An Overlay Automata Approach to Regular Expression Matching

Alex X. Liu Eric Torng
Michigan State University
East Lansing, Michigan 48824-1266, U.S.A.
Email: {alexliu, torng}@cse.msu.edu

***Abstract*—Regular expression (RegEx) matching, the core operation of intrusion detection and prevention systems, remains a fundamentally challenging problem. A desired RegEx matching scheme should satisfy four requirements: DFA speed, NFA size, automated construction, and scalable construction. Despite lots of work on RegEx matching, no prior scheme satisfies all four of these requirements. In this paper, we approach this holy grail by proposing OverlayCAM, a RegEx matching scheme that satisfies all four requirements. The theoretical underpinning of our scheme is OD^{2}FA, a new automata model proposed in this paper that captures both state and transition replication inherent in DFAs. Our RegEx matching solution processes one input character per lookup like a DFA, requires only the space of an NFA, is grounded in sound automata models, is easy to deploy in existing network devices, and comes with scalable and automated construction algorithms.**

I. Introduction

A. Background and Problem Statement

Deep packet inspection (DPI) is the core operation for a variety of devices, such as routers, Network Intrusion Detection (or Prevention) Systems (NIDS/NIPS), firewalls, and layer 7 switches, for a variety of services, such as malware filtering, attack detection, traffic monitoring, and application protocol identification. DPI is usually achieved by regular expression (RegEx) matching, *i.e.*, finding which RegExes in a set of predefined RegExes match the packet payload, because RegExes are expressive, efficient, and flexible for specifying attack or malware signatures [22]. Most open source and commercial NIDS/NIPS such as Snort, Bro, and HP TippingPoint use RegEx matching to implement DPI. Modern operating systems such as Cisco IOS and Linux have RegEx matching modules for layer 7 filtering as well.

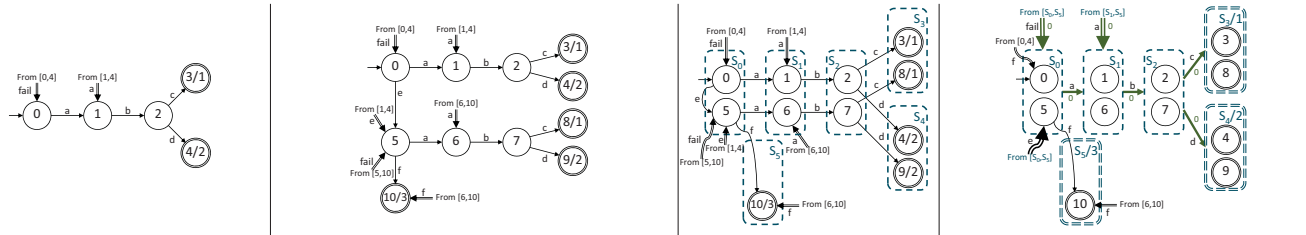
There are two standard automata models for implementing RegEx matching, Deterministic Finite State Automata (DFA) and Nondeterministic Finite State Automata (NFA). Each has its own advantage and corresponding disadvantage. The DFA advantage is that it maintains a single active state and thus processes one input character per lookup whereas the NFA maintains multiple active states and thus requires multiple lookups (one per active state) for each input character. The NFA advantage is that the number of NFA states and transitions is linear in the size and number of the RegExes whereas the number of DFA states and transitions can be exponential in the size and number of the RegExes due to the well known *state explosion* that results from a single NFA state being replicated many times.

A desired RegEx matching scheme should satisfy the following four requirements: (1) DFA Speed: Matching speed should be one memory lookup per character. This enables networking and security devices to process packets at line speed. (2) NFA Size: Memory size should be polynomial in the RegEx set size. For large RegEx sets, this enables storage of the corresponding automata in SRAM rather than DRAM, which is hundreds of times slower than SRAM. (3) Automated Construction: The construction of its memory image should be automated. This enables a RegEx matching scheme to be easily deployed in practice. (4) Scalable Construction: The automated construction algorithm should be scalable, *i.e.*, the required memory should be polynomial in the RegEx set size. This enables a RegEx matching scheme to be applied to large RegEx sets rather than just small RegEx sets.

B. Limitations of Prior Art

Although many RegEx matching schemes have been proposed, unfortunately, none of them satisfy all four of these requirements. Two schemes that come closest to satisfying the four requirements are XFA proposed by Smith *et al.* [20], [21] and Peng *et al.*'s Ternary Content Addressable Memory (TCAM) based RegEx matching scheme [18]. XFA reduces the number of states by augmenting each DFA state with a program that is executed upon reaching the state. XFA satisfies the NFA size requirement as it addresses state explosion and arguably satisfies the DFA speed requirement, although extra code must be fetched and executed along with each lookup. However, XFA does not satisfy the automated construction requirement as XFA construction requires a human expert to annotate the given RegEx set [24]. Furthermore, it is hard to implement XFA in ASIC although ASIC implementation is critical for such software based RegEx matching schemes to achieve high speed. A fundamental reason is that the ASIC implementation of XFA would require much of the complexity of a general purpose CPU to implement the programs associated with each state. Moreover, because each XFA state's program may take a very different amount of time to process, it may be difficult to pipeline the RegEx matching processing. Peng *et al.*'s TCAM-based scheme satisfies the DFA speed and NFA size requirements on their RegEx sets; however, although automated, their construction algorithm is not scalable because

978-14799-3360-0/14/$31.00 ©2014 IEEE



(a) DFA for {/abc/, /abd/} (b) DFA for {/abc/, /abd/, /e.*f/} (c) Corresponding ODFA (d) ODFA with super-state transitions

Fig. 1. Example DFA, state replication, and Overlay DFA.

it first must generate a DFA from the given RegEx set. Because this DFA may be exponentially larger than the corresponding NFA, their algorithms cannot be applied to RegEx sets that correspond to large DFAs even though their final TCAM implementation may be relatively small. This limits the size of the RegEx set that their scheme can practically handle.

C. Proposed Overlay Automata Approach

To address the limitations of prior DFA based automata, we propose an overlay automata approach. First, we propose *Overlay Deterministic Finite State Automata (ODFA)* that models state replication in DFAs. The basic idea is to overlay all the DFA states that are replicas of the same NFA state vertically together into what we call a *super-state*. Fig. 1 depicts the DFA and ODFA for the RegEx set {/abc/, /abd/, /e.*f/}. The ODFA model gives us the following key benefits. *First, we can compactly reference all replicas of the same NFA state using super-states.* In Fig. 1, we merge states 0 and 5 and states 1 and 6 into super-states S_0 and S_1, respectively. *Second, we can compactly represent replicas of the same NFA transition by one super-state transition between two super-states.* In Fig. 1, we merge the two transitions from states 0 and 5 on character "a" into one super-state transition on character "a".

