions [11]–[15].

B. Summary of Prior Art

Given the input RE set $\mathcal{R}$, any solution that builds a D^{2}FA for $\mathcal{R}$ will have to do the following two operations: 1) union the automata corresponding to each RE in $\mathcal{R}$, and 2) minimize the automata, both in terms of the number of states and the number of edges. Previous solutions [8], [16] employ a "Union then Minimize" framework where they first build automata for each RE within $\mathcal{R}$, then perform union operations on these automata to arrive at one combined automaton for all the REs in $\mathcal{R}$, and only then minimize the resulting combined automaton. In particular, previous solutions typically perform a computationally expensive NFA to DFA subset construction followed by or composed with DFA minimization (for states) and D^{2}FA minimization (for edges).

Consider the D^{2}FA construction algorithm proposed by Kumar *et al.* [8]. They first apply the Union then Minimize framework to produce a DFA that corresponds to $\mathcal{R}$ and then construct the corresponding minimum-state DFA. Next, in order to maximize transition compression, they solve a maximum-weight spanning tree problem on the following weighted graph, which they call a *space reduction graph* (SRG). The SRG has DFA states as its vertices. The SRG is a complete graph with the weight of an edge $w(u, v)$ equal to the number of common transitions between DFA states u and v. Once the spanning tree is selected, a root state is picked, and all edges are directed toward the root. These directed edges give the deferred state for each state. Fig. 1(b) shows the SRG built for the DFA in Fig. 1(a).

Becchi and Crowley also use the Union then Minimize framework to arrive at a minimum-state DFA [16]. At this point, rather than using an SRG to set deferment states for each state, Becchi and Crowley use state levels where the level of a DFA state u is the length of the shortest string that takes the DFA from the start state to state u. Becchi and Crowley observed that if all states defer to a state that is at a lower level than itself, then the deferment function F can never produce a cycle. Furthermore, when processing any input string of length n, at most $n-1$ deferred transitions will be processed. Thus, for each state u, among all the states at a lower level than u, Becchi and Crowley set $F(u)$ to be the state that shares the most transitions with u. The resulting D^{2}FA typically has a few more transitions than the minimal D^{2}FA that results by applying the Kumar *et al.* algorithm.

C. Limitations of Prior Art

Prior methods have three fundamental limitations. First, they follow the Union then Minimize framework, which means they create large automata and only minimize them at the end. This also means they must employ the expensive NFA to DFA subset construction. Second, prior methods build the corresponding minimum-state DFA before constructing the final D^{2}FA. This is very costly in both space and time. The D^{2}FA is typically 50 to 100 times smaller than the DFA, so even if the D^{2}FA would fit in available memory, the intermediate DFA might be too large, making it impractical to build the D^{2}FA. This is exacerbated in the case of the Kumar *et al.* algorithm, which needs the SRG that

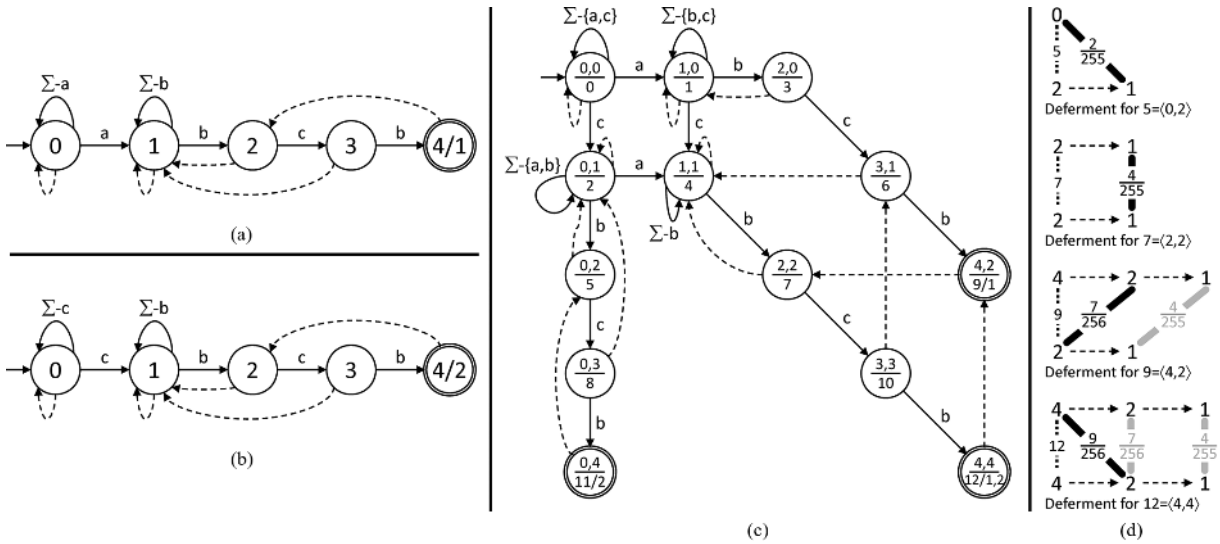


Fig. 2. (a) D_1: D^{2}FA for $.*\texttt{a}.*\texttt{bcb}$. (b) D_2: D^{2}FA for $.*\texttt{c}.*\texttt{bcb}$. (c) D_3: merged D^{2}FA. (d) Illustration of setting deferment for some states in D_3.

ranges from about the size of the DFA itself to over 50 times the size of the DFA. The resulting space and time required to build the DFA and SRG impose serious limits on the D^{2}FA that can be practically constructed. We do observe that the method proposed in [16] does not need to create the SRG. Furthermore, as the authors have noted, there is a way to go from the NFA directly to the D^{2}FA, but implementing such an approach is still very costly in time as many transition tables need to be repeatedly recreated in order to realize these space savings. Third, none of the previous methods provides efficient algorithms for updating the D^{2}FA when a new RE is added to $\mathcal{R}$.

D. Our Approach

To address these limitations, we propose a Minimize then Union framework. Specifically, we first minimize the small automata corresponding to each RE from $\mathcal{R}$, and then union the minimized automata together. A key property of our method is that our union algorithm automatically produces a minimum-state D^{2}FA for the regular expressions involved without explicit state minimization. Likewise, we choose deferment states efficiently while performing the union operation using deferment information from the input D^{2}FAs. Together, these optimizations lead to a vastly more efficient D^{2}FA construction algorithm in both time and space.

In more detail, given $\mathcal{R}$, we first build a DFA and D^{2}FA for each individual RE in $\mathcal{R}$. The heart of our technique is the D^{2}FA merge algorithm that performs the union. It merges two smaller D^{2}FAs into one larger D^{2}FA such that the merged D^{2}FA is equivalent to the union of REs that the D^{2}FAs being merged were equivalent to. Starting from the initial D^{2}FAs for each RE, using this D^{2}FA merge subroutine, we merge two D^{2}FAs at a time until we are left with just one final D^{2}FA. The initial D^{2}FAs are each equivalent to their respective REs, so the final D^{2}FA will be equivalent to the union of all the REs in $\mathcal{R}$. Fig. 2(a) and (b) shows the initial D^{2}FAs for the RE set $\{.*\texttt{a}.*\texttt{bcb}, .*\texttt{c}.*\texttt{bcb}\}$. The resulting D^{2}FA from merging these two D^{2}FAs using the D^{2}FA merge algorithm is shown in Fig. 2(c).

The D^{2}FA produced by our merge algorithm can be larger than the minimal D^{2}FA produced by the Kumar *et al.* algorithm. This is because the Kumar *et al.* algorithm does a global optimization over the whole DFA (using the SRG), whereas our merge algorithm efficiently computes state deferment in the merged D^{2}FA based on state deferment in the two input D^{2}FAs. In most cases, the D^{2}FA produced by our approach is sufficiently small to be deployed. However, in situations where more compression is needed, we offer an efficient final compression algorithm that produces a D^{2}FA very similar in size to that produced by the Kumar *et al.* algorithm. This final compression algorithm uses an SRG; we improve efficiency by using the deferment already computed in the merged D^{2}FA to greatly reduce the size of this SRG and thus significantly reduce the time and memory required to do this compression.

1) Advantages of our Algorithm: One of the main advantages of our algorithm is a dramatic increase in time and space efficiency. These efficiency gains are partly due to our use of the Minimize then Union framework instead of the Union then Minimize framework. More specifically, our improved efficiency comes about from the following four factors. First, other than for the initial DFAs that correspond to individual REs in $\mathcal{R}$, we build D^{2}FA bypassing DFAs. Those initial DFAs are very small (typically $<$50 states), so the memory and time required to build the initial DFAs and D^{2}FAs is negligible. The D^{2}FA merge algorithm directly merges the two input D^{2}FAs to get the output D^{2}FA without creating the DFA first. Second, other than for the initial DFAs, we never have to perform the NFA to DFA subset construction. Third, other than for the initial DFAs, we never have to perform DFA state minimization. Fourth, when setting deferment states in the D^{2}FA merge algorithm, we use deferment information from the two input D^{2}FA. This typically involves performing only a constant number of comparisons per state rather than a linear in the number of states comparison per state as is required by previous techniques. All told, our algorithm has a practical time complexity of $O(n|\Sigma|)$, where n is the number of states in the final D^{2}FA and $|\Sigma|$ is the size of the input alphabet. In contrast, Kumar *et al.*'s algorithm [8] has a time complexity of $O(n^2(\log(n)+|\Sigma|))$, and Becchi and Crowley's algorithm [16] has a time complexity of $O(n^2|\Sigma|)$ just for setting the deferment state for each state and ignoring the cost

of the NFA subset construction and DFA state minimization. See Section V-D for a more detailed complexity analysis.

These efficiency advantages allow us to build much larger D^{2}FAs than are possible with previous methods. For the synthetic RE set that we consider in Section VI, given a maximum working memory size of 1 GB, we can build a D^{2}FA with 80 216 064 states with our D^{2}FA merge algorithm, whereas the Kumar *et al.* algorithm can only build a D^{2}FA with 397 312 states. Also from Section VI, our algorithm is typically 35–250 times faster than previous algorithms on our RE sets.

Besides being much more efficient in constructing D^{2}FA from scratch, our algorithm is very well suited for frequent RE updates. When an RE needs to be added to the current set, we just need to merge the D^{2}FA for the RE to the current D^{2}FA using our merge routine which is a very fast operation.

