Fast Regular Expression Matching using Small TCAMs for Network Intrusion Detection and Prevention Systems

Chad R. Meiners Jignesh Patel Eric Norige Eric Torng Alex X. Liu
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824-1226, U.S.A.
{*meinersc, patelji1, norigeer, torng, alexliu*}*@cse.msu.edu*

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 Memories (TCAMs), which are off-the-shelf chips and have been widely deployed in modern networking devices for 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 DFAs and achieve potentially high RE matching throughtput. For space, we were able to store each of the corresponding 8 DFAs with as many as 25,000 states in a 0.59Mb TCAM chip where the number of TCAM bits required per DFA state were 12, 12, 12, 13, 14, 26, 28, and 42. Using a different TCAM encoding scheme that facilitates processing multiple characters per transition, we were able to achieve potential RE matching throughputs of between 10 and 19 Gbps for each of the 8 DFAs using only a single 2.36 Mb TCAM chip.

1 Introduction

1.1 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 [27]. 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 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, $\delta\colon \Sigma \times Q \to Q$ 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 DFA with the minimum number of states. 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.

Prior RE matching algorithms are either software-based [4, 6, 7, 12, 16, 18, 19] or FPGA-based [5, 7, 13, 14, 22, 24, 29]. 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 [18].

1.2 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 [15]. 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 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.

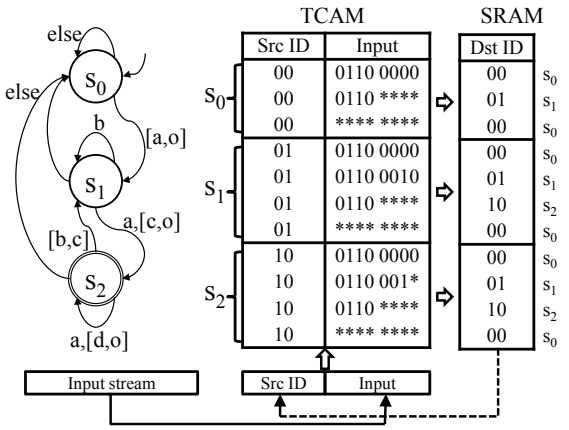

Figure 1: A DFA with its TCAM table

Advantages of TCAM-based RE Matching 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 [6], 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 [1] and the resulting required TCAM sizes for the 8 RE sets, we show that it may be possible to achieve throughputs 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.

Technical Challenges There are two key technical challenges in TCAM-based RE matching. The first is encoding a large DFA in a small TCAM. Directly encoding a DFA in a TCAM using one TCAM entry per transition will lead to a prohibitive amount of TCAM space. For example, consider a DFA with 25000 states that consumes one 8 bit character per transition. We would need a total of $140.38\,\text{Mb}$ $(= 25000 \times 2^8 \times (8 + \lceil \log 25000 \rceil))$. 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 second challenge is improving RE matching speed and thus throughput. 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 use the novel idea of *variable striding*.

Key Idea 1 - Transition Sharing The basic idea 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.

Key Idea 2 - Table Consolidation The basic idea 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.

Key Idea 3 - Variable Striding The basic idea is to store transitions with a variety of strides in the TCAM 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 lookup table.

The rest of this paper proceeds as follows. We review related work in Section 2. In Sections 3, 4, and 5, we describe transition sharing, table consolidation, and variable striding, respectively. We present implementation issues, experimental results, and conclusions in Sections 6, 7, and 8, respectively.

2 Related Work

In the past, deep packet inspection typically used string matching (often called pattern matching) as a core operator; string matching solutions have been extensively studied [2, 3, 28, 30, 32, 33, 35]). TCAM-based solutions have been proposed for string matching, but they do not generalize to RE matching because they only deal with independent strings [3, 30, 35].

Today deep packet inspection often uses RE matching as a core operator because strings are no longer adequate to precisely describe attack signatures [25, 27]. 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 [6, 7, 12, 16] and default transitions that exploit state redundancy [4, 6, 18, 19]. 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 [16]. 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 [3, 11]. 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. 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 [5, 26, 34]. 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 [8,17,25]. 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 applyied 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) [5, 7, 13, 14, 22, 24,29] 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 that is processed 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.

