How to implement complex policies on existing network infrastructure

Pavel Chuprikov
IMDEA Networks Institute
Steklov Institute of Mathematics at St. Petersburg

Kirill Kogan
IMDEA Networks Institute

Sergey Nikolenko
Steklov Institute of Mathematics at St. Petersburg

ABSTRACT

Transport networks satisfy requests to forward data in a given topology. At the level of a network element, forwarding decisions are defined by flows. To implement desired data properties during forwarding, a network operator imposes economic models by applying policies to flows, ideally without dealing with underlying resource constraints. Policy splitting over multiple network elements under resource constraints is a hard optimization problem [6, 7]. We discuss limitations of the proposed methods and existing Boolean minimization techniques. The major contribution of this work is an optimal solution with linear time complexity at the price of a single bit forwarded in every packet. The results are supported by a comprehensive evaluation study that compares previous and currently proposed methods.

CCS CONCEPTS

• **Networks** → **Packet classification**;

KEYWORDS

network management; software-defined networking

ACM Reference Format:
Pavel Chuprikov, Kirill Kogan, and Sergey Nikolenko. 2018. How to implement complex policies on existing network infrastructure. In *SOSR '18: ACM SIGCOMM Symposium on SDN Research, March 28–29, 2018, Los Angeles, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3185467.3185477

1 INTRODUCTION

The modern network edge has to perform tasks with heterogeneous complexity, including features such as advanced VPN services, deep packet inspection, firewall, intrusion detection, to list just a few. It is not always possible to use cloud resources to implement desired policies because of proximity issues and multiple management domains. Each required feature may require a different amount of processing at the level of individual network elements and may introduce new challenges for traditional architectures, leading to serious implementation and performance issues. In particular, increasing complexity of services implemented in-network can require additional computing and memory resources, which usually leads to the need to upgrade already existing (and expensive) network infrastructure. The challenge, however, is to find scalable and manageable methods to support these complexities without upgrading the capabilities of individual network elements. One possibility is to define a "virtual pipeline" by splitting required resources along the path of a packet flow using some knowledge of the network. The problem of splitting a policy into several lookup tables while minimizing the maximal local table size has been broadly studied in [6, 7] and found to be an intractable optimization problem. Heuristics proposed in [6, 7] suffer from three main problems: they have high computational cost, the resulting total classifier size can grow significantly (exponentially in the worst case), and finally, none of them can handle dynamic fields where a header can change along the routing path as a result of actions applied on a switch. The main contribution of this work is an optimal algorithm with linear time complexity that can handle dynamic fields at the price of a single bit of metadata prepended to every packet.

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

2 MODEL DESCRIPTION

We begin with formal definitions needed for further exposition, starting with the basic notions of a packet header and classifier.

A packet *header* $H = (h_1, \ldots, h_w)$ is a sequence of bits: each bit $h_i \in H$ has a value of either zero or one, $h_i \in \{0, 1\}$, $1 \le i \le w$. For example, (1 0 0 0) is a 4-bit header. A *classifier* $\mathcal{K} = \{R_1, \ldots, R_n\}_{\prec}$ is an ordered set of rules with ordering $\prec$ (in all examples we assume that $R_i \prec R_{i+1}$), where each *rule* $R_i = (F_i, A_i)$ consists of a *filter* F_i and a pointer to the corresponding *action* A_i. A filter $F = (f_1, \ldots, f_w)$ is a sequence of, again, w values corresponding to bits in the headers, but this time possible bit values are 0, 1, and $*$ ("don't care").

Example 2.1. Consider a sample classifier $\mathcal{K}$ on $w = 4$ bits:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1
R_2	1	0	*	*	A_2
R_3	0	0	*	*	A_3
R_4	*	*	1	1	A_4

A classifier's main purpose is to find the action corresponding to the highest priority rule that matches a given header. A header H *matches* a rule R iff it matches R's filter, and it matches a filter F iff for every bit of H the corresponding bit of F has either the same value or $*$. The set of rules has a non-cyclic priority ordering $\prec$; if a header matches both R and R' for $R \prec R'$, the action of rule R is applied. In particular, in Example 2.1 the header (1 0 1 0) matches both R_1 and R_2, but A_1 is applied.

Two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$ are *equivalent* if they choose the same actions for every possible incoming packet; we assume that no action is applied if the packet does not match any rule.

