Minimizing Rulesets for TCAM Implementation

Rick McGeer, Praveen Yalagandula
HP Labs, Palo Alto, CA

***Abstract*—Packet classification is a function increasingly used in a number of networking appliances and applications. Typically, this consists of a set of abstract *classifications*, and a set of *rules* which sort packets into the various classifications. For packet classification at line speeds, Ternary Content-Addressable Memories (TCAMs) have become a norm in most networking hardware. However, TCAMs are expensive and power-hungry. Hence, a packet classification ruleset need to be minimized before populating the TCAM. In this paper, we formulate the *Ruleset Minimization Problem* for TCAM as an abstract optimization problem based on two-level logic minimization, and propose an exact solution and a number of heuristics. We present experimental results with two different datasets—artificial filter sets generated using ClassBench tool suite and a real firewall Access Control List (ACL) from a large enterprise. We observe an average reduction of 41% in artificial filter sets and 72.5% reduction in the firewall ACL using the proposed heuristics.**

I. Introduction

Packet classification is a core function of several network appliances, including firewalls, switches, network address translators (NATs), and routers. The problem of packet classification can be formulated as follows: given a sequence of overlapping *rules*, each of which is defined as a Boolean expression over a set of header fields, match the incoming packet to the first matching rule. Simple examples are packet forwarding rules based on source and destination IP addresses, traffic monitoring based on transport protocol, and traffic shaping based on source or destination port number.

Packet classification is in the critical path for packet forwarding and hence must be performed at the line speeds. Assuming an average packet size of 300 bytes, a 24-port Gig switch must process each packet in about 0.1μsec. For this reason, network appliance manufacturers often use ternary content-addressable memories (TCAMs) to speed processing. But, TCAMs are expensive, power-hungry, and not expandable (once the appliance is built, the TCAM size is fixed). On the other hand, rulesets can be large and are specified with focus on readability by human operators. All these factors strongly motivate minimization of ruleset size before populating the TCAMs with those rules.

While efficient ruleset search has received a great deal of attention, the problem of minimizing rulesets is only of recent interest, and previous approaches [1]–[5] have been heuristic in nature and focused mainly on reducing the rules with port ranges in either the source port or destination port fields. Recent work by Liu et. al. [6], [7], use decision diagram based heuristics to minimize the rulesets. In contrast to these approaches, our focus is to formulate and study the ruleset minimization problem in the light of Boolean logic minimization.

The problem of ruleset minimization is superficially similar to the general problem of two-level logic minimization [8], a classic problem from undergraduate digital design. In fact, the similarity is quite striking. In both cases, one is given a specification consisting of a list of fixed-width bit fields, where each bit may be 1, 0, or - (wildcard); the problem is to minimize the length of the list while retaining the semantics of the original list. This resemblance is best seen by considering an example. Consider a toy domain with four-bit addresses and two protocols, TCP and UDP. Imagine a firewall with two rules; accept everything with addresses 0-6, and UDP packets with addresses 12-15. If addresses are encoded in the usual way in the first four bits, and the TCP protocol is encoded with a 1 and the UDP protocol with a 0 in bit 5, then these two rules become:

```
00--- /* accept any protocol on ports 0-3 */
0-0-- /* accept any protocol on ports 0,1,4,5 */
0--0- /* accept any protocol on ports 0,2,4,6 */
11--0 /* accept UDP packets on ports 12-15 */
```

These are exactly the rows that would appear in a TCAM implementation of these rules. They are also the input specification to a two-level minimization problem (and, actually, in this case also the output – this is the minimum representation, given the encoding of the problem).

There are two distinct differences between the ruleset minimization problem and the two-level logic minimization problem. The first, trivial distinction is given by the example above: each rule in a ruleset minimization problem leads to several terms in a TCAM implementation or two-level function specification. All this means is that a ruleset specification is a (slightly) high-level specification of a two-level logic function, and generating the appropriate function is a mechanical exercise. The second, major distinction is that two-level logic minimization assumes that each term of the two-level function will be tested in parallel. However, rulesets are specified assuming that they are tested in a sequential manner. And, since TCAMs output the index of the first matching entry, TCAM operation is also semantically equivalent to testing rows in a sequential order. This has major consequences where there are multiple targets of a TCAM (in almost all cases, there are always at least two: `Permit` and `Deny` or equivalent actions). Entries in a TCAM specification can assume the preceding entries have failed, and therefore each entry can implicitly incorporate the negative, or complement, of all of the preceding entries. Formulating this into a variant of two-level minimization problem, *Minimal Sequential Cover* problem, is one of our major contributions.

Our strategy for the ruleset optimization for TCAM implementation is as follows:

1) Expand each rule into a set of terms (or in the two-level minimization or logic synthesis jargon, *cubes*).
2) For each target (e.g., `Permit` or `Deny`), find the *explicit* logic function associated with the target, as a set of cubes.
3) Use a variant of two-level logic minimization, the *Minimal Sequential Cover algorithm*, to exactly find a minimal set of cubes.

Overall, our contribution is to formulate the *Ruleset Minimization Problem* as a mathematical optimization problem strongly related to the problem of two-level logic synthesis [8], and offer both exact and approximate algorithms to solve this problem. We present experimental results with two different datasets—artificial filter sets generated using ClassBench tool suite [9] and a real firewall Access Control List (ACL) from a large enterprise with about 1400 rules. We observe an average reduction of 41% in the artificial filter sets and 72.5% reduction in the firewall ACL using our proposed heuristics.

II. Terminology

A *Boolean space of dimension n*, denoted $\mathbf{B}^n$ is a space of n variables, $x_1, ..., x_n$, where each $x_i \in \{0, 1\}$. A *Boolean function* over $\mathbf{B}^n$ is a mapping $\mathbf{B}^n \rightarrow \{0, 1\}$. If the domain of f is $\mathbf{B}^n$, then f is said to be *completely-specified*. If the domain of f is not $\mathbf{B}^n$, then the set $\mathbf{B}^n - Dom(f)$ is said to be the *don't-care* set of f. Appropriate assignment of the values 0 and 1 to elements of the don't-care set is an important technique in logic minimization. The set $T \subset \mathbf{B}^n$ defined as $\{T : \mathbf{x} \in \mathbf{B}^n | f(\mathbf{x}) = 1\}$ is called the *on-set* of f, and the members of T are called *minterms* of f. The set $F \subset \mathbf{B}^n$ defined as $\{F : \mathbf{x} \in \mathbf{B}^n | f(\mathbf{x}) = 0\}$ is called the *off-set* of f. The function $\overline{f}(x) = 1 \iff f(x) = 0$ is called the complement of f.

