A Ternary Unification Framework for Optimizing TCAM-Based Packet Classification Systems

Eric Norige Alex X. Liu Eric Torng
Dept. of Computer Science and Engineering, Michigan State University
East Lansing, Michigan, USA
{norigeer, alexliu, torng}@cse.msu.edu

ABSTRACT

Packet classification is the key mechanism for enabling many networking and security services. Ternary Content Addressable Memory (TCAM) has been the industrial standard for implementing high-speed packet classification because of its constant classification time. However, TCAM chips have small capacity, high power consumption, high heat generation, and large area size. This paper focuses on the TCAM-based Classifier Compression problem: given a classifier C, we want to construct the smallest possible list of TCAM entries T that implement C. In this paper, we propose the Ternary Unification Framework (TUF) for this compression problem and three concrete compression algorithms within this framework. The framework allows us to find more optimization opportunities and design new TCAM-based classifier compression algorithms. Our experimental results show that the TUF can speed up the prior algorithm TCAM Razor by twenty times or more and leads to new algorithms that improve compression performance over prior algorithms by an average of 13.7% on our largest real life classifiers.

1. INTRODUCTION

1.1 Background and Motivation

Packet classification is the core mechanism that enables many networking devices, such as routers and firewalls, to perform services such as packet filtering, virtual private networks (VPNs), network address translation (NAT), quality of service (QoS), load balancing, traffic accounting and monitoring, differentiated services (Diffserv), etc. A packet classifier is a function that maps packets to a set of decisions, allowing packets to be classified according to some criteria. These classifiers are normally written as a sequence of rules where each rule consists of a *predicate* that specifies what packets match the rule and a *decision* for packets that match the rule. For convenience in specifying rules, more than one predicate is allowed to match a given packet. In such cases, the decision used comes from the first rule in the sequence whose predicate matches the packet. Table 1 shows a simplified example classifier with three rules. This packet classifier's predicates examine 5 fields of the packet, and has decision set {accept, discard}, as might be used on a firewall. Note that 1.2.0.0/16 denotes the IP prefix 1.2.*.*, which represents the set of IP addresses from 1.2.0.0 to 1.2.255.255. The final rule, r_3, is the default rule; it matches all packets, guaranteeing a decision for each packet.

Rule	Source IP	Dest. IP	Source Port	Dest. Port	Protocol	Action
r_1	1.2.0.0/16	192.168.0.1	[1,65534]	[1,65534]	TCP	accept
r_2	*	*	*	6881	TCP	discard
r_3	*	*	*	*	*	accept

Table 1: An example packet classifier

Packet classification is often the performance bottleneck for Internet routers as they need to classify every packet. Current generation fiber optic links can operate at over 40 Gb/s, or 12.5 ns per packet for processing. With the explosive growth of Internet-based applications, efficient packet classification becomes more and more critical. The de facto industry standard for high speed packet classification uses Ternary Content Addressable Memory (TCAM). Since 2003, most packet classification devices shipped were TCAM-based [1]. Although a large body of work has been done on software-based packet classification ([23]), because of the parallel search capability, TCAM remains the fastest and most scalable solution for packet classification because it is constant time regardless of the number of rules.

The high speed that TCAM offers for packet classification does not come for free. First, a TCAM chip consumes a large amount of power and generates lots of heat. This is because every occupied TCAM entry is tested on every query. The power consumption of a TCAM chip is about 1.85 Watts per Mb [3], which is roughly 30 times larger than an SRAM chip of the same size [18]. Second, TCAM chips have *large die area* on line cards - 6 times (or more) board space than an equivalent capacity SRAM chip [10]. Area efficiency is a critical issue for networking devices. Third, TCAMs are *expensive* - often costing more than network processors [11]. This high price is mainly due to the large die area, not their market size [10]. Finally, TCAM chips have small capacities. Currently, the largest TCAM chip has 72 megabits (Mb). TCAM chip size has been slow to grow due to their extremely high circuit density. The TCAM industry has not been able to follow Moore's law in the past, and it is unlikely to do so in the future. In practice, smaller TCAM chips are commonly used due to lower power consumption, heat generation, board space, and cost. For example, TCAM chips are often restricted to at most 10% of an entire board's power budget; thus, even a 36 Mb TCAM may not be deployable on many routers due to power consumption reasons.

Furthermore, the well known range expansion problem exacerbates the problem of limited capacity TCAMs. That is,

[1]Alex X. Liu is the corresponding author of this paper. Email: alexliu@cse.msu.edu. Tel: +1 (517) 353-5152.

converting packet classification rules to ternary format typically results in a much larger number of TCAM entries. In a typical packet classification rule, the three fields of source and destination IP addresses and protocol type are specified as prefixes which are easily written as ternary strings. However, the source and destination port fields are specified in ranges (*i.e.*, integer intervals), which may need to be expanded to one or more prefixes before being stored in a TCAM. This can lead to a significant increase in the number of TCAM entries needed to encode a rule. For example, 30 prefixes are needed to represent the single range $[1, 65534]$, so $30 \times 30 = 900$ TCAM entries are required to represent the single rule r_1 in Table 1.

1.2 Problem Statement

Formally, a classifier C is a function from binary strings of length w to some decision set D; that is, $C : \{0,1\}^w \rightarrow D$. In firewalls, the classifier commonly takes 104 bits of packet header and returns either *Accept* or *Reject*. A TCAM classifier T is an ordered list of rules $r_1, r_2, ..., r_n$. Each rule has a ternary predicate $p_i \in \{0,1,*\}^w$ and a decision $d_i \in D$. A ternary predicate p matches a binary string b if all non-star entries in p match the corresponding entries in b, that is,

$$\bigwedge_{i=1}^{w} (p_i = b_i \vee p_i = *).$$

The decision of a TCAM classifier T for an input $p \in \{0,1\}^w$ $T(p)$ is the decision of the first matching rule in T; that is, TCAMs use first-match semantics. A TCAM classifier T implements a classifier C if $T(p) = C(p)$ for all $p \in \{0,1\}^w$, that is, if all packets are classified the same by both.

This paper focuses on the fundamental *TCAM Classifier Compression* problem: given a classifier C, construct a minimum size TCAM classifier T that implements C. TCAM classifier compression helps to address all the aforementioned TCAM limitations - small sizes, high power consumption, high heat generation, and large die area. Note that even for the same TCAM chip, storing fewer rules will consume less power and generate less heat because the unoccupied entries can be disabled in blocks.