3 SINGLE FLOW

As a building block in the automatic translation of network-wide policies to individual switch configurations, the work [6] introduced policy splitting over the path of the corresponding flow. To represent the equivalence between the original policy and its distributed representation, we define a *splitting* of a given classifier $\mathcal{K}$ to be a *sequence* of classifiers that is equivalent to $\mathcal{K}$. A sequence $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ operates as follows: an incoming packet p is matched in each $\mathcal{K}_i$ in order, immediately applying actions of the matched rule. We define an *l-splitting of $\mathcal{K}$* as a splitting of $\mathcal{K}$ into l classifiers.

Originally, the work [6] introduced the following problem that captures policy splitting for a single flow.

Problem 1 (FlowSplit). *Given a classifier $\mathcal{K}$ and a sequence of switch capacities $c_1, c_2, \ldots, c_l$, find an l-splitting $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ of $\mathcal{K}$ such that $\mathcal{K}_i$ has at most c_i rules.*

Policy splitting becomes a nontrivial problem if there are rules R and R' in a classifier $\mathcal{K}$ that *intersect*, i.e., there exists a header that matches both rules. The reason is that a packet can now be matched twice, as the following example illustrates.

Example 3.1. Let us split $\mathcal{K}$ from Example 2.1 in two parts:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1
R_3	0	0	*	*	A_3

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
R_2	1	0	*	*	A_2
R_4	*	*	1	1	A_4

Now a header (1 0 1 0) in the sequence $(\mathcal{K}_1, \mathcal{K}_2)$ is matched twice, and both A_1 and A_2 are applied to a packet.

One way to fix Example 3.1 is to add a special **nop** action to $\mathcal{K}_2$ that does nothing but forward a packet further.

$\mathcal{K}'_2$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	**nop**
R_2	1	0	*	*	A_2
R_3	0	0	*	*	**nop**
R_4	*	*	1	1	A_4

Now the sequence $\mathcal{K}_1, \mathcal{K}'_2$ is a splitting of classifier $\mathcal{K}$ from Example 2.1.

4 SINGLE FLOW WITHOUT METADATA

Two major works on policy splitting, *Palette* (Palette) [7] and *One-Big-Switch* (OBS) [6], employ different methods to avoid the undesirable effect of rule intersection and preserve equivalence. In this section, we discuss the main ideas behind previously proposed methods. Next, we show how to apply *Boolean Minimization* (BM) techniques [1, 9] in a way that can give more flexibility but ultimately also limited. Finally, we discuss three major drawbacks that are common to all three approaches: slow running time, extensive rule duplication, and lack of support for dynamic fields.

4.1 Palette

Palette's idea [7] is to produce a splitting $\mathcal{K}_1, \ldots, \mathcal{K}_l$ that avoids any rule intersection between $\mathcal{K}_i$ and $\mathcal{K}_j$ for $i \neq j$. Two methods for building such a splitting were proposed: pivot-based and cut-based. Both methods expand "don't care" bits of possibly intersecting rules in order producing several non-intersecting sets of rules, as the following example shows.

Example 4.1. Expanding bit #1 in the classifier $\mathcal{K}$ from Example 2.1 yields two non-intersecting classifiers:

$\mathcal{K}^0$	#1	#2	#3	#4	*Action*
R_1^1	1	*	1	0	A_1
R_2	1	0	*	*	A_2
R_4^1	1	*	1	1	A_4

$\mathcal{K}^1$	#1	#2	#3	#4	*Action*
R_1^0	0	*	1	0	A_1
R_3	0	0	*	*	A_3
R_4^0	0	*	1	1	A_4

The rules from different subclassifiers do not intersect, hence, $\mathcal{K}^0$ and $\mathcal{K}^1$ form a 2-splitting of $\mathcal{K}$.