2) Technical Challenges: For our approach to work, the main challenge is to figure out how to efficiently union two minimum-state D^{2}FAs D_1 and D_2 so that the resulting D^{2}FA D_3 is also a minimum-state D^{2}FA. There are two aspects to this challenge. First, we need to make sure that D^{2}FA D_3 has the minimum number of states. More specifically, suppose D_1 and D_2 are equivalent to RE sets R_1 and R_2, respectively. We need to ensure that D_3 has the minimum number of states of any D^{2}FA equivalent to $R_1 \cup R_2$. We use an existing *union cross-product construction* for this and prove that it results in a minimum-state D^{2}FA for our purpose. We emphasize that this is not true in general, but holds for applications where D_3 must identify which REs from $R_1 \cup R_2$ match a given input string. Many security applications meet this criteria.

Our second challenge is building the D^{2}FA D_3 without building the entire DFA equivalent to D_3 while ensuring that D_3 achieves significant transition compression; that is, the number of actual edges stored in D^{2}FA D_3 must be small. More concretely, as each state in D_3 is created, we need to immediately set a deferred state for it; otherwise, we would be storing the entire DFA. Furthermore, we need to choose a good deferment state that eliminates as many edges as possible. We address this challenge by efficiently choosing a good deferment state for D_3 by using the deferment information from D_1 and D_2. Typically, the algorithm only needs to compare a handful of candidate deferment states.

3) Key Contributions: In summary, we make the following contributions. 1) We propose a novel Minimize then Union framework for efficiently constructing D^{2}FA for network security RE sets that we believe will generalize to other RE matching automata. Note, Smith *et al.* used the Minimize then Union Framework when constructing extended finite automata (XFA) [12], [13], though they did not prove any properties about their union algorithms. 2) To implement this framework, we propose a very efficient D^{2}FA merge algorithm for performing the union of two D^{2}FAs. 3) When maximum compression is needed, we propose an efficient final compression step to produce a nearly minimal D^{2}FA. 4) To prove the correctness of our D^{2}FA merge algorithm, we prove a fundamental property about the standard union cross-product construction and minimum-state DFAs when applied to network security RE sets that can be applied to other RE matching automata.

We implemented our algorithms and conducted experiments on real-world and synthetic RE sets. Our experiments indicate the following. 1) Our algorithm generates D^{2}FA with a fraction of the memory required by existing algorithms, making it feasible to build D^{2}FA with many more states. For the real-world RE sets we consider in Section VI, our algorithm requires an average of 1500 times less memory than the algorithm proposed in [8] and 30 times less memory than the algorithm proposed in [16]. 2) Our algorithm runs much faster then existing algorithms. For the real-world RE sets we consider in Section VI, our algorithm runs an average of 154 times faster than the algorithm proposed in [8] and 19 times faster than the algorithm proposed in [16]. 3) Even with the huge space and time efficiency gains, our algorithm generates D^{2}FA only slightly larger than existing algorithms in the worst case. If the resulting D^{2}FA is too large, our efficient final compression algorithm produces a nearly minimal D^{2}FA.

II. Related Work

Initially, network intrusion detection and prevention systems used string patterns to specify attack signatures [17]–[23]. Sommer and Paxson [5] first proposed using REs instead of strings to specify attack signatures. Today, network intrusion detection and prevention systems mostly use REs for attack signatures. RE matching solutions are typically software-based or hardware-based [field-programmable gate array (FPGA) or application-specific integrated circuit (ASIC)].

Software-based approaches are cheap and deployable on general-purpose processors, but their throughput may not be high. To achieve higher throughput, software solutions can be deployed on customized ASIC chips at the cost of low versatility and high deployment cost. To achieve deterministic throughput, software-based solutions must use DFAs, which face a space explosion problem. Specifically, there can be state explosion where the number of states increases exponentially in the number of REs, and the number of transitions per state is extremely high. To address the space explosion problem, transition compression and state minimization software-based solutions have been developed.

Transition compression schemes that minimize the number of transitions per state have mostly used one of two techniques. One is alphabet re-encoding, which exploits redundancy within a state, [16], [24]–[26]. The second is default transitions or deferment states, which exploit redundancy among states [8], [9], [16], [27]. Kumar *et al.* [8] originally proposed the use of default transitions. Becchi and Crowley [16] proposed a more efficient way of using default transitions. Our work falls into the category of transition compression via default transitions. Our algorithms are much more efficient than those of [8], [16] and thus can be applied to much larger RE sets. For example, if we are limited to 1 GB of memory to work with, we show that Kumar *et al.*'s original algorithm can only build a D^{2}FA with less than 400 000 states, whereas our algorithm can build a D^{2}FA with over 80 000 000 states.

Two basic approaches have been proposed for state minimization. One is to partition the given RE set and build a DFA for each partition [28]. When inspecting packet payload, each input symbol needs to be scanned against each partition's DFA. Our work is orthogonal to this technique and can be used in combination with this technique. The second approach is to modify the automata structure and/or use extra memory to remember history and thus avoid state duplication [11]–[15]. We believe our merge technique can be adopted to work with some of these

approaches. For example, Smith *et al.* also use the Minimize then Union framework when constructing XFA [12], [13]. One potential drawback with XFA is that there is no fully automated procedure to construct XFAs from a set of regular expressions. Paraphrasing Yang *et al.* [29], constructing an XFA from a set of REs requires manual analysis of the REs to identify and eliminate ambiguity.

FPGA-based solutions typically exploit the parallel processing capabilities of FPGAs to implement a nondeterministic finite automata (NFA) [14], [25], [30]–[34] or to implement multiple parallel DFAs [35]. TCAM-based solutions have been proposed for string matching in [21]–[23] and [36] and for REs in [10]. Our work can potentially be applied to these solutions as well.

Recently and independently, Liu *et al.* proposed to construct DFA by hierarchical merging [37]. That is, they essentially propose the Minimize then Union framework for DFA construction. They consider merging multiple DFAs at a time rather than just two. However, they do not consider D^{2}FA, and they do not prove any properties about their merge algorithm including that it results in minimum-state DFAs.

III. Pattern-Matching DFAs

A. Pattern-Matching DFA Definition

In a standard DFA, defined as a 5-tuple $(Q, \Sigma, \delta, q_0, A)$, each accepting state is equivalent to any other accepting state. However, in many pattern-matching applications where we are given a set of REs $\mathcal{R}$, we must keep track of which REs have been matched. For example, each RE may correspond to a unique security threat that requires its own processing routine. This leads us to define pattern-matching deterministic finite state automata (PMDFA). The key difference between a PMDFA and a DFA is that for each state q in a PMDFA, we cannot simply mark it as accepting or rejecting; instead, we must record which REs from $\mathcal{R}$ are matched when we reach q. More formally, given as input a set of REs $\mathcal{R}$, a PMDFA is a 5-tuple $(Q, \Sigma, \delta, q_0, M)$ where the last term M is now defined as $M : Q \rightarrow 2^{\mathcal{R}}$.

B. Minimum-State PMDFA Construction

Given a set of REs $\mathcal{R}$, we can build the corresponding minimum-state PMDFA using the standard Union then Minimize framework: First build an NFA for the RE that corresponds to an OR of all the REs $r \in \mathcal{R}$, then convert the NFA to a DFA, and finally minimize the DFA treating accepting states as equivalent if and only if they correspond to the same set of regular expressions. This method can be very slow, mainly due to the NFA to DFA conversion, which often results in an exponential growth in the number of states. Instead, we propose a more efficient Minimize then Union framework.

Let R_1 and R_2 denote any two disjoint subsets of $\mathcal{R}$, and let D_1 and D_2 be their corresponding minimum-state PMDFAs. We use the standard *union cross-product* construction to construct a minimum-state PMDFA D_3 that corresponds to $R_3 = R_1 \cup R_2$. Specifically, suppose we are given the two PMDFAs $D_1 = (Q_1, \Sigma, \delta_1, q_{01}, M_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_{02}, M_2)$. The union cross-product PMDFA of D_1 and D_2, denoted as $\mathrm{UCP}(D_1, D_2)$, is given by $D_3 = \mathrm{UCP}(D_1, D_2) = (Q_3, \Sigma, \delta_3, q_{03}, M_3)$, where $Q_3 = Q_1 \times Q_2$, $\delta_3(\langle q_i q_j \rangle, x) = \langle \delta_1(q_i, x)\delta_2(q_j, x) \rangle$, $q_{03} = \langle q_{01}, q_{02} \rangle$, and $M_3(\langle q_i, q_j \rangle) = M_1(q_i) \cup M_2(q_j)$.

Each state in D_3 corresponds to a pair of states, one from D_1 and one from D_2. For notational clarity, we use $\langle$ and $\rangle$ to enclose an ordered pair of states. Transition function δ_3 just simulates both δ_1 and δ_2 in parallel. Many states in Q_3 might not be reachable from the start state q_{03}. Thus, while constructing D_3, we only create states that are reachable from q_{03}.

We now argue that this construction is correct. This is a standard construction, so the fact that D_3 is a PMDFA for $R_3 = R_1 \cup R_2$ is straightforward and covered in standard automata theory textbooks (e.g. [6]). We now show that D_3 is also a minimum-state PMDFA for R_3 assuming $R_1 \cap R_2 = \emptyset$, a result that does not follow for standard DFAs.

Theorem III.1: Given two RE sets, R_1 and R_2, and equivalent minimum-state PMDFAs, D_1 and D_2, the union cross-product DFA $D_3 = \mathrm{UCP}(D_1, D_2)$, with only reachable states constructed, is the minimum-state PMDFA equivalent to $R_3 = R_1 \cup R_2$ if $R_1 \cap R_2 = \emptyset$.

