The Small, the Fast and the Lazy (SFL): A General Approach for Fast and Flexible Packet Classification

Sven Hager Samuel Brack Björn Scheuermann

Computer Engineering Group
Humboldt University of Berlin, Germany
Email: {hagersve, samuel.brack, scheuermann}@informatik.hu-berlin.de

***Abstract*—Packet classification—the matching of packet headers against a predefined rule set—is a crucial functionality of firewalls, intrusion detection systems, and SDN switches. Most existing classification algorithms trade setup time for classification speed—that is, the packet classification is fast, but the transformation of rules set into the corresponding search data structure takes a considerable amount of time. This preprocessing time, however, poses a significant challenge for systems where rule sets can often change. Hence, these systems often use slow classification algorithms that support frequent rule set updates, which drastically limits their achievable throughput. In this work, we present a novel algorithmic technique which is able to "upgrade" an arbitrary existing classification algorithm to support fast updates, while still providing high lookup performance. Our evaluation demonstrates that our proposed technique exceeds the matching performance of existing dynamically updatable algorithms by an order of magnitude while providing the same level of update responsiveness.**

***Index Terms*—Packet classification; Hybrid search structure**

I. Introduction

Network packet classification is a core functionality at the heart of packet processing systems such as routers, SDN switches, and firewalls. These devices distinguish between incoming network packets based on header information carried within the packets to implement QoS routing, forwarding tables, or security policies [1], [2]. To this end, a set of selected header fields, such as IP addresses, protocol numbers, or transport layer ports, is matched against a *rule set* installed on the device which specifies how different classes of packets should be treated [3], [4]. For example, a firewall could be configured to drop every incoming packet which is not addressed to an institution's web server, whereas an OpenFlow switch could notify the controller for each packet directed to a specific application. The matching process itself is driven by a *classification algorithm* which determines the most highly prioritized matching rule for the currently regarded packet.

The difficulty of network packet classification arises from the high performance requirements in order to meet line speed. Accordingly, the research community has proposed a wide variety of approaches to accelerate the classification process, ranging from fast classification algorithms [5]–[11] and rule set optimization techniques [12]–[17] to hardware-centric approaches [18]–[24]. Most of these works require significant preprocessing times to set up their search data structures, which in turn can be traversed quickly when a packet enters the classification system. In consequence, they provide excellent lookup performance in setups where the rule set does not change often, such as static security policies. However, if the classification system is used in dynamic environments with frequent rule set changes at run time, such as an SDN, the ability to quickly update the search structure is of paramount importance. Unfortunately, existing approaches that support dynamic updates either come with slow classification performance [9] or require specific hardware setups [19], [24].

In this work, we contribute the *SFL* approach, which is a technique to equip a given classification algorithm with the ability to quickly process updates while still maintaining high lookup performance. Specifically, we can augment an arbitrary existing classification algorithm $\mathcal{A}$ (the *F*ast) with a list-based update buffer $\mathcal{B}$ (the *S*mall). Rule set updates for the classification system, which are applied at system run time, are not installed immediately in the search structure of $\mathcal{A}$, but are inserted in the update buffer $\mathcal{B}$ (the *L*azy). When a network packet is to be classified, its header fields are first matched using $\mathcal{A}$'s search data structure to compute a preliminary classification decision. Subsequently, this decision is checked based on the buffer contents whether it is in conflict with a rule set update and is potentially modified. After sufficiently many updates have been collected, the classification data structure can be re-built once, thereby flushing the update buffer.

The main results of our evaluation are threefold: first, we demonstrate that existing fast classification algorithms fail to meet the requirements of highly dynamic environments, which results in severe throughput penalties. Second, we show that existing algorithms which support high update rates fall short in terms of throughput. Third, we show that fast SFL-"upgraded" algorithms perform significantly faster in dynamic environments than both existing fast and updateable classification algorithms. Specifically, some SFL-equipped algorithms can perform about an order of magnitude faster than the state-of-the-art dynamic algorithm *Tuple Space Search* [9], [25] while processing up to 60 updates per second.