The pivot-based method is a top-down iterative approach. At every iteration, we consider the current splitting $\mathcal{K}_1, \ldots, \mathcal{K}_k$ of $\mathcal{K}$, starting from a single classifier $\mathcal{K}$. Among the current splitting, we choose the classifier $\mathcal{K}_{i^*}$ with the largest number of rules and search for a bit index j whose expansion results in $\mathcal{K}^0_{i^*}$ and $\mathcal{K}^1_{i^*}$ that minimize the value of $\max\{|\mathcal{K}^1_{i^*}|, |\mathcal{K}^0_{i^*}|\}$. Finally, we replace $\mathcal{K}_{i^*}$ with $\mathcal{K}^1_{i^*}$ and $K^0_{i^*}$. Example 4.1 in fact presents the result of the first iteration of this procedure; note that if bit #2 were expanded instead, the expansions

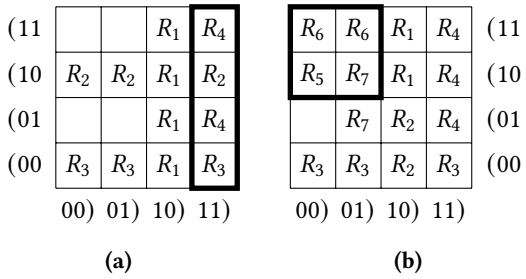

Figure 1: Classifiers from Example 2.1 and Example 4.2 represented in 2D. Bits #1 and #2 correspond to rows; bits #3 and #4, to columns. Each square shows the highest priority rule that matches the corresponding header.

would be of size 2 and 4. The time complexity of this algorithm is $O(nwl)$, where $n = |\mathcal{K}|$, w is the classification width, and l is the path length.

One of Palette's drawbacks is that it minimizes the maximal table size, which essentially assumes that switch capacities are equally utilized, $c_1 = c_2 = \ldots = c_n$. Since routing decisions of packet flows are decoupled from applying policies, this assumption significantly reduces the applicability of Palette in practice. Palette's cut-based approach uses graph-cut heuristics as a black box, and it is even harder to adjust for heterogeneous capacities.

4.2 One Big Switch

Palette not only optimizes a slightly different objective but also has fundamental limitations. First, an early decision to expand a bit propagates further to where it may be a suboptimal decision. Second, Palette always cuts the header space in half, which leads to worse performance when the path length is not a power of two. The *One Big Switch* (OBS) approach [6] avoids both limitations by taking a different *incremental* path.

The OBS algorithm also begins with a classifier $\mathcal{K}^{(0)} = \mathcal{K}$ but constructs the splitting $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_l$ incrementally, one classifier at a time. At step i, the algorithm chooses a subregion r_i of the header space that overlaps with at most c_i rules from $\mathcal{K}^{(i-1)}$. These overlapping rules are put into $\mathcal{K}_i$; if an overlap is only partial then the rule is first cut by r's boundary. Finally, $\mathcal{K}^{(i)}$ is constructed by taking all rules from $\mathcal{K}^{(i-1)}$ that are not completely covered by r_i together with a special rule $(r_i, \textbf{nop})$ with the highest priority that guarantees that packets will not be matched more than once. Note that partially overlapping rules are present both in $\mathcal{K}_i$ and $\mathcal{K}^{(i)}$, which is similar to the rule duplication effect of Palette.

To illustrate this idea, we draw the header space in two dimensions (see Figure 1), with half of the bits shown along one axis, and the other half shown along the other axis. The classifier $\mathcal{K}$ from Example 2.1 can be represented in this space as in Figure 1a. Each square shows the highest priority rule from $\mathcal{K}$ that matches the corresponding header; e.g., for $H = (1\,0\,1\,1)$ the highest priority matching rules is R_2 (shown in the intersection of the second row and fourth column). A possible subregion $r_1 = (*\ *\ 1\ 1)$ for $c_1 = 3$ is shown in bold.

The OBS algorithm chooses r to be the rectangle in this two-dimensional space that maximizes the ratio $\frac{n_\subseteq - 1}{n_\cap}$, where $n_\subseteq$ is the number of rules that lie inside r and $n_\cap$ is the number of rules that overlap with r. The time complexity of the overall algorithm is $O(n^5 wl)$, where $n = |\mathcal{K}|$, w is the classification width, and l is the path length. In practice, however, we will see in Section 7 that OBS is not quite as slow as one might think.

Comparing OBS and Palette using the classifier from Example 2.1, it may seem that OBS is inferior since it cannot fit into the required capacity: any r_1 leaves four rules for the next switch. However, according to the evaluation study in [6] this is a rather rare case, and most of the time higher flexibility of OBS leads to better results, as in the example below.