Proof: First, since only reachable states are constructed, D_3 cannot be trivially reduced. Now assume D_3 is not minimum. That would mean there are two states in D_3, say $\langle p_1, p_2 \rangle$ and $\langle q_1, q_2 \rangle$, that are indistinguishable. This implies that

$$\forall x \in \Sigma^*, \qquad M_3\left(\delta_3\left(\langle p_1, p_2 \rangle, x\right)\right) = M_3\left(\delta_3\left(\langle q_1, q_2 \rangle, x\right)\right).$$

Working on both sides of this equality, we get $\forall x \in \Sigma^*$

$$\begin{aligned} M_3\left(\delta_3\left(\langle p_1, p_2 \rangle, x\right)\right) &= M_3\left(\langle \delta_1(p_1, x), \delta_2(p_2, x) \rangle\right) \\ &= M_1\left(\delta_1(p_1, x)\right) \cup M_2\left(\delta_2(p_2, x)\right) \end{aligned}$$

as well as $\forall x \in \Sigma^*$

$$\begin{aligned} M_3\left(\delta_3\left(\langle q_1, q_2 \rangle, x\right)\right) &= M_3\left(\langle \delta_1(q_1, x), \delta_2(q_2, x) \rangle\right) \\ &= M_1\left(\delta_1(q_1, x)\right) \cup M_2\left(\delta_2(q_2, x)\right) \end{aligned}$$

This implies that

$$\begin{aligned} &\forall x \in \Sigma^* M_1\left(\delta_1(p_1, x)\right) \cup M_2\left(\delta_2(p_2, x)\right) \\ &\quad = M_1\left(\delta_1(q_1, x)\right) \cup M_2\left(\delta_2(q_2, x)\right). \end{aligned}$$

Now since $R_1 \cap R_2 = \emptyset$, this gives us

$$\begin{aligned} &\forall x \in \Sigma^*, \qquad M_1\left(\delta_1(p_1, x)\right) = M_1\left(\delta_1(q_1, x)\right) \\ &\forall x \in \Sigma^*, \qquad M_2\left(\delta_1(p_2, x)\right) = M_2\left(\delta_1(q_2, x)\right). \end{aligned}$$

This implies that p_1 and q_1 are indistinguishable in D_1 and p_2 and q_2 are indistinguishable in D_2, implying that both D_1 and D_2 are not minimum-state PMDFAs, which is a contradiction, and the result follows. ■

Our efficient construction algorithm works as follows. First, for each RE $r \in \mathcal{R}$, we build an equivalent minimum-state PMDFA D for r using the standard method, resulting in a set of PMDFAs $\mathcal{D}$. Then, we merge two PMDFAs from $\mathcal{D}$ at a time using the above UCP construction until there is just one PMDFA left in $\mathcal{D}$. The merging is done in a greedy manner: In each step, the two PMDFAs with the fewest states are merged together. Note the condition $R_1 \cap R_2 = \emptyset$ is always satisfied in all the merges.

In our experiments, our Minimize then Union technique runs exponentially faster than the standard Union then Minimize technique because we only apply the NFA to DFA step to the NFAs that correspond to each individual regular expression

rather than the composite regular expression. This makes a significant difference even when we have a relatively small number of regular expressions. For example, for our C7 RE set that contains seven REs, the standard technique requires 385.5 s to build the PMDFA, but our technique builds the PMDFA in only 0.66 s. For the remainder of this paper, we use DFA to stand for minimum-state PMDFA.

IV. D^{2}FA Construction

In this section, we first formally define what a D^{2}FA is, and then describe how we can extend the Minimize then Union technique to D^{2}FA bypassing DFA construction.

A. D^{2}FA Definition

Let $D = (Q, \Sigma, \delta, q_0, M)$ be a DFA. A corresponding D^{2}FA D' is defined as a 6-tuple $(Q, \Sigma, \rho, q_0, M, F)$. Together, function $F : Q \rightarrow Q$ and partial function $\rho : Q \times \Sigma \rightarrow Q$ are equivalent to DFA transition function δ. Specifically, F defines a unique deferred state for each state in Q, and ρ is a partially defined transition function. We use $dom(\rho)$ to denote the domain of ρ, i.e., the values for which ρ is defined. The key property of a D^{2}FA D' that corresponds to DFA D is that $\forall \langle q, c\rangle \in Q \times \Sigma, \langle q, c\rangle \in dom(\rho) \iff (F(q) = q \vee \delta(q, c) \neq \delta(F(q), c))$; that is, for each state, ρ only has those transitions that are different from that of its deferred state in the underlying DFA. When defined, $\rho(q, c) = \delta(q, c)$. States that defer to themselves must have all their transitions defined. We only consider D^{2}FA that correspond to minimum-state DFA, though the definition applies to all DFA.

The function F defines a directed graph on the states of Q. A D^{2}FA is well defined if and only if there are no cycles of length > 1 in this directed graph, which we call a deferment forest. We use $p \rightarrow q$ to denote $F(p) = q$, i.e., p directly defers to q. We use $p \twoheadrightarrow q$ to denote that there is a path from p to q in the deferment forest defined by F. We use $p \sqcap q$ to denote the number of transitions in common between states p and q; i.e., $p \sqcap q = |\{c \mid c \in \Sigma \wedge \delta(p, c) = \delta(q, c)\}|$.

The total transition function for a D^{2}FA is defined as

$$\delta'(u, c) = \begin{cases} \rho(u, c), & \text{if } \langle u, c\rangle \in dom(\rho) \\ \delta'(F(u), c), & \text{else.} \end{cases}$$

It is easy to see that δ' is well defined and equal to δ if the D^{2}FA is well defined.

B. D^{2}FA Merge Algorithm

The UCP construction merges two DFAs together. We extend the UCP construction to merge two D^{2}FAs together as follows. During the UCP construction, as each new state u is created, we define $F(u)$ at that time. We then define ρ to only include transitions for u that differ from $F(u)$.

To help explain our algorithm, Fig. 2 shows an example execution of the D^{2}FA merge algorithm. Fig. 2(a) and (b) show the D^{2}FA for the REs $.*\texttt{a}.*\texttt{bcb}$ and $.*\texttt{c}.*\texttt{bcb}$, respectively. Fig. 2(c) shows the merged D^{2}FA for the D^{2}FAs in Fig. 2(a) and 2(b). We use the following conventions when depicting a D^{2}FA. The dashed lines correspond to the deferred state for a given state. For each state in the merged D^{2}FA, the pair of numbers above the line refer to the states in the original D^{2}FAs that correspond to the state in the merged D^{2}FA. The number below the line is the state in the merged D^{2}FA. The number(s) after the "/" in accepting states give(s) the id(s) of the pattern(s) matched. Fig. 2(d) shows how the deferred state is set for a few states in the merged D^{2}FAs D_3. We explain the notation in this figure as we give our algorithm description.

For each state $u \in D_3$, we set the deferred state $F(u)$ as follows. While merging D^{2}FAs D_1 and D_2, let state $u = \langle p_0, q_0\rangle$ be the new state currently being added to the merged D^{2}FA D_3. Let $p_0 \rightarrow p_1 \rightarrow \cdots \rightarrow p_l$ be the maximal deferment chain DC_1 (i.e., p_l defers to itself) in D_1 starting at p_0, and $q_0 \rightarrow q_1 \rightarrow \cdots \rightarrow q_m$ be the maximal deferment chain DC_2 in D_2 starting at q_0. For example, in Fig. 2(d), we see the maximal deferment chains for $u = 5 = \langle 0, 2\rangle$, $u = 7 = \langle 2, 2\rangle$, $u = 9 = \langle 4, 2\rangle$, and $u = 12 = \langle 4, 4\rangle$. For $u = 9 = \langle 4, 2\rangle$, the top row is the deferment chain of state 4 in D_1, and the bottom row is the deferment chain of state 2 in D_2. We will choose some state $\langle p_i, q_j\rangle$ where $0 \leq i \leq l$ and $0 \leq j \leq m$ to be $F(u)$. In Fig. 2(d), we represent these candidate $F(u)$ pairs with edges between the nodes of the deferment chains. For each candidate pair, the number on the top is the corresponding state number in D_3, and the number on the bottom is the number of common transitions in D_3 between that pair and state u. For example, for $u = 9 = \langle 4, 2\rangle$, the two candidate pairs represented are state 7 ($\langle 2, 2\rangle$), which shares 256 transitions in common with state 9, and state 4 ($\langle 1, 1\rangle$), which shares 255 transitions in common with state 9. Note that a candidate pair is only considered if it is reachable in D_3. In Fig. 2(d) with $u = 9 = \langle 4, 2\rangle$, three of the candidate pairs corresponding to $\langle 4, 1\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$ are not reachable, so no edge is included for these candidate pairs. Ideally, we want i and j to be as small as possible though not both 0. For example, our best choices are typically $\langle p_0, q_1\rangle$ or $\langle p_1, q_0\rangle$. In the first case, $p_0 \sqcap p_1 = \langle p_0, q_0\rangle \sqcap \langle p_1, q_0\rangle$, and we already have $p_0 \rightarrow p_1$ in D_1. In the second case, $q_0 \sqcap q_1\rangle = \langle p_0, q_0\rangle \sqcap \langle p_0, q_1\rangle$, and we already have $q_0 \rightarrow q_1$ in D_2. In Fig. 2(d), we set $F(u)$ to be $\langle p_0, q_1\rangle$ for $u = 5 = \langle 0, 2\rangle$ and $u = 12 = \langle 4, 4\rangle$, and we use $\langle p_1, q_0\rangle$ for $u = 9 = \langle 4, 2\rangle$. However, it is possible that both states are not reachable from the start state in D_3. This leads us to consider other possible $\langle p_i, q_j\rangle$. For example, in Fig. 2(d), both $\langle 2, 1\rangle$ and $\langle 1, 2\rangle$ are not reachable in D_3, so we use reachable state $\langle 1, 1\rangle$ as $F(u)$ for $u = 7 = \langle 2, 2\rangle$.

We consider a few different algorithms for choosing $\langle p_i, q_j\rangle$. The first algorithm, which we call the *first match method*, is to find a pair of states (p_i, q_j) for which $\langle p_i, q_j\rangle \in Q_3$ and $i + j$ is minimum. Stated another way, we find the minimum $z \geq 1$ such that the set of states $Z = \{\langle p_i, q_{z-i}\rangle \mid (\max(0, z - m) \leq i \leq \min(l, z)) \wedge (\langle p_i, q_{z-i}\rangle \in Q_3)\} \neq \emptyset$. From the set of states Z, we choose the state that has the most transitions in common with $\langle p_0, q_0\rangle$ breaking ties arbitrarily. If Z is empty for all $z > 1$, then we just pick $\langle p_0, q_0\rangle$, i.e., the deferment pointer is not set (or the state defers to itself). The idea behind the first match method is that $\langle p_0, q_0\rangle \sqcap \langle p_i, q_j\rangle$ decreases as $i + j$ increases. In Fig. 2(d), all the selected $F(u)$ correspond to the first match method.