1.3 Limitations of Prior Art

Prior TCAM Classifier Compression algorithms fall into two categories: list based algorithms and tree based algorithms. List based algorithms (*e.g.*, Bit Weaving [18], Redundancy Removal [13], Dong's scheme [8]) take as input a list of TCAM rules and search for optimization opportunities between rules. These algorithms are sensitive to the representation of their input ruleset which means they can produce very different results for equivalent inputs. Tree based algorithms (*e.g.*, TCAM Razor [14] and Ternary Razor [18]) convert the input rules into a decision tree and search for optimization opportunities in the tree. Tree based algorithms typically produce better results because they can find optimization opportunities based on the underlying classifier without being misled by a specific instantiation of that classifier. A key limitation of prior tree based algorithms is that at each tree level, they only try to optimize the current dimension and therefore miss some optimization opportunities.

1.4 Proposed Approach

In this paper, we propose the Ternary Unification Framework (TUF) for TCAM classifier compression, which consists of three basic steps. First, TUF converts the given classifier to its BDD representation. Second, TUF collapses the BDD, converting leaves into sets of equivalent ternary data structures and combining these at internal nodes to produce a set of ternary data structures that represent the classifier. Finally, TUF converts the ternary data structures to TCAM rules and chooses the smallest as the final result. Broadly, the two decisions that define a specific TUF algorithm are (1) the ternary data structure to represent the intermediate classifiers and (2) the procedure to combine intermediate classifiers.

TUF advances the state of the art on TCAM classifier compression from two perspectives. First, it is a general framework, encompassing prior tree based classifier compression algorithms as special cases. Because of the structure that TUF imposes on tree based classifier compression algorithms, it allows us to understand them better and to easily identify optimization opportunities missed by those algorithms. Second, this framework provides a fresh look at the TCAM classifier compression problem and allows us to design new algorithms along this direction.

1.5 Key Contributions

We make three key contributions in this paper. First, we give a general framework for optimizing ternary classifiers. The framework allows us to find more optimization opportunities and design new TCAM classifier compression algorithms. More specifically, the choices of which ternary data structures to use and how to combine them give new flexibility in designing such algorithms. Second, by designing novel ternary data structures and combining procedures, we develop three concrete compression algorithms for three types of classifier compression. Third, we implemented our algorithms and conducted side-by-side comparison with the prior algorithms on both real-world and synthetic classifiers. The experimental results show that our new algorithms outperform the best prior algorithms by increasing amounts as classifier size and complexity increases. In particular, on our largest real life classifiers, the TUF algorithms improve compression performance over prior algorithms by an average of 13.7%.

2. RELATED WORK

Several papers have addressed the problem of optimizing TCAM packet classifiers. Some major categories of this research include techniques for redundancy removal, compressing one and two-dimensional packet classifiers and techniques for compressing higher-dimensional packet classifiers.

Redundancy removal techniques identify rules within a classifier whose removal does not change the semantics. Since firewall policy is specified indirectly, redundant rules are commonly introduced into real life classifiers. Discovering these redundant rules and removing them reduces the storage requirements of the classifier in TCAM, as well as potentially simplifying maintenance of the classifier. This technique has been investigated by Liu [12, 15], using FDD variants to efficiently identify a maximal set of redundant rules. More recently, Acharya and Gouda [2] have shown a correspondence between redundancy testing and firewall veri-

fication and used this to build a novel redundancy removal algorithm not based on trees. Redundancy removal is an important component of our algorithms, but alone it misses many opportunities to combine or re-represent policy using new rules.

While the problem of efficiently producing minimum prefix classifiers for a one-dimensional classifier has been solved [9,22], there are still ongoing efforts to solve the same problem in two dimensions. Suri *et al.* [22] solved the case of optimizing a two-dimensional classifier where any two rules are either disjoint or nested, and Applegate *et al.* [4] solved the special case for strip rules where all rules must span one dimension as well as providing approximation guarantees for the general two-dimensional case. These solutions do not generalize to higher dimensions, so they provide little assistance with typical five-dimensional classifiers.

Dong *et al.* [8] proposed the first five-dimensional prefix classifier minimization algorithm. Meiners *et al.* [14] improved upon this in their TCAM Razor algorithm. Meiners *et al.* [18] then developed two ternary classifier minimization algorithms Bit Weaving and Ternary Razor. Ternary Razor adds the bit merging algorithm from Bit Weaving into TCAM Razor. McGeer *et al.* [17] demonstrated an algorithm for finding the minimum representation of a given classifier, but this algorithm is impractical for all but the smallest classifiers due to its exponential runtime.

In addition to this work, there are many papers that also try to optimize TCAM packet classification using non-standard TCAM architectures [19–21] or by reencoding packet fields [5–7,25]. While this type of work may have theoretical elegance, the cost of engineering new TCAM architectures or re-encoding makes these works less practical. Algorithms like TUF that work within the constraints of the standard TCAM architecture have the advantage that they can be deployed immediately on existing hardware.

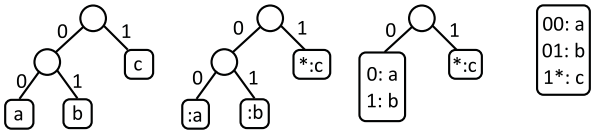
3. TUF FRAMEWORK

In this section, we outline our Ternary Unification Framework (TUF), which gives a general structure for ternary classifier compression algorithms. Section 3.1 gives the basic structural recursion to compress a TCAM classifier. Section 3.2 specifies how TUF facilitates efficient merging of partial solutions in the structural recursion step.

3.1 TUF Outline

We first present the basic steps of TUF. TUF takes a TCAM classifier as input and returns an optimized TCAM classifier as output. Because TCAM classifiers are written as rule lists, determining a simple property such as whether a TCAM classifier has a decision for every input is NP-complete. The first step of TUF is to represent the classifier as a BDD, where every node has zero or two children and the decisions are in the leaves. An example BDD with three leaves is shown in Figure 1a. Converting to BDD resolves the priorities of overlapping classifier rules and gives exact decisions for any input value or range. Note that the construction is dependent on the order of the bits in the BDD, and the resulting classifier can be different with different orderings. By considering multiple bit orderings, typically organized by packet header fields, better compression can be achieved.

The second step of TUF converts the leaves of the BDD into instances of a ternary data structure. As each leaf



(a) Input (b) Leaves (c) Merge (d) Output

Figure 1: Structural Recursion Example

represents some collection of input values assigned a single decision, we can convert it into an equivalent ternary data structure, such as a trie or a TCAM classifier. This is demonstrated in Figure 1b, where the BDD leaves are replaced by TCAM classifiers. The predicate for each classifier depends on the height of the leaf; in this case, the bottommost leaves are zero-bit classifiers, while the upper c leaf is a one-bit classifier as its height is 1.

The third step, the core of TUF, merges these ternary data structures to form ternary data structures that encode larger sections of the input space. Figure 1c shows the result of merging the left subtree of Figure 1b. It is in the merging process that compression is possible; similarities in the two halves can be identified and a smaller merge result constructed. After all BDD nodes have been merged, a ternary data structure that represents the entire classifier is created. If the ternary data structure used is not a TCAM classifier, then it is converted to a TCAM classifier as the final step. The TCAM classifier for this example is shown in Figure 1d.

