Fast Regular Expression Matching Using Small TCAM

Chad R. Meiners Jignesh Patel Eric Norige Alex X. Liu* Eric Torng

Abstract—**Regular expression (RE) matching is a core component of deep packet inspection in modern networking and security devices. In this paper, we propose the first hardware-based RE matching approach that uses Ternary Content Addressable Memory (TCAM), which is available as off-the-shelf chips and has been widely deployed in modern networking devices for tasks such as packet classification. We propose three novel techniques to reduce TCAM space and improve RE matching speed: transition sharing, table consolidation, and variable striding. We tested our techniques on 8 real-world RE sets, and our results show that small TCAMs can be used to store large Deterministic Finite Automata (DFAs) and achieve potentially high RE matching throughput. For space, we can store each of the corresponding 8 DFAs with 25,000 states in a 0.59Mb TCAM chip. Using a different TCAM encoding scheme that facilitates processing multiple characters per transition, we can achieve potential RE matching throughput of 10 to 19 Gbps for each of the 8 DFAs using only a single 2.36 Mb TCAM chip.**

I. Introduction

A. Background and Problem Statement

Deep packet inspection is a key part of many networking devices on the Internet such as Network Intrusion Detection (or Prevention) Systems (NIDS/NIPS), firewalls, and layer 7 switches. In the past, deep packet inspection typically used *string matching* as a core operator, namely examining whether a packet's payload matches any of a set of predefined strings. Today, deep packet inspection typically uses *regular expression (RE) matching* as a core operator, namely examining whether a packet's payload matches any of a set of predefined regular expressions, because REs are fundamentally more expressive, efficient, and flexible in specifying attack signatures [1]. Most open source and commercial deep packet inspection engines such as Snort, Bro, TippingPoint X505, and many Cisco networking appliances use RE matching. Likewise, some operating systems such as Cisco IOS and Linux have built RE matching into their layer 7 filtering functions. As both traffic rates and signature set sizes are rapidly growing over time, fast and scalable RE matching is now a core network security issue.

RE matching algorithms are typically based on the Deterministic Finite State Automata (DFA) representation of regular expressions. A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, A)$, where Q is a set of states, Σ is an alphabet, $\delta\colon \Sigma \times Q \rightarrow Q$ is the transition function, q_0 is the start state, and $A \subseteq Q$ is a set of accepting states. Any set of regular expressions can be converted into an equivalent minimum state DFA. The fundamental issue with DFA-based algorithms is the large amount of memory required to store transition table δ. We have to store $\delta(q, a) = p$ for each state q and character a.

B. Summary and Limitations of Prior Art

Prior RE matching algorithms are either software-based [2], [3], [4], [5], [6], [7], [8] or FPGA-based [9], [4]. Software-based solutions have to be implemented in customized ASIC chips to achieve high-speed, the limitations of which include high deployment cost and being hard-wired to a specific solution and thus limited ability to adapt to new RE matching solutions. Although FPGA-based solutions can be modified, resynthesizing and updating FPGA circuitry in a deployed system to handle regular expression updates is slow and difficult; this makes FPGA-based solutions difficult to be deployed in many networking devices (such as NIDS/NIPS and firewalls) where the regular expressions need to be updated frequently [5].

C. Our Approach

To address the limitations of prior art on high-speed RE matching, we propose the first Ternary Content Addressable Memory (TCAM) based RE matching solution. We use a TCAM and its associated SRAM to encode the transitions of the DFA built from an RE set where one TCAM entry might encode multiple DFA transitions. TCAM entries and lookup keys are encoded in ternary as 0's, 1's, and *'s where *'s stand for either 0 or 1. A lookup key matches a TCAM entry if and only if the corresponding 0's and 1's match; for example, key 0001101111 matches entry 000110****. TCAM circuits compare a lookup key with all its occupied entries in parallel and return the index (or sometimes the content) of the first address for the content that the key matches; this address is then used to retrieve the corresponding decision in SRAM.

Given an RE set, we first construct an equivalent minimum state DFA [10]. Second, we build a two column TCAM lookup table where each column encodes one of the two inputs to δ: the *source* state ID and the *input* character. Third, for each TCAM entry, we store the *destination* state ID in the same

∗ Alex X. Liu is the corresponding author of this paper.
C. Meiners is now with MIT Lincoln Laboratory, Lexington, MA, but the work was conducted while he was at Michigan State University, MI. chad.meiners@ll.mit.edu. J. Patel, E. Norige, A. Liu, and E. Torng are at Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824. Alex X. Liu is also with the Department of Computer Science and Technology at Nanjing University. The preliminary version of this paper titled "Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems" was published in USENIX Security 2010. This work is supported in part by the National Science Foundation under Grant Numbers CNS-1017588 and CNS-0845513 and by the National Natural Science Foundation of China (Grant No. 61272546).
Email: {patelji1, norigeer, alexliu, torng}@cse.msu.edu.

entry of the associated SRAM. Fig. 1 shows an example DFA, its TCAM lookup table, and its SRAM decision table. We illustrate how this DFA processes the input stream "01101111, 01100011". We form a TCAM lookup key by appending the current input character to the current source state ID; in this example, we append the first input character "01101111" to "00", the ID of the initial state s_0, to form "0001101111". The first matching entry is the second TCAM entry, so "01", the destination state ID stored in the second SRAM entry is returned. We form the next TCAM lookup key "0101100011" by appending the second input character "011000011" to this returned state ID "01", and the process repeats.

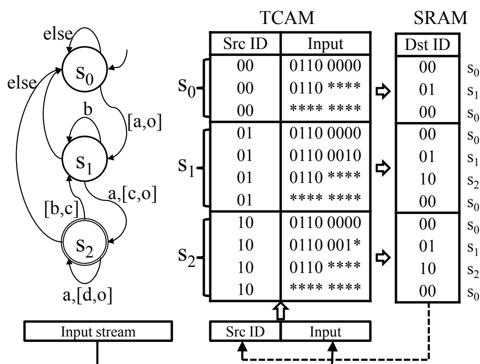


Fig. 1. A DFA with its TCAM table

There are three key reasons why TCAM-based RE matching works well. First, *a small TCAM is capable of encoding a large DFA* with carefully designed algorithms leveraging the ternary nature and first-match semantics of TCAMs. Our experimental results show that each of the DFAs built from 8 real-world RE sets with as many as 25,000 states, 4 of which were obtained from the authors of [3], can be stored in a 0.59Mb TCAM chip. The two DFAs that correspond to primarily string matching RE sets require 28 and 42 TCAM bits per DFA state; 5 of the remaining 6 DFAs which have a sizeable number of '.*' patterns require 12 to 14 TCAM bits per DFA state whereas the 6th DFA requires 26 TCAM bits per DFA state. Second, *TCAMs facilitate high-speed RE matching* because TCAMs are essentially high-performance parallel lookup systems: any lookup takes constant time (*i.e.*, a few CPU cycles) regardless of the number of occupied entries. Using Agrawal and Sherwood's TCAM model [11] and the resulting required TCAM sizes for the 8 RE sets, we show that it may be possible to achieve throughput ranging between 5.36 and 18.6 Gbps using only a single 2.36 Mb TCAM chip. Third, because TCAMs are off-the-shelf chips that are widely deployed in modern networking devices, *it should be easy to design networking devices that include our TCAM-based RE matching solution*. It may even be possible to immediately deploy our solution on some existing devices.

D. Technical Challenges and Proposed Solutions

1) Challenge 1: Encoding Large DFA in Small TCAM: Directly encoding a DFA in a TCAM using one TCAM entry per transition is infeasible. For example, consider a DFA with 25,000 states that consumes one 8 bit character per transition. We would need a total of 140.38 Mb ($= 25000 \times 2^8 \times (8 + \lceil \log 25000 \rceil)$). This is infeasible given the largest available TCAM chip has a capacity of only 72 Mb. To address this challenge, we use two techniques that minimize the TCAM space for storing a DFA: *transition sharing* and *table consolidation*.

The basic idea of transition sharing is to combine multiple transitions into one TCAM entry by exploiting two properties of DFA transitions: (1) character redundancy where many transitions share the same source state and destination state and differ only in their character label, and (2) state redundancy where many transitions share the same character label and destination state and differ only in their source state. One reason for the pervasive character and state redundancy in DFAs constructed from real-world RE sets is that most states have most of their outgoing transitions going to some common "failure" state; such transitions are often called default transitions. The low entropy of these DFAs opens optimization opportunities. We exploit character redundancy by *character bundling* (*i.e.*, input character sharing) and state redundancy by *shadow encoding* (*i.e.*, source state sharing). In character bundling, we use a ternary encoding of the input character field to represent multiple characters and thus multiple transitions that share the same source and destination states. In shadow encoding, we use a ternary encoding for the source state ID to represent multiple source states and thus multiple transitions that share the same label and destination state.

The basic idea of table consolidation is to merge multiple transition tables into one transition table using the observation that some transition tables share similar structures (*e.g.*, common entries) even if they have different decisions. This shared structure can be exploited by consolidating similar transition tables into one consolidated transition table. When we consolidate k TCAM lookup tables into one consolidated TCAM lookup table, we store k decisions in the associated SRAM decision table.

2) Challenge 2: Improving RE Matching Speed: One way to improve the throughput by up to a factor of k is to use k-stride DFAs that consume k input characters per transition. However, this leads to an exponential increase in both state and transition spaces. To avoid this space explosion, we propose the novel idea of *variable striding*. The basic idea of variable striding is to use transitions with a variety of strides so that we increase the average number of characters consumed per transition while ensuring all the transitions fit within the allocated TCAM space. This idea is based on two key observations. First, for many states, we can capture many but not all k-stride transitions using relatively few TCAM entries whereas capturing all k-stride transitions requires prohibitively many TCAM entries. Second, with TCAMs, we can store transitions with different strides in the same TCAM table.

3) Challenge 3: Handling REs with Counting Constraints: For REs with large counting constraints (such as `.*a.{n}bc`), the corresponding DFA may have an exponential number of states in the length n of the counting constraint. Other researchers have developed automata models to handle REs with counting constraints such as the counting-DFA proposed by Becchi *et al.* [12]. We propose to handle such REs using counting-DFA, which works well with RegCAM.

II. Related Work

In the past, deep packet inspection typically used string matching (often called pattern matching), which have been extensively studied [13], [14], [15], [16]). TCAM-based solutions have been proposed for string matching, but they do not generalize to RE matching because they only deal with independent strings [14], [15], [16].

Today, deep packet inspection often uses RE matching as a core operator because strings are no longer adequate to precisely describe attack signatures [17], [1]. Most prior work on RE matching falls into two categories: software-based and FPGA-based. Prior software-based RE matching solutions focus on either reducing memory by minimizing the number of transitions/states or improving speed by increasing the number of characters per lookup. Such solutions can be implemented on general purpose processors, but customized ASIC chip implementations are needed for high speed performance. For transition minimization, two basic approaches have been proposed: alphabet encoding that exploits character redundancy [2], [3], [4], [8] and default transitions that exploit state redundancy [5], [6], [3], [7]. Previous alphabet encoding approaches cannot fully exploit local character redundancy specific to each state. Most use a single alphabet encoding table that can only exploit global character redundancy that applies to every state. Kong *et al.* proposed using 8 alphabet encoding tables by partitioning the DFA states into 8 groups with each group having its own alphabet encoding table [8]. Our work improves upon previous alphabet encoding techniques because we can exploit local character redundancy specific to each state. Our work improves upon the default transition work because we do not need to worry about the number of default transitions that a lookup may go through because TCAMs allow us to traverse an arbitrarily long default transition path in a single lookup. Some transition sharing ideas have been used in some TCAM-based string matching solutions for Aho-Corasick-based DFAs [16], [18]. However, these ideas do not easily extend to DFAs generated by general RE sets, and our techniques produce at least as much transition sharing when restricted to string matching DFAs. One recently published transition sharing idea is the offset-DFA idea proposed by Liu *et al.* that exploits similarity in the transition function structure between two states even if they do not have the same destination state [19]. For state minimization, two fundamental approaches have been proposed. One approach is to first partition REs into multiple groups and build a DFA from each group; at run time, packet payload needs to be scanned by multiple DFAs [20], [9], [21]. This approach is orthogonal to our work and can be used in combination with our techniques. In particular, because our techniques achieve greater compression of DFAs than previous software-based techniques, less partitioning of REs will be required. The other approach is to use scratch memory to store variables that track the traversal history and avoid some duplication of states [22], [17], [12]. The benefit of state reduction for scratch memory-based FAs does not come for free. The size of the required scratch memory may be significant, and the time required to update the scratch memory after each transition may be significant. This approach is orthogonal to our approach. While we have only applied our techniques to DFAs in this initial study of TCAM-based RE matching, our techniques may work very well with scratch memory-based automata.