A second more complete algorithm for setting $F(u)$ is the *best match method* where we always consider all $(l+1) \times (m+1) - 1$ pairs and pick the pair that is in Q_3 and has the most transitions in common with $\langle p_0, q_0\rangle$. The idea behind the best match method is that it is not always true that $\langle p_0, q_0\rangle \sqcap \langle p_x, q_y\rangle \geq \langle p_0, q_0\rangle \sqcap \langle p_{x+i}, q_{y+j}\rangle$ for $i + j > 0$. For instance, we can have $p_0 \sqcap p_2 < p_0 \sqcap p_3$, which would mean $\langle p_0, q_0\rangle \sqcap \langle p_2, q_0\rangle < \langle p_0, q_0\rangle \sqcap \langle p_3, q_0\rangle$. In such cases, the first match method will

Algorithm 1: D2FAMerge(D_1, D_2)

Input: A pair of D^{2}FAs, $D_1 = (Q_1, \Sigma, \rho_1, q_{0_1}, M_1, F_1)$ and $D_2 = (Q_2, \Sigma, \rho_2, q_{0_2}, M_2, F_2)$, corresponding to RE sets, say R_1 and R_2, with $R_1 \cap R_2 = \emptyset$.
Output: A D^{2}FA corresponding to the RE set $R_1 \cup R_2$

```
1  Initialize D_3 to an empty D^2FA;
2  Initialize queue as an empty queue;
3  queue.push (⟨q_{0_1}, q_{0_2}⟩);
4  while queue not empty do
5      u = ⟨u_1, u_2⟩ := queue.pop();
6      Q_3 := Q_3 ∪ {u};
7      foreach c ∈ Σ do
8          nxt := ⟨δ'_1(u_1, c), δ'_2(u_2, c)⟩;
9          if nxt ∉ Q_3 ∧ nxt ∉ queue then queue.push (nxt);
10         ;
11         Add (u, c) → nxt transition to ρ_3;
12     M_3(u) := M_1(u_1) ∪ M_2(u_2);
13     F_3(u) := FindDefState(u);
14     Remove transitions for u from ρ_3 that are in common with F_3(u);
15 foreach u ∈ Q_3 do
16     newDptr := FindDefState(u);
17     if (newDptr ≠ F_3(u)) ∧ (newDptr ⊓ u > F_3(u) ⊓ u) then
18         F_3(u) := newDptr;
19         Reset all transitions for u in ρ_3 and then remove ones that are in
           common with F_3(u);
20 return D_3;

21 FindDefState (⟨v_1, v_2⟩)
22     Let ⟨p_0 = v_1, p_1, ..., p_l⟩ be the list of states on the deferment chain
       from v_1 to the root in D_1;
23     Let ⟨q_0 = v_2, q_1, ..., q_m⟩ be the list of states on the deferment chain
       from v_2 to the root in D_2;
24     for z = 1 to (l + m) do
25         S := {⟨p_i, q_{z−i}⟩ | (max(0, z − m) ≤ i ≤ min(l, z)) ∧
           (⟨p_i, q_{z−i}⟩ ∈ Q_3)};
26         if S ≠ ∅ then return argmax_{v∈S}(⟨v_1, v_2⟩ ⊓ v);
27         ;
28     return ⟨v_1, v_2⟩;
```

not find the pair along the deferment chains with the most transitions in common with $\langle p_0, q_0 \rangle$. In Fig. 2(d), all the selected $F(u)$ also correspond to the best match method. It is difficult to create a small example where first match and best match differ.

When adding the new state u to D_3, it is possible that some state pairs along the deferment chains that were not in Q_3 while finding the deferred state for u will later on be added to Q_3. This means that after all the states have been added to Q_3, the deferment for u can potentially be improved. Thus, after all the states have been added, for each state we again find a deferred state. If the new deferred state is better than the old one, we reset the deferment to the new deferred state. Algorithm 1 shows the pseudocode for the D^{2}FA merge algorithm with the first match method for choosing a deferred state. Note that we use u and $\langle u_1, u_2 \rangle$ interchangeably to indicate a state in the merged D^{2}FA D_3, where u is a state in Q_3, and u_1 and u_2 are the states in Q_1 and Q_2, respectively, that state u corresponds to.

C. Original D^{2}FA Construction for one RE

Before we can merge D^{2}FAs, we first must construct a D^{2}FA for each RE. One option, which we used in the NDSS preliminary version of this paper [1], is the D^{2}FA construction algorithm proposed in [10], which is based on the original D^{2}FA construction algorithm proposed in [8]. This is an effective algorithm that we now describe in more detail.

The first step is to build the SRG: a complete graph where the vertices represent DFA states and the weight of each SRG edge is the number of common transitions between its endpoints in the DFA. Meiners *et al.* note that for real-world RE sets, the distribution of edge weights in the SRG is bimodal, with edge weights typically either very small (<10) or very large (>180). They chose to omit low (<10) weight edges from the SRG, which then produced a forest with many distinct connected components. They then construct a maximum spanning forest of the SRG using Kruskal's algorithm.

The next step is to choose a root state for each connected component of the SRG. For this, Meiners *et al.* choose self-looping states to be root states. A state is a *self-looping state* if it has more than half (i.e., 128) of its transitions looping back to itself. Each component of the SRG has at most one self-looping state. For components that do not have a self-looping state, they choose one of the states in the center of the spanning tree as the root state. After choosing the root for each tree, all the edges in the spanning tree are directed toward the root, giving the deferment pointer for each state.

One subtle point of this algorithm is that there are many cases where multiple edges can be added to the spanning tree. Specifically, Kruskal's algorithm always chooses the edge with the maximum weight from the remaining edges. Since there are only 256 possible edge weights, there often are multiple edges with the same maximum weight. Meiners *et al.* use the following tie-breaking order among edges having the current maximum weight.

1) Edges that have a self-looping state as one of their endpoints are given the highest priority.
2) Next, priority is given to edges with higher sum of degrees (in the current spanning tree) of their end vertices.

D. Improved D^{2}FA Construction for one RE

We now offer an improved algorithm for constructing a D^{2}FA for one RE. This algorithm is similar to that of Meiners *et al.*'s algorithm [10]. The difference is we modify and extend the tie-breaking strategy as follows.

For each state u, we store a value, $deg'(u)$, which is initially set to 0. During Kruskal's algorithm, when an edge $e = (u, v)$ is added to the current spanning tree, $deg'(u)$ is incremented by 2 if $level(u) \leq level(v)$; otherwise it is incremented by 1. Recall that $level(u)$ is the length of the shortest string that takes the DFA from the start state to state u. We similarly update $deg'(v)$. Then, we use the following tie-breaking order among edges having the current maximum weight.

1) Edges that have a self-looping state as one of their end points are given the highest priority.
2) Next, priority is given to edges with higher sum of deg' of their end vertices.
3) Next, priority is given to edges with higher difference between the levels of their end vertices.

The sum of degrees of end vertices is used for tie breaking in order to prioritize states that are already highly connected. However, we also want to prioritize connecting to states at lower levels, so we use deg' instead of just the degree. Using the difference between levels of endpoints for tie breaking also prioritizes states at a lower level. This helps reduce the deferment depth and the D^{2}FA size for RE sets whose D^{2}FAs have a higher average deferment depth. We observe in our experiments section that the improved algorithm does outperform the original algorithm.

E. D²FA Construction for RE set $\mathcal{R}$

We now have methods for constructing a D²FA given one RE and merging two D²FAs into one D²FA. We combine these methods in the natural way to build one D²FA for a set of REs. That is, we first build a D²FA for each RE in $\mathcal{R}$. We then merge the D²FAs together using a balanced binary tree structure to minimize the worst-case number of merges that any RE experiences. We do use two different variations of our D2FAMerge algorithm. For all merges except the final merge, we use the first match method for setting $F(u)$. When doing the final merge to get the final D²FA, we use the best match method for setting $F(u)$. It turns out that using the first match method results in a better deferment forest structure in the D²FA, which helps when the D²FA is further merged with other D²FAs. The local optimization achieved by using the best match method only helps when used in the final merge.

F. Optional Final Compression Algorithm

When there is no bound on the deferment depth (see Section V-B), the original D²FA algorithm proposed in [8] results in a D²FA with smallest possible size because it runs Kruskal's algorithm on a large SRG. Our D²FA merge algorithm results in a slightly larger D²FA because it uses a greedy approach to determine deferment. We can further reduce the size of the D²FA produced by our algorithm by running the following compression algorithm on the D²FA produced by the D²FA merge algorithm.

We construct an SRG and perform a maximum weight spanning tree construction on the SRG, but we only add edges to the SRG that have the potential to reduce the size of the D²FA. More specifically, let u and v be any two states in the current D²FA. We only add the edge $e = (u, v)$ in the SRG if its weight $w(e)$ is $\geq \min(u \sqcap F(u), v \sqcap F(v))$. Here, $F(u)$ is the deferred state of u in the current D²FA. As a result, very few edges are added to the SRG, so we only need to run Kruskal's algorithm on a small SRG. This saves both space and time compared to previous D²FA construction methods. However, this compression step does require more time and space than the D²FA merge algorithm because it does construct an SRG and then runs Kruskal's algorithm on the SRG.

V. D²FA Merge Algorithm Properties

A. Proof of Correctness

The D²FA merge algorithm exactly follows the UCP construction to create the states. Hence, the correctness of the underlying DFA follows from the correctness of the UCP construction.

Theorem V.1 shows that the merged D²FA is also well defined (no cycles in deferment forest).

Lemma V.1: In the D²FA $D_3 = \text{D2FAMerge}(D_1, D_2)$, $\langle u_1, u_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle \Rightarrow u_1 \twoheadrightarrow v_1 \wedge u_2 \twoheadrightarrow v_2$.

Proof: If $\langle u_1, u_2\rangle = \langle v_1, v_2\rangle$, then the lemma is trivially true. Otherwise, let $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle$ be the deferment chain in D_3. When selecting the deferred state for $\langle u_1, u_2\rangle$, D2FAMerge always choose a state that corresponds to a pair of states along deferment chains for u_1 and u_2 in D_1 and D_2, respectively. Therefore, we have that $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \Rightarrow u_1 \twoheadrightarrow w_1 \wedge u_2 \twoheadrightarrow w_2$. By induction on the length of the deferment chain and the fact that the $\twoheadrightarrow$ relation is transitive, we get our result. □

Theorem V.1: If D²FAs D_1 and D_2 are well defined, then the D²FA $D_3 = \text{D2FAMerge}(D_1, D_2)$ is also well defined.