TUF can use a number of different ternary data structures such as tries, nested tries (tries of tries), and TCAM classifiers. To support a particular ternary data structure, TUF requires that the data structure support two operations: `Singleton` and `LRMerge`. `Singleton` converts a BDD leaf to a ternary data structure and `LRMerge` joins two ternary data structures A and B into one, $A + B$. Pseudocode for the TUF Core recursive merging is given in Algorithm 1.

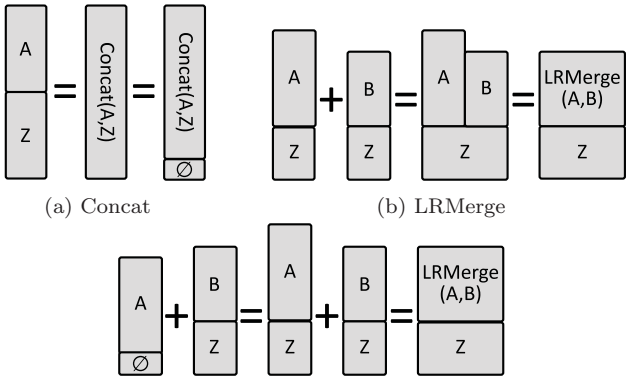
Algorithm 1: TUFCore(c)

```
   Input: A 2-tree c representing a classifier
   Output: An equivalent ternary data structure
1  switch c do
2      case Leaf dec
3          return Singleton (dec);
4      case Node(left,right)
5          LeftSol := TUFCore (left);
6          RightSol := TUFCore (right);
7          return LRMerge (LeftSol,RightSol);
```

To support classifier compression, the `LRMerge` operation should find commonalities between the two halves and use ternary rules to represent any found commonalities only once in the ternary data structure. This may be a complex operation, spending significant effort to both find and then efficiently represent such commonalities. We next describe how we can simplify the task required by `LRMerge` by tracking backgrounds.

3.2 Efficient Solution Merging

The goal of `LRMerge` is to combine two ternary data structures into one ternary data structure representing both. Using just a single ternary data structure at each step can cause overspecialization. The minimum-size solution for a small



(a) Concat (b) LRMerge

(c) LRMerge with EB

Figure 2: TUF operations w/ backgrounds

part of the input space is often not the best representation for it in the context of the complete classifier. By keeping multiple representations at each step, the right representation of a subtree can be used to create the final solution. Spending more time to keep extra representations of each subtree allows smaller ternary classifiers to be constructed. Taking this idea to the logical extreme, an algorithm could keep all combinations of sub-solutions every time two solution sets are merged. This could cause an exponential amount of work to be spent creating and combining them, leading to an impractical algorithm. The rest of this section explores the use of backgrounds as a way to limit the number of combinations that are created, allow pruning of useless solutions from the solution set and improve the speed of merging solutions.

As ternary classifiers can have multiple matching decisions for an input, they have a precedence structure. For TCAM classifiers, earlier rules in the list have higher precedence than rules later in the list. Using this relationship, a ternary classifier can be divided into two classifiers: a foreground of higher precedence and a background of lower precedence. The operation $\texttt{Concat}(A, Z)$, denoted $\langle A, Z\rangle$, joins a foreground and background ternary classifier into a single ternary classifier, as shown in Figure 2a. Intuitively, the joined classifier searches A first, and if no decision is found, the result from searching Z is used. If the background is non-empty, the foreground classifier should be an incomplete classifier so that some inputs do not have a decision. We denote the classifier that has no decision for any input as $\emptyset$ and note that it is the identity element for the `Concat` operation, *e.g.* $\langle x, \emptyset\rangle = x = \langle \emptyset, x\rangle$.

We write $\frac{F}{B}$ to represent a classifier split into separate foreground, F, and background, B, ternary data structures. For each BDD leaf, we make will create a set of solutions that encodes that leaf in different ways. This solution set has two split classifiers, one that encodes the decision in the foreground, and one that encodes it in the background. The solution set for a BDD leaf with decision d is

$$\left\{\frac{\texttt{Singleton}(d)}{\emptyset}, \frac{\emptyset}{\texttt{Singleton}(d)}\right\} \tag{1}$$

One of these two solutions will be used in the final classifier to represent this part of the input having decision d. The first solution will be used if the decision d is sufficiently rare in this subtree that it is best to encode this part of the classifier function as its own rule. The second solution will be used if d is sufficiently common in this will be common in this subtree, and that this decision will be encoded as part of a rule with a more general predicate.

TUF maintains the invariant that every solution set will have a solution with an empty background. Because the empty background solution will be handled differently from the other pairs, we give it the name EB, for Empty Background. In (1), the first solution is the EB as its background is empty. The EB has the best complete encoding of the classifier represented by a BDD subtree, while the other solutions are the best encoding that assumes some background.

TUF uses multiple solutions and backgrounds to efficiently merge its two input sets of solutions into a new set of solutions. TUF will create a new solution for every distinct background in either input set. For a background found in both input sets, TUF merges the two associated foregrounds together to make the solution for that background. For ternary data structures A, B, and Z, to merge $\frac{A}{Z}$ with $\frac{B}{Z}$, the result is $\frac{A+B}{Z}$, as shown in Figure 2b. When one child has a solution with a background that the other lacks, TUF substitutes the EB for the missing foreground. This will produce correct results because the EB is a complete classifier, so can be placed over any background without changing the semantics, as shown in Figure 2c. An example merge of two solution sets can be written as

$$\left\{\frac{A}{X}, \frac{B}{Y}, \frac{C}{\emptyset}\right\} + \left\{\frac{D}{X}, \frac{E}{\emptyset}\right\} = \left\{\frac{A+D}{X}, \frac{B+E}{Y}, \frac{C+E}{\emptyset}\right\}.$$

This is implemented as $\texttt{SetMerge}(l, r)$ in Algorithm 2.

Backgrounds simplify `LRMerge`'s search for commonalities by allowing `LRMerge` to focus on merging sibling ternary data structures that have the same background. The use of backgrounds also simplifies the merging process by producing the most useful solutions; instead of trying to merge all pairs of solutions ($O(mn)$ merges), we instead merge solutions with the same background ($O(m + n)$ merges). Finally, the use of backgrounds allow a set of solutions to be simplified.