Example 4.2. Consider a FlowSplit instance with $c_1 = 3$, $c_2 = 6$, and the following classifier:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	*	1	0	A_1
R_2	0	*	1	0	A_2
R_3	0	0	*	*	A_3
R_4	*	*	1	1	A_4
R_5	1	0	0	0	A_5
R_6	1	1	0	*	A_5
R_7	*	*	0	1	A_5

The pivot-based method will not satisfy the capacities, since expanding any bit gives a subclassifier with at least four rules. From the two-dimensional representation on Figure 1b we conclude that OBS can satisfy the requirements by taking $r_1 = (1\ *\ 0\ *)$, which covers R_5 and R_6 and partially overlaps with R_7. $\mathcal{K}_2$ is left with R_1, R_2, R_3, R_4, R_7, and $(r_1, \textbf{nop})$.

4.3 Boolean Minimization

The OBS is more flexible than Palette, but it requires the covered region to be a rectangle. We suggest an even more flexible *Boolean Minimization* (BM) approach that drops this requirement and allows to assign arbitrary subsets of rules to a switch, at the expense of adding more **nop** rules to subsequent switches.

Example 4.3. The classifier from Example 2.1 can be represented more efficiently than Palette with general **nop**

rules:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1
R_2	1	0	*	*	A_2
R_3	0	0	*	*	A_3

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
$R_{\textbf{nop}}$	*	0	*	*	**nop**
R_4	*	*	1	1	A_4

Now $R_{\textbf{nop}}$ is not a prefix on its first two bits, so it would not be a rectangle on Figure 1a.

We use an incremental approach similar to OBS. At step i, we allow an *arbitrary* subset $\mathcal{K}'_i \subseteq \mathcal{K}^{(i-1)}$ to be a candidate for $\mathcal{K}_i$. Once $\mathcal{K}'_i$ has been chosen, we consider $\mathcal{K}^{(i-1)}$, set the action to **nop** for all rules in $\mathcal{K}^{(i-1)} \setminus \mathcal{K}'$, and run one of the Boolean minimization heuristics [1, 9] to remove redundant rules (in Example 4.3 we choose $\mathcal{K}'_1 = \{R_1, R_2, R_3\}$). If the result fits in c_i, it is selected as $\mathcal{K}_i$. To construct $\mathcal{K}^{(i)}$ we take $\mathcal{K}^{(i-1)}$, set actions for all rules in $\mathcal{K}' \cap \mathcal{K}^{(i-1)}$ to **nop**, and run Boolean minimization again (in Example 4.3 this leads to $\mathcal{K}^{(2)} = \mathcal{K}_2$).

We use heuristics for Boolean minimization since it is NP-hard to find the exact solution. The time complexity of those heuristics is $O(n^3 l w \log n)$, where n is the number of rules in the classifier, w is the rule width in bits, and l is the path length.

One disadvantage of BM is that when a low-priority rule intersects many higher-priority rules, the switch with the low-priority rule must contain all the intersecting higher-priority rules, either with **nop** or with the original action. Consider the following example:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	1	0	1	A_1
R_2	1	0	1	0	A_2
R_3	1	1	1	0	A_3
R_4	*	0	0	1	A_4
R_5	1	*	*	*	A_5

BM is not able to satisfy capacity requirements $c_1 = c_2 = 4$ since a switch with R_5 must also hold the other four rules to avoid spurious applications of A_5. Both OBS and Palette are able to find a feasible solution.

4.4 Incomparability of considered methods

We have seen that OBS, Palette, and BM are all incomparable in general. OBS potentially has greater flexibility than Palette but cannot be extended to arbitrary bit strings. BM has higher flexibility in rule placement than OBS and Palette but performs badly when one rule intersects too many others.

At the same time, there are limitations common to all three methods:

(1) *memory expansion* due to rule duplication from expanded bits (in Palette), partially overlapping rules (in OBS), or **nop** rules (in BM);
(2) *slow running time* due to the high complexity of underlying computational problems;
(3) *inability to handle dynamic fields*, i.e., fields that are changed by actions and participate in the classification process.

In the worst case, rule duplication may cause a memory increase proportional to the length of the path. Below we show such an example for Palette:

$\mathcal{K}$	#1	#2	#3	#4	*Action*
R_1	1	1	*	*	A_1
R_2	1	*	1	*	A_2
R_3	1	*	*	1	A_3
R_4	*	1	1	*	A_4
R_5	*	1	*	1	A_5
R_6	*	*	1	1	A_6

Correctness under dynamic fields is a more subtle issue. Consider the splitting $(\mathcal{K}^0, \mathcal{K}^1)$ from Example 4.1. Assume that A_2 sets the first bit to zero, while A_3 sets it to one. Let us now try to match the header (1 0 0 0):

(1) (1 0 0 0) matches R_2 of $\mathcal{K}^0$ and changes to (0 0 0 0);
(2) (0 0 0 0) matches R_3 of $\mathcal{K}^1$, changing back to (1 0 0 0).