Proof: Since D_1 and D_2 are well defined, there are no cycles in their deferment forests. Now assume that D_3 is not well defined, i.e., there is a cycle in its deferment forest. Let $\langle u_1, u_2\rangle$ and $\langle v_1, v_2\rangle$ be two distinct states on the cycle. Then, we have that

$$\langle u_1, u_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle \wedge \langle v_1, v_2\rangle \twoheadrightarrow \langle u_1, u_2\rangle$$

Using Lemma V.1, we get

$$(u_1 \twoheadrightarrow v_1 \wedge u_2 \twoheadrightarrow v_2) \wedge (v_1 \twoheadrightarrow u_1 \wedge v_2 \twoheadrightarrow u_2)$$
$$i.e.\ (u_1 \twoheadrightarrow v_1 \wedge v_1 \twoheadrightarrow u_1) \wedge (u_2 \twoheadrightarrow v_2 \wedge v_2 \twoheadrightarrow u_2).$$

Since $\langle u_1, u_2\rangle \neq \langle v_1, v_2\rangle$, we have $u_1 \neq v_1 \vee u_2 \neq v_2$, which implies that at least one of D_1 or D_2 has a cycle in their deferment forest which is a contradiction. □

B. Limiting Deferment Depth

Since no input is consumed while traversing a deferred transition, in the worst case, the number of lookups needed to process one input character is given by the depth of the deferment forest. As previously proposed, we can guarantee a worst-case performance by limiting the depth of the deferment forest.

For a state u_1 of a D²FA D_1, the *deferment depth* of u_1, denoted as $\psi(u_1)$, is the length of the maximal deferment chain in D_1 from u_1 to the root. $\Psi(D_1) = \max_{v \in Q_1} \psi(v)$ denotes the deferment depth of D_1 (i.e., the depth of the deferment forest in D_1).

Lemma V.2: In the D²FA $D_3 = \text{D2FAMerge}(D_1, D_2)$, $\forall \langle u_1, u_2\rangle \in Q_3$, $\psi(\langle u_1, u_2\rangle) \leq \psi(u_1) + \psi(u_2)$.

Proof: Let $\psi(\langle u_1, u_2\rangle) = d$. If $\psi(\langle u_1, u_2\rangle) = 0$, then $\langle u_1, u_2\rangle$ is a root and the lemma is trivially true. Thus, we consider $d \geq 1$ and assume the lemma is true for all states with $\psi < d$. Let $\langle u_1, u_2\rangle \rightarrow \langle w_1, w_2\rangle \twoheadrightarrow \langle v_1, v_2\rangle$ be the deferment chain in D_3. Using the inductive hypothesis, we have

$$\psi(\langle w_1, w_2\rangle) \leq \psi(w_1) + \psi(w_2).$$

Given $\langle u_1, u_2\rangle \neq \langle w_1, w_2\rangle$, we assume without loss of generality that $u_1 \neq w_1$. Using Lemma V.1, we get that $u_1 \twoheadrightarrow w_1$. Therefore, $\psi(w_1) \leq \psi(u_1) - 1$. Combining the above, we get $\psi(\langle u_1, u_2\rangle) = \psi(\langle w_1, w_2\rangle) + 1 \leq \psi(w_1) + \psi(w_2) + 1 \leq (\psi(u_1) - 1) + \psi(u_2) + 1 \leq \psi(u_1) + \psi(u_2)$. □

Lemma V.2 directly gives us the following theorem.

Theorem V.2: If $D_3 = \text{D2FAMerge}(D_1, D_2)$, then $\Psi(D_3) \leq \Psi(D_1) + \Psi(D_2)$.

For an RE set $\mathcal{R}$, if the initial D²FAs have $\Psi = d$, in the worst case, the final merged D²FA corresponding to $\mathcal{R}$ can have $\Psi = d \times |\mathcal{R}|$. Although Theorem V.2 gives the value of Ψ in the worst case, in practical cases, $\Psi(D_3)$ is very close to $\max(\Psi(D_1), \Psi(D_2))$. Thus, the deferment depth of the final merged D²FA is usually not much higher than d.

Let Ω denote the desired upper bound on Ψ. To guarantee $\Psi(D_3) \leq \Omega$, we modify the `FindDefState` subroutine in Algorithm 1 as follows: When selecting candidate pairs for the deferred state, we only consider states with $\psi < \Omega$. Specifically, we replace line 23 with the following:

$$S := \{\langle p_i, q_{z-i}\rangle \mid (\max(0, z-m) \leq i \leq \min(l, z)) \wedge \langle p_i, q_{z-i}\rangle \in Q_3) \wedge (\psi(\langle p_i, q_{z-i}\rangle) < \Omega)\}.$$

When we do the second pass (lines 14–19), we may increase the deferment depth of nodes that defer to nodes that we readjust. We record the affected nodes and then do a third pass to reset their deferment states so that the maximum depth bound is satisfied. In practice, this happens very rarely.

When constructing a D^{2}FA with a given bound Ω, we first build D^{2}FAs without this bound. We only apply the bound Ω when performing the final merge of two D^{2}FAs to create the final D^{2}FA.

C. Deferment to a Lower Level

In [16], the authors propose a technique to guarantee an amortized cost of two lookups per input character without limiting the depth of the deferment tree. They achieve this by having states only defer to lower-level states, where the level of any state u in a DFA (or D^{2}FA), denoted $level(u)$, is defined as the length of the shortest string that ends in that state (from the start state). More formally, they ensure that for all states u, $level(u) > level(F(u))$ if $u \neq F(u)$. We call this property the *back-pointer* property. If the back-pointer property holds, then every deferred transition taken decreases the level of the current state by at least 1. Since a regular transition on an input character can only increase the level of the current state by at most 1, there have to be fewer deferred transitions taken on the entire input string than regular transitions. This gives an amortized cost of at most two transitions taken per input character.

In order to guarantee the D^{2}FA D_3 has the back-pointer property, we perform a similar modification to the `FindDefState` subroutine in Algorithm 1 as we performed when we wanted to limit the maximum deferment depth. When selecting candidate pairs for the deferred state, we only consider states with a lower level. Specifically, we replace line 23 with the following:

$$S := \{\langle p_i, q_{z-i}\rangle \mid (\max(0, z-m) \leq i \leq \min(l, z)) \wedge (\langle p_i, q_{z-i}\rangle \in Q_3) \wedge (level(\langle v_1, v_2\rangle) > level(\langle p_i, q_{z-i}\rangle))\}.$$

For states for which no candidate pairs are found, we just search through all states in Q_3 that are at a lower level for the deferred state. In practice, this search through all the states needs to be done for very few states because if D^{2}FAs D_1 and D_2 have the back-pointer property, then almost all the states in D^{2}FAs D_3 have the back-pointer property. As with limiting maximum deferment depth, we only apply this restriction when performing the final merge of two D^{2}FAs to create the final D^{2}FA.

D. Algorithmic Complexity

The time complexity of the original D^{2}FA algorithm proposed in [8] is $O(n^2(\log(n) + |\Sigma|))$. The SRG has $O(n^2)$ edges, and $O(|\Sigma|)$ time is required to add each edge to the SRG and $O(\log(n))$ time is required to process each edge in the SRG during the maximum spanning tree routine. The time complexity of the D^{2}FA algorithm proposed in [16] is $O(n^2|\Sigma|)$. Each state is compared to $O(n)$ other states, and each comparison requires $O(|\Sigma|)$ time.

The time complexity of our new D2FAMerge algorithm to merge two D^{2}FAs is $O(n\Psi_1\Psi_2|\Sigma|)$, where n is the number of states in the merged D^{2}FA, and Ψ_1 and Ψ_2 are the maximum deferment depths of the two input D^{2}FAs. When setting the deferment for any state $u = \langle u_1, u_2\rangle$, in the worst case the algorithm compares $\langle u_1, u_2\rangle$ with all the pairs along the deferment chains of u_1 and u_2, which are at most Ψ_1 and Ψ_2 in length, respectively. Each comparison requires $O(|\Sigma|)$ time. In practice, the time complexity is $O(n|\Sigma|)$ as each state needs to be compared to very few states for the following three reasons. First, the maximum deferment depth Ψ is usually very small. The largest value of Ψ among our eight primary RE sets in Section VI is 7. Second, the length of the deferment chains for most states is much smaller than Ψ. The largest value of average deferment depth $\overline{\psi}$ among our eight RE sets is 2.54. Finally, many of the state pairs along the deferment chains are not reachable in the merged D^{2}FA. Among our eight RE sets, the largest value of the average number of comparisons needed is 1.47.

When merging all the D^{2}FAs together for an RE set $\mathcal{R}$, the total time required in the worst case would be $O(n\Psi_1\Psi_2|\Sigma|\log(|\mathcal{R}|))$. The worst case would happen when the RE set contains strings and there is no state explosion. In this case, each merged D^{2}FA would have a number of states roughly equal to the sum of the sizes of the D^{2}FAs being merged. When there is state explosion, the last D^{2}FA merge would be the dominating factor, and the total time would just be $O(n\Psi_1\Psi_2|\Sigma|)$.

When modifying the D2FAMerge algorithm to maintain back-pointers, the worst-case time would be $O(n^2|\Sigma|)$ because we would have to compare each state with $O(n)$ other states if none of the candidate pairs are found at a lower level than the state. In practice, this search needs to be done for very few states, typically less than 1%.

The worst-case time complexity of the final compression step is the same as that of Kumar *et al.*'s D^{2}FA algorithm, which is $O(n^2(\log(n)+|\Sigma|))$, since both involve computing a maximum weight spanning tree on the SRG. However, because we only consider edges that improve upon the existing deferment forest, the actual size of the SRG in practice is typically linear in the number of nodes. In particular, for the real-world RE sets that we consider in Section VI, the size of the SRG generated by our final compression step is on average 100 times smaller than the SRG generated by Kumar *et al.*'s algorithm. As a result, the optimization step requires much less memory and time compared to the original algorithm.

VI. Experimental Results

In this section, we evaluate the effectiveness of our algorithms on real-world and synthetic RE sets. We consider three variants of our D^{2}FA merge algorithm. We denote the main variant as D^{2}FAMERGE; this variant uses our improved D^{2}FA construction algorithm for one RE. The other two variants are D^{2}FAMergeOld, which uses the D^{2}FA construction algorithm in [10] to build D^{2}FA for each RE and was used exclusively in the preliminary version of this paper, and D^{2}FAMergeAlgoOpt, which applies our final compression algorithm after running D^{2}FAMERGE. We compare our algorithms to *ORIGINAL*, the original D^{2}FA construction algorithm proposed in [8] that optimizes transition compression, and *BACKPTR*, the D^{2}FA construction algorithm proposed in [16] that enforces the back-pointer property described in Section V-C.