Prior FPGA-based solutions exploit the parallel processing capabilities of FPGA technology to implement nondeterministic finite automata (NFA) [9], [4] or parallel DFAs [23]. While NFAs are more compact than DFAs, they require more memory bandwidth to process each transition as an NFA may be in multiple states whereas a DFA is always only in one state. Thus, each character might be processed in up to $|Q|$ transition tables. Prior work has looked at ways for finding good NFA representations of the REs that limit the number of states that need to be processed simultaneously. However, FPGA's cannot be quickly reconfigured, and they have clock speeds that are slower than ASIC chips. Note that the RE matching solutions that can be implemented in ASIC and simulated by FPGA, these do not fall into this category.

III. Transition Sharing

The basic idea of transition sharing is to combine multiple transitions into a single TCAM entry. We propose two transition sharing ideas: character bundling and shadow encoding. Character bundling exploits intra-state optimization opportunities and minimizes TCAM tables along the input character dimension. Shadow encoding exploits inter-state optimization opportunities and minimizes TCAM tables along the source state dimension.

A. Character Bundling

Character bundling exploits character redundancy by combining multiple transitions from the same source state to the same destination into one TCAM entry. Character bundling consists of four steps. (1) Assign each state a unique ID of $\lceil \log |Q| \rceil$ bits. (2) For each state, enumerate all 256 transition rules where for each rule, the predicate is a transition's label and the decision is the destination state ID. (3) For each state, treating the 256 rules as a 1-dimensional packet classifier and leveraging the ternary nature and first-match semantics of TCAMs, we minimize the number of transitions using the optimal 1-dimensional TCAM minimization algorithm in [24], [25]. (4) Concatenate the $|Q|$ 1-dimensional minimal prefix classifiers together by prepending each rule with its source state ID. The resulting list can be viewed as a 2-dimensional classifier where the two fields are source state ID and transition label and the decision is the destination state ID. Fig. 1 shows an example DFA and its TCAM lookup table built using character bundling. The three chunks of TCAM entries encode the 256 transitions for s_0, s_1, and s_2, respectively. Without character bundling, we would need 256×3 entries.

B. Shadow Encoding

Whereas character bundling uses ternary codes in the input character field to encode multiple input characters, shadow encoding uses ternary codes in the source state ID field to encode multiple source states.

1) Observations: We use our running example in Fig. 1 to illustrate shadow encoding. We observe that all transitions with source states s_1 and s_2 have the same destination state except for the transitions on character c. Likewise, source state s_0 differs from source states s_1 and s_2 only in the character range $[a, o]$. This implies there is a lot of state redundancy. The table in Fig. 2 shows how we can exploit state redundancy to further reduce required TCAM space. First, since states s_1 and s_2 are more similar, we give them the state IDs 00 and 01, respectively. State s_2 uses the ternary code of 0* in the state ID field of its TCAM entries to share transitions with state s_1. We give state s_0 the state ID of 10, and it uses the ternary code of $**$ in the state ID field of its TCAM entries to share transitions with both states s_1 and s_2. Second, we order the state tables in the TCAM so that state s_1 is first, state s_2 is second, and state s_0 is last. This facilitates the sharing of transitions among different states where earlier states have incomplete tables deferring some transitions to later tables.

TCAM			SRAM
Src State	ID	Input	Dest State ID
s_1	00	0110 0011	01: s_2
	0*	0110 001*	00: s_1
s_2	0*	0110 0000	10: s_0
	0*	0110 ****	01: s_2
	**	0110 0000	10: s_0
s_0	**	0110 ****	00: s_1
	**	**** ****	10: s_0

Fig. 2. TCAM table with shadow encoding

In the rest of this section, we solve the following three problems in order to implement shadow encoding: (1) Find the best order of the state tables in the TCAM (any order is allowed). (2) Choose binary IDs and ternary codes for each state given the state table order. (3) Identify entries to remove from each state table.

Our shadow encoding technique builds upon prior work with default transitions [5], [6], [3], [7] by exploiting the same state redundancy observation and using their concepts of default transitions and Delayed input DFAs (D^{2}FA). However, our final technical solutions are different because we work with TCAM whereas prior techniques work with RAM. For example, the concept of a ternary state code has no meaning when working with RAM. The key advantage of shadow encoding in TCAM over prior default transition work is speed. Shadow encoding in TCAM incurs no delay whereas prior default transition techniques incur significant delay because a DFA may have to traverse multiple default transitions before consuming an input character.

2) Determining Table Order: We first describe how we compute the order of tables within the TCAM. We use some concepts such as default transitions and D^{2}FA that were originally defined by Kumar *et al.* [5] and subsequently refined in [6], [3], [7].

A D^{2}FA is a DFA with default transitions where each state p can have at most one default transition to one other state q in the D^{2}FA. In a legal D^{2}FA, the directed graph consisting of only default transitions must be acyclic; we call this graph a *deferment forest*. It is a forest rather than a tree since more than one node may not have a default transition. We call a tree in a deferment forest a *deferment tree*.

We determine the order of state tables in TCAM by constructing a deferment forest F and then using the partial order defined by F. If there is a directed path from state p to state q in F, we say that state p *defers* to state q, denoted $p \succ q$. If $p \succ q$, we say that state p is in state q's shadow. Specifically, state q's transition table must be placed after the transition tables of all states in state q's shadow.

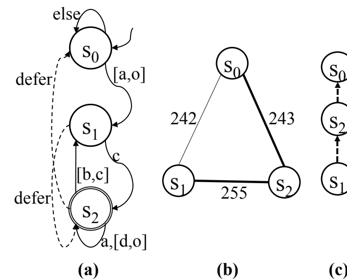


Fig. 3. D^{2}FA, SRG, and deferment tree of the DFA in Fig. 1

Our algorithm to compute a deferment forest that minimizes the TCAM representation of the resulting D^{2}FA builds upon algorithms from prior work [5], [6], [3], [7], but there are several key differences. First, unlike prior work, we do not pay a speed penalty for long default transition paths. Thus, we achieve better transition sharing than prior work. Second, to maximize the potential gains from our variable striding technique described in Section V and table consolidation, we choose states that have lots of self-loops to be the roots of our deferment trees. Prior work has typically chosen roots in order to minimize the distance from a leaf node to a root, though Becchi and Crowley do consider related criteria when constructing their D^{2}FA [3]. Third, we explicitly ignore transition sharing between states that have few transitions in common. This has been done implicitly in the past, but we show how doing so leads to better results with table consolidation.

Our algorithm consists of four steps. First, we construct a Space Reduction Graph (SRG), [5], from a given DFA. Given a DFA with $|Q|$ states, an SRG is a clique with $|Q|$ vertices each representing a distinct state. The weight of each edge is the number of common outgoing transitions between the two connected states. Second, we trim away edges with small weight from the SRG. In our experiments, we use a cutoff of 10. This pruning is effective because the distribution of edge weights in our experiments is bimodal: usually either very small (< 10) or very large (> 180). Using these low weight edges as default transitions leads to more TCAM entries and reduces the number of deferment trees which hinders our table consolidation technique (Section IV). Third, we compute a deferment forest by running Kruskal's algorithm to find a maximum weight spanning forest. Fourth, for each deferment tree, we pick the state that has largest number of transitions going back to itself as the root. Fig. 3(b) and (c) show the SRG and the deferment tree, respectively, for the DFA in Fig. 1.

In most deferment trees, more than 128 (*i.e.*, half) of the root state's outgoing transitions lead back to the root state; we call such a state a *self-looping state*. Based on the pigeonhole principle and the observed bimodal distribution, each deferment tree typically has exactly one self-looping state, and it is the root state. We choose self-looping states as roots to improve the effectiveness of variable striding which

we describe in Section V.

When we apply Kruskal's algorithm, we use a tie breaking strategy because many edges have the same weight. To have most deferment trees centered around a self-looping state, we give priority to edges that have the self-looping state as one endpoint.

3) Shadow Encoding Algorithm: We now describe our shadow encoding algorithm which takes as input a deferment forest F with one node per state and outputs a *shadow encoding* which consists of a ternary *shadow code* ($SC(q)$) and a binary *state ID* ($ID(q)$) for each state q. State IDs are used in the destination state ID field of transition rules. Shadow codes are used in the source state ID field of transition rules. The *shadow length* of a shadow encoding is the common length of every state ID and shadow code. A valid shadow encoding for a given deferment forest F must satisfy the following four *Shadow Encoding Properties* (SEP):

1) *Uniqueness Property*: For any two distinct states p and q, $ID(p) \neq ID(q)$ and $SC(p) \neq SC(q)$.
2) *Self-Matching Property*: For any state p, $ID(p) \in SC(p)$ (*i.e.*, $ID(p)$ matches $SC(p)$).
3) *Deferment Property*: For any two states p and q, $p \succ q$ (*i.e.*, q is an ancestor of p in the given deferment forest) if and only if $SC(p) \subset SC(q)$.
4) *Non-interception Property*: For any two distinct states p and q, $p \succ q$ if and only if $ID(p) \in SC(q)$.

We prove that these properties are sufficient to properly encode DFA transitions in the appendix, Theorem A.1.

We give a shadow encoding algorithm where the deferment forest is a single deferment tree DT. We handle deferment forests by simply creating a virtual root node whose children are the roots of the deferment trees in the forest and then running the algorithm on this tree. In the following, we refer to states as nodes. Our algorithm uses the following internal variables for each node v: a local binary ID denoted $L(v)$, a global binary ID denoted $G(v)$, and an integer weight denoted $W(v)$ that is the shadow length we would use for the subtree of DT rooted at v. Intuitively, the state ID of v will be $G(v)|L(v)$ where $|$ denotes concatenation, and the shadow code of v will be the prefix string $G(v)$ followed by the required number of *'s; some extra padding characters may be needed. We use $\#L(v)$ and $\#G(v)$to denote the number of bits in $L(v)$ and $G(v)$, respectively. Our algorithm processes nodes bottom-up. For all v, we initially set $L(v) = G(v) = \emptyset$ and $W(v) = 0$. Each leaf node of DT is now processed, which we denote by marking them red. We process an internal node v when all its children $v_1, \cdots, v_n$ are red. Once a node v is processed, its weight $W(v)$ and its local ID $L(v)$ are fixed, but we will prepend additional bits to its global ID $G(v)$ when we process its ancestors in DT.