Second, combining our overlay idea, which models state and transition replication, and the delayed input idea in D^{2}FA [12], which models sharing non-replicated transitions among non-replicated DFA states through a state deferment relationship, we propose *Overlay Delayed Input DFA (OD^{2}FA)* to model state replication, transition replication, and transition sharing. The relationship among these automata models, DFA, D^{2}FA, ODFA, and OD^{2}FA, is illustrated in Fig. 2. *OD^{2}FA provides us with the automata model that satisfies the DFA speed and NFA size requirements for designing RegEx matching schemes.* A key benefit of OD^{2}FA is that we can represent the deferment relationship among D^{2}FA states more compactly using deferment among OD^{2}FA super-states. From the perspective of transitions, OD^{2}FA optimizes both deferred transitions (*i.e.*, common transitions among states) and replicated transitions.

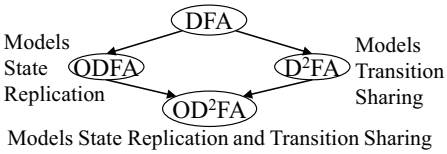


Fig. 2. Relationship of Automata Models

Third, we propose an *automated and scalable algorithm for constructing OD^{2}FA from a RegEx set* that will multiply the compression benefits of both ODFA and D^{2}FA. Our algorithm builds the OD^{2}FA M for RegEx set $R_1 \cup R_2$ by merging the OD^{2}FA M_1 for R_1 with the OD^{2}FA M_2 for R_2. That is, we first construct the equivalent OD^{2}FA for each RegEx, and then recursively merge OD^{2}FAs until only a single OD^{2}FA for the entire RegEx set is left. Our algorithm automatically identifies and groups together replicate states in M into super-states and replicate transitions into super-state transitions without having to perform an expensive analysis of the final DFA structure. The space required for constructing the final OD^{2}FA is essentially the size of the final OD^{2}FA.

Fourth, we propose *OverlayCAM*, an algorithm for implementing OD^{2}FA in TCAM. Because TCAMs are off-the-shelf chips and have been widely deployed in modern networking devices, OverlayCAM can be deployed in most current core networking and security devices without any architectural or hardware change. A bit in TCAM can have three values: 0, 1, or $*$. For a TCAM of w-bit width, where w is configurable, given a lookup key of w binary bits, the chip will compare the key with every TCAM entry in parallel and then report the index of the first TCAM entry that matches the key, where a $*$ can match both 0 and 1. This index is then used to retrieve the corresponding decision in the SRAM associated with the TCAM. Prior work has shown that TCAM-based RegEx matching significantly outperforms prior software or FPGA based RegEx matching schemes [14], [18]. The key issue in TCAM-based RegEx matching is to minimize TCAM space because TCAM chips have small capacities (maximum size of 72 megabits by current technology), consume lots of power, and generate lots of heat. Our idea is to encode the transitions in TCAM exploiting state replication, replicated transitions, and transition sharing simultaneously; in particular, representing transitions that end in different states using one TCAM entry. We propose a TCAM based OD^{2}FA implementation algorithm, OverlayCAM, that exploits the transition sharing relationship among the states within a super-state. OverlayCAM represents multiple transitions that end in different states within a super-state using offset decisions where the decision is the numerical difference between the source and destination state IDs. Based on OD^{2}FA, OverlayCAM generates fewer TCAM entries than NFA transitions by encoding multiple deferred transitions using one TCAM entry and encoding multiple non-deferred transitions that are replicas of the same NFA transition using only one TCAM entry.

D. Meeting RegEx Matching Requirements

Our RegEx matching scheme satisfies the aforementioned four RegEx matching requirements: DFA speed, NFA size, automated construction, and scalable construction. Our scheme has DFA speed because our deterministic automata consumes one input character per lookup. Our scheme has NFA size because we use one super-state to represent all replicas of the same NFA state and one super-state transition to represent all replicas of a transition among two NFA states. Our scheme has a fully automated and scalable construction algorithm, whose memory footprint is essentially the memory size of the final OD^{2}FA. In particular, our construction algorithm never generates the entire underlying DFA. On average, in our 8 real-world RegEx sets, the number of super-states per OD^{2}FA is 1.6 times the number of NFA states and is never more than 2.6 times the number of NFA states. Likewise, on average, the number of TCAM entries is 0.63 times the number of NFA transitions and is never more than 1.65 times the number of NFA transitions. *In summary, our RegEx matching solution processes one character per lookup like a DFA, requires only the space of an NFA, is grounded in automata models, is easy to deploy in existing network devices, and comes with scalable and automated construction algorithms.*

II. Related Work

There are two main approaches to developing RegEx matching solutions. The first one is to start with a DFA and compress it. Great work has been done in reducing the number of transitions stored per DFA state such as D^{2}FA [12], [13], [4], [2], [9], [17]. Some work has attempted to address state explosion. One approach is to partition RegExes into groups and then build an automata for each group [25], [3], [21]. This partition approach is orthogonal to our approach and can be combined with our approach to deal with extremely large RegEx sets. Another way is to use "scratch memory" to manage state replication and avoid state explosion [11], [20], [6], [21]. However, the size of the required scratch memory may itself be significant, and the processing required to update the scratch memory after each transition may be significant. Furthermore, many such approaches (such as [20], [21]) are not fully automated. The second approach is to start with an NFA and develop methods for coping with multiple active states. Much of the NFA work has exploited the parallel processing capabilities of FPGA technology to cope with the multiple active states that arise from NFA [19], [7], [16], [8], [23], [15], [3], [5]. The main limitation of this approach is that FPGAs are not commonly embedded in network processors as TCAMs commonly are.

Two TCAM-based RegEx matching schemes have recently been proposed: RegCAM [14] and Peng *et al.*'s scheme [18]. Compared with OverlayCAM, both methods do not satisfy the scalable construction requirement as both construction algorithms need to first convert the given RegEx set into a DFA. Furthermore, while Peng *et al.*'s scheme arguably satisfies the NFA size requirement based on their experimental results, RegCAM does not. RegCAM uses a table consolidation approach where k TCAM tables are combined together into one TCAM table. Table consolidation can only lead to a constant factor reduction in TCAM storage.

III. Overlay Automata

In this section, we formally define Overlay DFA (ODFA) and Overlay D^{2}FA (OD^{2}FA). Table I summarizes notations used in this paper.