A. Methodology

1) Data Sets: Our main results are based on eight real RE sets: four proprietary RE sets C7, C8, C10, and C613 from a large networking vendor and four public RE sets Bro217,

Snort 24, Snort31, and Snort 34, that we partition into three groups, STRING, WILDCARD, and SNORT, based upon their RE composition. For each RE set, the number indicates the number of REs in the RE set. The STRING RE sets, C613 and Bro217, contain mostly string matching REs. The WILDCARD RE sets, C7, C8, and C10, contain mostly REs with multiple wildcard closures ".*"'. The SNORT RE sets, Snort24, Snort31, and Snort34, contain a more diverse set of REs, roughly 40% of which have wildcard closures. To test scalability, we use Scale, a synthetic RE set consisting of 26 REs of the form $.* c_{\mathrm{u}}0123456.* c_{\mathrm{l}}789!\#\%\&/$, where c_{u} and c_{l} are the 26 uppercase and lowercase alphabet letters. Even though all the REs are nearly identical, differing only in the character after the two .*'s, we still get the full multiplicative effect where the number of states in the corresponding minimum-state DFA roughly doubles for every RE added.

2) Metrics: We use the following metrics to evaluate the algorithms. First, we measure the resulting D^{2}FA size (# transitions) to assess transition compression performance. Our D^{2}FAMERGE algorithm typically performs almost as well as the other algorithms even though it builds up the D^{2}FA incrementally rather than compressing the final minimum-state DFA. Second, we measure the maximum deferment depth (Ψ) and average deferment depth ($\overline{\psi}$) in the D^{2}FA to assess how quickly the resulting D^{2}FA can be used to perform regular expression matching. Smaller Ψ and $\overline{\psi}$ mean that fewer deferment transitions that process no input characters need to be traversed when processing an input string. Our D^{2}FAMERGE significantly outperforms the other algorithms. Finally, we measure the space and time required by the algorithm to build the final automaton. Again, our D^{2}FAMERGE significantly outperforms the other algorithms. When comparing the performance of D^{2}FAMERGE with another algorithm A on a given RE or RE set, we define the following quantities to compare them: transition increase is (D^{2}FAMERGE D^{2}FA size – A D^{2}FA size) divided by A D^{2}FA size, transition decrease is (A D^{2}FA size – D^{2}FAMERGE D^{2}FA size) divided by A D^{2}FA size, average (maximum) deferment depth ratio is A average (maximum) deferment depth divided by D^{2}FAMERGE average (maximum) deferment depth, space ratio is A space divided by D^{2}FAMERGE space, and time ratio is A build time divided by D^{2}FAMERGE build time.

Since we have a new D^{2}FAMERGE algorithm, we needed to rerun our experiments. We ran them on faster processors than in our conference version, so all of the algorithms report smaller processing times than before. One interesting note is that while the new D^{2}FAMERGE performs better than D^{2}FAMergeOld, the running times are essentially the same.

3) Measuring Space: When measuring the required space for an algorithm, we measure the maximum amount of memory required at any point in time during the construction and then final storage of the automaton. This is a difficult quantity to measure exactly; we approximate this required space for each of the algorithms as follows. For D^{2}FAMERGE and D^{2}FAMergeOld, the dominant data structure is the D^{2}FA. For a D^{2}FA, the transitions for each state can be stored as pairs of input character and next state id, so the memory required to store a D^{2}FA is calculated as $= (\#transitions) \times 5$ B. However, the maximum amount of memory required while running D^{2}FAMERGE may be higher than the final D^{2}FA size because of the following two reasons. First, when merging two D^{2}FAs, we need to maintain the two input D^{2}FAs as well as the output D^{2}FA. Second, we may create an intermediate output D^{2}FA that has more transitions than needed; these extra transitions will be eliminated once all D^{2}FA states are added. We keep track of the worst-case required space for our algorithm during D^{2}FA construction. This typically occurs when merging the final two intermediate D^{2}FA to form the final D^{2}FA.

For ORIGINAL, we measure the space required by the minimized DFA and the SRG. For the DFA, the transitions for each state can be stored as an array of size Σ with each array entry requiring 4 B to hold the next state id. For the SRG, each edge requires 17 B as observed in [16]. This leads to a required memory for building the D^{2}FA of $= |Q| \times |\Sigma| \times 4 + (\#edges\ in\ SRG) \times 17$ B.

For D^{2}FAMergeOpt, the space required is the size of the final D^{2}FA resulting from the merge step, plus the size of the SRG used by the final compression algorithm. The sizes are computed as in the case of D^{2}FAMERGE and ORIGINAL.

For BACKPTR, we consider two variants. The first variant builds the minimized DFA directly from the NFA and then sets the deferment for each state. For this variant, no SRG is needed, so the space required is the space needed for the minimized DFA, which is $|Q| \times |\Sigma| \times 4$ B. The second variant goes directly from the NFA to the final D^{2}FA; this variant uses less space, but is much slower as it stores incomplete transition tables for most states. Thus, when computing the deferment state for a new state, the algorithm must recreate the complete transition tables for each state to determine which has the most common transitions with the new state. For this variant, we assume the only space required is the space to store the final D^{2}FA, which is $= (\#transitions) \times 5$ B even though more memory is needed at various points during the computation. We also note that both implementations must perform the NFA to DFA subset construction on a large NFA, which means even the faster variant runs much more slowly than D^{2}FAMERGE.

4) Correctness: We tested correctness of our algorithms by verifying the final D^{2}FA is equivalent to the corresponding DFA. Note we can only do this check for our RE sets where we were able to compute the corresponding DFA. Thus, we only verified correctness of the final D^{2}FA for our eight real RE sets and the smaller-scale RE sets.

B. D^{2}FAMERGE versus ORIGINAL

We first compare D^{2}FAMERGE to ORIGINAL that optimizes transition compression when both algorithms have unlimited maximum deferment depth. These results are shown in Table I for our eight primary RE sets. Tables I–III summarize these results by RE group. We make the following observations.

1) D^{2}FAMERGE Uses Much Less Space Than ORIGINAL: On average, D^{2}FAMERGE uses 1500 times less memory than ORIGINAL to build the resulting D^{2}FA. This difference is most extreme when the SRG is large, which is true for the two STRING RE sets and Snort24 and Snort34. For these RE sets, D^{2}FAMERGE uses between 1422 and 4568 times less memory than ORIGINAL. For the RE sets with relatively small SRGs such as those in the WILDCARD and Snort31, D^{2}FAMERGE uses between 35 and 231 times less space than ORIGINAL.

2) D^{2}FAMERGE is Much Faster Than ORIGINAL: On average, D^{2}FAMERGE builds the D^{2}FA 155 times faster

TABLE I
PERFORMANCE DATA OF ORIGINAL AND D^{2}FAMERGE

RE set	# States	ORIGINAL					D^{2}FAMERGE				
		# Trans	Def. depth		RAM	Time	# Trans	Def. depth		RAM	Time
			Avg.	Max.	(MB)	(s)		Avg.	Max.	(MB)	(s)
Bro217	6533	9816	3.90	8	179.3	119.4	11737	2.15	5	0.13	3.2
C613	11308	21633	4.38	11	1042.7	326.0	26709	2.69	7	0.23	9.7
C7	24750	205633	16.38	27	47.4	397.7	207540	1.14	3	1.07	0.9
C8	3108	23209	8.60	14	4.9	14.5	23334	1.14	2	0.14	0.2
C10	14868	96793	16.39	26	25.5	141.0	97296	1.18	3	0.52	0.6
Snort24	13886	38485	9.67	18	861.2	299.2	39409	1.56	4	0.32	0.2
Snort31	20068	70701	9.17	16	298.5	244.3	92284	2.00	6	1.29	2.6
Snort34	13825	40199	10.95	18	795.2	309.9	43141	1.38	5	0.27	1.8

TABLE II
PERFORMANCE DATA OF D^{2}FAMergeOld AND D^{2}FAMergeOpt

RE set	# States	D^{2}FAMergeOld					D^{2}FAMergeOpt				
		# Trans	Def. depth		RAM	Time	# Trans	Def. depth		RAM	Time
			Avg.	Max.	(MB)	(s)		Avg.	Max.	(MB)	(s)
Bro217	6533	12325	2.16	5	0.10	3.2	9816	2.44	7	2.64	99.2
C613	11308	34991	2.54	7	0.29	9.7	21633	3.04	8	7.48	940.4
C7	24750	208564	1.14	3	1.07	0.9	207540	1.14	3	2.49	45.7
C8	3108	24604	1.14	2	0.14	0.2	23334	1.14	2	0.32	1.0
C10	14868	99124	1.17	3	0.53	0.6	97296	1.17	2	1.61	14.8
Snort24	13886	44883	1.56	4	0.35	0.2	38601	1.57	4	2.67	19.9
Snort31	20068	94339	1.97	6	0.86	2.6	70780	2.17	8	15.61	59.1
Snort34	13825	45642	1.38	5	0.28	1.8	40387	1.42	8	2.60	14.2

TABLE III
COMPARING D^{2}FAMergeOld, D^{2}FAMERGE, AND D^{2}FAMergeOpt TO ORIGINAL

RE set group	D^{2}FAMergeOld					D^{2}FAMERGE					D^{2}FAMergeOpt				
	Trans increase	Def. depth ratio		Space ratio	Time ratio	Trans increase	Def. depth ratio		Space ratio	Time ratio	Trans increase	Def. depth ratio		Space ratio	Time ratio
		Avg.	Max.				Avg.	Max.				Avg.	Max.		
All	20.1%	6.5	4.4	1388.6	154.5	10.8%	6.4	4.4	1499.8	154.5	0.4%	6.0	3.7	113.1	9.4
STRING	44.0%	2.5	2.0	2667.0	35.4	21.5%	2.4	2.0	2994.8	35.4	0.0%	2.2	1.6	103.5	0.8
WILDCARD	3.0%	12.1	8.8	42.7	246.6	1.0%	12.1	8.8	42.8	246.6	1.0%	12.1	10.0	16.8	10.8
SNORT	21.3%	6.3	4.0	1882.3	141.8	13.3%	6.3	4.0	1960.3	141.8	0.0%	6.0	3.0	215.8	13.7

than ORIGINAL. This time difference is maximized when the deferment chains are shortest. For example, D^{2}FAMERGE only requires an average of 0.05 and 0.09 ms per state for the WILDCARD and SNORT RE sets, respectively, so D^{2}FAMERGE is, on average, 247 and 142 times faster than ORIGINAL for these RE sets, respectively. For the STRING RE sets, the deferment chains are longer, so D^{2}FAMERGE requires an average of 0.67 ms per state, and is, on average, 35 times faster than ORIGINAL.