Dynamic fields can result in surprising interactions of seemingly independent actions, and equivalence can be violated in ways that are very hard to detect.

In the following, we propose a novel method that avoids all the above drawbacks by actually *exploiting* dynamic fields.

5 ONE-BIT PRICE FOR SIMPLICITY

In this section, we discuss the benefits of having a single extra bit of metadata that can be set, passed between network elements, and used in classification. In particular, we show how to satisfy any switch capacities once they sum up to at least $|\mathcal{K}|$. Moreover, the transformation will remain equivalent even in the presence of dynamic fields.

To understand how a single bit of metadata helps, we consider a very naive approach to FlowSplit. For a classifier $\mathcal{K}$ and a sequence of switch capacities $c_1, c_2, \ldots, c_l$ such that $\sum_i c_i \geq |\mathcal{K}|$, we simply put the first (top priority) c_1 rules to the first switch, then next c_2 rules to the second switch, and so on until all rules in $\mathcal{K}$ are covered. Unfortunately, a packet can be matched twice in a way similar to Example 2.1.

The remedy for this problem is obvious: do not let a packet to be matched twice! To implement this we use an extra metadata bit, which we call **matched**, that shows whether a packet was already matched. This bit is initially set to 0 and changed to 1 on any match. We perform classification for a packet only if **matched** is not set. Algorithm 1 (OneBit) formalizes this idea.

Example 5.1. Consider the classifier $\mathcal{K}$ from Example 2.1. The OneBit algorithm splits its rules into two classifiers $\mathcal{K}_1$ and $\mathcal{K}_2$, augmenting every non-default action of $\mathcal{K}_1$ with **matched** $\leftarrow 1$ and the default **nop** action with **matched** $\leftarrow 0$:

$\mathcal{K}_1$	#1	#2	#3	#4	*Action*
R_1	*	*	1	0	A_1, **matched** $\leftarrow 1$
R_2	1	0	*	*	A_2, **matched** $\leftarrow 1$
R_3	0	0	*	*	A_3, **matched** $\leftarrow 1$
default : **matched** $\leftarrow 0$					

$\mathcal{K}_2$ runs classification only if **matched** has not been set:

$\mathcal{K}_2$	#1	#2	#3	#4	*Action*
if matched $= 0$ **then**					
R_4	*	*	1	1	A_4, **matched** $\leftarrow 1$

Note that without the **matched** bit the header (1 0 1 1) would be erroneously matched to both R_2 in $\mathcal{K}_1$ and R_4 in $\mathcal{K}_2$.

Instead of checking the **matched** bit prior to classification we could incorporate this check into the classifier itself, similar to **nop**-actions of OBS and BM. But in this case we would be wasting processing cycles for packets that were already matched. One of the advantages of the **matched** bit approach is that it avoids these redundant efforts.

The following theorem demonstrates that OneBit operates correctly.

Theorem 5.2. *Given a FlowSplit's instance $(\mathcal{K}, \{c_i\})$ with $\sum_i c_i \geq |\mathcal{K}|$, the OneBit algorithm constructs, in time $O(|\mathcal{K}|)$, an l-splitting of $\mathcal{K}$ that remains correct in the presence of dynamic fields and has the same total number of rules $|\mathcal{K}|$.*

Proof. Assuming $|\mathcal{K}| \gg n$, the running time is dominated by line 5 that adds rules to $\mathcal{K}_i$. Since every added rule is removed from $\mathcal{K}$ in line 10, $\sum_i \mathcal{K}_i = |\mathcal{K}|$. Lines 7 and 9 use the **matched** bit to make sure that no packet is matched more than once; since none of $\mathcal{K}$'s rules depend on **matched**, dynamic fields do not violate this guarantee. Finally, consider an arbitrary header H that has been matched in $R \in \mathcal{K}$ and put into $\mathcal{K}_j$. H does not match any rule $R' \prec R$; thus, due to line 3 it is not a match to any $\mathcal{K}_{j'}$ with $j' < j$, and necessarily H is matched by R in $\mathcal{K}_j$. □

Hence, a single **matched** bit allows to construct a simple heuristic for the FlowSplit problem, which (1) does not expand rules, (2) has linear time complexity, and (3) remains correct even in the presence of dynamic fields.