We assign v and each of its children a variable-length binary code *HCode* such that no HCode is a prefix of another HCode. One option is to assign each node a binary number from 0 to n using $\lg(n+1)$ bits. To minimize the shadow code length $W(v)$, we use a Huffman coding style algorithm to compute the HCodes and $W(v)$. This algorithm uses two data structures: a binary encoding tree T with $n+1$ leaf nodes, one for v and each of its children, and a min-priority queue PQ, initialized with $n+1$ elements (one for v and each of its children) that is ordered by node weight. While PQ has more than one element, we remove the two elements x and y with lowest weight from PQ, create a new internal node z in T with two children x and y, and set weight(z)=maximum(weight(x), weight(y))+1, and then put element z into PQ. When PQ has only one element, T is complete. The HCode assigned to each leaf node v' is the path in T from the root node to v' where left edges have value 0 and right edges have value 1.

We update the internal variables of v and its descendants in DT as follows. We set $L(v)$ to be its HCode, and $W(v)$ to be the weight of the root node of T; $G(v)$ is left empty. For each child v_i, we prepend v_i's HCode to the global ID of every node in the subtree rooted at v_i including v_i itself. We then mark v as red. This continues until all nodes are red. We now set state IDs and a shadow codes. The shadow length is k, the weight of the root node of DT. We use $\{*\}^m$ to denote a ternary string with m *'s and $\{0\}^m$ to denote a binary string with m 0's. For each node v, $ID(v) = G(v)|L(v)|\{0\}^{k-\#G(v)-\#L(v)}$, $SC(v) = G(v)|\{*\}^{k-\#G(v)}$.

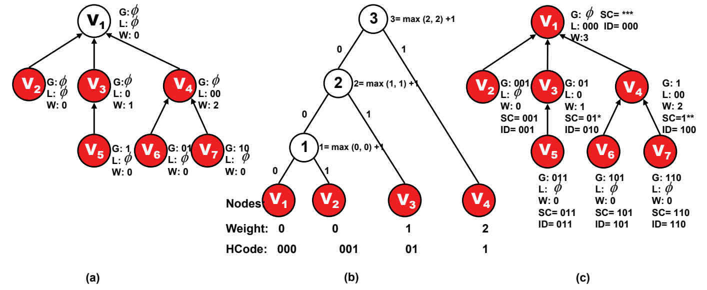


Fig. 4. Shadow encoding example

We illustrate our shadow encoding algorithm in Figure 4. Figure 4(a) shows all the internal variables just before v_1 is processed. Figure 4(b) shows the Huffman style binary encoding tree T built for node v_1 and its children v_2, v_3, and v_4 and the resulting HCodes. Figure 4(c) shows each node's final weight, global ID, local ID, state ID and shadow code.

We prove the correctness and optimality of our algorithm in Proofs A.2, A.3 in the appendix. Experimentally, no DFA had a shadow length larger than $\lceil \log_2 |Q| \rceil + 3$ where $\lceil \log_2 |Q| \rceil$ is the shortest possible shadow length.

4) Choosing Transitions: For a given DFA and a corresponding deferment forest, we construct a D^{2}FA by choosing which transitions to encode in each transition table as follows. If state p has a default transition to state q, we identify p's deferrable transitions which are the transitions that are common to both p's transition table and q's transition table. These deferrable transitions are optional for p's transition table; that is, they can be removed to create an incomplete transition table or included if that results in fewer TCAM entries. Fig. 2 is an example where including a deferrable transition produces a smaller classifier. The second entry in s_2's table in Fig. 2 can be deferred to state s_0's transition table. However, this results in a classifier with at least 4 TCAM entries whereas specifying the transition allows a classifier with just 3 TCAM entries. This leads us to the following problem for which we give an optimal solution.

Definition 3.1: (Partially Deferred Incomplete One-dimensional TCAM Minimization Problem)
Given a one-dimensional packet classifier f on $\{*\}^b$ and a subset $\mathcal{D} \subseteq \{*\}^b$, find the minimum cost prefix classifier f' such that $Cover(f') \supseteq \{*\}^b \setminus \mathcal{D}$ and is equivalent to f over $Cover(f')$.

Here b is the field width (in bits), $\mathcal{D}$ is the set of packets that can be deferred, and $Cover(c)$ is the union of the predicates of all the rules in c (*i.e.* all the packets matched by c). For simplicity of description, we assume that f has flattened rule set (*i.e.* one rule for each packet with the packet as the rule predicate). Assuming the packet is a one byte character, this implies f has 256 rules.

We provide a dynamic programming formulation for solving this problem that is similar to the dynamic programming formulation used in [27] and [24] to solve the related problem when all transitions must be specified. In these previous solutions for complete classifiers, for each prefix, the dynamic program maintains an optimal solution for each possible final decision. It then specifies how to combine these optimal solutions for matching prefixes into an optimal solution for the prefix that is the union of the two matching prefixes; in this step, two final rules for each prefix that have the same decision can be replaced by a single final rule for the combined prefix resulting in a savings of one TCAM entry. The main change is to maintain an optimal solution for each prefix where we defer some transitions within the prefix. Our formal characterization of this algorithm and proof of its result size is given in Theorem A.4, in the appendix.

Next, we discuss RE set updates. RE set updating is typically infrequent, unlike IP lookup. When an RE set is updated, we can use another computer to run our algorithms to compute the TCAM entries. Note that TCAM chips are often deployed in tandem. Thus, when one chip is updating, the other chip can be continuously queried.

IV. Table Consolidation

We now present *table consolidation* where we combine multiple transition tables for different states into a single transition table such that the combined table takes less TCAM space than the total TCAM space used by the original tables. To define table consolidation, we need two new concepts: k-decision rule and k-decision table. A *k-decision rule* is a rule whose decision is an array of k decisions. A *k-decision table* is a sequence of k-decision rules following the first-match semantics. Given a k-decision table $\mathbb{T}$ and i $(0 \leq i < k)$, if for any rule r in $\mathbb{T}$ we delete all the decisions except the i-th decision, we get a 1-decision table, which we denote as $\mathbb{T}[i]$. In table consolidation, we take a set of k 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$ and construct a k-decision table $\mathbb{T}$ such that for any i $(0 \leq i < k)$, the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds where $\mathbb{T}_i \equiv \mathbb{T}[i]$ means that $\mathbb{T}_i$ and $\mathbb{T}[i]$ are equivalent (*i.e.*, they have the same decision for every search key). We call the process of computing k-decision table $\mathbb{T}$ *table consolidation*, and we call $\mathbb{T}$ the *consolidated table*.

A. Observations

Table consolidation is based on three observations. First, semantically different TCAM tables may share common entries with possibly different decisions. For example, the three tables for s_0, s_1 and s_2 in Fig. 1 have three entries in common: 01100000, 0110****, and ********. Table consolidation provides a novel way to remove such information redundancy. Second, given any set of k 1-decision tables $\mathbb{T}_0, \cdots, \mathbb{T}_{k-1}$, we can always find a k-decision table $\mathbb{T}$ such that for any i $(0 \leq i < k)$, the condition $\mathbb{T}_i \equiv \mathbb{T}[i]$ holds. This is easy to prove as we can use one entry per each possible binary search key in $\mathbb{T}$. Third, a TCAM chip typically has a build-in SRAM module that is commonly used to store lookup decisions. For a TCAM with n entries, the SRAM module is arranged as an array of n entries where SRAM[i] stores the decision of TCAM[i] for every i. A TCAM lookup returns the index of the first matching entry in the TCAM, which is then used as the index to directly find the corresponding decision in the SRAM. In table consolidation, we essentially trade SRAM space for TCAM space because each SRAM entry needs to store multiple decisions. As SRAM is cheaper and more efficient than TCAM, moderately increasing SRAM usage to decrease TCAM usage is worthwhile.

Fig. 5 shows the TCAM lookup table and the SRAM decision table for a 3-decision consolidated table for states s_0, s_1, and s_2 in Fig. 1. In this example, by table consolidation, we reduce the number of TCAM entries from 11 to 5 for storing the transition tables for states s_0, s_1, and s_2. This consolidated table has an ID of 0. As both the table ID and column ID are needed to encode a state, we use the notation $< Table\ ID > @ < Column\ ID >$ to represent a state.

TCAM		SRAM		
Consolidated Src Table ID	Input Character	Column ID		
		00	01	10
0	0110 0000	s_0	s_0	s_0
0	0110 0010	s_1	s_1	s_1
0	0110 0011	s_1	s_2	s_1
0	0110 ****	s_1	s_2	s_2
0	**** ****	s_0	s_0	s_0

Fig. 5. 3-decision table for 3 states in Fig. 1

There are two key technical challenges in table consolidation. The first challenge is how to consolidate k 1-decision transition tables into a k-decision transition table. The second challenge is which 1-decision transition tables should be consolidated together. Intuitively, the more similar two 1-decision transition tables are, the more TCAM space saving we can get from consolidating them together. However, we have to consider the deferment relationship among states. We present our solutions to these two challenges.

B. Computing a k-decision table

In this section, we assume we know which states need to be consolidated together and present a local state consolidation algorithm that takes a k_1-decision table for state set S_i and a k_2-decision table for another state set S_j as its input and outputs a consolidated $(k_1 + k_2)$-decision table for state set $S_i \cup S_j$. For ease of presentation, we first assume that $k_1 = k_2 = 1$.

Let s_1 and s_2 be the two input states which have default transitions to states s_3 and s_4. The consolidated table will be

assigned a common table ID X. We assign state s_1 column ID 0 and state s_2 column ID 1. Thus, we encode s_1 as $X@0$ and s_2 as $X@1$. We enforce a constraint that if we do not consolidate s_3 and s_4 together, then s_1 and s_2 cannot defer any transitions at all. If we do consolidate s_3 and s_4 together, then s_1 and s_2 may have incomplete transition tables due to default transitions to s_3 and s_4, respectively.

The key concepts underlying this algorithm are breakpoints and critical ranges. To define breakpoints, it is helpful to view Σ as numbers ranging from 0 to $|\Sigma|-1$; given 8 bit characters, $|\Sigma| = 256$. For any state s, we define a character $i \in \Sigma$ to be a *breakpoint* for s if $\delta(s,i) \neq \delta(s,i-1)$. For the end cases, we define 0 and $|\Sigma|$ to be breakpoints for every state s. Let $b(s)$ be the set of breakpoints for state s. We then define $b(S) = \bigcup_{s \in S} b(s)$ to be the set of breakpoints for a set of states $S \subset Q$. Finally, for any set of states S, we define $r(S)$ to be the set of ranges defined by $b(S)$: $r(S) = \{[0, b_2 - 1], [b_2, b_3 - 1], \ldots, [b_{|b(S)|-1}, |\Sigma|-1]\}$ where b_i is ith smallest breakpoint in $b(S)$. Note that $0 = b_1$ is the smallest breakpoint and $|\Sigma|$ is the largest breakpoint in $b(S)$. Within $r(S)$, we label the range beginning at breakpoint b_i as r_i for $1 \leq i \leq |b(S)| - 1$. If $\delta(s, b_i)$ is deferred, then r_i is a deferred range.

When we consolidate s_1 and s_2 together, we compute $b(\{s_1, s_2\})$ and $r(\{s_1, s_2\})$. For each $r' \in r(\{s_1, s_2\})$ where r' is not a deferred range for both s_1 and s_2, we create a consolidated transition rule where the decision of the entry is the ordered pair of decisions for state s_1 and s_2 on r'. For each $r' \in r(\{s_1, s_2\})$ where r' is a deferred range for one of s_1 but not the other, we fill in r' in the incomplete transition table where it is deferred, and we create a consolidated entry where the decision of the entry is the ordered pair of decisions for state s_1 and s_2 on r'. Finally, for each $r' \in r(\{s_1, s_2\})$ where r' is a deferred range for both s_1 and s_2, we do not create a consolidated entry. This produces a non-overlapping set of transition rules that may be incomplete if some ranges do not have a consolidated entry. If the final consolidated transition table is complete, we minimize it using the optimal 1-dimensional TCAM minimization algorithm in [24], [25]. If the table is incomplete, we minimize it using the 1-dimensional incomplete classifier minimization algorithm in [27]. We generalize this algorithm to cases where $k_1 > 1$ and $k_2 > 1$ by simply considering $k_1 + k_2$ states when computing breakpoints and ranges.