A point of $\mathbf{B}^n$ is conventionally written as a vector of n integers, each taken from the set $\{0, 1\}$. A *rectangular subspace* or, more commonly, *cube* of $\mathbf{B}^n$ is a vector over the space $x_1, ..., x_n$, where each $x_i \in \{\{0\}, \{1\}, \{0, 1\}\}$. A cube of $\mathbf{B}^n$ is conventionally written as a vector of n integers over the set $\{0, 1, 2\}$, where 2 indicates the set $\{0, 1\}$. It may also be written $x_1^{i_1}...x_n^{i_n}$, where each i_j indicates the value of x_j. A more conventional shorthand, which we will use here, writes $x_i^{\{1\}}$ as x_i, $x_i^{\{0\}}$ as $\overline{x_i}$, and elides $x_i^{\{0,1\}}$. Thus, over the space $\mathbf{B}^3$, the subspace $\{1\}\{0\}\{0, 1\}$ is written as 102 or $x_1\overline{x_2}$.

A subspace c of $\mathbf{B}^n$ such that $f(x) = 1 \forall x \in c$ is called an *implicant* of f, and a maximal implicant of f is called a *prime implicant* or simply *prime* of f. Any function f may be written as a union of implicants of f. A set C of implicants of f such that every minterm of f is contained in some member of C is called a *cover* of f. The goal of the two-level logic synthesis algorithm is simply to find a minimal cover of an input function.

The problem of two-level logic synthesis has been extensively studied since it was formulated by W. V. Quine over 50 years ago. Exact methods of solution were devised as soon as the problem was formulated, and simple exact algorithms have been a staple of elementary digital design since the 1950's. Excellent approximate methods were devised in the 1970's and early 1980's, culminating in the `Espresso` algorithms [8]. Though the problem has been shown to be NP-hard (and is thought to be complete in Σ_2), the availability of this large body of literature and algorithms permits us to use these powerful techniques to attack the ruleset minimization problem.

III. The Ruleset Minimization Problem

The mapping of rulesets to a TCAM offers us a guide to the formulation of the ruleset minimization problem as a variant of the classic two-level logic synthesis problem. Any TCAM entry realizes a cube of $\mathbf{B}^n$, where n is the number of bits in the TCAM entry. Realization of a rule as a set of TCAM entries therefore corresponds to a disjunction of spaces where the rule is met – in other words, to a Boolean function. In the following, we define few terms before formulating the precise definition of a ruleset specification.

A *header space* of a packet is the space of bits of some subset of the fields of the packet header. For example, the space (Source IP, Dest IP, Source Port, Dest Port, Protocol) defines a 104-bit space. We do not specify in advance which fields of the header make up the header space for a particular ruleset: we merely observe that any ruleset inherently defines a header space, and that every header space is a Boolean space of some dimension.

A *target* is a Boolean function over a header space. Classic examples of targets are `Accept`, `Deny`, `Accept-with-log`, and `Deny-with-log`. Intuitively, a target represents the total set of conditions forming the classification of a packet.

A *rule* is a cube over a header space, and is a component of a function. For rules with ranges, we expand them into multiple cubes. For example, for the rule `Accept 0-6` discussed in the Section I, we simply expand the rule into the appropriate cube set with three cubes. Henceforward, we will use *cube* and *rule* interchangeably.

Definition 1: A *specification* of a TCAM is a sequence of pairs $(r_1, t_1)...(r_N, t_N)$, and a set of targets, henceforth referred to as *target universe*, $T = \{T_1, .., T_m\}$, where each r_i is a cube (a rule) and $t_i \in T$.

It is tempting to consider targets as simply disjunctions of rules, but this misses important semantic information. Consider the rules, for targets `Deny` and `Permit`, taken from [1], shown in Table I. Note that the rule for `Permit` is simple: unity. But this does not mean that every packet is permitted. The rule implicitly incorporates the complement of the preceding `Deny` rules. The actual rule for `Permit` is shown in Figure 1.

With this observation in hand, the semantics of rules and targets are complete. Since the actual semantics of a rule depends on where it appears with respect to other rules, the rules have a total order imposed. In the following, we

```
not ((dest IP = 10.112.*.* &  5000 < source port < 65535 & protocol =  UDP)  |
 (source IP = 32.75.226.* & 1001 < dest port < 2000 & protocol = ICMP) |
 (source IP = 199.36.184.* &  49152 < source port < 65535 & protocol = ICMP))
```

Fig. 1. Actual Rule for `Permit` in Table I

Rule	Source IP	Dest IP	Source Port	Dest Port	Prot	Action
1	*	10.112.*.*	5000-65535	*	UDP	deny
2	32.75.226.*	*	*	1001-2000	ICMP	deny
3	199.36.184.*	*	49152-65535	*	ICMP	deny
4	*	*	*	*	*	permit

TABLE I
SAMPLE RULES

formulate the exact function associated with each rule in a TCAM specification.

Suppose the target universe be $T = \{T_1, ..., T_m\}$, and the ruleset be $(r_1, t_1), ..., (r_n, t_n)$ in a given TCAM specification. For each target T_j, we define an index set I_j, s.t. $\forall i \in I_j, t_j = T_j$, which essentially is the set of indices of the rules with T_j as the target. Plainly, $I_j \cap I_k = \emptyset$ (no rule is a component of two targets), and, for each $1 \leq i \leq n$, $i \in I_j$ for exactly one j (each r_i is a component of exactly one target). Let f_j be the logic function associated with rule r_j. This function is the intersection of the explicit function associated with r_j (Boolean space defined by the cube) and the implicit function (complement of the logic functions for every other target in the preceding rules). Let g_j denote the explicit function associated with a rule r_j. We can write the implicit function $\widehat{h_j}$ as follows:

$$\widehat{h_j} = \prod_{i=1}^{j-1} \overline{f_i} \tag{1}$$

We then have:

$$f_j = g_j \widehat{h_j} \tag{2}$$

We can simplify (1) by substituting (2) and expanding using De Morgan's law,

$$\widehat{h_j} = \prod_{i=1}^{j-1} (\overline{g_i} + \overline{\widehat{h_i}}) \tag{3}$$

Computing the implicit function just based on the above equation can be a tedious procedure. We simplify it further in the following.