Notation	Meaning
D	A DFA/D^{2}FA
$\mathcal{D}$	An ODFA/OD^{2}FA
Q	The set of states in a DFA/D^{2}FA/ODFA/OD^{2}FA
$\mathcal{S}$	The set of super-states in an ODFA/OD^{2}FA
$\mathcal{O}$	The set of overlays an ODFA/OD^{2}FA
s, q, u	A DFA/D^{2}FA/ODFA/OD^{2}FA state
S	An ODFA/OD^{2}FA super-state
O	An ODFA/OD^{2}FA overlay
X	A set of overlays in an ODFA/OD^{2}FA
M(s)	Set of RegExes accepted by state s
$\mathcal{M}(S)$	Set of RegExes accepted by all states in super-state S
$F(s)$	Deferred state of state s
$\mathcal{F}(S)$	Deferred super-state of super-state S
$F^{-1}(s)$	The set of states that defer to state s
$p \rightarrow q$	State p defers to state q
$p \twoheadrightarrow q$	State p descendant of state q
$\perp$	NULL state/empty location
$\rho(s, \sigma)$	Partial state transition function for a D^{2}FA
$\delta'(s, \sigma)$	Total transaction function derived from ρ
$\Delta(S, X, \sigma)$	Super-state transition function for a ODFA/OD^{2}FA
$\rho'(s, \sigma)$	Partial state transition function derived from Δ
$\delta''(s, \sigma)$	Total transaction function derived from Δ/ρ'

TABLE I
TABLE OF NOTATIONS.

A. Overlay DFA

There are two ideas behind ODFA. The first is to group all DFA states that are replications of the same NFA state into a single super-state. The second is to merge as many transitions from the replicate states within a super-state as possible. To define ODFA, we will use the concepts of super-states, overlays, and super-state transitions. We begin by informally defining ODFA and these concepts using the ODFA in Fig. 1 as a running example.

We first define some of our notation for the DFA in Figure 1(a) for the RegEx set {`/abc/`, `/abd/`}. To simplify the diagram, we condense many transitions that have a common destination state on common characters as follows. These transitions are denoted with double arrows with their character labels next to the double arrow. The source states for these transitions are denoted as "`From [x..y]`" which represents the set of states with state IDs in the range $[x..y]$. For example, we represent four transitions starting in states 1 through 4 that end in state 1 on character '`a`' using double arrows beneath "`From [1..4]`" and an '`a`' next to the double arrow. When the text next to a double arrow is "fail", this represents all character transitions not explicitly shown in the figure. For example, the "fail" transition in Figure 1(a) includes all transitions out of state 0 for characters that are not 'a'. Finally, in an accepting state, the number following the '/' represents the ID of the RegEx matched by that accepting state.

The DFA in Figure 1(b) shows the DFA after the RegEx `/e.*f/` is added. This DFA illustrates the potential for ODFA as the entire DFA for the RegEx set {`/abc/`, `/abd/`} is replicated twice. The corresponding ODFA is shown in Figure 1(c). In Figure 1(c), we overlay the two copies of the DFA for the RegEx set {`/abc/`, `/abd/`}) on top of each other. Each pair of replicated DFA states is a *super-state* in the ODFA. Each layer of states is called an *overlay*. The ODFA in Figure 1(c) has six super-states $S_0, \ldots, S_5$ and two overlays. Each overlay contains a subset of the states in the entire DFA; in Figure 1(c), the first overlay does not contain a state from super-state S_5.

We now introduce the concept of *super-state transitions*. One super-state transition represents multiple DFA transitions much as one super-state represents a group of DFA states. In a standard DFA transition, the source state is a DFA state. In a super-state transition, the source state is an ODFA super-state and represents transitions from all the replicated DFA states within the super-state. The destination state is usually an ODFA super-state but can sometimes be a DFA state. The two super-state transition forms are $S_1 \xrightarrow{\sigma} S_2, o, 1$ and $S_1 \xrightarrow{\sigma} S_2, O, 0$ (distinguished by the last bit value 1/0). In the first form, the semantics are that each DFA state q in super-state S_1 transitions on character σ to a DFA state q' in super-state S_2, with $o = (\text{overlay of } q' - \text{overlay of } q) \bmod \#\text{overlays}$. The value of o is usually 0. In the second form, the semantics are that each DFA state q in super-state S_1 transitions on character σ to the DFA state located in super-state S_2 at overlay O. For example, consider the two DFA transitions $1 \xrightarrow{b} 2$ and $6 \xrightarrow{b} 7$ in Figure 1(c). These two transitions can be represented by one super-state transition $S_1 \xrightarrow{b} S_2, 0, 1$; the 0 denotes no change in overlay. As a second example, consider the two DFA transitions $3 \xrightarrow{e} 5$ and $8 \xrightarrow{e} 5$ in Figure 1(c). These two transitions can be represented by one super-state transition $S_3 \xrightarrow{e} S_0, 1, 0$.

In the ideal case, all DFA transitions can be replaced by super-state transitions which reduces the total number of transitions by the number of overlays in the ODFA. In some cases, not all states in a super-state have transitions that can be merged. We generalize super-state transitions to allow super-state transitions to be defined for a specific set of overlays X within a given super-state. Technically, traditional transitions from a single state s are super-state transitions where X contains only s's overlay. We refer to these as *singleton super-state transitions*.

Figure 1(d) shows the ODFA for our running example with non-singleton super-state transitions denoted with thick edges. For example, the two transitions $0 \xrightarrow{a} 1$ and $5 \xrightarrow{a} 6$ from Figure 1(c) are represented with one super-state transition $S_0 \xrightarrow{a} S_1, 0, 1$. For super-state transitions of the form $S_1 \xrightarrow{\sigma} S_2, o, 1$ (*i.e.* destination is also a super-state), the number besides the thick edge gives the change in overlay value o. As we use double arrows to represent multiple transitions, we use thick double arrows to represent multiple non-singleton super-state transitions. For example, the two transitions $0 \xrightarrow{e} 5$ and $5 \xrightarrow{e} 5$ from Figure 1(c) are included in one super-state transition $S_0 \xrightarrow{e} S_0, 1, 0$ which is part of the thick double arrow labeled with "e" ending at state 5. The DFA in Figure 1(b) has $11 \times 256 = 2816$ total transitions; the ODFA in Figure 1(d) has 1542 total super-state transitions which is close to the best possible result of $2816/2 = 1408$ total super-state transitions; only a few of these transitions are singleton super-state transitions.

While we have defined super-state transitions where the destination state is a super-state, practical implementation is challenging because each DFA transition represented by such a super-state transition has a different destination DFA state. We describe in Section V how we can represent such super-state transitions using one TCAM entry.

We now review the formal definition of DFA and then formally define ODFA. Given a set of RegExes $\mathcal{R}$, the corresponding DFA is a 5-tuple $(Q, \Sigma, q_0, M, \delta)$ where Q is a set of states, Σ is an alphabet, $q_0 \in Q$ is the starting state, $M\colon Q \to 2^{\mathcal{R}}$ gives the subset of RegExes accepted by each state, and $\delta\colon Q \times \Sigma \to Q$ is the transition function. Note that in a traditional DFA definition, rather than M, each state is simply an accepting or rejecting state. The language accepted by the DFA would simply be $\cup_{r \in \mathcal{R}} L(r)$. However, in security settings where each regular expression corresponds to a unique threat, the system must know which regular expressions have been matched. Thus, M stores the subset of RegExes matched when each state is reached, and the language of strings accepted by each state q is $\cup_{r \in M(q)} L(r)$. For example, in Figure 1(b), the language of strings accepted by state 3 are those that end in `/abc/` which corresponds to RegEx 1 and the language of strings accepted by state 10 are those that end in `/e.*f/` which corresponds to RegEx 3.