C. Choosing States to Consolidate

We now describe our global consolidation algorithm for determining which states to consolidate together. As we observed earlier, if we want to consolidate two states s_1 and s_2 together, we need to consolidate their parent nodes in the deferment forest as well or else lose all the benefits of shadow encoding. Thus, we propose to consolidate two deferment trees together.

A consolidated deferment tree must satisfy the following properties. First, each node is to be consolidated with at most one node in the second tree; some nodes may not be consolidated with any node in the second tree. Second, a level i node in one tree must be consolidated with a level i node in the second tree. The level of a node is its distance from the root. We define the root to be a level 0 node. Third, if two level i nodes are consolidated together, their level $i-1$ parent nodes must also be consolidated together. An example legal matching of nodes between two deferment trees is depicted in Fig. 6.

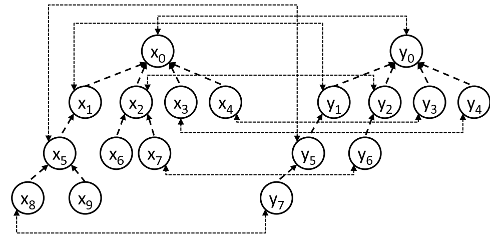


Fig. 6. Consolidating two trees

Given two deferment trees, we start the consolidation process from the roots. After we consolidate the two roots, we need to decide how to pair their children together. For each pair of nodes that are consolidated together, we again must choose how to pair their children together, and so on. We make an optimal choice using a combination of dynamic programming and matching techniques. Suppose we wish to compute the minimum cost $C(x, y)$, measured in TCAM entries, of consolidating two subtrees rooted at nodes x and y where x has u children $X = \{x_1, \ldots, x_u\}$ and y has v children $Y = \{y_1, \ldots, y_v\}$. We first recursively compute $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$ using our local state consolidation algorithm as a subroutine. We then construct a complete bipartite graph $K_{X,Y}$ such that each edge (x_i, y_j) has the edge weight $C(x_i, y_j)$ for $1 \leq i \leq u$ and $1 \leq j \leq v$. Here $C(x, y)$ is the cost of a minimum weight matching of $K(X, Y)$ plus the cost of consolidating x and y. When $|X| \neq |Y|$, to make the sets equal in size, we pad the smaller set with null states that defer all transitions.

Finally, we must decide which trees to consolidate together. We assume that we produce k-decision tables where k is a power of 2. We describe how we solve the problem for $k = 2$ first. We create an edge-weighted complete graph with where each deferment tree is a node and where the weight of each edge is the cost of consolidating the two corresponding deferment trees together. We find a minimum weight matching of this complete graph to give us an optimal pairing for $k = 2$. For larger $k = 2^l$, we then repeat this process $l-1$ times. Our matching is not necessarily optimal for $k > 2$.

In some cases, the deferment forest may have only one tree. In such cases, we consider consolidating the subtrees rooted at the children of the root of the single deferment tree. We also consider similar options if we have a few deferment trees but they are not structurally similar.

1) Greedy Matching: Our algorithm using the matching subroutines gives the optimal pairing of deferment trees but can be relatively slow on larger DFAs. When running time is a concern, we present a greedy matching routine. When we need to match children of two nodes, x and y, we consider one child at a time from the node with fewer children (say x). First all children of y are set *unmarked*. For each child, x_i, of x, we find the *best match* from the unmarked children of y, match them up, and set the matched child in y as *marked*.

The best match for x_i is given by

$$argmin_{y_j \in \{\text{unmarked children of } y\}} \frac{C(x_i, y_j)}{C(x_i) + C(y_j)}$$

where $C(x)$ is just the cost (in TCAM entries) of the subtree rooted at x. If $C(x_i)+C(y_j) = 0$, then we set the ratio to 0.5. All unmarked children of y at the end are matched with null states. We consider the children of x in decreasing order of $C(x_i)$ to prioritize the larger children of x. We use the same approach for matching roots. First all roots are set unmarked. Each time we consider the largest unmarked root, find the best match for it, and then mark the newly matched roots.

In our experiments, this greedy approach runs much faster than the optimal approach and the resulting classifier size is not much larger. We also observe that another greedy approach that uses $C(x_i, y_j)$ instead of $\frac{C(x_i,y_j)}{C(x_i)+C(y_j)}$ produces classifiers with much larger TCAM sizes. This approach often matches a large child of x with a small child of y that it does not align well with.

D. Effectiveness of Table Consolidation

We now explain why table consolidation works well on real-world RE sets. Our algorithm proceeds as follows. Most real-world RE sets contain REs with wildcard closures '.*' where the wildcard '.' matches any character and the closure '*' allows for unlimited repetitions of the preceding character. Wildcard closures create deferment trees with lots of structural similarity. For example, consider the D[2]FA in Fig. 7 for RE set `{.*a.*bc, .*cde}` where we use dashed arrows to represent the default transitions. The second wildcard closure '.*' in the RE `.*a.*bc` duplicates the entire DFA sub-structure for recognizing string `cde`. Thus, table consolidation of the subtree $(0, 1, 2, 3)$ with the subtree $(4, 5, 6, 7)$ will lead to significant space saving.

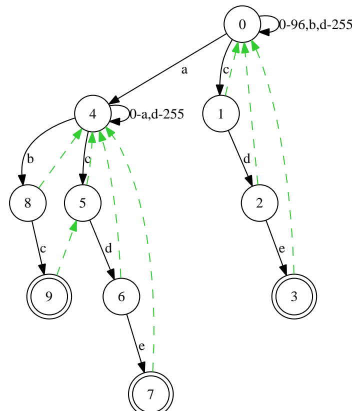


Fig. 7. D[2]FA for {a.*bc, cde}

V. Variable Striding

We explore ways to improve RE matching throughput by consuming multiple characters per TCAM lookup. One possibility is a k-stride DFA which uses k-stride transitions that consume k characters per transition. Although k-stride DFAs can speed up RE matching by up to a factor of k, the number of states and transitions can grow exponentially in k. To limit the state and transition space explosion, we propose variable striding using *variable-stride DFAs*. A k-var-stride DFA consumes between 1 and k characters in each transition with at least one transition consuming k characters. Conceptually, each state in a k-var-stride DFA has 256^k transitions, and each transition is labeled with (1) a unique string of k characters and (2) a stride length j ($1 \le j \le k$) indicating the number of characters consumed.

In TCAM-based variable striding, each TCAM lookup uses the next k consecutive characters as the lookup key, but the number of characters consumed in the lookup varies from 1 to k; thus, the lookup decision contains both the destination state ID and the stride length.

A. Observations

We use an example to show how variable striding can achieve a significant RE matching throughput increase with a small and controllable space increase. Fig. 8 shows a 3-var-stride transition table that corresponds to state s_0 in Figure 1. This table only has 7 entries as opposed to 116 entries in a full 3-stride table for s_0. If we assume that each of the 256 characters is equally likely to occur, the average number of characters consumed per 3-var-stride transition of s_0 is $1*1/16+2*15/256+3*225/256 = 2.82$.

TCAM		SRAM
SRC	Input	DEC : Stride
s_0	0110 0000 **** **** **** ****	s_0 : 1
s_0	0110 **** **** **** **** ****	s_1 : 1
s_0	**** **** 0110 0000 **** ****	s_0 : 2
s_0	**** **** 0110 **** **** ****	s_1 : 2
s_0	**** **** **** **** 0110 0000	s_0 : 3
s_0	**** **** **** **** 0110 ****	s_1 : 3
s_0	**** **** **** **** **** ****	s_0 : 3

Fig. 8. 3-var-stride transition table for s_0

B. Eliminating State Explosion

We first explain how converting a 1-stride DFA to a k-stride DFA causes state explosion. For a source state and a destination state pair (s, d), a k-stride transition path from s to d may contain $k-1$ intermediate states (excluding d); for each unique combination of accepting states that appear on a k-stride transition path from s to d, we need to create a new destination state because a unique combination of accepting states implies that the input has matched a unique combination of REs. This can be a very large number of new states.

We eliminate state explosion by ending any k-var-stride transition path at the first accepting state it reaches. Thus, a k-var-stride DFA has the exact same state set as its corresponding 1-stride DFA. Ending k-var-stride transitions at accepting states does have subtle interactions with table consolidation and shadow encoding. We end any k-var-stride consolidated transition path at the first accepting state reached in any one of the paths being consolidated which can reduce the expected throughput increase of variable striding. There is a similar but even more subtle interaction with shadow encoding which we describe in the next section.

C. Controlling Transition Explosion

In a k-stride DFA converted from a 1-stride DFA with alphabet Σ, a state has $|\Sigma|^k$ outgoing k-stride transitions. Although we can leverage our techniques of character bundling and shadow encoding to minimize the number of required TCAM entries, the rate of growth tends to be exponential with respect to stride length k. We have two key ideas to control transition explosion: self-loop unrolling and k-var-stride transition sharing.

1) Self-Loop Unrolling Algorithm: We now consider root states, most of which are self-looping. We have two methods to compute the k-var-stride transition tables of root states. The first is direct expansion (stopping transitions at accepting states) since these states do not defer to other states which results in an exponential increase in table size with respect to k. The second method, which we call *self-loop unrolling*, scales linearly with k.

Self-loop unrolling increases the stride of all the self-loop transitions encoded by the last default TCAM entry. Self-loop unrolling starts with a root state j-var-stride transition table encoded as a compressed TCAM table of n entries with a final default entry representing most of the self-loops of the root state. Note that given any complete TCAM table where the last entry is not a default entry, we can always replace that last entry with a default entry without changing the semantics of the table. We generate the (j+1)-var-stride transition table by expanding the last default entry into n new entries, which are obtained by prepending 8 *s as an extra default field to the beginning of the original n entries. This produces a (j+1)-var-stride transition table with $2n-1$ entries. Fig. 8 shows the resulting table when we apply self-loop unrolling twice on the DFA in Fig. 1.

2) k-var-stride Transition Sharing Algorithm: Similar to 1-stride DFAs, there are many transition sharing opportunities in a k-var-stride DFA. Consider two states s_0 and s_1 in a 1-stride DFA where s_0 defers to s_1. The deferment relationship implies that s_0 shares many common 1-stride transitions with s_1. In the k-var-stride DFA constructed from the 1-stride DFA, all k-var-stride transitions that begin with these common 1-stride transitions are also shared between s_0 and s_1. Furthermore, two transitions that do not begin with these common 1-stride transitions may still be shared between s_0 and s_1. For example, in the 1-stride DFA fragment in Fig. 9, although s_1 and s_2 do not share a common transition for character a, when we construct the 2-var-stride DFA, s_1 and s_2 share the same 2-stride transition on string aa that ends at state s_5.

To promote transition sharing among states in a k-var-stride DFA, we first need to decide on the deferment relationship among states. The ideal deferment relationship should be calculated based on the SRG of the final k-var-stride DFA. However, the k-var-stride DFA cannot be finalized before we need to compute the deferment relationship among states because the final k-var-stride DFA is subject to many factors such as available TCAM space. There are two approximation options for the final k-var-stride DFA for calculating the deferment relationship: the 1-stride DFA and the full k-stride DFA. We have tried both options in our experiments, and the difference in the resulting TCAM space is negligible. Thus, we simply use the deferment forest of the 1-stride DFA in computing the transition tables for the k-var-stride DFA.