There has been work [7, 12] on creating multi-stride DFAs and NFAs. This work primarily applies to FPGA NFA implementations since multiple character SRAM based DFAs have only been evaluated for a small number of REs. The ability to increase stride has been limited by the constraint that all transitions must be increased in stride; this leads to excessive memory explosion for strides larger than 2. With variable striding, we increase stride selectively on a state by state basis. Alicherry *et al.* have explored variable striding for TCAM-based string matching solutions [3] but not for DFAs that apply to arbitrary RE sets.

3 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.

3.1 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 [20, 31]. (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.

3.2 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.

3.2.1 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

Figure 2: TCAM table with shadow encoding

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

Our shadow encoding technique builds upon prior work with default transitions [4, 6, 18, 19] by exploiting the same state redundancy observation and using their

concepts of default transitions and Delayed input DFAs (D^2FA). 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 techniques is speed. Specifically, shadow encoding incurs no delay while prior default transition techniques incur significant delay because a DFA may have to traverse multiple default transitions before consuming an input character.

3.2.2 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^2FA that were originally defined by Kumar *et al.* [18] and subsequently refined [4, 6, 19].

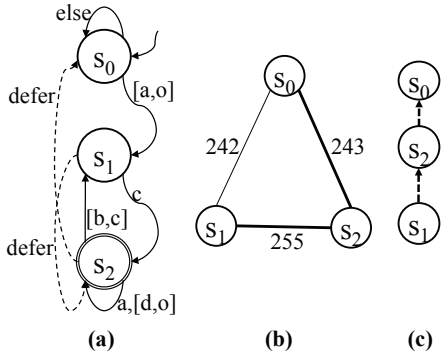

Figure 3: D^2FA, SRG, and deferment tree

A D^2FA 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^2FA. In a legal D^2FA, 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 and then using the partial order defined by the deferment forest. Specifically, if there is a directed path from state p to state q in the deferment forest, 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. We use the partial order of a deferment forest to determine the order of state transition tables in the TCAM. Specifically, state q's transition table must be placed after the transition tables of all states in state q's shadow.

We compute a deferment forest that minimizes the TCAM representation of the resulting D^2FA as follows. Our algorithm builds upon algorithms from prior work [4, 6, 18, 19], 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 5 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^2FA [6]. 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 when we use table consolidation.

The algorithm for constructing deferment forests consists of four steps. First, we construct a Space Reduction Graph (SRG), which was proposed in [18], 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. We justify this step based on the following observations. A key property of SRGs that we observed in our experiments is that the weight distribution is bimodal: an edge weight is typically either very small (< 10) or very large (> 180). If we use these low weight edges for default transitions, the resulting TCAM often has more entries. Plus, we get fewer deferment trees which hinders our table consolidation technique (Section 4). 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.

We make the following key observation about the root states in our deferment trees. 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 can have at most one self-looping state, and it is clearly the root state. We choose self-looping states as roots to improve the effectiveness of variable striding which we describe in Section 5. Intuitively, we have a very space-efficient method, self-loop unrolling, for increasing the stride of self-looping root states. The resulting increase in stride applies to all states that defer transitions to this self-looping root state.

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. If we still have a

tie, we favor edges by the total number of edges in the current spanning tree that both endpoints are connected to prioritize nodes that are already well connected.

3.2.3 Choosing Transitions

For a given DFA and a corresponding deferment forest, we construct a D^{2}FA as follows. If state p has a default transition to state q, we remove any transitions that are common to both p's transition table and q's transition table from p's transition table. We denote the default transition in the D^{2}FA with a dashed arrow labeled with defer. Fig. 3(a) shows the D^{2}FA for the DFA in Fig. 1 given the corresponding deferment forest (a deferment tree in this case) in Figure 3(c). We now compute the TCAM entries for each transition table.

(1) For each state, enumerate all individual transition rules except the deferred transitions. For each transition rule, the predicate is the label of the transition and the decision is the *state ID* of the destination state. For now, we just ensure each state has a unique state ID. Thus, we get an incomplete 1-dimensional classifier for each state. (2) For each state, we minimize its transition table using the 1-dimensional incomplete classifier minimization algorithm in [21]. This algorithm works by first adding a default rule with a unique decision that has weight larger than the size of the domain, then applying the weighted one-dimensional TCAM minimization algorithm in [20] to the resulting complete classifier, and finally remove the default rule, which is guaranteed to remain the default rule in the minimal complete classifier due to its huge weight. In our solution, the character bundling technique is used in this step. We also consider some optimizations where we specify some deferred transitions to reduce the total number of TCAM entries. For example, the second entry in s_2's table in Fig. 2 is actually a deferred transition to state s_0's table, but not using it would result in 4 TCAM entries to specify the transitions that s_2 does not share with s_0.