To simplify a set of solutions, TUF incorporates a cost function $\texttt{Cost}(C)$ which returns the cost of any ternary classifier. Let A be the foreground of the EB in a solution set. For any solution $\frac{X}{Y}$, if $\texttt{Cost}(A) \leq \texttt{Cost}(X)$, then TUF removes $\frac{X}{Y}$ from the set. It is a useful simplification because substituting A for X in future merges will supply `LRMerge` with lower cost inputs, which should produce a lower cost result while maintaining correctness. TUF can also replace the EB by $\frac{\langle X,Y\rangle}{\emptyset}$ if there is a solution $\frac{X}{Y}$ for which $\texttt{Cost}(A) > \texttt{Cost}(\langle X, Y\rangle)$. The result of these simplifications is that the EB will have the highest cost foreground of any solution, and the difference in cost between the EB and any other solution must be less than that solution's background cost.

So far, we have treated the classifier as an unstructured string of bits. In actual usage, the bits being tested are made up of distinct fields, and there is structure to the classifier rules related to these fields. For example, ACL rulesets often have 5 fields: Protocol, Source IP, Source Port, Destination IP, and Destination Port. Once we have developed a good ternary classifier for a section of one field, it is often beneficial to simply store that classifier without modification as we extend it to consider bits from other fields. To support this, we use a function `Encap(d)` that creates a field break by encapsulating the ternary data structure as a decision of another 1-dimensional classifier. The `LRMerge` operation is not

Algorithm 2: SetMerge(l,r)

```
   Input: solution sets l and r
   Output: merged solution set
1  Out = empty solution set;
2  NullLeft = foreground of ∅ in l;
3  NullRight = foreground of ∅ in r;
4  foreach bg in l.backgrounds ∪ r.backgrounds do
5      ForeLeft = foreground of bg in l or NullLeft;
6      ForeRight = foreground of bg in r or NullRight;
7      Out.add(LRMerge (ForeLeft,ForeRight),bg);
8  return Out
```

Algorithm 3: TUFCore(b) with backgrounds

```
   Input: A BDD b
   Output: A solution set of (fg,bg) pairs
1  switch b do
2      case Leaf d          /* BDD Leaf w/ decision d */
3          return {(Singleton (d),∅),
4                  (∅,Singleton (d))};
5      case Node(left,right)          /* Internal Node */
6          LeftPairs := TUFCore (left);
7          RightPairs := TUFCore (right);
8           //next line uses LRMerge;
9          MergedPairs := SetMerge (LeftPairs,
           RightPairs);
10         Solutions := Concat to all of MergedPairs;
11         BestSol := lowest Cost solution in Solutions;
12         MergedPairs.removeIf(Cost (x) ≥ Cost
           (BestSol));
13         MergedPairs.add(BestSol, ∅);
14         if at field boundary then
15             Encap (MergedPairs);
16             MergedPairs.add(∅, Encap (BestSol));
17         return MergedPairs;
```

required to respect this field break, although doing so will reduce the complexity of merging, as it will have fewer bits to consider. While encapsulating, we also promote the EB to be a background to make it easy to find as a commonality.

Pseudocode for this enhanced TUF Core is given in Algorithm 3. In it, the ordered pair (X, Y) is used to represent $\frac{X}{Y}$.

4. PREFIX MINIMIZATION USING TRIES

The TUF framework can be used to create a multi-dimensional prefix classifier compression algorithm by using tries. Prefix classifiers have prefix predicates where all stars follow all 0's and 1's. TUF will represent multi-dimensional prefix rules with tries of tries.

In this paper, tries are binary trees with nodes optionally labeled with decisions. As with BDDs, the binary search key determines a path from the root, taking the left (right) child if the next bit is 0 (1). The decision of a trie for a search key is the last label found on the path for that key. Each labeled node corresponds to a prefix rule; the path to it from the root matches a prefix of the search key, and all other bits are ignored. Tries are commonly used to represent Internet routing tables where the longest matching prefix determines the route taken. To handle multi-dimensional prefix classifiers, the solution is to use tries where the decision is another trie.

We now describe 1-dimensional and multi-dimensional prefix classifier minimization in TUF.

4.1 1-Dimensional Prefix Minimization

Adapting tries into the TUF framework is very natural. The empty classifier, $\emptyset$, is a trie node with no decision. To implement `Singleton`(d) and produce a classifier where all inputs have decision d, simply create a trie node with decision d. The `Cost(t)` of a trie t is the number of nodes with a decision, which corresponds to the number of TCAM rules needed for that trie. To `LRMerge(l,r)` two tries, we create a new trie node with no decision and assign the left child as l and the right child as r. The `Concat(f,b)` operation only has to handle the case where the foreground has a root node without decision and the background is a singleton trie node with a decision. This is because backgrounds are always singleton tries and because `Concat` is applied immediately after `LRMerge` which produces a foreground trie where the root has no decision. In this situation, `Concat` just moves the decision of the background to the root node of the foreground.

Figure 3 illustrates the compression of an example classifier into a trie using these operations. The input classifier assigns decision `a` to binary input value 01 and `b` to binary input values 00, 10 and 11. The BDD representation of this classifier is Figure 3a, which has a leaf labeled `a` in position 01 (left, right), and leaves with decision `b` elsewhere. At each BDD leaf, two solutions are created, one with the decision in the foreground and one with the decision in the background, shown in Figure 3b. As backgrounds will always be a singleton trie, we will show them as the decision of that trie over a shaded background. The merging step combines solutions that have the same background. In the case where the other solution set is missing a solution with the same background, a solution is combined with the EB of the other solution set. We will apply this merging step twice to our example BDD, as shown in Figures 3c and 3d. The first merge produces three solutions: the EB, one solution with `a` as background, and one solution with `b` as background. To produce the EB, the foregrounds of the existing EBs are merged. To produce the solution with background `a` or `b`, the corresponding foreground is merged with the EB of the opposite solution set. Note that `LRMerge` is not symmetric, producing differently shaped tries. The second merge is done similarly, with the two `b` solutions merged, the two EBs merged, and the `a` solution merged with the EB.

After we finish merging two solution sets, we optimize the result. Optimization has no effect after the first merge in our example, but it does improve the solution set after the second merge. Recall that if any solution has total (foreground + background) cost less than the EB, then the EB can be replaced by a `Concat` of that solution. In the example, the total cost of the solution in Figure 3d with background `b` is 2; the foreground cost is 1 and the background cost is 1. The EB has a higher cost of 3, which is greater than the total for `b`, so we replace the EB. The result of `Concat` is to put the background decision into the root node of the trie, as shown in the final solution set, where the EB has decision `b` in its root node. Next, recall that if any solution has foreground cost no smaller than the EB, it is replaceable by the EB and