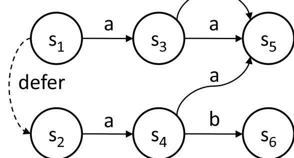


Fig. 9. s_1 and s_2 share transition aa

Second, for any two states s_1 and s_2 where s_1 defers to s_2, we need to compute s_1's k-var-stride transitions that are not shared with s_2 because those transitions will constitute s_1's k-var-stride transition table. Although this computation is trivial for 1-stride DFAs, this is a significant challenge for k-var-stride DFAs because each state has too many (256^k) k-var-stride transitions. The straightforward algorithm that enumerates all transitions has a time complexity of $O(|Q|^2|\Sigma|^k)$, which grows exponentially with k. We propose a dynamic programming algorithm with a time complexity of $O(|Q|^2|\Sigma|k)$, which grows linearly with k. Our key idea is that the non-shared transitions for a k-stride DFA can be quickly computed from the non-shared transitions of a (k-1)-var-stride DFA. For example, consider the two states s_1 and s_2 in Fig. 9 where s_1 defers to s_2. For character a, s_1 transits to s_3 while s_2 transits to s_4. Assuming that we have computed all (k-1)-var-stride transitions of s_3 that are not shared with the (k-1)-var-stride transitions of s_4, if we prepend all these (k-1)-var-stride transitions with character a, the resulting k-var-stride transitions of s_1 are all not shared with the k-var-stride transitions of s_2, and therefore should all be included in s_1's k-var-stride transition table. Formally, using $n(s_i, s_j, k)$ to denote the number of k-stride transitions of s_i that are not shared with s_j, our dynamic programming algorithm uses the following recursive relationship between $n(s_i, s_j, k)$ and $n(s_i, s_j, k-1)$:

$$n(s_i, s_j, 0) = \begin{cases} 0 & \text{if } s_i = s_j \\ 1 & \text{if } s_i \neq s_j \end{cases} \quad (1)$$

$$n(s_i, s_j, k) = \sum_{c \in \Sigma} n(\delta(s_i, c), \delta(s_j, c), k-1) \quad (2)$$

The above formulae assume that the intermediate states on the k-stride paths starting from s_i or s_j are all non-accepting. For state s_i, we stop increasing the stride length along a path whenever we encounter an accepting state on that path or on the corresponding path starting from s_j. The reason is similar to why we stop a consolidated path at an accepting state, but the reasoning is more subtle. Let p be the string that leads s_j to an accepting state. The key observation is that we know that any k-var-stride path that starts from s_j and begins with p ends at that accepting state. This means that s_i cannot exploit transition sharing on any strings that begin with p.

The above dynamic programming algorithm produces non-overlapping and incomplete transition tables that we compress using the 1-dimensional incomplete classifier minimization algorithm in [27].

D. Variable Striding Selection Algorithm

We now propose solutions for the third key challenge - which states should have their stride lengths increased and by how much, *i.e.*, how should we compute the transition function δ. Note that each state can independently choose its variable striding length as long as the final transition tables are composed together according to the deferment forest. This can be easily proven based on the way that we generate k-var-stride transition tables. For any two states s_1 and s_2 where s_1 defers to s_2, the way that we generate s_1's k-var-stride transition table is seemingly based on the assumption that s_2's transition table is also k-var-stride; actually, we do not have this assumption. For example, if we choose k-var-stride

$(2 \leq k)$ for s_1 and 1-stride for s_2, all strings from s_1 will be processed correctly; the only issue is that strings deferred to s_2 will process only one character.

We view this as a packing problem: given a TCAM capacity C, for each state s, we select a variable stride length value K_s, such that $\sum_{s \in Q} |\mathbb{T}(s, K_s)| \leq C$, where $\mathbb{T}(s, K_s)$ denotes the K_s-var-stride transition table of state s. This packing problem has a flavor of the knapsack problem, but an exact formulation of an optimization function is impossible without making assumptions about the input character distribution. We propose the following algorithm for finding a feasible δ that strives to maximize the minimum stride of any state. First, we use all the 1-stride tables as our initial selection. Second, for each j-var-stride $(j \geq 2)$ table t of state s, we create a tuple $(l, d, |t|)$ where l denotes variable stride length, d denotes the distance from state s to the root of the deferment tree that s belongs to, and $|t|$ denotes the number of entries in t. As stride length l increases, the individual table size $|t|$ may increase significantly, particularly for the complete tables of root states. To balance table sizes, we set limits on the maximum allowed table size for root states and non-root states. If a root state table exceeds the root state threshold when we create its j-var-stride table, we apply self-loop unrolling once to its $(j-1)$-var-stride table to produce a j-var-stride table. If a non-root state table exceeds the non-root state threshold when we create its j-var-stride table, we simply use its $(j-1)$-var-stride table as its j-var-stride table. Third, we sort the tables by these tuple values in increasing order first using l, then using d, then using $|t|$, and finally a pseudorandom coin flip to break ties. Fourth, we consider each table t in order. Let t' be the table for the same state s in the current selection. If replacing t' by t does not exceed our TCAM capacity C, we do the replacement.

VI. Handling REs with Large Counts

In this section, we consider the issue presented by REs with large counting constraints. As an example, consider the RE `.*a.{n}bc`; in this RE, *exactly* n *characters* must appear between some occurrence of `a` and `bc`. The issue is that each time an `a` is encountered in the input stream, the automaton must check whether `bc` appears exactly n characters later. For example, suppose $n = 3$ and the input stream is `aaaddbbc`. The second `a` leads to a match with the final `bc` in positions 7 and 8. The automaton must keep track of all the `a`s that appear and, in this case, check positions 5 and 6, 6 and 7, and 7 and 8 for the pattern `bc` to determine if the RE matches the input stream. In general, the corresponding DFA may have an exponential number of states in the length n of the counting constraint. The counting constraint might not be a single value n but rather a range $\{m, n\}$ which means the length must be at least m and not more than n.

We do not directly handle REs with large counting constraints. However, other researchers have developed automata models to handle REs with counting constraints such as the counting-DFA proposed by Becchi *et al.* [12]. We propose to handle REs with large counting constraints by using other automata such as a counting-DFA and then extending RegCAM to implement these other automata. We demonstrate this approach by showing how we can extend RegCAM to implement counting-DFA. We first briefly review counting-DFA and then show how to extend RegCAM to implement counting-DFA.

A. Counting-DFA

A counting-DFA uses extra scratch memory apart from the automaton to manage the count. As a result, the size of the automaton is independent of the value of the count in the counting constraint, making it practical to build the counting-DFA for REs with large counting constraints. Fig. 10 shows the counting-DFA example from [12] for the RE `.*a.{n}bc`. We briefly explain here how the counting-DFA works using this example; complete details can be found in [12].

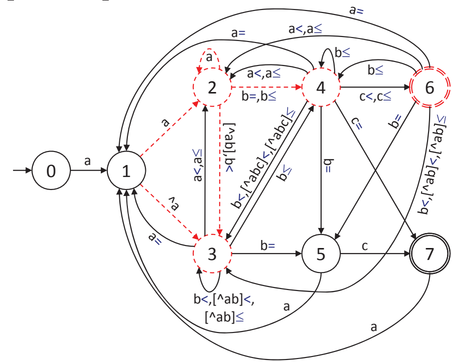


Fig. 10. Counting-DFA for RE `.*a.{n}bc`

The counting-DFA maintains a count variable *cnt* for each counting constraint; our example has only one constraint. The counting-DFA maintains all active instances of *cnt* in a queue with the *cnt* values strictly decreasing from front to back so that the largest *cnt* value is the first value. Some transitions create a new instance of *cnt* and are marked with red, dashed arrows in Fig. 10. Likewise, some states increment every active instance of *cnt* by 1; these are the red dashed states in Fig. 10. Such states are *counting* states; all other states are *standard* states.

The outgoing transitions of counting states depend on the current status of the *cnt* queue in addition to the input character. For a queue corresponding to the range $\{m, n\}$, there are three possibilities for its status: (1) The largest *cnt* value is strictly less than m, i.e. no value in the queue satisfies the condition. (2) The largest and smallest *cnt* values are both between m and n, i.e. all values in the queue satisfy the condition. (3) The largest instance of *cnt* is between m and n, but the smallest is strictly less than m, i.e. some values in the queue satisfy the condition and some do not.

Intuitively, counting states non-deterministically take one transition if the counter value is not satisfied, and take another if the counter value is satisfied. In the deterministic automaton, we have a set of counter values, so we may take either or both of these transitions depending on the values in the queue.

We denote queue status 1, 2 and 3 by the symbols '$<$', '$=$' and '$\leq$' respectively. In Fig. 10, the transition label for conditional transitions shows the input character concatenated with the required queue status. For example, the transition from DFA state 4 to DFA state 7 with label 'c=' will only be taken if the input character is 'c' and the queue status is 2.

As observed in [12] the *cnt* increment operation can be efficiently performed using a differential representation that only requires updating the first *cnt* instance in the queue. The checks needed to determine the queue status only involve testing the largest and smallest *cnt* instances and thus can be performed in constant time.

As proposed in [12], conditional transitions can be implemented as regular transitions by considering an expanded alphabet that is the combination of the input character and the *cnt* queue status. We can then compress the counting-DFA using deferred transitions to build an equivalent counting-D^{2}FA. Fig. 11 shows the SRG and and deferment forest constructed while building the counting-D^{2}FA for the counting-DFA in Fig. 10, and Fig. 12 shows the resulting counting-D^{2}FA.

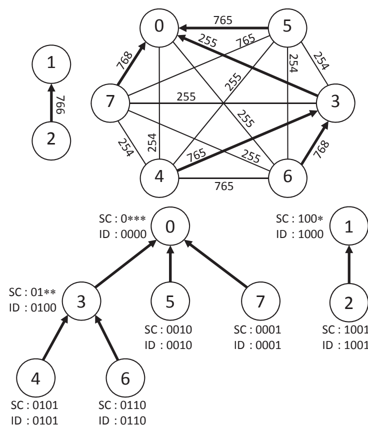


Fig. 11. SRG and deferment forest for the counting-DFA in Fig. 10

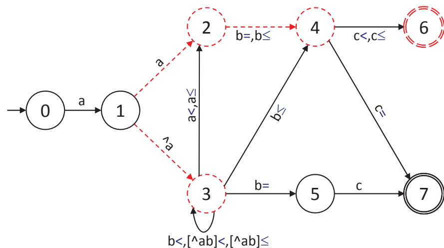


Fig. 12. Counting-D^{2}FA for RE `.*a.{n}bc`

B. Implementing Counting-D^{2}FA with RegCAM

Given REs with counting constraints, we first build the counting-DFA and the corresponding counting-D^{2}FA. Implementing counting-D^{2}FA has two parts: count queue management and automaton transitions. We propose implementing queue management in software. This involves allocating and deallocating *cnt* instances and determining the queue status, both of which can be done in constant time for each counting constraint in the RE set.

We implement the counting-D^{2}FA transition function in TCAM as follows taking into account two additional requirements. First, counting-D^{2}FA transitions can have an action associated with them; second, the transitions for counting states may depend on the *cnt* queue status. The first requirement can be handled by storing the action information in SRAM along with the next state. A simple scheme to handle conditional transitions is to implement counting states in software and only implement standard states in TCAM. With this scheme, we do not need to make any changes to RegCAM other than storing an extra bit in SRAM indicating whether the next state is stored in software or in TCAM.