6 NETWORK-WIDE POLICY SPLITTING

The FlowSplit problem considers the splitting of a policy controlling the behavior of a single flow. On the network-wide level, there are several flows, and each has its own policy and its own forwarding path. The authors of OBS suggested in [6] the following problem for joint multi-flow splitting.

Algorithm 1 OneBit$(\mathcal{K}_\prec, c_1, \ldots, c_l)$

```
1:  Initialize K_1, K_2, ... K_l to empty classifiers
2:  for i in 1, 2, ..., l do
3:      Let R_i be c_i highest priority rules of K
4:      for (F, A) ∈ sorted(R, ≺) do
5:          Append (F, [A, matched ← 1]) to K_i
6:      if i = 1 then
7:          Set default action to matched ← 0 in K_i
8:      else
9:          Make K_i conditioned on matched = 1
10:     Remove R from K
11: return K_1, K_2, ..., K_l
```

Problem 2 (MultiFlowSplit). *Given a set of nodes V, node capacities $c : V \to \mathbb{N}$, and a set of k path/classifier pairs $(P_i, \mathcal{K}_i)$, where P_i is a sequence of vertices $(v^i_1, \ldots, v^i_{l_i})$, find a capacity allocation $a : V \times [k] \to \mathbb{N}$ such that $\sum_{i=1}^{k} a(v, i) \leq c(v)$, and for each i the pair $(P_i, \mathcal{K}_i)$ is a solution for FlowSplit with $\mathcal{K} = \mathcal{K}_i$ and $c_j = a(v^i_j, j)$ for $j = 1, \ldots, l_i$.*

The approach of OBS to solving MultiFlowSplit [6] contains three steps that are repeated iteratively:

(1) estimate the total capacity η_i required by the ith flow;
(2) use η_i to allocate per-path per-node capacities by solving a linear programming problem;
(3) *try* to solve the FlowSplit problem for each flow.

If the OBS algorithm cannot solve FlowSplit for some flows on step (3), the algorithm goes back to (1), where η_i are increased for failed flows. Unfortunately, there is no clear bound on the number of iterations.

The main reason that iterations were needed at all is that η_i are mere estimations: the success of the OBS algorithm on each FlowSplit problem depends not only on the total capacity but also on the capacities of individual switches.

Fortunately, if we use OneBit with a similar approach, Theorem 5.2 implies that we succeed iff $\sum_i c_i \geq |\mathcal{K}|$. Thus, if we take $\eta_i = |\mathcal{K}_i|$ we will be able to solve MultiFlowSplit in a single iteration of steps (1)-(3), and the resulting algorithm is guaranteed to work in polynomial time.

7 EVALUATION RESULTS

To evaluate the proposed approaches for FlowSplit, we ran simulations on 12 classifiers from Classbench [18] generated with real parameters, each with ≈ 10K rules on 6 fields. In particular, we used the ACL test suite since it is a more likely example of a splittable policy. We investigated only the uniform version of FlowSplit, where all capacities are equal. Importantly, we applied the Boolean minimization procedure as described in [10] to make the comparison more fair for those algorithms that do not use any classifier simplification

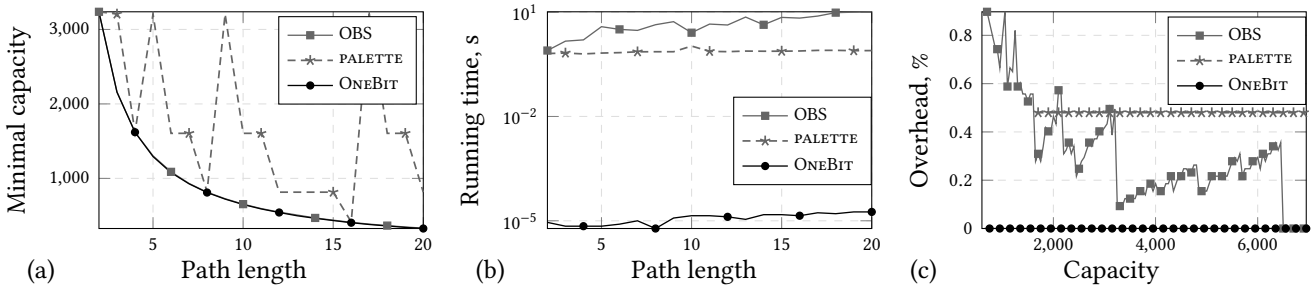

Figure 2: Experimental results on a ClassBench ACL classifier with 9928 rules.

internally. We used the pivot-based heuristic for Palette because the cut-based heuristic took too much time to complete. The code for our simulations is available at GitHub [4].