3) D^{2}FAMERGE Produces D^{2}FA With Much Smaller Average and Maximum Deferment Depths Than ORIGINAL: On average, D^{2}FAMERGE produces D^{2}FA that have average deferment depths that are 6.4 times smaller than ORIGINAL and maximum deferment depths that are 4.4 times smaller than ORIGINAL. In particular, the average deferment depth for D^{2}FAMERGE is less than 2 for all but the two STRING RE sets, where the average deferment depths are 2.15 and 2.69. Thus, the expected number of deferment transitions to be traversed when processing a length n string is less than n. One reason D^{2}FAMERGE works so well is that it eliminates low weight edges from the SRG so that the deferment forest has many shallow deferment trees instead of one deep tree. This is particularly effective for the WILDCARD RE sets and, to a lesser extent, the SNORT RE sets. For the STRING RE sets, the SRG is fairly dense, so D^{2}FAMERGE has a smaller advantage relative to ORIGINAL.

4) D^{2}FAMERGE Produces D^{2}FA With Only Slightly More Transitions Than ORIGINAL, Particularly on the RE Sets That Need Transition Compression the Most: On average, D^{2}FAMERGE produces D^{2}FA with roughly 11% more transitions than ORIGINAL does. D^{2}FAMERGE works best when state explosion from wildcard closures creates DFA composed of many similar repeating substructures. This is precisely when transition compression is most needed. For example, for the WILDCARD RE sets that experience the greatest state explosion, D^{2}FAMERGE only has 1% more transitions than ORIGINAL. On the other hand, for the STRING RE sets, D^{2}FAMERGE has, on average, 22% more transitions. For this group, ORIGINAL needed to build a very large SRG and thus used much more space and time to achieve the improved transition compression. Furthermore, transition compression is typically not needed for such RE sets as all string matching REs can be placed into a single group and the resulting DFA can be built.

In summary, D^{2}FAMERGE achieves its best performance relative to ORIGINAL on the WILDCARD RE sets (except for space used for construction of the D^{2}FA) and its worst performance relative to ORIGINAL on the STRING RE sets (except for space used to construct the D^{2}FA). This is desirable as the space- and time-efficient D^{2}FAMERGE is most needed on RE sets like those in the WILDCARD because those RE sets experience the greatest state explosion.

TABLE IV
PERFORMANCE DATA FOR ORIGINAL AND D²FAMERGE GIVEN MAXIMUM DEFERMENT DEPTH BOUNDS OF 1, 2, AND 4

RE set	ORIGINAL						D²FAMERGE					
	# Trans			Avg. def. depth			# Trans			Avg. def. depth		
	$\Omega=1$	$\Omega=2$	$\Omega=4$	Ω=1	Ω=2	Ω=4	$\Omega=1$	$\Omega=2$	$\Omega=4$	Ω=1	Ω=2	Ω=4
Bro217	698229	296433	52628	0.62	1.18	2.09	50026	15087	11757	1.00	1.83	2.15
C613	1204831	507613	102183	0.62	1.17	2.16	154548	51858	27735	1.00	1.94	2.64
C7	2044171	597544	206814	0.71	1.24	2.07	215940	208044	207540	0.97	1.13	1.14
C8	206897	40411	23261	0.77	1.32	2.51	24090	23334	23334	0.98	1.14	1.14
C10	1105160	325536	97137	0.75	1.31	2.39	101556	97326	97296	0.98	1.18	1.18
Snort24	1376779	543378	106211	0.66	1.25	2.39	68906	42176	39409	0.99	1.47	1.56
Snort31	2193679	1102693	405785	0.62	1.11	2.08	208136	119810	95496	1.00	1.52	1.97
Snort34	1357697	559255	85800	0.66	1.19	2.17	57187	44607	43231	1.00	1.34	1.38

5) Improvement of D²FAMERGE Over D²FAMergeOld: Using our improved algorithm to build the initial D²FAs results in significant reduction in the final size of the D²FA produced by the D²FA merge algorithm. On average, D²FAMergeOld produces a D²FA 8.2% larger than that produced by D²FAMERGE.

C. Assessment of Final Compression Algorithm

We now assess the effectiveness of our final compression algorithm by comparing D²FAMergeOpt to ORIGINAL and D²FAMERGE. As expected D²FAMergeOpt produces a D²FA that is almost as small as that produced by ORIGINAL; on average, the number of transitions increases by only 0.4%. There is a very small increase for WILDCARD and SNORT because ORIGINAL also considers all edges with weight >1 in the SRG, whereas D²FAMergeOpt does not use edges with weight <10. There is a significant benefit to not using these low-weight SRG edges; the deferment depths are much higher for the D²FA produced by ORIGINAL when compared to the D²FA produced by D²FAMergeOpt.

The final compression algorithm of D²FAMergeOpt does require more resources than are required by D²FAMERGE. In some cases, this may limit the size of the RE set for which D²FAMergeOpt can be used. However, as explained earlier, D²FAMERGE performs best on the WILDCARD (which has the most state explosion) and performs the worst on the STRING (which has the no or limited state explosion). Hence, the final compression algorithm is only needed for and is most beneficial for RE sets with limited state explosion. Finally, we observe that D²FAMergeOpt requires on average 113 times less RAM than ORIGINAL and, on average, runs 9 times faster than ORIGINAL.

D. D²FAMERGE Versus ORIGINAL With Bounded Maximum Deferment Depth

We now compare D²FAMERGE and ORIGINAL when they impose a maximum deferment depth bound Ω of 1, 2, and 4. Because time and space do not change significantly, we focus only on number of transitions and average deferment depth. These results are shown in Table IV. Note that for these data sets, the resulting maximum depth Ψ typically is identical to the maximum depth bound Ω; the only exception is for D²FAMERGE and $\Omega=4$, thus we omit the maximum deferment depth from Table IV. Table V summarizes the results by RE group highlighting how much better or worse D²FAMERGE does than ORIGINAL on the two metrics of number of transitions and average deferment depth $\overline{\psi}$.

TABLE V
COMPARING D²FAMERGE WITH ORIGINAL GIVEN MAXIMUM DEFERMENT DEPTH BOUNDS OF 1, 2, AND 4

RE set group	$\Omega=1$		$\Omega=2$		$\Omega=4$	
	Trans decr.	Avg. def. depth ratio	Trans decr.	Avg. def. depth ratio	Trans decr.	Avg. dptr len ratio
All	91.3%	0.7	79.4%	0.9	42.5%	1.5
STRING	90.0%	0.6	92.5%	0.6	75.5%	0.9
WILDCARD	89.3%	0.8	59.0%	1.1	0.0%	2.0
SNORT	94.0%	0.7	91.0%	0.8	63.0%	1.4

Overall, D²FAMERGE performs very well when presented a bound Ω. In particular, the average increase in the number of transitions for D²FAMERGE with Ω equal to 1, 2 and 4, is only 131%, 20%, and 1% respectively, compared to D²FAMERGE with unbounded maximum deferment depth. Stated another way, when D²FAMERGE is required to have a maximum deferment depth of 1, this only results in slightly more than twice the number of transitions in the resulting D²FA. The corresponding values for ORIGINAL are 3121%, 1216%, and 197%.

These results can be partially explained by examining the average deferment depth data. Unlike in the unbounded maximum deferment depth scenario, here we see that D²FAMERGE has a larger average deferment depth $\overline{\psi}$ than ORIGINAL except for the WILDCARD when Ω is 1 or 2. What this means is that D²FAMERGE has more states that defer to at least one other state than ORIGINAL does. This leads to the lower number of transitions in the final D²FA. Overall, for $\Omega=1$, D²FAMERGE produces D²FA with roughly 91% fewer transitions than ORIGINAL for all RE set groups. For $\Omega=2$, D²FAMERGE produces D²FA with roughly 59% fewer transitions than ORIGINAL for the WILDCARD RE sets and roughly 92% fewer transitions than ORIGINAL for the other RE sets.

E. D²FAMERGE Versus BACKPTR

We now compare D²FAMERGE to BACKPTR that enforces the back-pointer property described in Section V-C. We adapt D²FAMERGE to also enforce this back-pointer property. The results for all our metrics are shown in Section VI for our eight primary RE sets. We consider the two variants of BACKPTR described in Section VI-A.3, one that constructs the minimum-state DFA corresponding to the given NFA and one that bypasses the minimum-state DFA and goes directly to the D²FA from the given NFA. We note the second variant appears to

TABLE VI
PERFORMANCE DATA FOR BOTH VARIANTS OF BACKPTR AND D^{2}FAMERGE WITH THE BACK-POINTER PROPERTY

RE set	BACKPTR							D^{2}FAMERGE with back-pointer				
	# Trans	Def. depth		RAM	Time	RAM2	Time2	# Trans	Def. depth		RAM	Time
		Avg.	Max.	(MB)	(s)	(MB)	(s)		Avg.	Max.	(MB)	(s)
Bro217	11247	2.61	6	6.38	88.08	0.05	273.95	13567	2.33	6	0.13	6.24
C613	26222	2.50	5	11.04	55.91	0.13	971.45	33777	2.30	5	0.25	10.78
C7	217812	5.94	13	24.17	277.80	1.04	1950.00	219684	1.15	4	1.12	4.51
C8	34636	2.44	8	3.04	12.61	0.17	27.76	35476	1.20	4	0.19	0.69
C10	157139	2.13	7	14.52	96.86	0.75	476.54	158232	1.21	4	0.80	11.94
Snort24	46005	8.74	17	13.56	70.95	0.22	1130.00	58273	1.62	8	0.41	47.77
Snort31	82809	2.87	8	19.60	109.56	0.39	1110.00	124584	1.74	6	1.29	3.61
Snort34	46046	7.05	14	13.50	94.19	0.22	983.98	51557	1.42	5	0.30	6.06

TABLE VII
COMPARING D^{2}FAMERGE WITH BOTH VARIANTS OF BACKPTR

RE set group	Trans increase	Def. depth ratio		Space ratio	Time ratio	Space2 ratio	Time2 ratio
		Avg.	Max.				
All	17.9%	2.9	1.9	30.4	19.3	0.7	142.5
STRING	25.0%	1.1	1.0	47.3	9.7	0.5	67.0
WILDCARD	1.3%	3.0	2.3	18.5	29.3	0.9	170.8
SNORT	29.7%	4.0	2.1	31.1	15.8	0.5	164.5

use less space than D^{2}FAMERGE. This is partially true since BACKPTR creates a smaller D^{2}FA than D^{2}FAMERGE. However, we underestimate the actual space used by this BACKPTR variant by simply assuming its required space is the final D^{2}FA size. We ignore, for instance, the space required to store intermediate complete tables or to perform the NFA to DFA subset construction. Tables VI and VII summarize these results by RE group displaying ratios for many of our metrics that highlight how much better or worse D^{2}FAMERGE does than BACKPTR.