We define:

$$h_j = \prod_{i=1}^{j-1} \overline{g_i} \tag{4}$$

Lemma 1:

$$f_j = g_j h_j \tag{5}$$

Proof: Note that since $f_j = g_j \widehat{h_j}$, $f_j \subseteq g_j$ and therefore $\overline{f_j} \supseteq \overline{g_j}$. Hence,

$$\prod_{i=1}^{j-1} \overline{g_i} \subseteq \prod_{i=1}^{j-1} \overline{f_i}$$

or $g_j h_j \subseteq g_j \widehat{h_j}$. To complete the proof, we show that $g_j h_j \supseteq g_j \widehat{h_j}$. To see this, let m be an arbitrary minterm of f_j. Hence, $m \in g_j \widehat{h_j}$, which implies $m \in g_j$. Now, all we must show is that $m \in h_j$. But this is evident. Since $m \notin f_i$ for all $i < j$, $m \notin g_i$ for all $i < j$, since $f_i \supseteq g_i$ for all i. Hence $m \in h_j$, QED. ■

With this lemma, we can use the alternate, simpler form of f_j in Eq. 5. Once we have the functions for each rule, the logic function associated with target T_j is easy:

$$T_j = \sum_{i \in I_j} f_i \tag{6}$$

Note that this formulation is now independent of any rule order and we have an explicit form for each target function. We can now formalize this in a sequence of definitions, which gives us our mathematical optimization problem.

Definition 2: Given a sequence of cubes $c_1, .., c_n$, the *implicit function* of cube c_i, denoted $\tilde{c_i}$, is the function defined as:

$$\tilde{c_i} = c_i \overline{c_1 c_2} ... \overline{c_{i-1}}$$

Definition 3: Given a specification of a TCAM $(r_1, t_1)...(r_n, t_n)$, $\{T_1, ..., T_m\}$, the *explicit function* of a target T_j is the function:

$$T_j = \sum_{1 \leq i \leq n, t_i = T_j} \tilde{r_i}$$

It is clear that these definitions recapitulate in a formal sense the derivation ending in (6) above. The explicit functions T_j are those that must be realized by the cubes in a TCAM. With these in hand, we can now formally state the conditions under which a set of cube/target pairs is a valid TCAM realization of a specification.

Definition 4: A sequence of cube/target pairs $(c_1, d_1), ..., (c_k, d_k)$ is said to *implement* a specification $(r_1, t_1), ..., (r_n, t_n), T = \{T_1, ..., T_m\}$ of a TCAM if and only if,

- $1 \leq i \leq k, d_i \in T$.
- for each (c_j, d_j) pair with $d_j = T_m$, $\tilde{c_j} \in T_m$.
- for each explicit function of T_m,

$$T_m = \sum_{1 \leq j \leq k, d_j = T_m} \tilde{c_j}$$

The purpose of these definitions is to formalize the intuitive notion of what it means for a cube to be a valid entry in a TCAM, and what it means for a set of targets to be precisely implemented by a TCAM: each cube will explicitly select exactly one target T_j in the case that none of the cubes preceding it in the sequence are selected, and the set of cubes cover all the cases of the rules.

Formally, the TCAM Minimization Problem can be stated as follows:

Problem 1: Given a TCAM specification $(r_1, t_1), ..., (r_n, t_n), \{T_1, ..., T_m\}$, find a sequence of N cube/target pairs $(c_1, d_1), ..., (c_N, d_N)$, such that

$(c_1, d_1), ..., (c_N, d_N)$ implements the given TCAM specification and such that no other cube/target pair sequence smaller than N implements the given TCAM specification.

We can now turn to the complexity of this problem and the minimization procedure. The overall procedure is given by a two-step process:

1) Form the explicit functions (6) of the problem.
2) Minimize the two-level form of the functions.

Step (1) of this procedure emerges as a side effect of the complexity discussion in the next section. In the subsequent section, we turn to step (2), a much more complex procedure.

IV. Complexity

It is interesting to consider the complexity class of the Problem 1. In this section, we demonstrate that the formal problem has a verification oracle in Σ_1, and hence the problem itself is in Σ_2. Thus the problem is in the same class as the general circuit minimization problem.

We show that this problem is in Σ_2 by demonstrating that its verification procedure is polynomially reducible to circuit equivalence. Circuit equivalence is well-known to be polynomially transformable to satisfiability (a variant of this was used by Richard Karp [10] to demonstrate the NP-completeness of 3SAT). Since the verification procedure for TCAM Minimization is thus in Σ_1 (i.e., in NP), it follows that TCAM Minimization is in Σ_2.

We show the reduction by deriving a logic circuit realizing a cube/target pair sequence $(c_1, t_1)...,(c_k, t_k)$, target set $\{T_1, .., T_n\}$ and demonstrating that the circuit size is a polynomial function of k, n, and s, where s is the number of TCAM bits.

We present the procedure `buildCircuit` in Figure 2. The `AND` and `OR` operators in this algorithm create `AND` and `OR` gates in the network, and `ZERO` and `ONE` are the logic 0 and 1 functions, respectively. We refer to T_i, t_i, c_i, k, and n as `target_function[i]`, `target[i]`, `cube[i]`, `num_cubes`, and `num_targets`, respectively.

We must show:

1) The functions `target_function[i]` computed by the algorithm are the explicit functions T_i of the specification; and
2) The size of the circuit is polynomial in the size of the original specification.