Results of our experimental evaluation are summarized in Figure 2, which shows detailed results for an ACL classifier, and Table 1, which shows other classifiers in our study; OneBit's overhead is not shown in Table 1 because it is always 0. In the first experiment (Fig. 2a, $c_{\min}$ column in Table 1), we investigated what is the minimal switch capacity for which an algorithm is able to accommodate the classifier, and how it depends on the path length. We computed these results with a binary search of different capacity values. We see that, as expected, OneBit is optimal, with OBS being several times worse in some cases (e.g., acl3 and ipc2 in Table 1), and Palette often imposing significant overhead in terms of extra capacity. The second experiment (Fig. 2b, t in Table 1) shows the running time of different algorithms. We see that OneBit, as expected, is several orders of magnitude faster than both Palette and OBS. In Table 1, we computed the runtimes on the same capacity value (the largest of minimal capacities) so that they would be comparable. The third experiment (Fig. 2c, OH in Table 1) shows the total (relative) overhead of different algorithms' representations in terms of actual increase in the total size of the resulting classifiers. OneBit is the optimal algorithm with zero overhead, OBS performs well in terms of this metric on the ACL classifier in Fig. 2c, but leads to a substantial overhead for several other classifiers (e.g., acl3 and fw2 in Table 1), and Palette is noticeably worse. Thus, evaluations support our theoretical conclusion that OneBit is the best possible with respect to all metrics.

		Palette			OBS			OneBit	
	Size	$c_{\min}$	OH,%	t,s	$c_{\min}$	OH,%	t,s	$c_{\min}$	t,s
acl1	9928	2480	2.66	0.68	1030	3.49	11.03	997	$\leq 10^{-5}$
acl2	7433	2308	47.50	0.54	1214	60.85	105.81	746	$\leq 10^{-5}$
acl3	9149	3622	123.49	0.63	2687	157.15	1638.02	919	$\leq 10^{-5}$
acl4	8059	2870	98.52	0.59	1706	106.82	434.41	809	$\leq 10^{-5}$
acl5	9072	2265	0.04	0.63	912	0.20	2.60	910	$\leq 10^{-5}$
fw1	8902	3776	132.33	0.67	2423	159.35	80.42	891	$\leq 10^{-5}$
fw2	9968	4308	160.03	0.70	3688	250.71	651.36	997	$\leq 10^{-5}$
fw3	8029	3877	180.91	0.59	2818	243.77	433.64	804	$\leq 10^{-5}$
fw4	2633	1196	173.15	0.31	800	192.10	31.63	268	$\leq 10^{-5}$
fw5	8136	2651	81.31	0.58	1780	113.03	62.38	814	$\leq 10^{-5}$
ipc1	8338	2260	26.52	0.58	1088	29.89	42.49	837	$\leq 10^{-5}$
ipc2	10000	4348	134.10	0.67	2406	111.51	300.37	1002	$\leq 10^{-5}$

Table 1: Experimental results on ClassBench classifiers with path length $l = 10$: min capacity $c_{\min}$, overhead OH, and runtime t.

8 RELATED WORK

In addition to Palette [7] and OBS [6], there are other works that deal with application of network-wide policies; unfortunately, they do not consider switch resource constraints and as a result are of limited applicability in practice [2, 5, 21]. Other works consider the policy splitting while changing the routing [14, 15, 20]. This setting is different from Palette, OBS, and our approach, where routing is not affected as a result of policy splitting. Optimizing classifiers on a single switch is a well-developed line of research, with numerous heuristics suggested in works such as [3, 8, 11–13, 16, 17, 19]. In this light it is interesting to explore applications of Boolean minimization methods to reduce memory requirements that we apply on input classifiers [1, 9] prior to running policy splitting methods.

9 CONCLUSION

In this work, we have studied the automatic translation of a network-wide policy to individual switch configurations. In particular, we have explored efficient implementations of the path splitting algorithm used for such translations. We show that at the price of a single bit per packet the intractable combinatorial problem previously considered in [6, 7] can be solved optimally in linear time.