We now describe an extension to RegCAM that implements all transitions in TCAM. The key is adding a two bit *cnt* queue status field in the TCAM for each count queue. We only need two bits because there are only three queue statuses. For unconditional transitions, each *cnt* queue status field is set to '$**$'. For conditional transitions, we again set *cnt* queue status to '$**$' if the transition does not depend on the given *cnt* queue. If it does depend on the given *cnt* queue, we encode *cnt* queue status as follows: '01' for state 1, '10' for state 2, and '11' for state 3. We use this encoding because it allows us to represent *cnt* queue statuses 1 and 3 with one TCAM entry '$*1$' and *cnt* queue statuses 2 and 3 with one TCAM entry '$1*$'. For example, in Fig. 13 which shows the TCAM rules for implementing the counting-D^{2}FA in Fig. 12, the two transitions out of state 2 in Fig. 12 are combined into the first rule in the TCAM table. When doing a transition lookup in the TCAM, the queue management module determines the status of each *cnt* queue and sets the appropriate bits in the lookup key according to this encoding.

	TCAM				SRAM	
	Source SC	Input char.	Queue state		Dest. ID	Action
state 2	1001	b	1*	→	0101	New cnt
state 1	100*	a	**	→	1001	New cnt
	100*	*	**	→	0100	New cnt
state 5	0010	c	**	→	0001	
state 4	0101	c	10	→	0001	
	0101	c	*1	→	0110	
state 3	01**	a	*1	→	1001	
	01**	b	10	→	0010	
	01**	b	11	→	0101	
	01**	*	*1	→	0100	
state 0	0***	a	**	→	1000	
	0***	*	**	→	0000	

Fig. 13. TCAM rules for counting-D^{2}FA for RE `.*a.{n}bc`

VII. Implementation and Modeling

We now describe some implementation issues associated with our TCAM based RE matching solution. First, the only hardware required to deploy our solution is the off-the-shelf TCAM (and its associated SRAM). Many deployed networking devices already have TCAMs, but these TCAMs are likely being used for other purposes. Thus, to deploy our solution on existing network devices, we would need to share an existing TCAM with another application. Alternatively, new networking devices can be designed with an additional dedicated TCAM chip.

Second, we describe how we update the TCAM when an RE set changes. First, we must compute a new DFA and its corresponding TCAM representation. For the moment, we recompute the TCAM representation from scratch, but we believe a better solution can be found and is something we plan to work on in the future. We report some timing results in our experimental section. Fortunately, this is an offline process during which time the DFA for the original RE set can still be

used. The second step is loading the new TCAM entries into TCAM. If we have a second TCAM to support updates, this rewrite can occur while the first TCAM chip is still processing packet flows. If not, RE matching must halt while the new entries are loaded. This step can be performed very quickly, so the delay will be very short. In contrast, updating FPGA circuitry takes significantly longer.

We have not developed a full implementation of our system. Instead, we have only developed the algorithms that would take an RE set and construct the associated TCAM entries. Thus, we can only estimate the throughput of our system using TCAM models. We use Agrawal and Sherwood's TCAM model [11] assuming that each TCAM chip is manufactured with a $0.18\mu m$ process to compute the estimated latency of a single TCAM lookup based on the number of TCAM entries searched. These model latencies are shown in Table I. We recognize that some processing must be done besides the TCAM lookup such as composing the next state ID with the next input character; however, because the TCAM lookup latency is much larger than any other operation, we focus only on this parameter when evaluating the potential throughput of our system.

Entries	TCAM Chip size (36-bit wide)	TCAM Chip size (72-bit wide)	Latency ns
1024	0.037 Mb	0.074 Mb	0.94
2048	0.074 Mb	0.147 Mb	1.10
4096	0.147 Mb	0.295 Mb	1.47
8192	0.295 Mb	0.590 Mb	1.84
16384	0.590 Mb	1.18 Mb	2.20
32768	1.18 Mb	2.36 Mb	2.57
65536	2.36 Mb	4.72 Mb	2.94
131072	4.72 Mb	9.44 Mb	3.37

TABLE I
TCAM SIZE AND LATENCY

VIII. EXPERIMENTAL RESULTS

In this section, we evaluate our TCAM-based RE matching solution on real-world RE sets focusing on two metrics: TCAM space and RE matching throughput.

A. Methodology

We obtained 4 proprietary RE sets, namely C7, C8, C10, and C613, from a large networking vendor, and 4 public RE sets, namely Snort24, Snort31, Snort34, and Bro217 from the authors of [3] We do report a slightly different number of states for Snort31, 20068 to 20052; this may be due to Becchi *et al.* making slight changes to their Regular Expression Processor that we used. Quoting Becchi *et al.* [3], "Snort rules have been filtered according to the headers ($HOME_NET, any, $EXTERNAL_NET, $HTTP_PORTS/any) and ($HOME_NET, any, 25, $HTTP_PORTS/any). In the experiments which follow, rules have been grouped so to obtain DFAs with reasonable size and, in parallel, have datasets with different characteristics in terms of number of wildcards, frequency of character ranges and so on." Of these 8 RE sets, the REs in C613 and Bro217 are all string matching REs, the REs in C7, C8, and C10 all contain wildcard closures '.*', and about 40% of the REs in Snort 24, Snort31, and Snort34 contain wildcard closures '.*'.

Finally, to test the scalability of our algorithms, we use one family of 34 REs from a recent public release of the Snort rules with headers ($EXTERNAL_NET, $HTTP_PORTS, $HOME_NET, any), most of which contain wildcard closures '.*'. We added REs one at a time until the number of DFA states reached 305,339. We name this family Scale.

We calculate TCAM space by multiplying the number of entries by the TCAM width: 36, 72, 144, 288, or 576 bits. For a given DFA, we compute a minimum width by summing the number of state ID bits required with the number of input bits required. In all cases, we needed at most 16 state ID bits. For 1-stride DFAs, we need exactly 8 input character bits, and for 7-var-stride DFAs, we need exactly 56 input character bits. We then calculate the TCAM width by rounding the minimum width up to the smallest larger legal TCAM width. For all our 1-stride DFAs, we use TCAM width 36. For all our 7-var-stride DFAs, we use TCAM width 72.

We estimate the potential throughput of our TCAM-based RE matching solution by using the model TCAM lookup speeds we computed in Section VII to determine how many TCAM lookups can be performed in a second for a given number of TCAM entries and then multiplying this number by the number of characters processed per TCAM lookup. With 1-stride TCAMs, the number of characters processed per lookup is 1. For 7-var-stride DFAs, we measure the average number of characters processed per lookup in a variety of input streams. We use Becchi *et al.*'s network traffic generator [28] to generate a variety of synthetic input streams. This traffic generator includes a parameter that models the probability of malicious traffic p_M. With probability p_M, the next character is chosen so that it leads away from the start state. With probability $(1 - p_M)$, the next character is chosen uniformly at random.

RE set	#states	TS tcam Mbits	TS #rows per state	TS thru Gbps	TS + TC2 tcam Mbits	TS + TC2 #rows per state	TS + TC2 thru Gbps	TS + TC4 tcam Mbits	TS + TC4 #rows per state	TS + TC4 thru Gbps
Bro217	6533	0.31	1.40	3.64	0.21	0.94	4.35	0.17	0.78	4.35
C613	11308	0.63	1.61	3.11	0.52	1.35	3.64	0.45	1.17	3.64
C10	14868	0.61	1.20	3.11	0.31	0.61	3.64	0.16	0.32	4.35
C7	24750	1.00	1.18	3.11	0.53	0.62	3.64	0.29	0.34	3.64
C8	3108	0.13	1.20	5.44	0.07	0.62	5.44	0.03	0.33	8.51
Snort24	13886	0.55	1.16	3.64	0.30	0.64	3.64	0.18	0.38	4.35
Snort31	20068	1.43	2.07	2.72	0.81	1.17	2.72	0.50	0.72	3.64
Snort34	13825	0.56	1.18	3.11	0.30	0.62	3.64	0.17	0.36	4.35

TABLE II
TCAM SIZE AND THROUGHPUT FOR 1-STRIDE DFAS

B. Results on 1-stride DFAs

Table II shows our experimental results on the 8 RE sets using 1-stride DFAs. We use TS to denote our transition sharing algorithm including both character bundling and shadow encoding. We use TC2 and TC4 to denote our table consolidation algorithm where we consolidate at most 2 and 4 transition tables together, respectively. For each RE set, we measure the number states in its 1-stride DFA, the resulting TCAM space in megabits, the average number of TCAM table entries per state, and the projected RE matching throughput; the number of TCAM entries is the number of states times the average number of entries per state. The TS column shows our results when we apply TS alone to each RE set. The TS+TC2 and TS+TC4 columns show our results when we apply both TS and TC under the consolidation limit of 2 and 4, respectively, to each RE set.

We draw the following conclusions from Table II. (1) *Our RE matching solution is extremely effective in saving TCAM space*. Using TS+TC4, the maximum TCAM size for the 8 RE sets is only 0.50 Mb, which is two orders of magnitude smaller than the current largest commercially available TCAM chip size of 72 Mb. More specifically, the number of TCAM entries per DFA state ranges between .32 and 1.17 when we use TC4. We require 16, 32, or 64 SRAM bits per TCAM entry for TS, TS+TC2, and TS+TC4, respectively as we need to record 1, 2, or 4 state 16 bit state IDs in each decision, respectively. (2) *Transition sharing alone is very effective*. With the transition sharing algorithm alone, the maximum TCAM size is only 1.43Mb for the 8 RE sets. Furthermore, we see a relatively tight range of TCAM entries per state of 1.16 to 2.07. Transition sharing works extremely well with all 8 RE sets including those with wildcard closures and those with primarily strings. (3) *Table consolidation is very effective*. On the 8 RE sets, adding TC2 to TS improves compression by an average of 41% (ranging from 16% to 49%) where the maximum possible is 50%. We measure improvement by computing $(TS - (TS + TC2))/TS)$. Replacing TC2 with TC4 improves compression by an average of 36% (ranging from 13% to 47%) where we measure improvement by computing $((TS + TC2) - (TS + TC4))/(TS + TC2)$. Here we do observe a difference in performance, though. For the two RE sets Bro217 and C613 that are primarily strings without table consolidation, the average improvements of using TC2 and TC4 are only 24% and 15%, respectively. For the remaining six RE sets that have many wildcard closures, the average improvements are 47% and 43%, respectively. The reason, as we touched on in Section IV-D, is how wildcard closure creates multiple deferment trees with almost identical structure. Thus wildcard closures, the prime source of state explosion, is particularly amenable to compression by table consolidation. In such cases, doubling our table consolidation limit does not greatly increase SRAM cost. Specifically, while the number of SRAM bits per TCAM entry doubles as we double the consolidation limit, the number of TCAM entries required almost halves! (4) *Our RE matching solution achieves high throughput with even 1-stride DFAs*. For the TS+TC4 algorithm, on the 8 RE sets, the average throughput is 4.60Gbps (ranging from 3.64Gbps to 8.51Gbps).

We use our Scale dataset to assess the scalability of our algorithms' performance focusing on the number of TCAM entries per DFA state. Fig. 14(a) shows the number of TCAM entries per state for TS, TS+TC2, and TS+TC4 for the Scale REs containing 26 REs (with DFA size 1275) to 34 REs (with DFA size 305,339). The DFA size roughly doubled for every RE added. In general, the number of TCAM entries per state is roughly constant and actually decreases with table consolidation. This is because table consolidation performs better as more REs with wildcard closures are added as there are more trees with similar structure in the deferment forest.