Lemma 2: At the end of the mth iteration of the loop 6-9 in algorithm `buildCircuit`, the following invariants hold:

1) `last` = $\overline{c_1 c_2}...\overline{c_m}$
2) `cube` = $\tilde{c_m}$
3) $1 \leq j \leq n$, `target_function[j]` = $\bigvee_{i \leq m, t_i = T_j} \tilde{c_i}$

Proof: Induction on m. For $m = 1$, Invariant 2 is assured by line 6 of algorithm `buildCircuit`. Line 2 ensures that at the entry to the loop in line 5, `target_function[j]` = `0` for all `j`. Since Invariant 2 holds, by line 9 Invariant 1 holds. Since Invariant 2 holds, line 8 ensures Invariant 3 holds. Now assume the lemma is true after $m - 1$ iterations.

```
1. proc buildCircuit
2.   for i = 1 to num_targets
3.     target_function[i] = ZERO
4.   last = ONE;
5.   for (i = 1 to num_cubes)
6.     cube = AND(cubes[i], last)
7.     target_number = target[i]
8.     target_function[target_number] =
     OR(target_function[target_number], cube)
9.     last = AND(last, NOT(cubes[i]))
```

Fig. 2. Forming a Circuit from a Cube Set

```
1. proc tcam_formulation(cubes, targets)
2.      buildCircuit()
3.      for each target_function in circuit
4.          target_function.flatten()
5.      return target_functions
```

Fig. 3. Procedure for TCAM Formulation

In mth iteration, as before, Invariant 2 holds by line 6 and Invariant 3 by the induction hypothesis and line 8. By induction, at the entry to the loop `target_function[j]` = $\bigvee_{i \leq m, t_i = T_j} \tilde{c_i}$ for each `j`. Line 8 OR's in `cube[i]` for `target[m]`. By Invariant 2, this is simply $\tilde{c_m}$, and therefore `target_function[target[m]]` = $\bigvee_{i \leq m, t_i = t_m} \tilde{c_i}$ for `target[m]`. ■

The preceding lemma serves to demonstrate that the circuit constructed by `buildCircuit` yields the correct functions `target_function`. For polynomial complexity, note that each iteration of the main loop creates exactly one `OR` and one `AND` gate, and hence the resulting circuit is of polynomial size in the size of the cube set.

The verification procedure for Problem 1 is as follows: construct the circuit corresponding to the original cube set, the circuit corresponding to the optimized cube set, and run circuit verification on the two circuits. Since circuit verification is well known to be in Σ_1, it follows that Problem 1 is in Σ_2.

It is important to note that algorithm `buildCircuit` can be used to complete Step (1) of the TCAM minimization procedure, above. We introduce the operator `flatten`, which forms a set of cubes from a multi-level network i.e., convert the function into its Disjunctive Normal Form (OR of ANDs). This algorithm is shown in Figure 3. We stress that this is not a practical solution. A simple multi-level network can result in an exponential number of cubes – this procedure is an existence proof, not an efficient solution.

Now that the complexity of the problem is analyzed, and a procedure to generate a complete, order-independent, explicit set of TCAM rows is given in the Figure 3, we turn to step (2) of the TCAM minimization procedure: minimizing the set of rows given by the formulation.

V. Sequential Cover

The *Ruleset Minimization Problem*, naively, is the minimization of the functions T_j defined in (6), as a standard multi-valued minimization problem, which were derived by Figure 3. This neglects two important facts, however. First, by construction: $T_i.T_j = 0, i \neq j$, which indicates that standard multi-valued minimization will simply yield the same result

```
1. minimize_sequential_cover(f1, ..., fm, DC)
2.     if DC contains f1, ..., fm return ∅
3.     find the primes P of f1, ...fm using DC as a don't-care set
4.     best_solution_size = -1
5.     foreach prime p ∈ P
6.          solution = minimize_sequential_cover(f1, ..., fm, DC + p)
7.          if best_solution_size == -1 or size(solution) < best_solution_size
8.                 best_solution = p + solution
9.             best_solution_size = size(best_solution)
10.    return best_solution
```

Fig. 4. Exact Minimization Procedure for Rulesets

as a series of single-valued minimization problems. A more important omission is the neglect of the precedence of the TCAM evaluation of the rules. The realization of the Eq. 6 will be a sequence of cube-target pairs over the header space: $(c_1, t_1), ..., (c_k, t_k)$, where t_j is the target of rule c_j. A solution which neglects precedence ordering insists:

$$\forall j \ c_j \subseteq T_i, \text{where } T_i = t_j \tag{7}$$

It is easy to see that if Eq. 7 is the sole constraint, besides the obvious requirement of

$$T_i = \sum_{t_j = T_i} c_j, \tag{8}$$

then ruleset minimization is simply the standard logic synthesis problem on the sequence of disjoint functions $T_1, ...T_n$. However, evaluation over a TCAM can be seen as a sequence of ternary lookups where once one succeeds, the sequence is abandoned. This permits us to expand the constraint in Eq. 7:

$$\forall j \ c_j \subseteq T_i + \sum_{l=1}^{j-1} c_l, \text{where } T_i = t_j \tag{9}$$

The optimization problem (8)-(9) introduces an order dependence into this problem. Formally, we introduce the *two-level minimization problem with sequential selection*: *Given* a sequence of functions $T_1, ..., T_n$, $T_i.T_j = 0$ for $i \neq j$, find a sequence of cubes $c_1, ..., c_k$ such that, for each j there exists some i such that

$$c_j \subseteq T_i + \sum_{l=1}^{j-1} c_l \tag{10}$$

and, for each i

$$T_i \subseteq \sum_{j=1}^{k} c_j \tag{11}$$

We define such a sequence of cubes as a *sequential cover* of $T_1, ..., T_n$. The *ruleset minimization problem* is therefore to find a *minimal sequential cover*, a sequential cover with the smallest number of cubes, of the target functions in Eq. 6.