Acknowledgements. We are grateful to the authors of [7] and [6] for providing us with the source code for their algorithms. This project has been made possible in part by a grant from the Cisco University Research Program Fund, an advised fund of Silicon Valley Community Foundation.

REFERENCES

[1] Eric Allender, Lisa Hellerstein, Paul McCabe, Toniann Pitassi, and Michael E. Saks. 2008. Minimizing Disjunctive Normal Form Formulas and AC^0 Circuits Given a Truth Table. *SIAM J. Comput.* 38, 1 (2008), 63–84.

[2] Martín Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. 2009. Rethinking enterprise network control. *IEEE/ACM Trans. Netw.* 17, 4 (2009), 1270–1283.

[3] Pavel Chuprikov, Kirill Kogan, and Sergey I. Nikolenko. 2017. General ternary bit strings on commodity longest-prefix-match infrastructures. In *ICNP*. 1–10.

[4] Code for simulations 2017. Code for simulations. https://github.com/distributedpolicies/submission. (2017).

[5] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. 2000. Implementing a distributed firewall. In *CCS*. 190–199.

[6] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. 2013. Optimizing the "one big switch" abstraction in software-defined networks. In *CoNEXT*. 13–24.

[7] Yossi Kanizo, David Hay, and Isaac Keslassy. 2013. Palette: Distributing tables in software-defined networks. In *INFOCOM*. 545–549.

[8] Alexander Kesselman, Kirill Kogan, Sergey Nemzer, and Michael Segal. 2013. Space and speed tradeoffs in TCAM hierarchical packet classification. *J. Comput. Syst. Sci.* 79, 1 (2013), 111–121.

[9] Subhash Khot and Rishi Saket. 2008. Hardness of Minimizing and Learning DNF Expressions. In *FOCS*. 231–240.

[10] Kirill Kogan, Sergey I. Nikolenko, Patrick Eugster, and Eddie Ruan. 2014. Strategies for Mitigating TCAM Space Bottlenecks. In *HOTI*. IEEE, 25–32.

[11] Kirill Kogan, Sergey I. Nikolenko, Patrick Eugster, Alexander Shalimov, and Ori Rottenstreich. 2017. Efficient FIB Representations on Distributed Platforms. *IEEE/ACM Trans. Netw.* 25, 6 (2017), 3309–3322.

[12] Kirill Kogan, Sergey I. Nikolenko, Ori Rottenstreich, William Culhane, and Patrick Eugster. 2016. Exploiting Order Independence for Scalable and Expressive Packet Classification. *IEEE/ACM Trans. Netw.* 24, 2 (2016), 1251–1264.

[13] Chad R. Meiners, Alex X. Liu, and Eric Torng. 2012. Bit Weaving: A Non-Prefix Approach to Compressing Packet Classifiers in TCAMs. *IEEE/ACM Trans. Netw.* 20, 2 (2012), 488–500.

[14] Masoud Moshref, Minlan Yu, Abhishek B. Sharma, and Ramesh Govindan. 2012. vCRIB: Virtualized Rule Management in the Cloud. In *HotCloud*.

[15] Masoud Moshref, Minlan Yu, Abhishek B. Sharma, and Ramesh Govindan. 2013. Scalable Rule Management for Data Centers. In *NSDI*. 157–170.

[16] Sergey I. Nikolenko, Kirill Kogan, Gábor Rétvári, Erika R. Bérczi-Kovács, and Alexander Shalimov. 2016. How to represent IPv6 forwarding tables on IPv4 or MPLS dataplanes. In *INFOCOM Workshops*. 521–526.

[17] Eric Norige, Alex X. Liu, and Eric Torng. 2013. A Ternary Unification Framework for optimizing TCAM-based packet classification systems. In *ANCS*. 95–104.

[18] David E. Taylor and Jonathan S. Turner. 2007. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Trans. Netw.* 15, 3 (2007), 499–511.

[19] Rihua Wei, Yang Xu, and H. Jonathan Chao. 2012. Block permutations in Boolean Space to minimize TCAM for packet classification. In *INFOCOM*. 2561–2565.

[20] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. 2010. Scalable flow-based networking with DIFANE. In *SIGCOMM*. 351–362.

[21] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. 2006. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *S&P*. 199–213.