The remainder of this paper is structured as follows: in Section II, we discuss related work. In Section III, we introduce the packet classification and rule set update problems. Section IV describes the proposed SFL algorithm, which is subsequently evaluated in Section V. Finally, we conclude this paper in Section VI.

DOI 10.1109/LCN.2016.125

II. Related Work

Existing approaches to solve the packet classification problem can be subdivided into three not strictly disjoint categories: *classification algorithms*, *rule set optimization techniques*, and *hardware-based approaches*. This section discusses these existing schemes and points out the differences to the proposed SFL approach.

A. Classification Algorithms

Packet classification algorithms are at the heart of every networked system, which has to distinguish network packets based on their header data. The most straightforward classification algorithm is a linear search through the installed rule set: as the name suggests, each rule in the rule set is tested whether it matches the currently regarded packet, until the first matching rule is found. Tuple Space Search [9] is a hash-based classification algorithm that partitions the rule set into equivalence classes, which are searched by hashing certain parts of the packet header. Both linear search and Tuple Space Search have a small memory footprint and provide excellent update performance if the rule set is to be modified, as the search data structure can be updated incrementally. Accordingly, these algorithms are implemented in classification engines such as the `netfilter/iptables` [26] and `ipfw` [27] firewalls or the Open vSwitch [25].

They stand in contrast to most other advanced classification algorithms which trade memory footprint and especially update latency for significantly better classification performance. Decision tree algorithms, such as HiCuts [7] or HyperSplit [8], transform the rule set into a multi-dimensional search tree during a preprocessing step, which is traversed in order to classify incoming packets. Decomposition-based schemes like bit vector algorithms [5], [6] and crossproducting approaches [10], [11], such as RFC [10], first divide the multi-dimensional classification problem into one-dimensional search problems, which each can be solved efficiently through binary searches or lookups in precomputed tables. The partial solutions of the one-dimensional problems are subsequently merged in order to obtain the desired classification result. Decision tree or decompositional algorithms are typically significantly faster than linear search or Tuple Space Search when it comes to classification performance [25] for most but the smallest rule sets. However, their search data structure preprocessing and update times are comparatively time-consuming, as demonstrated in Section V. This restricts their practical use to scenarios where the underlying rule set does not change too frequently.

With SFL, we propose a methodology to combine the good classification performance of static algorithms with the adaptability of dynamic algorithms in order to obtain the benefits of both worlds.

B. Rule Set Optimization

Rule set optimization aims at improving the classification performance of a packet classification engine by adapting the rule set in a way that the resulting search data structure can be traversed faster at run time. To this end, rule set optimizers modify the rule set *before* it is installed in the classification engine. This is done by either reducing the number of rules [12]–[14] or by exploiting certain matching capabilities of the underlying classification engine in order to reduce the number of tests that have to be performed for incoming packets [15], [16]. Hence, rule set optimization can be considered orthogonal to SFL, as it can be executed on the initial rule set before it is installed in the classification engine.

C. Hardware-based Approaches

In order to push the achievable classification throughput to the limit, some classification architectures utilize special-purpose hardware with massively parallel computing capabilities, such as GPUs (Graphics Processing Unit), TCAMs (Ternary Content-Addressable Memory), or FPGAs (Field-Programmable Gate Array). These approaches are often based on a parallel and/or pipelined implementation of a classification algorithm [18], [19], [21], [22], [28] or match incoming packets against the entire rule set in parallel [20], [23], [24]. Although some of these architectures support dynamic rule set updates [19], [23], [24], [28], they are restricted to a specific hardware setup. In contrast, the proposed SFL approach is more general: it does neither require a specific hardware platform, nor does it assume a specific matching algorithm.

III. Problem Statement

In this section, we first introduce the reader to the packet classification problem. Building thereupon, we subsequently introduce the rule set update problem, which is mainly addressed in this paper.