3.2.4 Shadow Encoding Algorithm

To ensure that proper sharing of transitions occurs, we need to encode the source state IDs of the TCAM entries according to the following shadow encoding scheme. Each state is assigned a binary *state ID* and a ternary *shadow code*. State IDs are used in the decisions of transition rules. Shadow codes are used in the source state ID field of transition rules. In a valid assignment, every state ID and shadow code must have the same number of bits, which we call the *shadow length* of the assignment. For each state p, we use $ID(p)$ and $SC(p)$ to denote the state ID and shadow code of p. A valid assignment of state IDs and shadow codes for a deferment forest must satisfy the following four shadow encoding properties:

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 tree) 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 $ID(p) \in SC(q)$.

Intuitively, q's shadow code must include the state ID of all states in q's shadow and cannot include the state ID of any states not in q's shadow.

We give an algorithm for computing a valid assignment of state IDs and shadow codes for each state given 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 in a bottom-up fashion. For each node 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, which we call *HCode*. The HCode provides a unique signature that uniquely distinguishes each of the $n+1$ nodes from each other while satisfying the four required shadow code properties. One option would be to simply use $\lg(n+1)$ bits and assign each node a binary number from 0 to n. However, to minimize the shadow code length $W(v)$, we use a Huffman coding style algorithm instead 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, initialized with $n+1$ elements, one for v and each of its children, that is ordered by node weight. While the priority queue has more than one element, we remove the two elements x and y with lowest weight from the priority queue, create a new

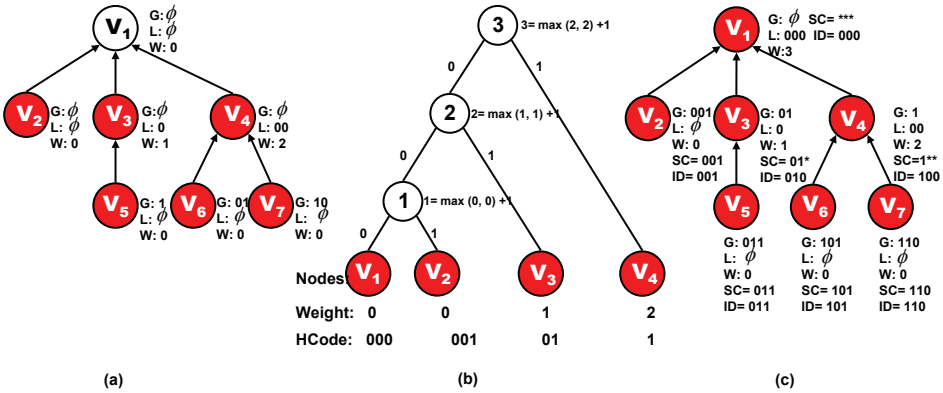

Figure 4: Shadow encoding example

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 the priority queue. When there is only a single element in the priority queue, the binary encoding tree 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 assign each node a state ID and a shadow code. First, we set the shadow length to be k, the weight of the root node of DT. We use $\{*\}^m$ to denote a ternary string with m number of *'s and $\{0\}^m$ to denote a binary string with m number of 0's. For each node v, we compute v's state ID and shadow code as follows: $ID(v) = G(v)|L(v)|\{0\}^{k-\#G(v)-\#L(v)}, SC(v) = G(v)|\{*\}^{k-\#G(v)}$. 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.

Experimentally, we found that our shadow encoding algorithm is effective at minimizing shadow length. No DFA had a shadow length larger than $\lceil \log_2 |Q| \rceil + 3$, and $\lceil \log_2 |Q| \rceil$ is the minimum possible shadow length.

4 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*.

4.1 Observations

Table consolidation is based 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

Consolidated Src Table ID	Input Character
0	0110 0000
0	0110 0010
0	0110 0011
0	0110 ****
0	**** ****

SRAM

Column ID		
00	01	10
s_0	s_0	s_0
s_1	s_1	s_1
s_1	s_2	s_1
s_1	s_2	s_2
s_0	s_0	s_0

Figure 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.