VI. Minimal Sequential Cover

The optimization problem given by (10)-(11) has not been extensively studied. However, it is clearly related to the classic two-level logic synthesis problem. Moreover, (10) simply suggests that each preceding cube in a minimal solution acts as a don't-care set for the cube c_j. This observation leads to an exact minimization procedure shown in Figure 4, where $f_1, f_2, ..., f_m$ are the target functions to implement in TCAM and `DC` is initially set to $\emptyset$. Given a set of target functions, this algorithm first lists all primes (all possible cubes) for each function. Then with each one of those primes as dont-care, it recursively calls itself to compute the best sized solution possible with the chosen prime as the first cube in the final sequence. In the following, we prove that this algorithm always returns a minimal sequential cover.

Lemma 3: Procedure `minimize_sequential_cover` returns an irredundant sequential cover of the functions $f_1, ..., f_m$ with respect to don't-care set DC.

Proof: We proceed by induction on the size of the on-set $(f_1 + f_2 + ... + f_m)\overline{DC}$. When DC contains $f_1, ..., f_m$, the only irredundant sequential cover is the empty set, and the procedure returns this. Now assume the lemma holds for $|(f_1 + f_2 + ... + f_m)\overline{DC}| < N$, and consider the case $|(f_1 + f_2 + ... + f_m)\overline{DC}| = N$. We choose a prime p, which contains at least one point of the on-set $(f_1+f_2+...+f_m)\overline{DC}$. Hence, the recursive call on line 6 is a call to the procedure where $(f_1 + f_2 + ... + f_m)\overline{DC} < N$, and so by induction the procedure returns an irredundant sequential cover of $(f_1+f_2+...+f_m)\overline{DC + p}$. Suppose the algorithm returns the sequential cover $p, p_1, p_2, ..., p_n$. Since recursive call returns irredundant sequential cover, we only need to show that p is essential for this cover. We prove this through a simple contradiction. Suppose p is not essential, then p_1 should have been in the set of primes found in line 3. In this case, when line 5 choose this prime p_1 and line 6 is executed with DC+p_1 as dont-care set, the recursive call would have returned with $p_2, ..., p_n$ (based on induction hypothesis and the assumption that p is not essential) which would have led to a smaller sized solution. Since line 7 always chooses the best solution, this function should have returned $p_1, p_2, ..., p_n$ as the solution. Hence the contradiction; so, the solution returned by the function is irredundant. In order to complete the proof all we need observe is that we execute line 9 at least once, but this is trivial: on the first iteration of the loop 5-9 best_solution_size is -1, and so the condition on line 7 is true, and so we'll execute line 8. ■

The preceding lemma demonstrates that `minimize_sequential_cover` returns an irredundant cover. In order to prove that this returns a *minimal cover*, we need one more lemma.

Lemma 4: Let $c_1+c_2+...+c_N$ be a *minimal* solution of the

sequential minimization problem $\{f_1, ..., f_m\}$ with don't-care set DC. Then there is a minimal solution $p_1 + c_2 + ... + c_N$, where p_1 is a prime of $\{f_1, ..., f_m\}$ with don't-care set DC.

Proof: If we choose a prime $p_1 \supseteq c_1$, the result follows immediately. Since $c_1 + ... + c_N$ is a cover, and $p_1 \supseteq c_1$, $p_1 + c_2 + ... + c_N \supseteq c_1 + ... + c_N$, hence $p_1 + ... + c_N$ is a cover. Further, $p_1 + ... + c_N$ is irredundant, since otherwise $p_1 + ... + c_{j-1} + c_{j+1} + ... + c_N$ is a cover, and $|p_1 + .. + c_N| < |c_1 + ... + c_N|$, implying $c_1 + ... + c_N$ is not minimal. $|p_1 + ... + c_N| = |c_1 + ... + c_N|$, hence $p_1 + ... + c_N$ is minimal. ∎

These two lemmas suffice to demonstrate the correctness of `minimize_sequential_cover`.

Theorem 1: Procedure `minimize_sequential_cover` returns a minimal sequential cover of the functions $f_1, ..., f_m$ with respect to don't-care set DC.

Proof: (Omitted for space) Straightforward proof by induction similar to Lemma 2. ∎

Procedure `minimize_sequential_cover` is clearly expensive. It is an exhaustive-search algorithm, which further involves a potentially exponential procedure (generate all primes) at each step. This is clearly more expensive than the classic Quine-McClusky logic synthesis algorithm, and far more expensive than the efficient McGeer-Sanghavi-Brayton procedure [11]. However, it serves as a baseline for heuristic procedures.

VII. Dual-target case: PERMIT and DENY

In all of the datasets we have access to (see Section VIII), only `PERMIT` rules are specified along with (sometimes implicit and sometimes explicit) `DENY-all` rule at the end. At first blush, it appears that such a case is indistinguishable from standard two-level minimization. However, appearances are deceiving: this common case does yield extra opportunities for optimization. The implicit complement function can be made explicit, its cubes incorporated in the solution, and the results can be used to minimize beyond that of minimizing a single `PERMIT` function.

To take a simple example, consider a ruleset with a four-bit address (0-15). The explicit rule given is to `PERMIT` all accesses for ports 0-7 and ports 9-15. The minimum two-level logic solution for this is

```
PERMIT 0---
PERMIT --1-
PERMIT ---1
```

which is three rules. However, the *implicit* complementary rule is `DENY 8`. Using this, and using this as an implicit don't-care, we arrive at a two-entry solution:

```
DENY 1000
PERMIT ----
```

This is obviously a toy, trivial example. It is easy to build this into more complex examples, involving multiple `PERMIT` and `DENY` rules intermixed. This example serves to show the fundamental feature exploited in TCAM minimization not available in two-level minimization, the use of early rules as don't-cares for subsequent rules.

```
1. proc single_target(cubes)
2.   complement = negate(cubes)
3.   minimize_sequential_cover(cubes,
       complement, nil)
```

Fig. 5. Algorithm for a dual-target function

```
1. proc optimize(rules, complement)
2.   dc = empty
3.   foreach cube c of complement
4.     next_dc = dc + c
5.     next_rules = minimize(rules, dc)
6.     if (size(next_rules) <
              size(rules))
7.           dc = next_dc
8.           rules = next_rules
9. return (dc, rules)
```

Fig. 6. Heuristic Complement Subset Algorithm

The general algorithm for a dual-target case is shown in Figure 5. Again, this procedure is highly inefficient; negating, or complementing, a set of cubes can lead to an exponential blowup in the number of cubes (the Achille's Heel function $x_1x_2x_3 + x_4x_5x_6 + ... + x_{3n-2}x_{3n-1}x_{3n}$ is one well-known example), and then of course procedure `minimize_sequential_cover` is exponential. However, it serves as a baseline for heuristic and approximate algorithms.