ODFA are formally defined as follows.

Definition 1 (Overlay DFA (ODFA)): An Overlay DFA (ODFA) for a set of RegExes $\mathcal{R}$ is defined as a 7-tuple $(Q, \Sigma, q_0, \mathcal{S}, \mathcal{O}, \mathcal{M}, \Delta)$. The first three terms are the same as those in the above DFA definition.

The next two terms define the overlay structure on top of a DFA: $\mathcal{S} = \{S_1, \ldots, S_{|\mathcal{S}|}\}$ is a set of super-states that partitions Q while $\mathcal{O} = \{O_1, \ldots, O_{|\mathcal{O}|}\}$ is a set of overlays that also partitions Q. We shall treat each overlay as a unique number in Δ. We overload notation and define $\mathcal{S}\colon Q \to \mathcal{S}$ and $\mathcal{O}\colon Q \to \mathcal{O}$ as functions mapping states to super-states and overlays, respectively. For any two states $s_i \neq s_j$, it must be the case that $(\mathcal{S}(s_i), \mathcal{O}(s_i)) \neq (\mathcal{S}(s_j), \mathcal{O}(s_j))$. For any super-state S and overlay O, $S \cap O$ is either empty or contains one state $s \in Q$.

The term $\mathcal{M}\colon \mathcal{S} \to 2^{\mathcal{R}}$ gives the subset of RegExes matched by any state within the given super-state. Of course, $\mathcal{M}$ is only correctly defined assuming Δ is correctly defined too. The final term $\Delta\colon \mathcal{S} \times 2^{\mathcal{O}} \times \Sigma \to \mathcal{S} \times [0..|\mathcal{O}|) \times \{0, 1\}$ defines the super-state transition function. For any $s \in Q$ and any $\sigma \in \Sigma$, all the transition $(\mathcal{S}(s), X, \sigma) \in \Delta$ with $\mathcal{O}(s) \in X$ have the same value; *i.e.* if we have two transitions $(\mathcal{S}(s), X, \sigma) \in \Delta$ and $(\mathcal{S}(s), Y, \sigma) \in \Delta$, with $\mathcal{O}(s) \in X \cap Y$, then we must have $\Delta(\mathcal{S}(s), X, \sigma) = \Delta(\mathcal{S}(s), Y, \sigma)$. We define $\delta''(s, \sigma)$ based on

this unique transition value, say (S', o, b) as follows. First, if $b = 0$, we call the transition a *non-offset transition*, and $\delta''(s, \sigma) = S' \cap o$. Otherwise ($b = 1$), we call the transition an *offset transition*, and $\delta''(s, \sigma) = S' \cap ((\mathcal{O}(s) + o) \bmod |\mathcal{O}|)$. In this definition, we treat overlays as integers. It must be the case that overlay $(\mathcal{O}(s) + o) \bmod |\mathcal{O}|$ does intersect S'. Normally for offset transitions $o = 0$, so the resulting overlay is just $\mathcal{O}(s)$.

Even though an ODFA has super-states and overlays, an ODFA processes an input string much like a DFA does. That is, the ODFA is always in a unique state and each character processed moves the ODFA to a potentially new state. The main difference is that the ODFA hopefully compresses multiple DFA transitions into a single ODFA super-state transition, and the RegEx matching information is stored at the super-state level rather than at the state level. For example, given the ODFA in Figure 1(d) and the input string `abea`, the ODFA begins in state 0. After processing character `a`, the ODFA moves to state 1. After processing character `b`, the ODFA moves to state 2. After processing character `e`, the ODFA moves to state 5. Finally, after processing character `a`, the ODFA moves to state 6. The first and fourth transitions are actually the same super-state transition. The third transition corresponds to the first form of super-state transition with specified destination state 5. In all cases, $\mathcal{M}(\mathcal{S}(s')) = \emptyset$, so no RegEx is ever matched.

Due to space limitations, we omit algorithms for constructing an ODFA from a given set of regular expressions. These algorithms are subsumed by our construction algorithms for OD^{2}FA in Section IV.

Overlays and super-states are two orthogonal partitionings of states in Q; intuitively, super-states partition Q vertically and overlays partition Q horizontally. There exist many possible ways to partition the states of a DFA into super-states and overlays. The benefits of an ODFA are only realized by a careful partitioning; for example, grouping replicate states of the same NFA state together in a super-state. Some super-states may not have DFA states in each overlay. In Figure 1(d), super-state S_5 contains only one DFA state 10 which belongs to the second overlay.

The compressive power of a super-state transition increases with the number of overlays that it includes. In the best case, all overlays are included in a super-state transition. In Figure 1(d), most super-state transitions include all overlays; there are only a few singleton super-state transitions. In more complex ODFA, there may be cases where a given super-state transition includes more than one overlay but not all overlays.

We could generalize the matching definition of ODFA to allow different states within a super-state to match different RegExes where the set of RegExes matched in state s is defined by $M(s) \cup \mathcal{M}(\mathcal{S}(s))$. However, in practice, this is rarely needed. It is also impractical if each state truly requires its own set of matched RegExes, given state explosion. Thus, ODFA must satisfy the following Condition (C1).

$$\forall S \in \mathcal{S}, \forall s_1, s_2 \in S,\ M(s_1) = M(s_2), \qquad \text{(C1)}$$

B. Overlay D^{2}FA

ODFAs address state explosion and D^{2}FAs address transition explosion. We propose overlay D^{2}FAs to address both state and transition explosion in DFAs.

We briefly review some highlights of D^{2}FA and refer the reader to Kumar *et al.*'s paper [12] for more details.

D^{2}FA use default transitions to compactly represent many common transitions between states in a DFA transition function δ. For example, consider two DFA states s_1 and s_2 where $\delta(s_1, \sigma) = \delta(s_2, \sigma)$ for all characters $\sigma \in C \subseteq \Sigma$. The DFA requires $|\Sigma|$ transitions for both s_1 and s_2; the D^{2}FA eliminates $\delta(s_2, \sigma)$ for all $\sigma \in C$ by adding a default transition from s_2 to s_1. If the D^{2}FA is in state s_2 and receives a character $\sigma \in C$, the D^{2}FA follows the default transition and changes to s_1 without consuming σ; the D^{2}FA will then process σ correctly because $\delta(s_1, \sigma) = \delta(s_2, \sigma)$. In this scenario, we say that s_2 *defers* to s_1 and the default transition from s_2 to s_1 is called a *deferment transition* (or edge). In many cases, almost every state in a D^{2}FA can eliminate all but one or two character transitions. For the above example, the D^{2}FA eliminates $|C|$ transitions at the cost of adding one deferment transition. In software implementations of D^{2}FA [12], there is a time penalty as each deferment transition taken does not advance the processing of the input. In TCAM implementations of D^{2}FA [14], there is no time penalty because of the first match functionality of TCAMs. We explain this more in Section V.