A. The Packet Classification Problem

The goal of packet classification is to find the most highly prioritized rule in a rule set that matches a selected set of header fields of an incoming network packet. Formally, the d-dimensional packet classification problem can be defined as follows [3]: let H be a *packet header*, which is modelled as a d-tuple

$$H = (h_1, \ldots, h_d) \in \mathcal{U},$$

where $\mathcal{U} = D_1 \times \ldots \times D_d$. The D_j are the domains of the individual header fields, e. g., the interval $[0, 2^{32} - 1]$ for IPv4 addresses or $[0, 2^{16} - 1]$ for TCP/UDP ports. Furthermore, let $\mathcal{R}$ be a *rule set over* $\mathcal{U}$ that consists of N *rules* R_i, i. e.,

$$\mathcal{R} = \langle R_1, \ldots, R_N \rangle.$$

The order of the rules R_i in $\mathcal{R}$ implies priorities in case more than one rule matches a given packet. Without loss of generality, we assume that a rule with a smaller index in $\mathcal{R}$ has a higher priority. Each rule $R_i \in \mathcal{R}$ is a $(d+1)$-tuple consisting of an *action* A^i and d *checks* C^i_j with

$$R_i = \left(C^i_1, \ldots, C^i_d, A^i\right).$$

Each check C^i_j is a function

$$C^i_j : D_j \to \{true, false\}$$

that performs a test on the jth header field of H. These tests are typically simple equality, range, or prefix tests. For example, it could be tested whether the protocol header field equals 17 (for UDP) or if the destination IP address is in the subnet 1.2.3.0/24. A rule R_i *matches* a packet header H iff all tests succeed on the corresponding header fields, i. e., iff

$$\forall j \in \{1, \ldots, d\} : C_j^i(h_j) = true.$$

Two rules R_i and R_k, $i \neq k$, are said to *overlap* iff a header $H \in \mathcal{U}$ exists, such that both R_i and R_k match H. The objective is to find the smallest index i^* for a given packet header H so that rule R_{i^*} matches H. Subsequently, this *matching index* i^* is used to look up the action A^{i^*} that determines the packet's fate (e. g., DROP or ACCEPT). We call an algorithm $\mathcal{A}$ a *classification algorithm*, if it computes the correct matching index i^* for a given rule set R and a given packet header $H \in \mathcal{U}$.

B. The Rule Set Update Problem

In an actual implementation of a packet classification system, the rule set $\mathcal{R}$ is transformed into a suitable search data structure $S_{\mathcal{R},\mathcal{A}}$ for a classification algorithm $\mathcal{A}$. $S_{\mathcal{R},\mathcal{A}}$ is traversed by $\mathcal{A}$ to compute the matching indices. We denote the computation of $S_{\mathcal{R},\mathcal{A}}$ by $\mathit{spawn}_{\mathcal{A}}(\mathcal{R})$ and the computation of the matching index for a packet header H using $S_{\mathcal{R},\mathcal{A}}$ by $\mathit{classify}(S_{\mathcal{R},\mathcal{A}}, H)$. Furthermore, we denote the *insertion* of a rule R^* at index $i \in \{1, \ldots, N\}$ into a rule set $\mathcal{R} = \langle R_1, \ldots, R_N \rangle$ by

$$\mathit{insert}(R^*, i, \mathcal{R}) := \langle R_1, \ldots, R_{i-1}, R^*, R_i, \ldots, R_N \rangle.$$

The *deletion* $\mathit{delete}(i, \mathcal{R})$ is defined likewise as

$$\mathit{delete}(i, \mathcal{R}) := \langle R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_N \rangle.$$

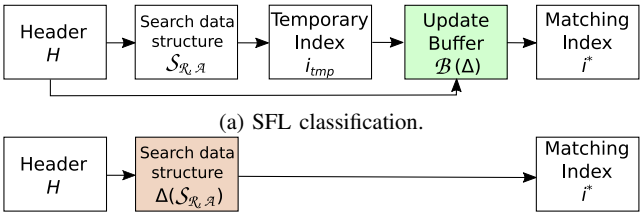
An *update* to a rule set $\mathcal{R}$ is either a deletion or an insertion.