We now analyze running time. We ran our experiments on the Michigan State University High Performance Computing Center (HPCC). The HPCC has several clusters; most of our experiments were executed on the fastest cluster which has nodes that each have 2 quad-core Xeons running at 2.3GHz. The total RAM for each node is 8GB. Fig. 14(b) shows the compute time per state in milliseconds. The build times are the time per DFA state required to build the non-overlapping set of transitions (applying TS and TC); these increase linearly because these algorithms are quadratic in the number of DFA states. For our largest DFA Scale 34 with 305,339 states, the total time required for TS, TS+TC2, and TS+TC4 is 19.25 mins, 118.6 hrs, and 150.2 hrs, respectively. These times are cumulative; that is going from TS+TC2 to TS+TC4 requires an additional 31.6 hours. This table consolidation time is roughly one fourth of the first table consolidation time because the number of DFA states has been cut in half by the first table consolidation and table consolidation has a quadratic running time in the number of DFA states. The BW times are the time per DFA state required to minimize these transition tables using the Bitweaving algorithm in [27]; these times are roughly constant as Bitweaving depends on the size of the transition tables for each state and is not dependent on the size of the DFA. For our largest DFA Scale 34 with 305,339 states, the total Bitweaving optimization time on TS, TS+TC2, and TS+TC4 is 10 hrs, 5 hrs, and 2.5 hrs. These times are not cumulative and fall by a factor of 2 as each table consolidation step cuts the number of DFA states by a factor of 2.

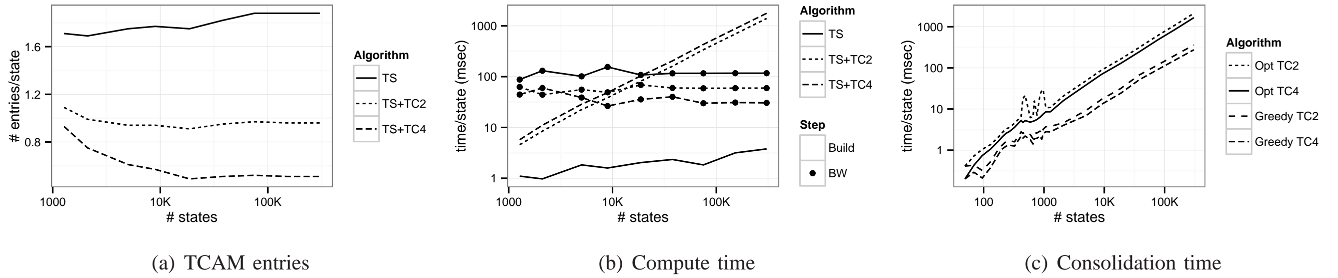
Fig. 14(c) shows the time required per state for the greedy and optimal consolidation algorithms on the Scale dataset. The greedy algorithm runs roughly 6 times faster than the optimal algorithm. The average increase in the number of resulting TCAM rules is around 4% for TC2 and around 9% for TC4.

The partially deferred algorithm given in section III-B4 always performs at least as well as the completely deferred minimization algorithm given in [27]. For the three Snort RE sets and C613, the partially deferred algorithm results in a reduction of 1, 2, 152, and 194 TCAM entries over the completely deferred algorithm. For the other RE sets, both algorithms perform equally well. The partially deferred algorithm is slower than the completely deferred algorithm because there are more unique decisions during minimization, so we use the completely deferred minimization algorithm for computing classifier sizes during consolidation, and we use the partially deferred minimization algorithm for generating the final TCAM classifiers for each state.

C. Results on 7-var-stride DFAs

We consider two implementations of variable striding assuming we have a 2.36 megabit TCAM with TCAM width 72 bits (32,768 entries). Using Table I, the latency of a lookup is 2.57 ns. Thus, the potential RE matching throughput of by a 7-var-stride DFA with average stride S is $8 \times S/.00000000257 = 3.11 \times S$ Gbps.

In our first implementation, we only use self-loop unrolling of root states in the deferment forest. Specifically, for each RE set, we first construct the 1-stride DFA using transition sharing. We then apply self-loop unrolling to each root state of the deferment forest to create a 7-var-stride transition table. Because of the linear increase in transition table size, we know that the resulting TCAM table will increase in size by at most a factor of 7. In all our experiments, the size never increased by more than a factor of 2.25, and the largest DFA (for C7)



(a) TCAM entries (b) Compute time (c) Consolidation time

Fig. 14. Per-DFA state statistics for Scale 26 through Scale 34.

required only 2.25 megabits. We can decrease the TCAM space by using table consolidation; this was very effective for all RE sets except the string matching RE sets Bro217 and C613. This was unnecessary since all self-loop unrolled tables fit within our available TCAM space.

Second, we apply full variable striding. We first create 1-stride DFAs using transition sharing and then apply variable striding with no table consolidation, table consolidation with 2-decision tables, and table consolidation with 4-decision tables. We use the best result that fits within the 2.36 megabit TCAM space. For the RE sets Bro217, C8, C613, Snort24 and Snort34, no table consolidation is used. For C10 and Snort31, we use table consolidation with 2-decision tables. For C7, we use table consolidation with 4-decision tables.

We now run both implementations of our 7-var-stride DFAs on traces of length 287484 to compute the average stride. For each RE set, we generate 4 traces using Becchi *et al.*'s trace generator tool using default values 35%, 55%, 75%, and 95% for the parameter p_M. These generate increasingly malicious traffic that is more likely to move away from the start state towards distant accept states of that DFA. We also generate a completely random string to model completely uniform traffic such as binary traffic patterns which we treat as $p_M = 0$.

We group the 8 RE sets into 3 groups: group (a) represents the two string matching RE sets Bro217 and C613; group (b) represents the three RE sets C7, C8, and C10 that contain all wildcard closures; group (c) represents the three RE sets Snort24, Snort31, and Snort34 that contain roughly 40% wildcard closures. Fig. 15 shows the average stride length and throughput for the three groups of RE sets according to the parameter p_M (the random string trace is $p_M = 0$).

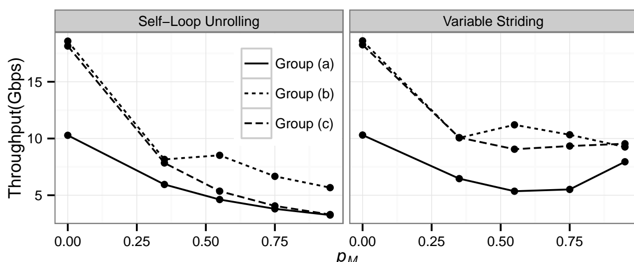


Fig. 15. The throughput and average stride length of RE sets.

We make the following observations. *(1) Self-loop unrolling is extremely effective on the uniform trace.* For the non string matching sets, it achieves an average stride length of 5.97 and 5.84 and RE matching throughput of 18.58 and 18.15 Gbps for groups (b) and (c), respectively. For the string matching sets in group (a), it achieves an average stride length of 3.30 and a resulting throughput of 10.29 Gbps. Even though only the root states are unrolled, self-loop unrolling works very well because the non-root states that defer most transitions to a root state will still benefit from that root state's unrolled self-loops. In particular, it is likely that there will be long stretches of the input stream that repeatedly return to a root state and take full advantage of the unrolled self-loops. *(2) The performance of self-loop unrolling does degrade steadily as p_M increases for all RE sets except those in group (b).* This occurs because as p_M increases, we are more likely to move away from any default root state. Thus, fewer transitions will be able to leverage the unrolled self-loops at root states. *(3) For the uniform trace, full variable striding does little to increase RE matching throughput.* Of course, for the non-string matching RE sets, there was little room for improvement. *(4) As p_M increases, full variable striding does significantly increase throughput, particularly for groups (b) and (c).* For example, for groups (b) and (c), the minimum average stride length is 2.91 for all values of p_M which leads to a minimum throughput of 9.06Gbps. Also, for all groups of RE sets, the average stride length for full variable striding is much higher than that for self-loop unrolling for large p_M. For example, when $p_M = 95\%$, full variable striding achieves average stride lengths of 2.55, 2.97, and 3.07 for groups (a), (b), and (c), respectively, whereas self-loop unrolling achieves average stride lengths of only 1.04, 1.83, and 1.06 for groups (a), (b), and (c), respectively.

These results indicate the following. First, self-loop unrolling is extremely effective at increasing throughput for random traffic traces. Second, other variable striding techniques can mitigate many of the effects of malicious traffic that lead away from the start state.

IX. Conclusions

We make four key contributions in this paper. (1) We propose the first TCAM-based RE matching solution. We prove that this unexplored direction not only works but also works well. (2) We propose two fundamental techniques, transition sharing and table consolidation, to minimize TCAM space. (3) We propose variable striding to speed up RE matching while carefully controlling the corresponding increase in memory.(4) We implemented our techniques and conducted experiments on real-world RE sets. We show that small TCAMs are capable of storing large DFAs. For example, in our experiments, we were able to store a DFA with 25K states in a 0.5Mb TCAM chip; most DFAs require at most 1 TCAM entry per DFA state. With variable striding, we show that a throughput of up to 18.6 Gbps is possible.

Appendix
Proofs

Lemma A.1: Given a valid shadow encoding for deferment forest F, for any state q and all states p in q's shadow, $ID(p) \in SC(q)$.

Proof: The deferment property implies that $SC(p) \subset SC(q)$. The self-matching property implies that $ID(p) \in SC(p)$. Thus, the result follows. ■

Lemma A.2: Given a valid shadow encoding for deferment forest F, for any state q and all states p not in q's shadow, $ID(p) \notin SC(q)$.

Proof: This follows immediately from the non-interception property. ■

Theorem A.1: Given a valid shadow encoding for a DFA M and deferment forest F and a TCAM classifier $\mathbb{C}$ that uses only binary state IDs for both source and destination state IDs in transition rules and that orders the state tables according to F, the TCAM classifier formed by replacing each source state ID in $\mathbb{C}$ with the corresponding shadow code and each destination state ID in $\mathbb{C}$ with the corresponding state ID will be equivalent to $\mathbb{C}$.

Proof: This follows from the first match nature of TCAMs, the state tables are ordered according to F, and Lemmas A.1 and A.2. ■

Theorem A.2: The state IDs and shadow codes generated by our Shadow Encoding algorithm satisfy the SEP.

Proof: We prove by induction on the height n of T. The base case where $n = 0$ is trivial since there is only a single node. For the inductive case, our inductive hypothesis is that the shadow codes and state IDs generated for T_i for $1 \leq i \leq c$ satisfy the SEP. Note, we do not process the root node s in this assumption. We now consider what happens when we process s. For each node $v \in T_i$ for $1 \leq i \leq c$, HCode(s_i) is prepended to the $SC(v)$ and $ID(v)$. Thus, the SEP still holds for all the nodes within T_i for $1 \leq i \leq c$. For any nodes p and q from different subtrees T_i and T_j, it follows that $ID(p) \notin SC(q)$ and $ID(q) \notin SC(p)$ because HCode(s_i) and HCode(s_j) are not prefixes of each other. Finally, for all nodes $v \in T$, $ID(v) \in SC(s)$ because $SC(s)$ contains only *'s. ■

We define a prefix shadow encoding as a shadow encoding where all shadow codes are prefix strings; that is, all *'s are after any 0's or 1's. For any prefix shadow encoding $\mathcal{E}$ of T, $\mathcal{E}_{T_i}$ denotes the subset of state ids and shadow codes for all $v \in T_i$. For any state id or shadow code X, ${}_p\lfloor X$ denotes the first p characters of X, and $X\rfloor_p$ denotes the last p characters of X. We define $\mathcal{E}_{T_i}\rfloor_p = \{X\rfloor_p \mid X \in \mathcal{E}_{T_i}\}$.