Given a DFA $D{=}(Q, \Sigma, q_0, M, \delta)$, its corresponding D^{2}FA D' is defined as a 6-tuple $(Q, \Sigma, q_0, M, \rho, F)$ where the combination of deferred state function $F\colon Q \to Q$ and partial function $\rho\colon Q \times \Sigma \to Q$ is equivalent to DFA transition function δ. To make F a complete function, for a state s that does not defer to any other state, we have s defer to itself by setting $F(s) = s$. The deferment relationship among states defined by F forms a *deferment forest*. A D^{2}FA is well defined if and only if there are no cycles other than self-loops in the deferment forest. The roots of the deferment trees in the forest are those states that defer to themselves. We use $q{\rightarrow}s$ to denote $F(q) = s$, *i.e.* q directly defers to s. We use $q{\twoheadrightarrow}s$ to denote that there is a path from q to s in the deferment forest defined by F. We now describe how F and ρ combine to define δ. Let $dom(\rho)$ denote the domain of partial function ρ, *i.e.* the values for which ρ is defined. The total transition function for a D^{2}FA is defined as

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

To ensure $\delta'(s, \sigma)$ is appropriately defined for all $s \in Q$ and $\sigma \in \Sigma$, the following conditions must be satisfied. For any $\langle s, \sigma \rangle \in dom(\rho)$, $\rho(s, \sigma) = \delta(s, \sigma)$. Furthermore, $\forall \langle s, \sigma \rangle \in Q \times \Sigma, \langle s, \sigma \rangle \in dom(\rho)$ if $(F(s) = s \vee \delta(s, \sigma) \neq \delta(F(s), \sigma))$.

Next we formally define the OD^{2}FA.

Definition 2 (Overlay D^{2}FA (OD^{2}FA)): We define an OD^{2}FA as an 8-tuple $(Q, \Sigma, q_0, \mathcal{F}, \mathcal{S}, \mathcal{O}, \mathcal{M}, \Delta)$, where the first three terms are same as in defining D^{2}FA, and the last four terms are the same as in defining ODFA. The only difference is that, we derive a partial transition function

$\rho'\colon Q \times \Sigma \to Q$ from Δ. Since ρ' is a partial function, we do not require the existence of a transition for each (s, σ) in Δ. $\mathcal{F}\colon \mathcal{S} \to \mathcal{S}$ is the super-state deferment function, and gives the deferred super-state for each super-state. We define the D^{2}FA state deferment function F from $\mathcal{F}$ as $F(s) = \mathcal{F}(\mathcal{S}(s)) \cap \mathcal{O}(s))$. To ensure this is a valid deferment function, $\mathcal{F}$ must satisfy the following two conditions. First,

$$\forall s \in Q, \mathcal{F}(\mathcal{S}(s)) \cap \mathcal{O}(s)) \neq \perp, \qquad \text{(C2)}$$

Second, the deferment forest of super-states defined by $\mathcal{F}$ has no cycles other than self-loops. Finally, ρ' and F define a total transition function δ'' as follows.

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

We say that $\langle s, \sigma\rangle \in dom(\rho')$ if there exists a transition $(\mathcal{S}(s), X, \sigma) \in \Delta$ with $\mathcal{O}(s) \in X$. If $\langle s, \sigma\rangle \in dom(\rho')$, then $\rho'(s, \sigma)$ is defined as δ'' is defined for ODFA.

We say that super-state S *overlay covers* super-state S' if $\forall O \in \mathcal{O}, (S \cap O = \perp) \to (S' \cap O = \perp)$. That is, every overlay that is empty in S is also empty in S'. Then, Condition (C2) says that for every super-state S, super-state $\mathcal{F}(S)$ overlay covers S.

The transition function δ'' is computed by finding the unique transition $(\mathcal{S}(s), X, \sigma) \in \Delta$ with $\mathcal{O}(s) \in X$ if such a transition exists. If not, the OD^{2}FA follows the super-state deferment function. In a software implementation of OD^{2}FA, performing these checks may incur a time penalty. However, in our proposed TCAM implementation in Section V, we can perform these checks with no penalty.

As defined, we store $\mathcal{F}$ rather than F; thus deferment information is stored only at the super-state level. Likewise, we store just RegEx matching information $\mathcal{M}$ at the super-state level. Finally, with Δ, many super-state transitions represent multiple singleton transitions. Combined, we can achieve significant savings. Figure 3(a) shows the D^{2}FA for the RegEx set {`/abc/`, `/abd/`, `/e.*f/`}. The dashed edges are deferment transitions. Figure 3(b) shows the corresponding OD^{2}FA. The D^{2}FA needs to store 518 actual transitions and 10 deferment transitions while the OD^{2}FA only needs to store 260 actual transitions, most of which are not singleton super-state transitions, and 5 super-state deferred transitions. For this example, we achieve near optimal compression given only two overlays in the OD^{2}FA when compared to the D^{2}FA.

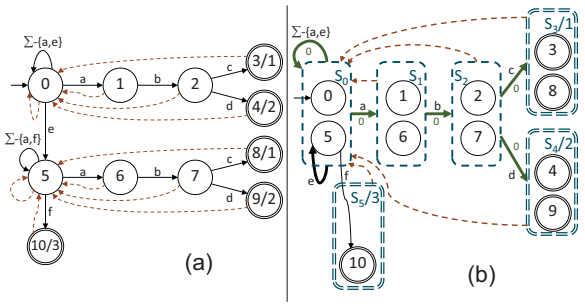


Fig. 3. (a) D^{2}FA for RegEx set {`/abc/`, `/abd/`, `/e.*f/`}. (b) Corresponding OD^{2}FA.

IV. OD^{2}FA Construction

Given a set of RegExes, we construct its equivalent OD^{2}FA incrementally in two phases. In the first phase, we construct an equivalent individual OD^{2}FA for each RegEx. In the second phase, we merge all the individual OD^{2}FAs in a balanced binary tree fashion; *i.e.* we merge two OD^{2}FAs into one OD^{2}FA at a time until there is only one OD^{2}FA for the entire given RegEx set.