If the rule set $\mathcal{R}$ changes to another rule set $\mathcal{R}'$ due to a sequence of update operations $\Delta = \langle \Delta_1, \ldots, \Delta_m \rangle$, $S_{\mathcal{R},\mathcal{A}}$ must be adjusted in order to correctly classify packets with respect to the rule set change. This can either happen incrementally or through a complete rebuild of the search data structure. Let the expression $\Delta(S_{\mathcal{R},\mathcal{A}})$ denote the incrementally updated search structure, and $S_{\mathcal{R}',\mathcal{A}}$ the search structure which is obtained through $\mathit{spawn}_{\mathcal{A}}(\mathcal{R}')$. It is indispensable that $\Delta(S_{\mathcal{R},\mathcal{A}})$ and $S_{\mathcal{R}',\mathcal{A}}$ are equivalent, i. e.,

$$\forall H \in \mathcal{U} : \mathit{classify}(\Delta(S_{\mathcal{R},\mathcal{A}}), H) = \mathit{classify}(S_{\mathcal{R}',\mathcal{A}}, H). \quad (1)$$

Unfortunately, for many fast classification algorithms, both $\mathit{spawn}_{\mathcal{A}}(\mathcal{R}')$ as well as incremental updates take a considerable amount of time. In fact, for most decompositional or decision tree algorithms there are no known efficient update operations, therefore, the computation of $\Delta(S_{\mathcal{R},\mathcal{A}})$ is actually implemented as $\mathit{spawn}_{\mathcal{A}}(\mathcal{R}')$. Hence, frequent rule set updates can severely diminish the achievable classification throughput.

In this paper, we address this problem and propose an approach that allows, for an arbitrary classification algorithm $\mathcal{A}$, the fast computation of $\Delta(S_{\mathcal{R},\mathcal{A}})$ that is correct with respect to (1) *and* can be searched efficiently.



(a) SFL classification.

(b) Non-SFL classification.

Fig. 1: Control flow in non-SFL and SFL classification.

IV. The SFL Approach

The main idea of the SFL approach is to equip an arbitrary classification algorithm $\mathcal{A}$, as defined in Section III, with the ability to perform quick updates on $\mathcal{A}$'s search data structure $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ for an initial rule set $\mathcal{R}$. Hence, the SFL technique can be regarded as an algorithm-agnostic "upgrade" for a classification algorithm $\mathcal{A}$. Accordingly, we denote an SFL-augmented algorithm $\mathcal{A}$ as $\mathit{SFL}(\mathcal{A})$. SFL($\mathcal{A}$) is a classification algorithm which mainly relies on the search structure $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ for an initial rule set $\mathcal{R}$ to classify network packets, and uses an *update buffer* $\mathcal{B}$ as well as a *master rule set* $\mathcal{R}_M$ to quickly process a sequence of incoming rule set updates Δ. At each point, the master rule set always stores the up-to-date rule set, where all updates $\Delta_i \in \Delta$ have been applied to $\mathcal{R}$. Initially, $\mathcal{R}_M$ is equal to $\mathcal{R}$.

If a packet with header H enters the classification system, it is first classified by the search data structure $\mathcal{S}_{\mathcal{R},\mathcal{A}}$, which computes a temporary index i_{tmp}. However, i_{tmp} is possibly incorrect due to updates to the rule set (for example, rule $R_{i_{\text{tmp}}}$ could have been deleted). Therefore, the update buffer $\mathcal{B}$ is used to correct the index i_{tmp} to the actual matching index i^*. Finally, i^* is used to look up the action A^* in $\mathcal{R}_M$ that belongs to R_{i^*}. In contrast, a non-SFL classification algorithm would simply apply its search structure to H to compute i^*. However, it is forced to directly apply all rule set updates to $\mathcal{S}_{\mathcal{R},\mathcal{A}}$, which could result in long update durations where no fast classification can happen. Figure 1 sketches both the SFL and the classic approach. In the remainder of this section, we explain in detail how *insert*, *delete*, and *classify* operations are performed using the update buffer $\mathcal{B}$, the (non-updated) search data structure $\mathcal{S}_{\mathcal{R},\mathcal{A}}$, and the master rule set $\mathcal{R}_M$.