4.2 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. 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. We assign state s_1 column ID 0 and state s_2 column ID 1. This consolidated table will be assigned a common table ID X. Thus, we encode s_1 as $X@0$ and s_2 as $X@1$.

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 [20, 31]. If the table is incomplete, we minimize it using the 1-dimensional incomplete classifier minimization algorithm in [21]. 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.

4.3 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.

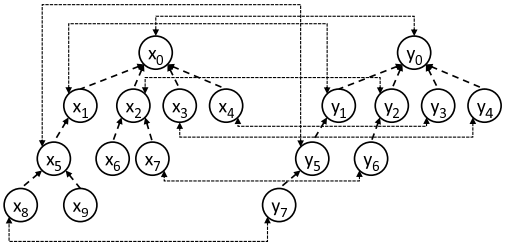

Figure 6: Consolidating two trees

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. Our algorithm proceeds as follows. 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 $k > 2$.

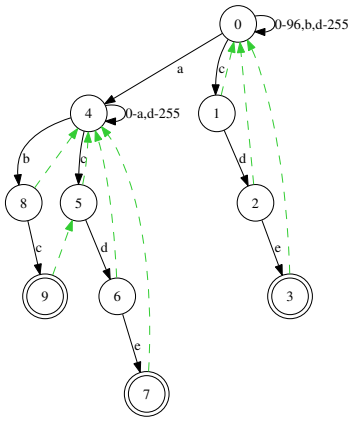

Figure 7: D^{2}FA for {a.*bc, cde}

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.

4.4 Effectiveness of Table Consolidation

We now explain why table consolidation works well on real-world RE sets. 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 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.

5 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 ($1 \leq j \leq k$) 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.

5.1 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

SRC	Input
s_0	0110 0000 **** **** **** ****
s_0	0110 **** **** **** **** ****
s_0	**** **** 0110 0000 **** ****
s_0	**** **** 0110 **** **** ****
s_0	**** **** **** **** 0110 0000
s_0	**** **** **** **** 0110 ****
s_0	**** **** **** **** **** ****

SRAM

DEC : Stride
s_0 : 1
s_1 : 1
s_0 : 2
s_1 : 2
s_0 : 3
s_1 : 3
s_0 : 3

Figure 8: 3-var-stride transition table for s_0

5.2 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.

5.3 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: k-var-stride transition sharing and self-loop unrolling.

5.3.1 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.

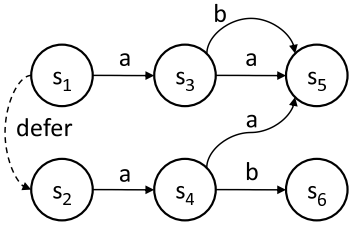

Figure 9: s_1 and s_2 share transition aa

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.

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 and incomplete transition tables that we compress using the 1-dimensional incomplete classifier minimization algorithm in [21].

5.3.2 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.

5.4 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.

6 Implementation and Modeling

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 1: TCAM size in Mb and Latency in ns

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 [1] 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 1. 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.

7 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.

7.1 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 [6] (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.* [6], "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 6 to determine how many TCAM lookups can be performed in a second

RE set	# states	TS			TS + TC2			TS + TC4		
		TCAM megabits	#Entries per state	throughput Gbps	TCAM megabits	#Entries per state	thru Gbps	TCAM megabits	#Entries per state	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 2: TCAM size and throughput for 1-stride DFAs

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 [9] 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.

7.2 Results on 1-stride DFAs

Table 2 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 2. (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 4.4, 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. 10(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

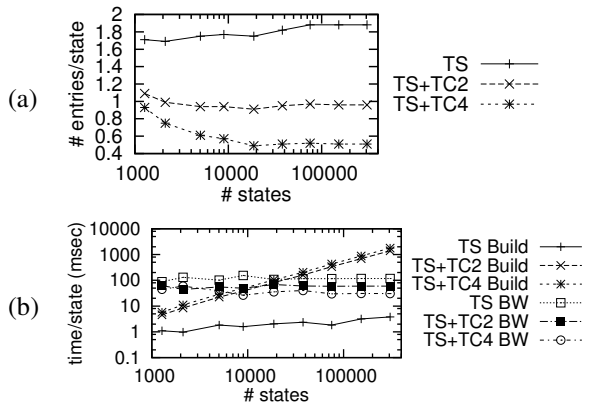

Figure 10: TCAM entries per DFA state (a) and compute time per DFA state (b) for Scale 26 through Scale 34.