Lemma A.1: Consider a deferment tree T with a valid length x prefix shadow encoding $\mathcal{E}$ that satisfies the SEP. For every child $s_i, 1 \leq i \leq c$, of the root of T, there exist two values p_i and q_i such that:

1) $\forall i,\ 0 < p_i \leq x$, $0 \leq q_i < x$ and $p_i + q_i = x$.
2) $\forall i,\ \forall v \in T_i,\ {}_{p_i}\lfloor ID(v) = {}_{p_i}\lfloor SC(v) = {}_{p_i}\lfloor SC(s_i)$.
3) $\forall i,\ \mathcal{E}_{T_i}\rfloor_{q_i}$ is a valid prefix shadow encoding of T_i.
4) The set $\mathcal{E}_{ID} = \{{}_{p_i}\lfloor SC(s_i) \mid 1 \leq i \leq c\}$ is prefix free.

Proof: Since $\mathcal{E}$ is a prefix shadow encoding, for any child s_i, $SC(s_i)$ must be of the form $\{0,1\}^a\{*\}^{x-a}$. Let $p_i = a$ and $q_i = x - a$. Now, $p_i > 0$, otherwise we would have $SC(s_i) = \{*\}^x$, which is not possible as it would violate the deferment and non-interception properties. This proves (1). Also, since $\mathcal{E}$ satisfies the deferment and self-matching properties, we must have (2) and (3). And we must have (4) because of the non-interception property. ■

Theorem A.3: For any deferment tree T, our shadow encoding algorithm generates the shortest possible prefix shadow encoding that satisfies the SEP.

Proof: First, our shadow encoding algorithm generates a prefix shadow encoding. We prove by induction on the height n of T that it is the shortest possible prefix shadow encoding. The base case where $n = 0$ is trivial since the encoding for a single node is empty and thus optimal. For the inductive case, our inductive hypothesis is that the prefix shadow encoding for T_i for $1 \leq i \leq c$ is the shortest possible.

Let $\mathcal{E}$ be the prefix shadow encoding generated by our shadow encoding algorithm and $\mathcal{F}$ be the optimal prefix shadow encoding. Let l and m be the lengths of $\mathcal{E}$ and $\mathcal{F}$ respectively. Let g_i and w_i be the values defined by Lemma A.1 for $\mathcal{E}$. And let p_i and q_i be the corresponding values for $\mathcal{F}$. By the inductive hypothesis, we have $w_i \leq q_i$ for $1 \leq i \leq c$.

If $m < l$, this implies that the optimal shortest prefix shadow encoding for T must compute a better set of HCode equivalents for each child node s_i. In particular, we have that $\max_i(p_i + q_i) < \max_i(g_i + w_i)$. *i.e.* given equal or larger initial lengths, $\{q_i\}$, optimal prefix shadow encoding computes prefix-free codes $\mathcal{F}_{ID}$ for the children that are shorter than the prefix-free codes $\mathcal{E}_{ID}$ computed by the HCode subroutine. However, this is a contradiction, since the Huffman style encoding used to compute the HCodes minimizes the term $\max_i(g_i + w_i)$ [26]. Therefore, we must have $l \leq m$. ■

To formally specify our solution for the Partially Deferred Incomplete One-dimensional TCAM Minimization Problem(Definition 3.1), we introduce the following notation. Let $d_i, i \geq 1$ denote the actual decisions in a classifier. For a prefix $\mathcal{P} = \{0,1\}^k\{*\}^{b-k}$, we use $\underline{\mathcal{P}}$ to denote the prefix $\{0,1\}^k 0\{*\}^{b-k-1}$, and $\overline{\mathcal{P}}$ to denote the matching prefix $\{0,1\}^k 1\{*\}^{w-k-1}$. For a classifier f on $\{*\}^b$ and a prefix $\mathcal{P} \subseteq \{*\}^b$, $f_{\mathcal{P}}$ denotes a classifier on $\mathcal{P}$ that is equivalent to f (*i.e.* the subset of rules in f with predicates that are in $\mathcal{P}$). So $f = f_{\{*\}^b}$. For $i \geq 1$, $f_{\mathcal{P}}^{d_i}$ denotes a classifier on $\mathcal{P}$ that is equivalent to f and the decision of the last rule is d_i. Note that all packets in $\mathcal{P}$ are specified by such classifiers. Classifier $f_{\mathcal{P}}^{d_0}$ denotes the optimal classifier that is equivalent to f except that it possibly defers some packets within $\mathcal{D}$. We use $C(f_{\mathcal{P}}^{d_i})$ to denote the cost of the minimum classifier equivalent to $f_{\mathcal{P}}^{d_i}$ for $i \geq 0$. $[P(x)]$ evaluates to 1 when the statement inside is true; otherwise it evaluates to 0. We use x to represent some packet in the prefix $\mathcal{P}$ currently being considered.

Theorem A.4: Given a one-dimensional classifier f on $\{*\}^b$ and a subset $\mathcal{D} \subseteq \{*\}^b$ with a set of possible decisions $\{d_1, d_2, \ldots, d_z\}$ and a prefix $\mathcal{P} \subseteq \{*\}^b$, we have that $C(f_{\mathcal{P}}^{d_i})$ is calculated as follows:

(1) For $i > 0$

$$C(f_{\mathcal{P}}^{d_i}) = \begin{cases} 1 + [f(x) \neq d_i] & \text{if } f \text{ is consistent on } \mathcal{P} \\ \min_{j=1}^{z}(C(f_{\underline{\mathcal{P}}}^{d_j}) + C(f_{\overline{\mathcal{P}}}^{d_j}) - 1 + [j \neq i]) & \text{else} \end{cases}$$

(2) For $i = 0$:

$$C(f_{\mathcal{P}}^{d_0}) = \begin{cases} 0 & \text{if } \mathcal{P} \subseteq \mathcal{D} \\ \min(\min_{i=1}^{z}(C(f_{\mathcal{P}}^{d_i})), C(f_{\underline{\mathcal{P}}}^{d_0}) + C(f_{\overline{\mathcal{P}}}^{d_0})) & \text{else} \end{cases}$$

Proof: (1) When $i > 0$, we just build a minimum cost complete classifier. The recursion and the proof is exactly the same as given in [27] Theorem 4.1 (with decision weights $= 1$). (2) We consider two possibilities. Either the optimal classifier is a complete classifier or the optimal classifier is an incomplete classifier. If the optimal classifier is incomplete, we consider two cases. If the entire prefix $\mathcal{P}$ is contained with $\mathcal{D}$ and can be deferred, the minimum cost classifier is to defer all transitions and has cost 0. Otherwise, the minimum cost classifier for $\mathcal{P}$ would just be the minimum cost classifier for $\underline{\mathcal{P}}$ concatenated with the minimum cost classifier for $\overline{\mathcal{P}}$. This is represented by the last term in the minimization for case (2). The possibility that the optimal classifier is a complete classifier is handled by the first term in the first minimization for case (2). ■

REFERENCES

[1] R. Sommer and V. Paxson, “Enhancing bytelevel network intrusion detection signatures with context,” in *Proc. ACM Conf. on Computer and Communication Security*, 2003, pp. 262–271.

[2] B. C. Brodie, D. E. Taylor, and R. K. Cytron, “A scalable architecture for high-throughput regular-expression pattern matching,” *SIGARCH Computer Architecture News*, 2006.

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

[4] ——, “Efficient regular expression evaluation: Theory to practice,” in *Proc. ACM/IEEE ANCS*, 2008.

[5] 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.

[6] S. Kumar, J. Turner, and J. Williams, “Advanced algorithms for fast and scalable deep packet inspection,” in *Proc. ANCS*, 2006, pp. 81–92.

[7] M. Becchi and S. Cadambi, “Memory-efficient regular expression search using state merging,” in *Proc. INFOCOM*. IEEE, 2007.

[8] S. Kong, R. Smith, and C. Estan, “Efficient signature matching with multiple alphabet compression tables,” in ACM *SecureComm*, 2008.

[9] M. Becchi and P. Crowley, “A hybrid finite automaton for practical deep packet inspection,” in *Proc. CoNext*, 2007.

[10] J. E. Hopcroft, *The Theory of Machines and Computations*. Academic Press, 1971, ch. An nlogn algorithm for minimizing the states in a finite automaton, pp. 189–196.

[11] 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*, 2006, pp. 120– 129.

[12] M. Becchi and P. Crowley, “Extending finite automata to efficiently match perl-compatible regular expressions,” in *Proc. CoNEXT*, 2008.

[13] A. V. Aho and M. J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[14] F. Yu, R. H. Katz, and T. V. Lakshman, “Gigabit rate packet pattern-matching using TCAM,” in *Proc. 12th IEEE Int. Conf. on Network Protocols (ICNP)*, 2004, pp. 174–183.

[15] 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.

[16] M. Alicherry, M. Muthuprasanna, and V. Kumar, “High speed pattern matching for network IDS/IPS,” in *ICNP*, 2006, pp. 187–196.

[17] R. Smith, C. Estan, and S. Jha, “XFA: Faster signature matching with extended automata,” in *Proc. IEEE Symposium on Security and Privacy*, 2008, pp. 187–201.

[18] A. Bremler-Barr, D. Hay, and Y. Koral, “CompactDFA: generic state machine compression for scalable pattern matching,” in *IEEE INFOCOM*, 2010, pp. 659–667.

[19] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo, “An efficient regular expressions compression algorithm from a new perspective,” in *IEEE INFOCOM*, 2011, pp. 2129–2137.

[20] 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)*, 2006, pp. 93–102.

[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*, 2008, pp. 207–218.

[22] 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.

[23] 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.

[24] S. Suri, T. Sandholm, and P. Warkhede, “Compressing two-dimensional routing tables,” *Algorithmica*, vol. 35, pp. 287–300, 2003.

[25] C. R. Meiners, A. X. Liu, and E. Torng, “TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs,” in *Proc. 15th IEEE Conf. on Network Protocols*, October 2007, pp. 266–275.

[26] D. E. Knuth, “Huffman’s algorithm via algebra,” *Journal of Combinatorial Theory, Series A*, vol. 32, no. 2, pp. 216 – 224, 1982.

[27] C. R. Meiners, A. X. Liu, and E. Torng, “Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs,” in *Proc. 17th IEEE Conf. on Network Protocols (ICNP)*, October 2009.

[28] M. Becchi, M. Franklin, and P. Crowley, “A workload for evaluating deep packet inspection architectures,” in *Proc. IEEE IISWC*, 2008.

Chad R. Meiners Chad Meiners received his Ph.D. in Computer Science at Michigan State University in 2009. He is currently a Member of Technical Staff with MIT Lincoln Laboratory, Lexington, MA. His research interests include networking, algorithms, and security.

Jignesh Patel Jignesh Patel is currently a Ph.D. student in the Department of Computer Science and Engineering at Michigan State University. His research interests include algorithms, networking, and security.

Eric Norige is working towards his Ph.D. degree in computer science from Michigan State University. He is currently working for NetSpeed Systems, a Silicon Valley startup in the area of NoC IP. His research interests are algorithms, optimization, and security.

Alex X. Liu received his Ph.D. degree in computer science from the University of Texas at Austin in 2006. He received the IEEE & IFIP William C. Carter Award in 2004 and an NSF CAREER award in 2009. He received the Withrow Distinguished Scholar Award in 2011 at Michigan State University. He is an Associate Editor of IEEE/ACM Transactions on Networking. He received Best Paper Awards from ICNP-2012, SRDS-2012, and LISA-2010. His research interests focus on networking and security.

Eric Torng received his Ph.D. degree in computer science from Stanford University in 1994. He is currently an associate professor and graduate director in the Department of Computer Science and Engineering at Michigan State University. He received an NSF CAREER award in 1997. His research interests include algorithms, scheduling, and networking.