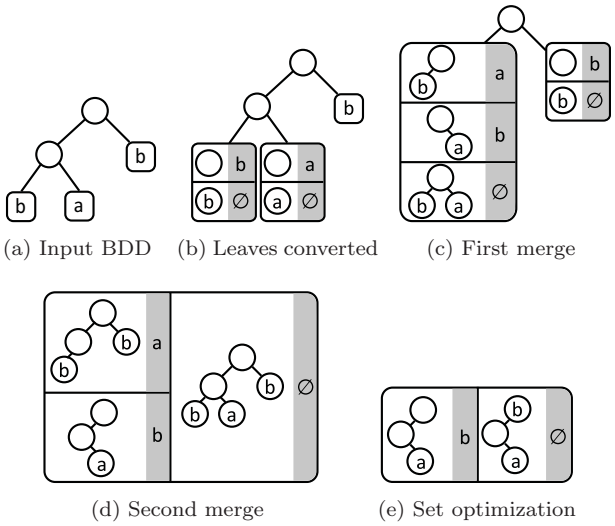


Figure 3: TUF Trie compression of simple classifier

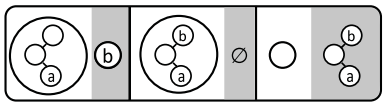


Figure 4: Encapsulation at a field boundary

can be removed. In this case, we removed the solution with background `a` because its foreground cost of 2 is the same cost as that of the new EB. As we have finished reducing our BDD to a single solution set, the result of compressing this classifier is that newly created EB. The final result is shown in Figure 3e.

4.2 Multi-dimensional prefix minimization

To represent a multi-dimensional prefix classifier, tries of tries are the natural solution. In a trie of tries, each decision of an n-dimensional trie is an $(n-1)$-dimensional trie. The lowest level 1-dimensional tries have the final decisions of the classifier in them. Tries of tries are turned into decisions for the next level trie at field boundaries using an `Encap` function which is run on both the foreground and background classifiers. In this case, `Encap` simply takes the existing classifier and sets it as the decision of a singleton trie, producing a $(n+1)$-dimensional trie from an n-dimensional trie. For the case of an empty trie such as the background of the EB, `Encap` returns an empty trie. This is analagous to how leaf solution sets are created for tries.

The result of encapsulating the solution set in Figure 3e is shown in Figure 4. The two existing solutions have been encapsulated as decisions of 2-dimensional tries, which is shown with the existing trie inside a new, large trie node. For the second solution, the background is just an empty background as encapsulation of the empty background is a null operation. One new solution is added at this step (the rightmost solution in Figure 4) where the EB is set as a background decision to an empty foreground.

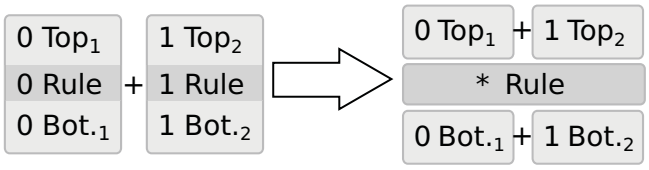


Figure 5: Recursive merging left and right ACLs

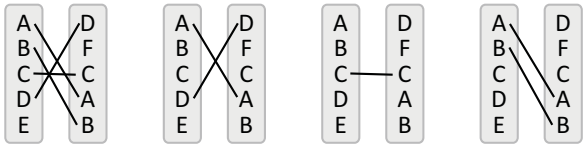


Figure 6: ACL pairing example

5. TERNARY MINIMIZATION USING ACLS

In this section, we use TCAM classifiers as the ternary data structure in the TUF algorithm. Using tree structures makes the `LRMerge` operation very natural to represent, but limits the variety of TCAM classifiers that can be produced. Using TCAM classifiers as the ternary data structure makes the `Concat` operation trivial, but has additional complexity in implementing a compressing `LRMerge` operation.

Merging TCAM classifiers without factoring out any commonalities can be done by prefixing all rules in the left input by 0 and those from the right input by 1 and concatenating them: $A + B = \langle 0A, 1B \rangle$. As there is no overlap between the two groups of rules, tThe order of concatenation doesn't matter, and the two prefixed inputs can be interleaved arbitrarily. Factoring out commonalities is superficially easy; find a rule that appears in both inputs and put a copy of that rule prefixed by * into the result instead of the two rules prefixed by 0 and 1. This simple method does not take into account the ordering of rules in the inputs, so it produces incorrect results. A rule that appears in both inputs may be needed to shadow other rules. We must take this into account when merging.

To preserve correctness in all cases, we must ensure that rules combined in this way stay in the same order relative to other rules in both inputs. Graphically, this is illustrated in Figure 5. This leads to a recursive procedure to merge two TCAM classifiers. After identifying a common rule, split both ACLs into the part before the common rule and the part after the common rule, called the tops and bottoms, respectively. Next, merge the two tops and the two bottoms, recursively and join the pieces back together. We can write this algebraically as

$\langle T_1, x, B_1 \rangle + \langle T_2, x, B_2 \rangle = \langle (T_1 + T_2), *x, (B_1 + B_2) \rangle$.

Given two rulesets, we can maximize the number of rules merged by examining the pattern of which rules could be merged. Figure 6a shows an abstracted pair of ACLs, with letters representing rules. Each pair of rules that can be merged is connected by a line. Two pairs of rules conflict if after merging one pair, the other pair cannot be merged. Graphically, two pairs of rules conflict if their corresponding lines intersect. We define a pairing to be a subset of the pairs of rules that can be merged without conflict. Figure 6b shows an invalid pairing, as splitting the rules for one pair prevents the other pair from merging. A maximal pairing is a valid pairing in which no pairs can be added without it

becoming invalid. Figure 6c shows a maximal pairing; when we split the rulesets into tops and bottoms, we can see there are no further pairings. A maximum pairing is a pairing with the property that no other pairing has more pairs. Figure 6d shows a maximum pairing; there is no larger set of pairs that has no conflict.

The problem of finding a maximum pairing can be reduced to the maximum common subsequence problem [16]. This problem is NP-complete for an arbitrary number of input sequences, but has polynomial-time solutions for the case of two input sequences. In our experiments on the real-life rulesets used in Section 7, we observe that there is little difference in the number of pairings identified between the optimal solution and our greedy solution.

6. REVISITING PRIOR SCHEMES IN TUF

TUF provides a new perspective for classifier compression that leads to new opportunities for compression and more efficient implementations. We illustrate this feature by studying previously developed one-dimensional and multi-dimensional prefix classifier minimization algorithms from the perspective of TUF. Specifically, we examine an existing algorithm for one-dimensional prefix classifier minimization [22] and a non-optimal but effective algorithm for multi-dimensional prefix classifier minimization [14]. Both Suri *et al.*'s one-dimensional algorithm and Meiners *et al.*'s multi-dimensional algorithms can be viewed as instantiations of the TUF framework. Furthermore, when viewed within TUF, we immediately find improvements to both algorithms.