A. Rule Insertion

The first *update* operation being considered is the insert operation $\mathit{insert}(R^*, i, \mathcal{R}_M)$ which inserts the rule R^* into $\mathcal{R}_M$ at position i. Inserting a rule into $\mathcal{R}_M$ causes an index shift of all rules located behind the inserted rule. If rules located in front of the inserted rule have been inserted or deleted previously, i changes accordingly, so that i equals the rule's actual position relative to the master rule set $\mathcal{R}_M$. Figure 2 shows a set of rules R_1 to R_4 that is incorporated into $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ using the $\mathit{spawn}_{\mathcal{A}}(\mathcal{R})$ function associated with $\mathcal{A}$. The update buffer is empty because no updates occurred so far. In this example, rules match one dimension and specify an action.

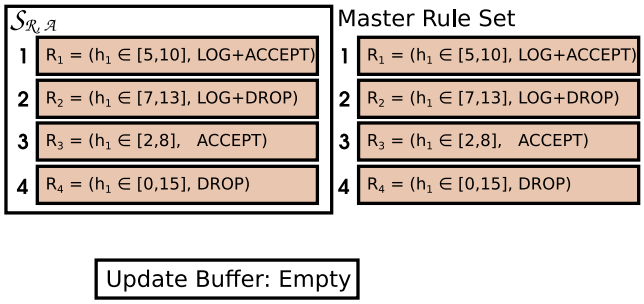


Fig. 2: SFL data structure without updates.

Now consider an insertion of a new rule R^* at index 3 into the master rule set $\mathcal{R}_M$, as shown in Figure 3. In consequence, the matching indices which are computed for all rules located *behind* the inserted rule must be incremented by one in order to be correct. For example, classification of a packet which is matching rule R_3 has to return a matching index of 4. Thus, only rules located behind R^* are actually affected by that insertion. If a rule R_{old} is located above R^*, neither is its priority changed nor can it be overruled by R^*. However, if R_{old} is located behind R^*, R^* and R_{old} may overlap, meaning that R^* could match some packets that originally would have matched R_{old}. For example, in Figure 2, the search structure $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ computes the temporary index $i_{\text{tmp}} = 3$ from R_3 for a header H_1 with value $h_1 = 2$. Since no updates have yet occured in Figure 2, $i^* = i_{\text{tmp}}$. According to $\mathcal{R}_M$ in Figure 3, the newly inserted rule R^* matches H_1, too, yielding $i^* = 3$ because R^* is the third rule in the updated master rule set. H_2 with $h_1 = 4$ matches R_3 in both cases, resulting in $i^* = 3$ before the insertion of R^* and in $i^* = 4$ after R^*'s insertion.

The required index transformations are implemented by the update buffer $\mathcal{B}$. In case of an insertion of rule R^* this buffer checks which rule R_j is the first one behind the inserted rule that exists in the initial rule set $\mathcal{R}$. When R_j is determined, a new *update node* with index j is created and inserted into a linked list. Update nodes store the index j and – in case they represent an insertion – a list of inserted rules. All rules inserted directly in front of a rule present in $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ are stored in the same node in the order of their priorities. If a rule is inserted behind the lowest-prioritized rule $R_{|\mathcal{R}|}$ in $\mathcal{R}$ (i. e., there is no rule index behind the new rule which determines the index of the update node), the update node with index $|\mathcal{R}|+1$ is chosen. The procedure that inserts R^* into the update buffer $\mathcal{B}$ is depicted in Algorithm 1. For a rule R, the $R.old\_index$ variable refers to R's index in the original rule set $\mathcal{R}$.

B. Rule Deletion

The second possible *update* operation supported by SFL is the delete operation $delete(i, \mathcal{R}_M)$, which deletes the rule R_i from $\mathcal{R}_M$ which is located at position i. Rules in the master rule set $\mathcal{R}_M$ either already exist in $\mathcal{R}$ (and are therefore incorporated in $\mathcal{S}_{\mathcal{R},\mathcal{A}}$) or have been inserted by an insert operation as described above. Therefore, a deletion falls into one of two categories: the first scenario is deleting a rule R^*