A. Subsets of the Complement

The complement does not need to be formed completely to get some of the benefits of optimization. A subset of the complement can be formed when the entire complement cannot be, and this can be used for optimization. This can either be done by enumerating cubes of the complement until a sufficient number of cubes have been reached, or by quantification.

The algorithm in Figure 6 is a greedy heuristic which uses a subset of the complements to reduce the number of cubes in the TCAM. This algorithm returns a subset of the complement as the first rows of a TCAM, and then a set of rules for the original function. For example, if the function were a set of rules for `PERMIT`, this function would first create a set of `DENY` rules which would precede *all* the `PERMIT` rules. The size of the original TCAM is reduced over the optimized TCAM since the algorithm ensures that each complement cube added deletes at least one original cube.

B. Quantification

A second strategy to form a subset of the complement set is through the mathematical operation of *quantification*.

Quantification is a specific subclass of a general operator calculus on Boolean functions; quantifiers act to remove variables. In particular, given a variable x and function f, the *existential quantifier of f with respect to x*, denoted $\exists_x f$, is given by the function: $\exists_x f = f_x + f_{\overline{x}}$, where f_x is function f simplified with x set to 1, and $f_{\overline{x}}$ is f simplified with x set to 0.

```
1. proc quant_heuristic(rules)
2.   rules1 = minimize(rules)
3.   vars = vars_to_quantify(rules1)
4.   rules2 = existential_quantify(rules1,
               vars)
5.   optimize(rules1, complement(rules2))
```

Fig. 7. Quantification Heuristic Procedure

The dual quantifier to the existential quantifier is the *universal quantifier* $\forall_x f$, and it is given by the function $\forall_x f = f_x f_{\overline{x}}$.

One can quantify against a set of variables by using the relations:

$$\exists_x \exists_y f = \exists_{xy} f, \forall_x \forall_y f = \forall_{xy} f$$

Critically for our application, an analog of DeMorgan's Law (and, in fact, a direct consequence of DeMorgan's Law) holds: $\overline{\exists_x f} = \forall_x \overline{f_x}$ with the obvious dual:$\overline{\forall_x f} = \exists_x \overline{f_x}$.

Note that: $\forall_x f \subseteq f \subseteq \exists_x f$. From these, we can conclude that:

$$\overline{\exists_x f} = \forall_x \overline{f} \subseteq \overline{f}$$

The above suggests that we can form a subset of the complement set for a function by quantifying the given function over few variables and taking complement of that quantified function. Further, in sum-of-products form, $||\exists_x f|| \leq ||f||$, implying that existential quantification can be performed safely. In fact, operationally, one takes the existential quantifier of f with respect to x simply by replacing 1 and 0 with $-$ in the entry for x in each cube of f.

The preceding discussion indicates an approximation procedure: quantify out some variables and complement the result. This will compute a subset of the complement. Consider an example with a set of rules expressing `PERMIT` cases, where addresses and flags are specified. Consider the existential quantification of the flag variables: this yields the set of addresses which should be permitted under some set of flag values; the complement is the set of addresses which should *never* be permitted, independent of the flag settings. This set then can be used as the complement set for the algorithm in Figure 6. The algorithm in Figure 7 gives a sketch of the procedure.

C. A Hybrid Procedure

The procedure of Figure 7 often yields no improvement over standard two-level minimization, because the existential quantification yields too large a superset of the original function – and thus too small a value for the complement. To see an example, consider the case where almost all rules specify a destination IP address – but a few do not. Quantifying out all the variables outside the destination IP address, rules that do not specify the destination address result in all wild-card rules. Hence, the complement will be an empty set. Formally, we consider the case where $f(x, y) = g(x, y) + h(x)$, where $||h(x)|| << ||g(x, y)||$. We believe that $\forall_x \overline{f(x, y)}$ is a useful don't care set to optimize f, but if we blindly apply the quantification we get:

$$\exists_x f(x, y) = \exists_x g(x, y) + \exists_x h(x) = \exists_x g(x, y) + 1 = 1$$

```
1. proc partial_quantification
                (f(x,y) = g(x,y) + h(x))
2.   result = quant_heuristic(g)
3.   return h, result
```

Fig. 8. Partial Quantification Procedure

and therefore $\forall_x \overline{f(x, y)} = 0$, which leads to no optimization.

The solution is to exclude h from the complemented don't-care optimization, and to put the rules for h directly into the TCAM before complementing and optimizing g. We show this procedure in Figure 8.

It is important to note that the ordering in the TCAM is important here: the order *must* be h, DC, g', where DC and g' are the result of the algorithm shown in Figure 6.

VIII. Experimental Results

In this section, we describe the results from applying the mechanisms described in the previous sections on several artificial filter sets and a real firewall Access Control List. All of the filter sets that we have access to belong to the dual-target case as described in Section VII (a series of `PERMIT` statements followed by a single `DENY-all` statement).

A. Datasets

There is a scarcity of real life filter sets that are available to the general research community. ISPs are reluctant to provide real filter sets for security and confidentiality reasons. ClassBench [9], a suite of tools for benchmarking packet classification algorithms and devices from Washington University at St. Louis, includes a filter set generator tool that is shown to accurately model the characteristics of real life filters. We use the 12 parameter files provided in the ClassBench distribution [12] to generate filter sets that model 12 real filter sets that authors of ClassBench had access to. For this initial set of experiments, we generated filter sets with approximately 100 rules in each of them. Each rule contained two IP address prefixes (source and destination), two port ranges (source and destination), 8-bit transport protocol number, and 16-bits of flags. An implicit `DENY-all` rule at the end is assumed for these filter sets.