Suri *et al.* first build a BDD out of the input trie, then apply a union/intersection operation to label interior nodes with sets of decisions, and finally traverse the tree once more from the root to give each tree node its final decision. The TUF Trie algorithm presented in Section 4 follows a very similar structure. The set of solutions generated at each step follows the same pattern of union/intersection. Because of the simple background structure, all foregrounds are always equal cost, except for the EB, which has a cost that is greater by one. When the children of an internal node have no matching backgrounds, all the merge results will have the same cost and will be preserved for the next step; this corresponds to the union case. When there are matching backgrounds with the same decision, the resulting solution will replace the EB and eliminate all non-paired solutions; this corresponds to the intersection case.

There is one important difference between the algorithms which is how they manage the tree. The existing algorithms are in-place algorithms that modify an existing tree in multiple passes. This requires the whole BDD to be generated and kept in memory. The TUF Trie algorithm does a reduction over the structure of the BDD, but does not require it all to be generated, or in memory at once. The BDD can be lazily generated, and only a few solution sets need be in memory at once. In this way, TUF Trie can be more efficient than existing algorithms for very large classifiers.

We next consider the multi-dimensional TCAM Razor algorithm developed by Meiners *et al.*. TCAM Razor builds a multi-dimensional algorithm from Suri *et al.*'s one-dimensional algorithm. It first builds an FDD from the input classifier and compresses the 1D classifiers at the leaves using a weighted version of 1D compression. It then treats the results of this compression as a single decision with weight

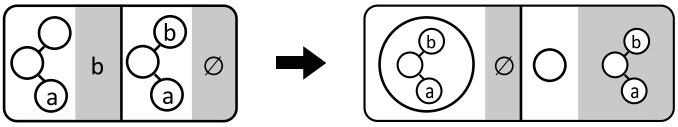


Figure 7: Razor hoisting the null solution as a decision

0*	10*	: A			
0*	***	: D		0*	10*	: A
1*	***	: D		**	***	: D

(a) Result of TCAM Razor

(b) Result of TUF Trie, same input

equal to the number of rules in the result, effectively reducing the dimension by one. By repeating this process until all the dimensions of the FDD have been compressed, Razor produces a single classifier as output.

Looking at TCAM Razor from the perspective of TUF, we identify ways to improve compression speed. Razor's weighted one-dimensional compression keeps a full solution for each decision. As the last dimension is compressed, its decisions are compressed tries for the other dimensions. There may be tens or hundreds of these, and it is wasteful to do merges for all these solutions, when few of them will be used. A TUF-based solution can remove solutions that are strictly inferior and start with a small set of solutions at the leaves to greatly decrease the amount of work done. These changes give an average of 20x speed improvement on the classifiers that take more than 1/4 second to compress.

The compression level achievable by TCAM Razor can also be improved. Looking at both algorithms from the perspective of solution sets, we see that when Razor finishes compressing a field, it keeps only a best solution (the EB). When the next level is processed, only this one solution is used, and it is treated as an opaque value. The resulting transformation is illustrated in Figure 7, which can be compared with Figure 4. The classifier with background b is discarded, and the EB is shown encapsulated on the far right and promoted as a new background in the middle.

The Multi-dimensional TUF Trie improves compression by encapsulating more solutions at field boundaries which can lead to the discovery of more potential similarities. Figures 8a and 8b give example outputs of TCAM Razor and the simple multi-dimensional prefix TUF algorithm described in section 4. TCAM Razor treats the rulesets from already compressed fields as opaque values, so once the last field is processed, the results have no flexibility. Because of TUF's ability to keep multiple possible solutions for already compressed fields, it is able to apply the default rule across the field boundary, resulting in better compression.

7. EXPERIMENTAL RESULTS

7.1 Evaluation Metrics

The critical metric of a classifier compression algorithm is the number of rules in the output TCAM classifier. As the input classifiers are in range form, they may contain rules that must be rewritten to be stored in TCAM. When computing compression ratios, we compare against the result of direct range expansion. This means we replace each non-ternary rule with a collection of prefix rules that compose the same predicate. We denote the result of this process Direct(C) for a classifier C. We denote the size of the result

of running algorithm A on classifier C as $|A(C)|$. For example, $|\text{Razor}(C)|$ is the number of rules after running TCAM Razor on a classifier C. Then, the compression ratio for an algorithm A on a classifier C is

$$CR(A, C) = \frac{|A(C)|}{|\text{Direct}(C)|}.$$

A smaller compression ratio indicates that the algorithm produced a smaller output and thus needs less TCAM space. To measure an algorithm's performance on a set of classifiers, we use Average Compression Ratio (ACR). For a set of classifiers S, the ACR of algorithm A is the mean compression ratio across those classifiers;

$$ACR(A, S) = \frac{1}{|S|} \sum_{C \in S} CR(A, C).$$

We evaluate how much TUF advances TCAM classifier compression using the following improvement metric. First, for any classifier C, we define $CR_{prior}(C)$ to be the best possible compression ratio for C using any algorithm excluding TUF algorithms. We then define $CR_{new}(C)$ to be the best possible compression ratio for C using any algorithm including TUF algorithms. In both cases, we use the best possible field permutation order for each algorithm for the given classifier. We define the percent improvement of TUF as $1 - \frac{CR_{new}}{CR_{prior}}$. In this case, a higher Improvement percentage means that TUF performs better and saves more TCAM space.

7.2 Results on real-world classifiers

We test these algorithms on a collection of real-life classifiers in three categories. The categories are based on the number of non-redundant rules and difficulty converting the rules to ternary format. Table 2 gives a breakdown and statistics on these categories.

Cat.	Count	Avg # Non-Red.	Avg. # Prefix Exp.
Small	13	9	1578
Large	8	3221	7525
Med.	17	209	641

Table 2: Classifier categories

Classifiers with an expansion ratio over 20 are categorized as Small RL. These 13 classifiers have an average of 9 non-redundant rules each, yet their prefix expansions have 1600 ternary rules. The remaining classifiers with more than 800 non-redundant rules are categorized as Large RL. The remaining 17 classifiers have neither extreme expansion ratios nor are extremely large, so we categorize them as Medium RL.

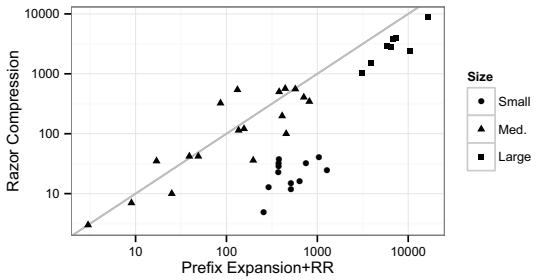


Figure 9: Razor vs. Redundancy Removal, for classifier grouping purposes

These groupings are visually distinguishable by plotting $|Direct(RR(C))|$, the prefix expansion of non-redundant rules, against the compressed size using TCAM Razor, as shown in Figure 9.