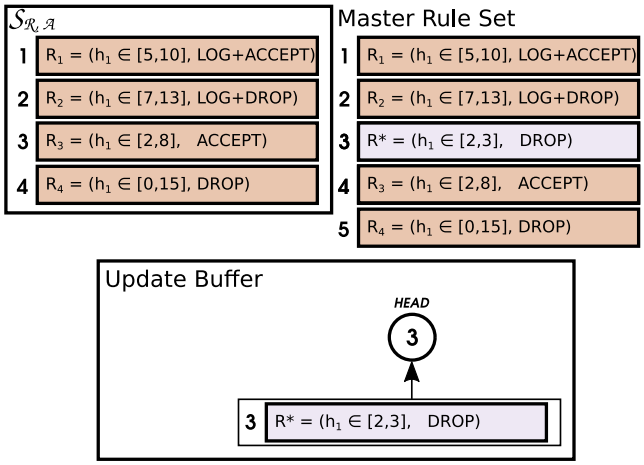


Fig. 3: SFL data structure after one rule insertion.

Algorithm 1 Insert rule R^* into update buffer $\mathcal{B}$ at index i.

```
1: function INSERT(Update Buffer B, Rule R*, Index i,
   Master Rule Set R_M)
2:     j ← GET_NEXT_OLD_RULE(R_M, R*).old_index
3:     node ← SEARCH_NODE(B, j)
4:     if node = invalid then
5:         node ← CREATE_NODE(B, j)
6:     INSERT_RULE_IN_NODE(node, R*)
7:     return
```

that has been inserted by a previous insert operation. In this case, R^* is simply removed from the corresponding hybrid node in the update buffer $\mathcal{B}$. In the other case, a rule from the original rule set $\mathcal{R}$ is to be deleted, so that this information has to be represented in $\mathcal{B}$.

Consider the situation in Figure 3 as a starting point and assume that R_2 is picked for deletion. R_2's index in $\mathcal{R}_M$ is 2. The resulting data structure is displayed in Figure 4, where $\mathcal{R}_M$ no longer contains R_2 and the update buffer contains one more update node. A deletion is encoded by creating a new (or retrieving the existing) update node with index i. In order to distinguish between insertions and deletions, every update node explicitly stores a *delete flag*, which is set in case of a deletion. This flag is denoted "D" in Figure 4.

Assume that a packet header H_3 with $h_1 = 13$ arrives at the SFL($\mathcal{A}$) search structure. $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ calculates a temporary matching index of $i_{\text{tmp}} = 2$. Before the deletion occurred, the update buffer passes this index unchanged as the final matching index i^* because no changes are stored that affect rule R_2. After the delete operation, however, the update buffer has indeed information on matching index $i_{\text{tmp}} = 2$. The active delete flag in update node 2 indicates that the highest-prioritized matching rule in $\mathcal{R}$ has been deleted and is no longer valid. Therefore, every rule located *behind* the deleted rule R_i in the master rule set $\mathcal{R}_M$ might match H_3 and has to be searched accordingly. This results in a search for the first matching rule in the master rule set suffix, i. e., all rules

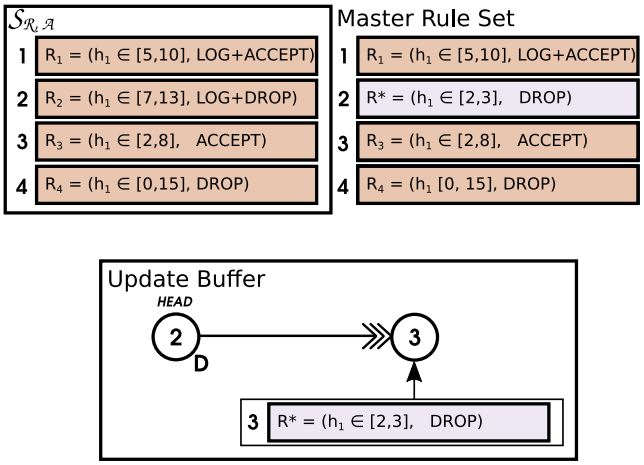


Fig. 4: SFL data structure with one insertion and one deletion.

that were located behind R_2 in $\mathcal{R}_M$ immediately before its deletion. Thus, rules R^*, R_3, and R_4 are tested. R_4 at index 4 in $\mathcal{R}_M$ matches H_3. Hence, the final matching index i^* calculated by SFL($\mathcal{A}$) is 4.