Fortunately, we were also able to get access to the Access Control List (ACL) on a firewall of our enterprise. This filter set has 1380 rules in total including a `DENY-all` rule at the end. All rules in this filter set are based on the standard 5-tuple (no flags field in contrast to the ClassBench generated rules). Source port in all rules is set to *any* and there are only few rules with destination port ranges specified (40/1380 = 2.89%). Only 9.85% rules have destination source port set to *any* and the remaining rules (87.2%) have a single number in the destination port (exact-match case). This distribution of destination ports differs dramatically from the average numbers reported in the ClassBench [9] paper: 40% wild-card entries, 49% exact match entries, and 11% ranges.

B. Methodology

In all our experiments, we used *espresso* [8] tool (now distributed as part of ABC software [13] from UC Berkeley)

Parameters	acl1	acl2	acl3	acl4	acl5	fw1	fw2	fw3	fw4	fw5	ipc1	ipc2	Avg
# Filters	99	100	100	100	100	96	93	96	100	96	100	90	
# Cubes	153	226	207	177	127	476	178	176	702	241	163	90	
Expansion Ratio	1.55x	2.26x	2.07x	1.77x	1.27x	4.96x	1.91x	1.83x	7.02x	2.51x	1.63x	1.03x	2.48x
Heuristic 1	81	119	201	173	127	461	3	43	611	125	142	38	
% Reduction	47.1%	47.3%	2.9%	2.3%	0.0%	3.2%	98.3%	75.6%	13.0%	48.1%	12.9%	59.1%	34.15%
Execution time (sec)	0.03	0.93	0.10	0.24	0.04	5.39	0.01	0.56	132.24	89.95	0.21	0.40	
# Complement cubes	1931	43737	4689	9467	3303	100599	31	15772	95754	173125	13117	13964	
Heuristic 2 $\|D_0\|$	2	3	39	18	5	0	0	0	0	0	13	0	
Heuristic 2 $\|P_1\|$	70	92	124	118	105	461	3	43	611	125	97	38	
Heuristic 2 Total	72	95	163	136	110	461	3	43	611	125	110	38	
% Reduction (over H1)	11.1%	20.2%	18.9%	21.4%	13.4%	0%	0%	0%	0%	0%	22.5%	0%	
% Reduction (overall)	52.9%	57.7%	21.3%	23.9%	13.4%	3.2%	98.3%	75.6%	13.0%	48.1%	32.5%	59.1%	41.6%
Execution Time(min.)	1.4	614	9.5	44.9	3.5	>720	0.01	108.8	>720	>720	47.4	64.4	

TABLE II
RESULTS FROM EXPERIMENTS WITH CLASSBENCH GENERATED FILTER SETS.

for boolean logic minimization and for computing complement of a given boolean function. To use espresso, we convert each filter set rules into a set of cubes. Note that expanding rules with port ranges results in multiple cubes [1], [2]. A rule in ClassBench filter sets results in one or more 120-bit cubes and a rule in our real ACL results in one or more 104-bit cubes.

We apply the following three heuristics on the datasets:

Heuristic 1: We run two-level logic minimization only on the `PERMIT` rules.

Heuristic 2: (Based on algorithm in Figure 6) We use espresso to compute complement of the permit rules. Because of the large complement sizes in most cases, espresso runs forever as it also performs a minimization step on the complement set of cubes. To avoid this, we run espresso with "-Dd1merge" option to only perform the expand operation and not perform minimization step. Then, we run logic minimization on the permit rules with each cube in the complement function (one at a time) as the dont-care cube. We select those cubes from the complement function that result in a reduction of cubes by more than one. We refer to this deny set as D_0 and this set is placed in front of the permit cubes, referred to as P_1, in the sequential cover.

Heuristic 3: As mentioned earlier, complement sets can be very large. So running Heuristic 2 for each of the cube in a large complement set can lead to large running times. So, we perform quantification on the cubes in the permit set with respect to all dimensions except for few chosen dimensions. For example, suppose we want to quantify on all bits except destination IP address bits. For each cube in the permit set, we modify all other bits to dont-cares except the destination address bits. Then we remove the cubes that become dont-cares in all bit positions as discussed in the Section VII-C. Suppose the set of original permit cubes corresponding to these removed quantified cubes be P_0. We compute complement function on the remaining cubes and run Heuristic 2 with those complement cubes to find D_0 and P_1. Hence the final order of result is P_0, D_0, and P_1. We automated this heuristic to quantify on source IP address, destination IP address, combination of both IP addresses, source port, destination port, and combination of both ports.

C. Results

ClassBench filter sets: We present the results from our experiments with the filter sets generated by the ClassBench tool in Table II. Note that ClassBench might output fewer filters than requested as it generates requested number of filters and then removes redundant filters, if any. We observe a varied expansion ratios as we expanded the filter rules into cubes, with an average of 2.48x. Most of them expanded by 2x or less except in two cases where there is an expansion of 5x and 7x. These two filter sets have a significant number of rules with port ranges that are not "any".

With Heuristic 1, we are able to achieve an average of 34% reduction in the number of cubes, though the reductions varied heavily across different sets. In *fw2* case, we observed a 98% reduction; upon analysis of the ClassBench filter set, we discovered one rule that subsumed a large number of other rules. For four sets (*acl3, acl4, acl5,* and *fw1*), we observed zero or very small reduction in the rules with this simple heuristic.

We ran Heuristic 2 with a time limit of 12 hours. In all cases where it found a non-empty D_0 set Heuristic 2 was able to achieve more than 10% reduction. Upon further analysis of these case, we observed that this heuristic is eliminating the cubes produced by range rules. Many of the range rules in the filter sets that lead to blow up in the cubes are of the type *1024-65535*, which expands into 6 cubes. But by having a single deny cube that represents range *0-1023*, all those 6 cubes can be merged down to a single cube. This observation has been previously made and leveraged by other researchers in [1], [2].

Heuristic 2 did not complete execution in some cases (denoted with ">720" in execution time) as the number of complement cubes is very high (hundreds of thousands) and also the execution time of an espresso minimization step is in the order of few seconds to hundred seconds. We ran Heuristic 3 for these rulesets with quantifications on source IP address, destination IP address, combination of both IP addresses, source port number, destination port number, and combination of both port numbers. But we did not find any further reduction in the ruleset size.