To test the sensitivity of algorithms to the number of decisions, we also compressed a variant of each of these classifiers. These variants are modified to have a unique decision for each rule. One practical reason for considering unique decisions is that we may need to know which rule matched; for example, we may be tracking hit counts for each rule.

	Algorithm	All	Large	Med.	Small
Orig.	TUF Trie	26.2%	30.8	41.8	0.8
	TUF ACL	22.8	22.8	38.4	0.7
Uniq.	TUF Trie	45.3	56.9	70.2	2.4
	TUF ACL	43.3	50.9	68.8	2.1

Table 3: ACR on real world classifiers

Table 3 shows the results of compressing the real-life classifiers and their unique variants with TUF Trie and TUF ACL. Both variants are very effective at compressing classifiers, but TUF ACL does outperform TUF Trie by roughly 13% on all classifiers and 26% on the Large classifiers where there are more compression opportunities to be exploited by a full ternary compression algorithm.

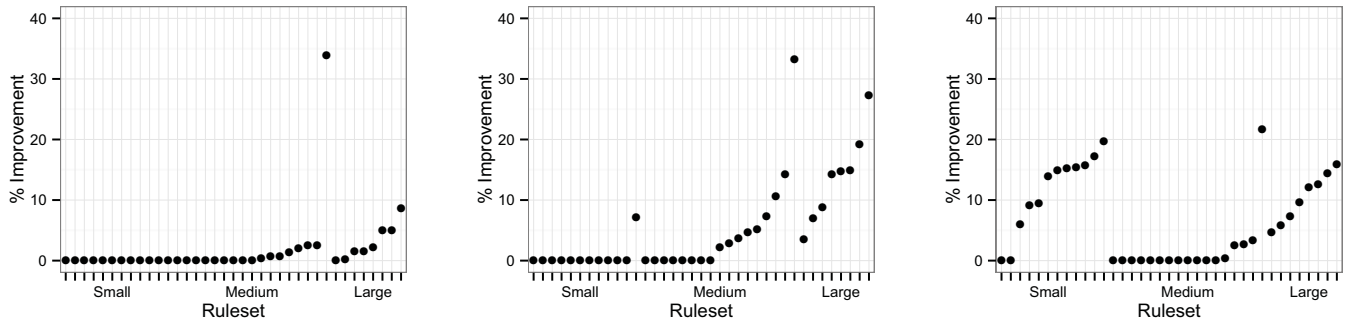
7.2.1 Sensitivity to number of unique decisions

When the input classifier has a unique decision for each rule, less compression is possible because there is less potential to apply a background that applies to multiple rules. As a result, TUF ACL and TUF Trie both have reduced compression performance, needing two to three times as many TCAM rules to represent the classifiers. TUF ACL still outperforms TUF Trie but by a smaller amount, roughly 4.5% on all classifiers and 10.5% on Large classifiers.

7.2.2 Comparison with state-of-the-art results

We present a direct comparison between TUF algorithms and the previous best algorithms in Figure 10. For prefix compression, $C_{prior}(C)$ uses only TCAM Razor [14] and $C_{new}(C)$ uses the best of TCAM Razor and TUF Trie. For ternary compression, $C_{prior}(C)$ uses the best of Ternary Razor and BitWeaving [18] and $C_{new}(C)$ uses the best of Ternary Razor, BitWeaving, and TUF ACL. Each graph shows the percent improvement of the TUF algorithm over the comparable state of the art for each of our real-life classifiers. The x-axis of each graph is broken into three parts corresponding to the Small, Medium and Large classifiers. Within each group, classifiers are sorted in order of increasing improvement from left to right.

We first consider prefix compression. In Figure 10a, we can see that adding TUF Trie improves performance by an average of 1.9 % on all classifiers. The improvement is small but does increase as we move from Small to Medium to Large classifiers from 0% to 2.6% to 3.0%. Furthermore, while the improvement is generally small, the percentage of classifiers where adding TUF Trie improves performance increases as we move to larger classifiers. Adding TUF Trie improves performance on 0 of the 13 Small classifiers (0%), 8 of the 17 Medium classifiers (47%), and 7 of the 8 Large Classifiers (87.5%). There is one notable outlier where TUF trie outperforms TCAM Razor by 34%.



(a) Prefix Compression (b) Ternary Compression (c) Ternary Compression, Unique decisions

Figure 10: Improvement of TUF over the state of the art for real life classifiers

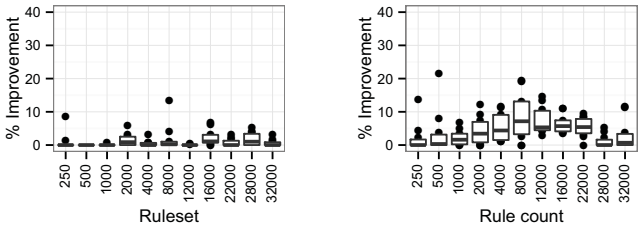
We next consider ternary compression. In Figure 10b, we see how much adding TUF ACL improves compression over using only Ternary Razor and BitWeaving on our set of real life classifiers. We see that the improvement is greater than for prefix compression. Specifically, adding TUF ACL improves performance by an average of 5.4 % on all classifiers. As with prefix compression, the improvement does increase as we move from Small to Medium to Large classifiers from 0.6% to 4.9% to 13.7%. As with prefix compression, the number of classifiers where adding TUF ACL improves performance increases as we move to larger classifiers. Specifically, adding TUF ACL improves performance on 1 of the 13 Small classifiers (7.7%), 11 of the 17 Medium classifiers (64.7%), and 8 of the 8 Large Classifiers (100%).

For prefix compression with unique decisions, TUF Trie offers almost no improvement over the state of the art, giving a maximum of 1.7% improvement and only improving 3 of the classifiers. We omit the plot for this uninteresting result.

For ternary compression with unique decisions, from Figure 10c, we see that TUF ACL improves performance but the pattern of improvement is quite different. Adding TUF ACL improves performance by an average of 6.2% on all classifiers with unique decisions, but now the best performance is on the Small Classifiers followed by the Large classifiers, with almost no improvement for the Medium classifiers. As we move from Small to Medium to Large classifiers, the average improvment goes from 11.4% to 1.8% to 10.3%. For the Medium classifiers, many of them are already in prefix form, so with unique decisions, the only optimization possible is removing redundant rules. The remainder are nearly in prefix form, so there is little opportunity for compression not already found by prior algorithms, although we still find some. For the Small classifiers, TUF ACL achieves improved performance by better ordering backgrounds to minimize the effects of prefix expansion. For the Large classifiers, TUF ACL is still better able to find global commonalities than previous algorithms.