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. 10(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 [21]; 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.

7.3 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 1, 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. In all cases, the increase in size due to self-loop unrolling is tiny. The bigger issue was that the TCAM width doubled from 36 bits to 72 bits. 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 only necessary for Snort31. All other self-loop unrolled tables fit within our available TCAM space.

Second, we apply full variable striding. Specifically, 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. 11 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 $p_M = 0$).

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 throughputs

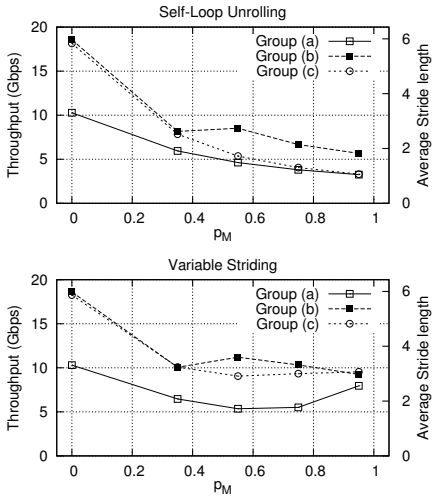

Figure 11: The throughput and average stride length of RE sets.

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.

8 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.

References

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

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 1975.

[3] M. Alicherry, M. Muthuprasanna, and V. Kumar. High speed pattern matching for network IDS/IPS. In *Proc. ICNP*, 2006.

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

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

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

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

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

[9] M. Becchi, M. Franklin, and P. Crowley. A workload for evaluating deep packet inspection architectures. In *Proc. IEEE IISWC*, 2008.

[10] M. Becchi, C. Wiseman, and P. Crowley. Evaluating regular expression matching engines on network and general purpose processors. In *Proc. ANCS*, 2009.

[11] A. Bremler-Bar, D. Hay, and Y. Koral. CompactDFA: generic state machine compression for scalable pattern matching In *Proc. INFOCOM*, 2010.

[12] 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.

[13] C. R. Clark and D. E. Schimmel. Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Proc. FPL*, pages 956–959, 2003.

[14] C. R. Clark and D. E. Schimmel. Scalable pattern matching for high speed networks. In *FCCM* 2004.

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

[16] S. Kong, R. Smith, and C. Estan. Efficient signature matching with multiple alphabet compression tables. In *Proc. ACM SecureComm*, Article 1, 2008.

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

[18] 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.

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

[20] C. R. Meiners, A. X. Liu, and E. Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. In *Proc. ICNP*, 2007.

[21] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. In *Proc. ICNP*, 2009.

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

[23] J. Moscola, J. Lockwood, R. P. Loui, and M. Pachos. Implementation of a content-scanning module for an internet firewall. In *FCCM*, 2003.

[24] R. Sidhu and V. K. Prasanna. Fast regular expression matching using fpgas. In *FCCM*, 2001.

[25] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proc. Symposium on Security and Privacy*, 2008.

[26] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *Proc. SIGCOMM*, pages 207–218, 2008.

[27] R. Sommer and V. Paxson. Enhancing bytelevel network intrusion detection signatures with context. In *Proc. ACM CCS*, pages 262–271, 2003.

[28] I. Sourdis and D. Pnevmatikatos. Pnevmatikatos: Fast, large-scale string match for a 10gbps fpga-based network intrusion detection system. In *Proc. FCCM*, pages 880–889, 2003.

[29] I. Sourdis and D. Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *Proc. FCCM*, 2004.

[30] 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.

[31] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 2003.

[32] L. Tan and T. Sherwood. A high throughput string matching architecture for intrusion detection and prevention. In *Proc. ISCA*, 2005.

[33] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. In *Proc. IEEE Infocom*, pages 333–340, 2004.

[34] 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. ANCS*, 2006.

[35] F. Yu, R. H. Katz, and T. V. Lakshman. Gigabit rate packet pattern-matching using TCAM. In *Proc. ICNP*, 2004.