A. OD^{2}FA Construction from One RegEx

Given one RegEx, we first build its equivalent D^{2}FA using the technique described in [14]. The deferment relationship among states in this D^{2}FA defines a deferment forest. Meiners *et al.* define the concept of a self-looping state which are states that transit to themselves for more than $|\Sigma|/2 = 128$ characters. Most failure transitions end in self-looping states. A key observation is that any D^{2}FA is also a valid OD^{2}FA with only a single overlay, singleton super-states, and singleton super-state transitions. We gradually convert the D^{2}FA into a more compact OD^{2}FA by first creating valid overlays and super-states and then updating the super-state transition function to combine multiple transitions into one super-state transition.

We begin by specifying the number of deferment trees in the super-state deferment forest and the number of overlays in a super-state. We accomplish these tasks by partitioning the self-looping root states of the D^{2}FA into two groups, accepting root states and rejecting root states. If either partition is empty, we create one deferment tree in the OD^{2}FA; otherwise there are two deferment trees. The number of overlays in the OD^{2}FA is the larger of the number of accepting root states and the number of rejecting root states. To ensure the number of overlays is a power of 2, we pad extra overlays as needed. This helps when we later compute super-state transitions.

B. OD^{2}FA Merge Algorithm

We present our OD^{2}FA merge algorithm, which we call *OD2FAMerge*, that constructs OD^{2}FA $\mathcal{D}_3$ with underlying D^{2}FA D_3 for the RegEx set $R_3 = R_1 \cup R_2$ given two OD^{2}FAs, $\mathcal{D}_1$ with underlying D^{2}FA D_1 for RegEx set R_1 and $\mathcal{D}_2$ with underlying D^{2}FA D_2 for RegEx set R_2 where $R_1 \cap R_2 = \emptyset$.

The first step is to create the merged D^{2}FA D_3. We use the space efficient D^{2}FA merge algorithm developed by Patel *et al.* in [17]. This algorithm extends the standard *Union Cross Product* (UCP) construction algorithm for merging DFAs [10].

We now construct OD^{2}FA $\mathcal{D}_3 = (Q_3, \Sigma, q_{03}, \mathcal{F}_3, \mathcal{S}_3, \mathcal{O}_3, \mathcal{M}_3, \Delta_3)$ from the input OD^{2}FAs $\mathcal{D}_1 = (Q_1, \Sigma, q_{01}, \mathcal{F}_1, \mathcal{S}_1, \mathcal{O}_1, \mathcal{M}_1, \Delta_1)$ and $\mathcal{D}_2 = (Q_2, \Sigma, q_{02}, \mathcal{F}_2, \mathcal{S}_2, \mathcal{O}_2, \mathcal{M}_2, \Delta_2)$ as well as the merged D^{2}FA D_3. The first three terms are derived from D_3. We then set $\mathcal{S}_3 = \mathcal{S}_1 \times \mathcal{S}_2$ and $\mathcal{O}_3 = \mathcal{O}_1 \times \mathcal{O}_2$. We reduce $\mathcal{S}_3$ to only include reachable super-states (a super-state is reachable if it contains at least one reachable state). We discuss how we handle empty overlays in Section V. For any super-state $S_3 = \langle S_1, S_2\rangle \in \mathcal{S}_3$, we set $\mathcal{M}_3(S_3) = \mathcal{M}_1(S_1) \cup \mathcal{M}_2(S_2)$.

We define the super-state deferment relationship $\mathcal{F}_3$ as follows: for any super-state S, which contains one or more states in Q_3, we defer it to the super-state that contains most of the states that the states in S defer to; *i.e.*, $\forall S \in \mathcal{S}$, $\mathcal{F}_3(S) := \text{mode}(\{\mathcal{S}_3(F_3(u)) \mid u \in S\})$ where mode is the function that returns the most common item in a given multi-set. After defining $\mathcal{F}_3$, we need to adjust the deferment relationship F for D^{2}FA D_3. Specifically, for each state s in a super-state S where S defers to super-state S', we let s defer to state s' in S' where s and s' are in the same overlay if $s' \neq \perp$. If $s' = \perp$, we split S into two super-states $S_1 = S \setminus \{s\}$ and $S_2 = \{s\}$, where S_2 defers to the super-state that contains the state that s defers to (*i.e.*, $\mathcal{F}_3(S_2) := \mathcal{S}_3(F_3(s))$). Note that the case that $s' = \perp$ rarely happens in our experimental RegEx sets. This super-state splitting ensures that Condition (C2) holds for $\mathcal{D}_3$.

V. OD^{2}FA Implementation in TCAM

In this section, we describe how OD^{2}FA can be implemented in TCAM and present our OverlayCAM algorithm for doing so. TCAM-based implementations of automata typically use two tables to represent an automata: a TCAM lookup table with a source state ID column and an input character column, and a corresponding SRAM decision table which contains the next state ID. To implement OD^{2}FA in TCAM, we use the unique pair of super-state ID and overlay ID as source state ID in the TCAM lookup table and next state ID in the SRAM decision table. The super-state ID and overlay ID columns in TCAM will be filled with ternary values that together match multiple states rather than a single state whereas the super-state ID and overlay ID columns in SRAM will be binary values that together match a single state. We add an extra bit in the SRAM decision table to specify the overlay bit in the super-state transition decision. We leverage the first match feature of TCAMs to ensure that the correct transition will be found in the TCAM lookup table. Specifically, if super-state S defers to super-state S', then we list all the super-state transitions for super-state S before those of super-state S'. We describe several of the key steps in OverlayCAM in the remainder of this section.

A. Generating Super-state IDs and Codes

For super-states, we apply the shadow encoding algorithm in [14] on the super-state deferment forest of the given OD^{2}FA to generate a binary super-state ID $SSID(S)$ and a ternary super-state shadow code $SSCD(S)$ for each super-state S that satisfy the following four properties: (1) Uniqueness Property: For any two distinct super-states S_1 and S_2, $ID(S_1) \neq ID(S_2)$ and $SC(S_1) \neq SC(S_2)$. (2) *Self-Matching Property*: For any super-state S, $ID(S) \in SC(S)$ (*i.e.*, $ID(S)$ matches $SC(S)$). (3) *Deferment Property*: For any two super-states S_1 and S_2, $S_1 \twoheadrightarrow S_2$ (*i.e.*, S_2 is an ancestor of S_1 in the given deferment tree) if and only if $SC(S_1) \subset SC(S_2)$. (4) *Non-interception Property*: For any two distinct super-states S_1 and S_2, $S_1 \twoheadrightarrow S_2$ if and only if $ID(S_1) \in SC(S_2)$.

B. Implementing Super-state Transitions

We now address the implementation of super-state transitions in TCAM. Let $(S_1, X) \xrightarrow{\sigma} (S_2, o, b)$ be the super-state transition we want to implement in TCAM. In the TCAM table, we use $SSCD(S_1)$ in the super-state ID column. Since we restrict the set of overlays in any super-state transition to ternary values, we can just use X in the overlay ID column of the TCAM. For the SRAM, in the super-state ID column, we use $SSID(S_2)$, In the overlay ID column, we use the binary representation of the overlay value o. The offset bit b is stored in the offset bit location in SRAM.