7.3 Synthetic Rules

We also use a collection of 110 synthetic classifiers which were created by ClassBench [24]. These include 10 classifiers from each of 11 sizes from 250 rules to 32K rules. Note that the size of the synthetic classifiers is not comparable to real-life classifiers as large (> 2000 rule) synthetic classifiers contain about 90% redundant rules. This means that the complexity of the classifier function to be optimized is much lower than a real-life classifier with the same number of rules. The ACR values for TUF ACL across this group ranges from 23% for the smallest classifiers to 6.1% for the largest.



(a) Synthetic Original (b) Synthetic Unique

Figure 11: Improvement of TUF over the state of the art ternary algorithms for synthetic classifiers

The improvement in compressing these classifiers over the state of the art is shown in Figure 11. These figures show box plots for each same-size group of 10 classifiers. Each boxplot shows the median value as a line in the middle of the box, the 1st and 3rd quartile shown as the top and bottom of the box, and the most extreme values plotted as points. Figure 11a shows that adding TUF ACL leads to a 1% improvement across the synthetic rule sets, even though 38% of rulesets had improvement. On the other hand, Figure 11b show that adding TUF ACL leads to larger improvement for the synthetic rule sets with unique decisions. In this case, 73% of classifiers experience improvement, including 93% from the groups with between 1000 and 22000 rules. Overall, the average improvement in the state of the art was 4.5%. The improvement percentage generally increases as the synthetic classifiers increase in size, up to 28000 rules, possibly indicating a saturation point in the complexity of the classifier where less compression is available. We omit prefix results as there is a less than 0.1% improvement in prefix compression of the synthetic classifiers.

7.4 Efficiency

For prefix compression, TUF-based TCAM Razor is much faster than the original TCAM Razor. For small classifiers, TUF-based Razor can be more than 1000 times faster than the original TCAM razor. For larger classifiers, the speed difference is an average of twenty times faster, achieving the exact same level of compression in much less time.

Ternary compression algorithms take longer, as their search for commonalities throughout the classifier increases the work they have to do. On a desktop PC with an AMD 3GHz Phenom CPU, these algorithms usually complete in under one minute. For some of our classifiers and some of our algorithms and some field permutations, the running time occasionally exceeds one minute; fortunately, these cases can be ignored as they almost always result in poor compression. That is, we can set a reasonable limit such as one minute or five minutes for running a compression algorithm and terminate the algorithm and discard the result for the given field permutation if it exceeds the given time limit. Furthermore, for many applications, a slow worst case compression time is acceptable as updates are performed offline.

8. CONCLUSIONS

In this paper, we propose a new framework for compressing TCAM-based packet classifiers and three new algorithms for implementing this framework. This framework allows us to look at the classifier compression problem from a fresh angle, unify many seemingly different algorithms, and discover many unknown compression opportunities. Our experimental results show that TUF gives insights that (1) significantly improve the speed of the existing TCAM Razor algorithm with no loss in compression and (2) lead to new algorithms that compress better than prior algorithms. More importantly, this framework opens a new direction for further work on TCAM-based packet classifier compression.

Acknowledgement

This work is partially supported by the National Science Foundation under Grant Numbers CNS-0916044, CNS-0845513, CNS-1017588, and CNS-1017598, and the National Natural Science Foundation of China under Grant Numbers 61272546 and 61370226.

9. REFERENCES

[1] A guide to search engines and networking memory. *http://www.linleygroup.com/pdf/NMv4.pdf*, 2004.

[2] H. Acharya and M. Gouda. Firewall verification and redundancy checking are equivalent. In *INFOCOM, 2011 Proceedings IEEE*, pages 2123–2128. IEEE, 2011.

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

[4] D. A. Applegate, G. Calinescu, D. S. Johnson, H. Karloff, K. Ligett, and J. Wang. Compressing rectilinear pictures and minimizing access control lists. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, January 2007.

[5] A. Bremler-Barr, D. Hay, D. Hendler, and B. Farber. Layered interval codes for TCAM based classification. In *Proc. of IEEE Infocom*, 2009.

[6] A. Bremler-Barr and D. Hendler. Space-efficient TCAM-based classification using gray coding. In *Proc. 26th Annual IEEE Conf. on Computer Communications (Infocom)*, May 2007.

[7] Y. Chang, C. Lee, and C. Su. Multi-field range encoding for packet classification in tcam. In *INFOCOM, 2011 Proceedings IEEE*, pages 196–200. IEEE, 2011.

[8] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla. Packet classifiers in ternary CAMs can be smaller. In *Proc. ACM Sigmetrics*, pages 311–322, 2006.

[9] R. Draves, C. King, S. Venkatachary, and B. Zill. Constructing optimal IP routing tables. In *Proc. IEEE INFOCOM*, pages 88–97, 1999.

[10] C. Lambiri. Senior staff architect IDT, private communication. 2008.

[11] P. C. Lekkas. *Network Processors - Architectures, Protocols, and Platforms*. McGraw-Hill, 2003.

[12] A. X. Liu and M. G. Gouda. Complete redundancy detection in firewalls. In *Proc. 19th Annual IFIP Conf. on Data and Applications Security, LNCS 3654*, pages 196–209, August 2005.

[13] A. X. Liu and M. G. Gouda. Complete redundancy removal for packet classifiers in tcams. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, to appear.

[14] A. X. Liu, C. R. Meiners, and E. Torng. TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 18(2):490–500, April 2010.

[15] A. X. Liu, C. R. Meiners, and Y. Zhou. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proc. 27th Annual IEEE Conf. on Computer Communications (Infocom)*, April 2008.

[16] D. Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, Apr. 1978.

[17] R. McGeer and P. Yalagandula. Minimizing rulesets for tcam implementation. In *Proc. IEEE Infocom*, 2009.

[18] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs. *IEEE/ACM Transactions on Networking*, 20(2):488–500, 2012.

[19] T. Mishra and S. Sahni. Petcam–a power efficient tcam architecture for forwarding tables. *Computers, IEEE Transactions on*, 61(1):3–17, 2012.

[20] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy. Exact worst-case tcam rule expansion. *IEEE Transactions on Computers*, 2012.

[21] E. Spitznagel, D. Taylor, and J. Turner. Packet classification using extended TCAMs. In *Proc. 11th IEEE Int. Conf. on Network Protocols (ICNP)*, pages 120– 131, November 2003.

[22] S. Suri, T. Sandholm, and P. Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35:287–300, 2003.

[23] D. E. Taylor. Survey & taxonomy of packet classification techniques. *ACM Computing Surveys*, 37(3):238–275, 2005.

[24] D. E. Taylor and J. S. Turner. Classbench: A packet classification benchmark. In *Proc. IEEE Infocom*, March 2005.

[25] R. Wei, Y. Xu, and H. Chao. Block permutations in boolean space to minimize tcam for packet classification. In *INFOCOM, 2012 Proceedings IEEE*, pages 2561 –2565, march 2012.