Real Firewall ACL: In Table III, we present the results from our experiments with the filter set from a real firewall ACL. As mentioned before, this filter set has very few range rules and hence a small expansion factor of 1.1 when expanding to cubes. Heuristic 1 is very effective on this filter

Number of Filters	1380
Number of Cubes	1504
Expansion Ratio	1.1x
Heuristic 1	900
Percentage Reduction	40.16%
Execution time (sec)	28.4
Number of complement cubes	145772
After Quantification (on all except Dest IP)	
Number of complement cubes	347
Heuristic 3 $\|P_0\|$	66
Heuristic 3 $\|D_0\|$	179
Heuristic 3 $\|P_1\|$	168
Heuristic 3 Total	413
Percentage Reduction (over H1)	54.1%
Percentage Reduction (overall)	72.5%

TABLE III
RESULTS FROM EXPERIMENTS WITH REAL FIREWALL ACL.

set achieving 40% reduction to 900 cubes. Heuristic 2 timed out with out resulting in any further reduction because of the large number of cubes in the complement. But Heuristic 3 was successful with quantification on all bits except destination IP address bits resulting in a 54% reduction on Heuristic 1 and 72.5% reduction compared to the original set.

IX. Related Work

Most of the previous work in TCAM rule optimization focused on optimizing the rules with ranges. Dong et. al. [1] propose a set of simple heuristics for minimizing the number of TCAM entries for a given set of filter rules. Their approach consists of applying four simple rules in an iterative fashion until no more entries can be reduced. Liu et. al. in TCAM-Razor [6] and Firewall Compressor [7] propose a dynamic programming algorithm that first constructs a reduced decision diagram, a canonical representation of a given ruleset, and minimize the number of prefixes associated with each uplink in the diagram. At the end, they run their all-match redundancy removal algorithm [14] that is shown to be optimal. The authors claim up to 96.1% reduction in some real world rulesets. As a future work, we plan to explore how these techniques map into the Boolean logic minimization framework that we pursue in this paper.

In [5], Liu leverages the common case of range rules [15]: there are only handful of distinct ranges in any particular field of a rule. Liu proposes using a separate bit for each range and hence claims great reduction in the number of TCAM entries. In contrast to encoding using standard binary notation, [2], [4] propose fence coding and gray coding of ranges in the rules to reduce the total number of TCAM entries. These techniques require converting the input from the standard binary encoding to the encoding chosen for TCAM programming.

There have been several proposals that present better TCAM architectures or even other novel hardware that can support large number of filters. In [16], Dong et. al. propose adding few registers and a bit of logic to the forwarding ASIC to perform packet classification at wire speeds. Che et. al. propose DRES [3], a dynamic range encoding scheme for TCAM co-processors, where ranges are encoded using P^2C algorithm [17] that allows N ranges to be encoded using only $\log_2(N+1)$ bits. This can be useful in designing next generation switches but can not be used with the existing network devices.

X. Conclusion

The goal of this paper is to formulate ruleset minimization for TCAM implementation in a boolean optimization framework. We present the formulation of the problem, analyze the complexity, present an exact minimization algorithm, and propose several heuristics. Our initial experimental results with artificial filter sets and a real firewall ACL are promising; we observe 72% reduction in the real firewall ACL rules with our heuristics.

We only scratched the surface in terms of exploring heuristics for the sequential cover problem and only for the dual-target case. But, we believe that our formulation provides a basic framework for incorporating heuristics proposed in other papers into the Boolean logic minimization framework. As future work, we plan to develop algorithms to tackle rulesets with much larger number of rules, rulesets with more than two actions, and incremental updates to the rulesets.

References

[1] Q. Dong, S. Banerjee, J. Wang, D. Agrawal, and A. Shukla, "Packet classifiers in ternary CAMs can be smaller," in *Proc. SIGMETRICS*, 2006, pp. 311–322.

[2] A. Bremler-Barr and D. Hendler, "Space-Efficient TCAM-based Classification Using Gray Coding," in *Proc. INFOCOM*, 2007, pp. 1388–1396.

[3] H. Che, Z. Wang, K. Zheng, and B. Liu, "DRES: Dynamic Range Encoding Scheme for TCAM Coprocessors," *IEEE Transactions on Computers*, 2008.

[4] K. Lakshminarayanan, A. Rangarajan, and S. Venkatachary, "Algorithms for advanced packet classification with ternary CAMs," in *Proc. SIGCOMM*, 2005, pp. 193–204.

[5] H. Liu, "Efficient Mapping of Range Classifier into Ternary-CAM," in *Proc. HOT Interconnects*, 2002, p. 95.

[6] C. R. Meiners, A. X. Liu, and E. Torng, "Tcam razor: A systematic approach towards minimizing packet classifiers in tcams," in *ICNP*, 2007.

[7] A. X. Liu, E. Torng, and C. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *INFOCOM*, Phoenix, Arizona, April 2008.

[8] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[9] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, 2007.

[10] R. M. Karp, *Complexity of Computer Computations*. Plenum, 1972, ch. Reducibility among Combinatorial Problems, pp. 85–103.

[11] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, "Espresso-signature: A new exact minimizer for logic functions," *IEEE Transactions on VLSI Systems*, vol. 1, no. 4, pp. 432–440, December 1993.

[12] http://www.arl.wustl.edu/~det3/ClassBench/index.htm.

[13] http://www.eecs.berkeley.edu/~alanmi/abc/.

[14] A. X. Liu, C. R. Meiners, and Y. Zhou, "All-match based complete redundancy removal for packet classifiers in TCAMs," in *INFOCOM*, April 2008.

[15] P. Gupta and N. McKeown, "Packet Classification on Multiple Fields," in *Proc. SIGCOMM*, 1999, pp. 147–160.

[16] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire Speed Packet Classification Without TCAMs: A Few More Registers (And A Bit of Logic) Are Enough," in *Proc. SIGMETRICS*, 2007.

[17] J. van Lunteren and A. Engbersen, "Fast and scalable packet classification," *IEEE Journal of Selected Areas in Communications*, 2003.