The RegEx matching process works as follows. Let S be the current super-state, O be the current overlay, and σ the current input character. So $s = SSID(S) \cdot O$ denotes the current state; s concatenated with σ is used as a TCAM lookup key. Let uid be the SSID stored in super-state ID column in SRAM and o be the value stored in the overlay ID column in SRAM and b be the value of the offset bit stored in SRAM. We compute the next super-state ID and overlay ID as follows. The next super-state ID will be uid. The next overlay ID will be $(b \times \mathcal{O}(s) + o) \bmod |\mathcal{O}|$. If $b = 0$, the next overlay ID is simply o. If $b = 1$, the next overlay ID is $(\mathcal{O}(s) + o) \bmod |\mathcal{O}|$; in most cases where $o = 0$, the next overlay ID is $(\mathcal{O}(s) + 0) \bmod |\mathcal{O}| = \mathcal{O}(s)$.

C. TCAM Table Generation

We now explain how we generate the TCAM entries for OD^{2}FA. We work on one super-state at a time. Let S be the current super-state. We create a TCAM table for S by creating one TCAM entry per super-state transition on S. After building this initial TCAM table for S, we reduce the TCAM entries by applying the bit merging algorithm on the TCAM entries generated for the super-state focusing on the current input character σ. This mostly helps with case insensitive searches where transitions on the alphabet characters will mostly occur in pairs and such pairs can be merged because they differ on only one bit in ASCII encoding.

We order the TCAM tables of the super-states according to the super-state deferment relationship (every super-state table occurs before its deferred super-state table).The overlay classifiers for the root super-state exclude all the self-looping transitions. All of these transitions are handled by the last rule added in the TCAM, which is all $*$s.

VI. Experimental Results

We implemented OverlayCAM using C++ and conducted experiments to evaluate its effectiveness and scalability. We verify our results by confirming that the TCAM table generated by OverlayCAM is equivalent to the original DFA. That is, for every pair of current state and input character, the next state returned by the TCAM lookup matches the next state returned by the DFA.

A. Data Sets and Methods

We performed experiments using two distinct groups of RegEx sets. One group consists of 8 real-world RegEx sets, some of which have been used in previous papers. The 8 real-world RegEx sets include 4 RegEx sets from a large networking vendor (*i.e.*, C7, C8, C10, and C613) and 4 RegEx sets from Bro and Snort (*i.e.*, Bro217, Snort24, Snort31, and Snort34). For each set, the number indicates the number of RegExes in the RegEx set. The second group SCALE is a synthetic RegEx set consisting of 13 RegExes from a recent release of the Snort rules. We use SCALE to test the scalability of OverlayCAM by adding one RegEx at a time from SCALE. Each SCALE RegEx contains closure on the wildcard or a range; these cause the DFA size to double as each SCALE RegEx is added. The final SCALE DFA has 225,040 states.

We define the following metric for measuring the amount of state replication in the DFA that corresponds to an RegEx set. *For any RegEx set R, we define $SR(R)$ to be the ratio of the number of states in the minimum state DFA corresponding to R divided by the number of states in the standard NFA without ϵ transitions corresponding to R.* Based on the characteristics of the RegExes, we partition 8 real-world RegEx sets into three groups, STRING ={C613, Bro217}, which contains mostly strings, causing little state replication ($SR(Bro271) = 3.0$, $SR(C613) = 2.1$); WILDCARD ={C7, C8 and C10}, which contains multiple wildcard closures '.*', causing lots of state replication ($SR(C7) = 231$, $SR(C8) = 43$, and $SR(C10) = 162$); and SNORT ={Snort24, Snort31, and Snort34}, which contain a diverse set of RegExes, roughly 40% of the RegExes have wildcard closures, causing moderate state replication ($SR(Snort24) = 24$, $SR(Snort31) = 22$, and $SR(Snort34) = 16$).

For each of the 8 RegEx sets, we built the corresponding NFA and minimum state DFA using standard automata theory algorithms. We also ran OverlayCAM, `RegCAM-TC` (RegCAM without Table Consolidation) and `RegCAM+TC` (RegCAM with Table Consolidation). For `RegCAM+TC`, we consolidated 4 tables together as was done in [14]. For TCAM space, we only report the number of TCAM entries because the TCAM widths for all TCAM tables generated by `RegCAM-TC`, `RegCAM+TC`, and OverlayCAM on all 8 RegEx sets are in the range [21,27]. Since TCAM width typically is only allowed to be configured as 36, 72, or 144 bits, we use a TCAM width of 36 in all cases. TCAM lookup speed is typically higher for smaller TCAM chips. We use the well adopted TCAM model proposed by Agrawal and Sherwood [1] to calculate RegEx matching throughput.

When comparing OverlayCAM to NFA, we use the following two metrics. The first is the *TCAM Expansion Factor (TEF)* of a RegEx set; this is the number of TCAM entries generated by OverlayCAM when given the RegEx set divided by the number of NFA transitions in the corresponding NFA. The second is the *super-state expansion factor (SEF)* of a RegEx set; this is the number of super-states in the ODFA divided by the number of NFA states. We use these ratios to assess how well we achieve the NFA size requirement.

B. Comparison with NFA

We now show that we satisfy the NFA size requirement by comparing the size of our RegEx matching solution for the 8 real-world data sets and SCALE to the size of the corresponding NFAs. The data for the 8 real world data sets is shown in Table II. As this data clearly illustrates, OverlayCAM does satisfy the NFA size requirement as the average value of TEF is 0.63 with a maximum value of 1.65. Thus, the number of TCAM entries is typically 63% of that of the number of NFA transitions. We do particularly well on the three WILDCARD sets that exhibit significant state replication. For these sets, the TEF values are all below 10%.