Similar to D^{2}FAMERGE versus ORIGINAL, we find that D^{2}FAMERGE with the back-pointer property performs well when compared to both variants of BACKPTR. Specifically, with an average increase in the number of transitions of roughly 18%, D^{2}FAMERGE runs on average 19 times faster than the fast variant of BACKPTR and 143 times faster than the slow variant of BACKPTR. For space, D^{2}FAMERGE uses on average almost 30 times less space than the first variant of BACKPTR and on average roughly 42% more space than the second variant of BACKPTR. Furthermore, D^{2}FAMERGE creates D^{2}FA with average deferment depth 2.9 times smaller than BACKPTR and maximum deferment depth 1.9 times smaller than BACKPTR. As was the case with ORIGINAL, D^{2}FAMERGE achieves its best performance relative to BACKPTR on the WILDCARD RE sets and its worst performance relative to BACKPTR on the STRING RE sets. This is desirable as the space- and time-efficient D^{2}FAMERGE is most needed on RE sets like those in the WILDCARD because those RE sets experience the greatest state explosion.

F. Scalability Results

Finally, we assess the improved scalability of D^{2}FAMERGE relative to ORIGINAL using the Scale RE set assuming that we have a maximum memory size of 1 GB. For both ORIGINAL and D^{2}FAMERGE, we add one RE at a time from Scale until the space estimate to build the D^{2}FA goes over the 1-GB limit. For ORIGINAL, we are able only able to add 12 REs; the final D^{2}FA has 397 312 states and requires over 71 h to compute. As explained earlier, we include the SRG edges in the RAM size estimate. If we exclude the SRG edges and only include the DFA size in the RAM size estimate, we would only be able to add one more RE before we reach the 1-GB limit. For D^{2}FAMERGE, we are able to add 19 REs; the final D^{2}FA has 80 216 064 states and requires only 77 min to compute. This data set highlights the quadratic versus linear running time of ORIGINAL and D^{2}FAMERGE, respectively. Fig. 3 shows how the space and time requirements grow for ORIGINAL and D^{2}FAMERGE as REs from Scale are added one by one until 19 have been added.

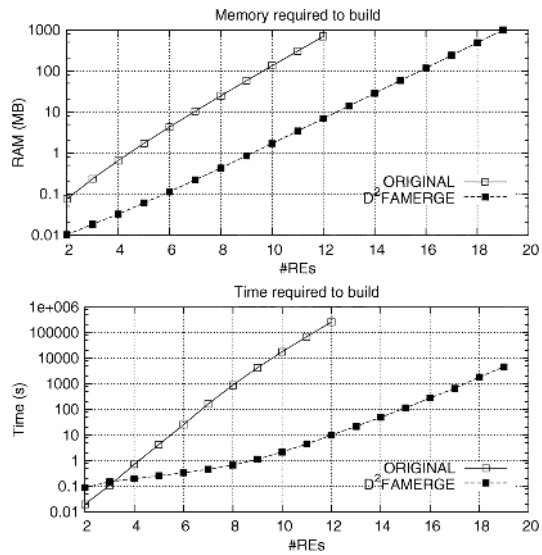


Fig. 3. Memory and time for ORIGINAL's D^{2}FA and D^{2}FAMERGE's D^{2}FA.

VII. CONCLUSION

In this paper, we propose a novel Minimize then Union framework for constructing D^{2}FAs using D^{2}FA merging. Our approach requires a fraction of memory and time compared to current algorithms. This allows us to build much larger D^{2}FAs than was possible with previous techniques. Our algorithm naturally supports frequent RE set updates. We conducted experiments on real-world and synthetic RE sets that verify our claims. For example, our algorithm requires an average of 1500 times less memory and 150 times less time than the original D^{2}FA construction algorithm of Kumar *et al.*. We also provide an optimization post-processing step that produces D^{2}FAs that are essentially as good as those produced by the original D^{2}FA construction algorithm; the optimization step requires on average 113 times less memory and 9 times less time than the original D^{2}FA construction algorithm.

REFERENCES

[1] J. Patel, A. Liu, and E. Torng, "Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems," in *Proc. 19th Annu. NDSS*, San Diego, CA, USA, Feb. 2012.

[2] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proc. 13th USENIX LISA*, Nov. 1999, pp. 229–238.

[3] "Snort," [Online]. Available: http://www.snort.org/

[4] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23–24, pp. 2435–2463, 1999.

[5] R. Sommer and V. Paxson, "Enhancing byte-level network intrusion detection signatures with context," in *Proc. ACM Conf. Comput. Commun. Sec.*, 2003, pp. 262–271.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA, USA: Addison-Wesley, 2000.

[7] J. E. Hopcroft, "An nlogn algorithm for minimizing the states in a finite automaton," in *The Theory of Machines and Computations*. New York, NY, USA: Academic Press, 1971, pp. 189–196.

[8] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc. SIGCOMM*, 2006, pp. 339–350.

[9] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. IEEE/ACM ANCS*, 2006, pp. 81–92.

[10] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proc. 19th USENIX Security Symp.*, Washington, DC, USA, Aug. 2010.

[11] S. Kumar, B. Chandrasekaran, J. Turner, and G. Varghese, "Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia," in *Proc. ACM/IEEE ANCS*, 2007, pp. 155–164.

[12] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc. IEEE Symp. Security Privacy*, 2008, pp. 187–201.

[13] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata," in *Proc. SIGCOMM*, 2008, pp. 207–218.

[14] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. CoNext*, 2007, Art. no. 1.

[15] M. Becchi and P. Crowley, "Extending finite automata to efficiently match perl-compatible regular expressions," in *Proc. ACM CoNEXT*, 2008, Art. no. 25.

[16] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM/IEEE ANCS*, 2007, pp. 145–154.

[17] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[18] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, 2004, pp. 333–340.

[19] I. Sourdis and D. Pnevmatikatos, "Pnevmatikatos: Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion detection system," in *Proc. Int. Field Program. Logic Appl.*, 2003, pp. 880–889.

[20] L. Tan and T. Sherwood, "A high throughput string matching architecture for intrusion detection and prevention," in *Proc. 32nd Annu. ISCA*, 2005, pp. 112–122.

[21] F. Yu, R. H. Katz, and T. V. Lakshman, "Gigabit rate packet pattern-matching using TCAM," in *Proc. 12th IEEE ICNP*, 2004, pp. 174–183.

[22] J.-S. Sung, S.-M. Kang, Y. Lee, T.-G. Kwon, and B.-T. Kim, "A multi-gigabit rate deep packet inspection algorithm using TCAM," in *Proc. IEEE GLOBECOM*, 2005, pp. 453–457.

[23] M. Alicherry, M. Muthuprasanna, and V. Kumar, "High speed pattern matching for network ids/ips," in *Proc. IEEE Int. Conf. Netw. Protocols*, 2006, pp. 187–196.

[24] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 191–202, 2006.

[25] M. Becchi and P. Crowley, "Efficient regular expression evaluation: Theory to practice," in *Proc. ACM/IEEE ANCS*, 2008, pp. 50–59.

[26] S. Kong, R. Smith, and C. Estan, "Efficient signature matching with multiple alphabet compression tables," in *Proc. 4th SecureComm*, 2008, p. 1.

[27] M. Becchi and S. Cadambi, "Memory-efficient regular expression search using state merging," in *Proc. IEEE INFOCOM*, 2007, pp. 1064–1072.

[28] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM/IEEE ANCS*, 2006, pp. 93–102.

[29] L. Yang, R. Karim, V. Ganapathy, and R. Smith, "Fast, memory-efficient regular expression matching with NFA-OBDDs," *Comput. Netw.*, vol. 55, no. 55, pp. 3376–3393, 2011.

[30] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *Proc. IEEE FCCM*, 2001, pp. 227–238.

[31] C. R. Clark and D. E. Schimmel, "Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns," in *Proc. Field-Program. Logic Appl.*, 2003, pp. 956–959.

[32] C. R. Clark and D. E. Schimmel, "Scalable pattern matching for high speed networks," in *Proc. 12th Annu. IEEE FCCM*, Washington, DC, USA, 2004.

[33] I. Sourdis and D. Pnevmatikatos, "Pre-decoded cams for efficient and high-speed NIDs pattern matching," in *Proc. Field-Program. Custom Comput. Mach.*, 2004, pp. 258–267.

[34] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for accelerating SNORT IDS," in *Proc. ACM/IEEE ANCS*, 2007, pp. 127–136.

[35] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos, "Implementation of a content-scanning module for an internet firewall," in *Proc. IEEE Field Program. Custom Comput. Mach.*, 2003, pp. 31–38.

[36] A. Bremler-Barr, D. Hay, and Y. Koral, "CompactDFA: Generic state machine compression for scalable pattern matching," in *Proc. IEEE INFOCOM*, 2010, pp. 1–9.

[37] Y. Liu, L. Guo, M. Guo, and P. Liu, "Accelerating DFA construction by hierarchical merging," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. Appl.*, 2011, pp. 1–6.

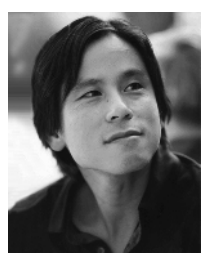
Jignesh Patel is currently pursuing the Ph.D. degree in computer science and engineering at Michigan State University, East Lansing, MI, USA.

His research interests include algorithms, networking, and security.

Alex X. Liu received the Ph.D. degree in computer science from The University of Texas at Austin, Austin, TX, USA, in 2006.

He is currently an Associate Professor with the Department of Computer Science and Engineering, Michigan State University, East Lansing, USA. His research interests focus on networking and security.

Dr. Liu is an Associate Editor of the IEEE/ACM TRANSACTIONS ON NETWORKING. He is the TPC Co-Chair of ICNP 2014. He received the IEEE & IFIP William C. Carter Award in 2004, an NSF CAREER Award in 2009, and the Michigan State University Withrow Distinguished Scholar Award in 2011. He received Best Paper Awards from ICNP 2012, SRDS 2012, LISA 2010, and TSP 2009.



Eric Torng received the Ph.D. degree in computer science from Stanford University, Stanford, CA, USA, in 1994.

He is currently an Associate Professor and Graduate Director with the Department of Computer Science and Engineering, Michigan State University, East Lansing, MI, USA. His research interests include algorithms, scheduling, and networking.

Dr. Torng received an NSF CAREER Award in 1997.