RE set	SR	NFA		OverlayCAM			Ratios	
		# States	# Trans.	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{O}\rvert$	# TCAM	SEF	TEF
C8	43.17	72	2177	85	72	125	1.18	0.06
C10	161.61	92	2982	133	288	263	1.45	0.09
C7	231.31	107	3261	127	648	234	1.19	0.07
Snort24	24.15	575	4054	897	30	1426	1.56	0.35
Snort34	15.52	891	4731	1151	48	2293	1.29	0.48
Snort31	21.88	917	5738	2395	32	9478	2.61	1.65
Bro217	3.06	2132	5424	3401	2	6028	1.60	1.11
C613	2.12	5343	14563	11308	1	18256	2.12	1.25

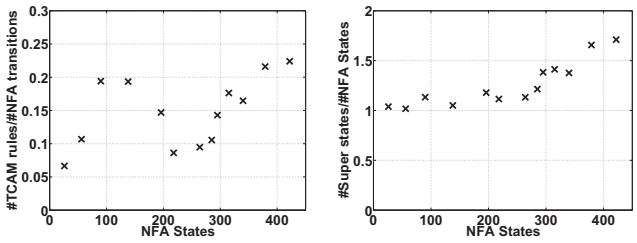
TABLE II
EXPERIMENTAL RESULTS OF OVERLAYCAM ON 8 REAL-WORLD REGEX SETS IN COMPARISON WITH NFA. REGEX SETS ORDERED BY NUMBER OF NFA STATES

There are two key reasons why OverlayCAM is able to perform as well if not better than NFAs. (1) *OverlayCAM is very effective in conquering state replication*. OverlayCAM effectively and automatically identifies all NFA state replicates and groups them together into super-states. Examining the SEF ratio data, the number of super-states is, on average, 1.62 times the number of NFA states and is never more than 2.61 times the number of NFA states. Looking at this another way, the number of overlays is proportional to the $SR(R)$ value. (2) *OverlayCAM effectively multiplies the compression benefits of conquering state replication and transition sharing*. That is, OverlayCAM effectively multiplies the benefits of ODFA and D^{2}FA. The average number of TCAM entries per super-state is only 2.14, even when super-states have hundreds of constituent states. The addition of effective transition sharing allows OverlayCAM to outperform NFAs, particularly when there is significant state replication.

We now consider the SCALE dataset. We plot the SEF and TEF ratios for OverlayCAM in Fig. 4. As this data shows, TEF is very stable with essentially no growth as the number of NFA states increases. SEF is also relatively stable, though there may be a slow linear growth in SEF as the number of NFA states increases.

C. Comparison with TCAM-based RegEx Matching Schemes

We briefly summarize our results comparing OverlayCAM to RegCAM and Peng *et al.*'s scheme. Recall that both RegCAM and Peng *et al.*'s schemes fail to satisfy the scalable construction requirement as they both build the DFA



(a) TEF (b) SEF

Fig. 4. SEF and TEF vs. # NFA states for OverlayCAM on the SCALE RegEx sets

whereas OverlayCAM satisfies this requirement by only building OD^{2}FA. We now show that OverlayCAM is not only more efficient than RegCAM and Peng *et al.*'s schemes, it also outperforms them.

For the two string-based RegEx sets Bro217 and C613, we observe that OverlayCAM does not significantly outperform the two RegCAM algorithms. This is expected as OverlayCAM is designed to handle state replication and string-based RegEx sets have little state replication. For the other RegEx sets, OverlayCAM significantly outperforms RegCAM. (1) *OverlayCAM uses orders of magnitude fewer TCAM entries than RegCAM.* On average, OverlayCAM uses 41 times fewer TCAM entries than `RegCAM-TC` and 12 times fewer TCAM entries than `RegCAM+TC`. The difference in performance is well-predicted by the $SR(R)$ ratio and would grow significantly as the state replication value increases. (2) *OverlayCAM has significantly higher throughput than RegCAM.* On average, OverlayCAM has 2.5 and 1.93 times higher throughput than `RegCAM-TC` and `RegCAM+TC`, respectively. We compute OverlayCAM achieves an average throughput of 7.59 Gbps and achieves a maximum throughput of 12.5 Gbps for the C8. The lowest throughput is 3.11 Gbps for C613. We are only able to compare results with Peng *et al.*'s scheme for the two public RegEx sets Snort24 and Snort34. For these two sets, OverlayCAM requires 2.15 and 1.44 times less TCAM space.

VII. Conclusions

We make four key contributions. First, we propose the ODFA model to capture state and transition replication and the OD^{2}FA model to capture state and transition replication as well as transition sharing. Second, we propose an automated and scalable algorithm for constructing OD^{2}FA. Third, we propose the OverlayCAM algorithm for encoding OD^{2}FA transitions in TCAM. Finally, we implemented our scalable OD^{2}FA construction algorithm and OverlayCAM and experimentally validated our assertion that we can build an automata that uses the space of an NFA while achieving one lookup per input character like a DFA.

Acknowledgements

We would like to thank Jignesh Patel for his participation in the early stage of this work. He has helped us in implementing the proposed algorithms and conducting experiments. This work is supported by the National Science Foundation under Grant Numbers CCF-1347953.

References

[1] B. Agrawal and T. Sherwood. Modeling TCAM power for next generation network devices. In *Proc. IEEE Int. Symposium on Performance Analysis of Systems and Software*, pages 120– 129, 2006.

[2] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *Proc. INFOCOM*. IEEE, 2007.

[3] M. Becchi and P. Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. CoNext*, 2007.

[4] M. Becchi and P. Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. ACM/IEEE ANCS*, 2007.

[5] M. Becchi and P. Crowley. Efficient regular expression evaluation: Theory to practice. In *Proc. ACM/IEEE ANCS*, 2008.

[6] M. Becchi and P. Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proc. ACM CoNEXT*, 2008. Article Number 25.

[7] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proc. Field-Programmable Logic and Applications*, pages 956–959, 2003.

[8] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *Proc. 12th FCCM*, Washington, DC, 2004.

[9] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, G. Antichi, and A. D. Pietro. An improved DFA for fast regular expression matching. *Computer Communication Review*, 38(5):29–40, 2008.

[10] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2000.

[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*, pages 155–164, 2007.

[12] 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*, pages 339–350, 2006.

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

[14] 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 Symposium*, Washington, DC, August 2010.

[15] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. ACM/IEEE ANCS*, 2007.

[16] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *Proc. IEEE Field Programmable Custom Computing Machines*, 2003.

[17] J. Patel, A. X. Liu, and E. Torng. Bypassing space explosion in regular expression matching for network intrusion detection and prevention systems. In *Proc. Network and Distributed System Security Symposium (NDSS'12)*, February 2012.

[18] K. Peng, S. Tang, M. Chen, and Q. Dong. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM. In *Proc. ACM ANCS*, pages 24–35, 2011.

[19] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines FCCM*, pages 227–238, 2001.

[20] R. Smith, C. Estan, and S. Jha. Xfa: Faster signature matching with extended automata. In *Proc. IEEE Symposium on Security and Privacy*, pages 187–201, 2008.

[21] 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*, pages 207–218, 2008.

[22] R. Sommer and V. Paxson. Enhancing bytelevel network intrusion detection signatures with context. In *Proc. ACM Conf. on Computer and Communication Security*, pages 262–271, 2003.

[23] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *Proc. Field-Programmable Custom Computing Machines*, 2004.

[24] L. Yang, R. Karim, V. Ganapathy, and R. Smith. Fast, memory-efficient regular expression matching with NFA-OBDDs. *Computer Networks*, 55(55):3376–3393, 2011.

[25] 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 Symposium on Architecture for Networking and Communications Systems (ANCS)*, pages 93–102, 2006.