The procedure for deleting rule R_i is listed in Algorithm 2. In case R_i does not exist in the original rule set $\mathcal{R}$ (i. e., its *old_index* equals 0), it is deleted from $\mathcal{R}_M$ and its containing update node is removed from the update buffer. If the node is otherwise empty, the node itself is removed, too. If $R_i \in \mathcal{R}$, the mentioned activation of the delete flag in the corresponding update node takes place. Update nodes can concurrently hold information on both deletions and insertions by having an active delete flag and a non-empty list of inserted rules.

C. Classification

Despite the newly introduced ability to handle dynamic rule set updates, the most commonly executed function of the SFL($\mathcal{A}$) search structure will be the *classification* function *classify*($\mathcal{S}_{\mathcal{R}_M,\text{SFL}(\mathcal{A})}, H$) for a header H. Algorithm 3 shows a detailed overview of this function. Internally, the first step of a call to that procedure generates a temporary matching index i_{tmp} using $\mathcal{S}_{\mathcal{R},\mathcal{A}}$. After that, the first update node in $\mathcal{B}$'s list of nodes is retrieved and checked if it has an index less or equal than i_{tmp}. If true, all rules that have been inserted into that node are checked if one of them matches H. Such rules are more highly prioritized than the rule found at i_{tmp}. Therefore, if a rule R_k of these rules matches, k is retrieved, i^* is set to k, and the procedure finishes. If no matching rule is found, an offset counter is updated where the number of inserted rules in the current node is added to the counter. A traversed node with an active delete flag decreases that counter. This operation is necessary due to the shifted indices in $\mathcal{R}_M$ after updates have been applied. For example, in Figure 3 rule R_4 is shifted to index 5 in $\mathcal{R}_M$ due to the insertion of R^*. After updating that offset counter, the next update node is retrieved.

When a node is found that carries i_{tmp} as its index, that node is checked for an active delete flag. If that check is successful, two deductions can be made: first, there was no insertion of a more highly prioritized matching rule in $\mathcal{R}_M$, otherwise classification would have ended already. Second, the rule i_{tmp} has been deleted and therefore is no longer valid. These two conditions imply that any possibly matching rule must be located behind the original position of rule i_{tmp} in $\mathcal{R}_M$. Thus, a linear search in $\mathcal{R}_M$ starting at the position of the removed rule is executed that determines if there exists any matching rule. The result of that search is returned as the final matching index i^*. However, if i_{tmp} *is* the highest prioritized matching index, the final matching index i^* is set to i_{tmp}, shifted by the calculated index offset.

D. Rebuild Operations

The downside of storing rule set updates in the update buffer is the degradation of classification performance with an increasing number of updates. Each update causes the creation of a new node or at least the insertion of a pointer to a rule from $\mathcal{R}_M$ which may be traversed during the lookup process.

Algorithm 2 Delete rule at index i from the update buffer $\mathcal{B}$.

```
1:  function DELETE(Update Buffer B, Index i, Master Rule Set R_M)
2:      rule ←LOOKUP_IN_MASTER_RULESET(R_M, i)
3:      if rule.old_index= 0 then
4:          n ← DETERMINE_CONTAINING_NODE(B, rule)
5:          DELETE_RULE_FROM_NODE(B, n, rule)
6:      else
7:          node ← SEARCH_NODE(B, rule.old_index)
8:          if node = invalid then
9:              node ← CREATE_NODE(B, rule.old_index)
10:         SET_DELETE_FLAG(B, node)
11:     return
```

Algorithm 3 Perform a *classify* operation with SFL($\mathcal{A}$).

```
1:  function CLASSIFY(Master Rule Set R_M, S_{R_M,SFL(A)} =
        (Search Structure S_{R,A}, Update Buffer B), Header H)
2:      i* ← invalid
3:      offset ← 0
4:      i_tmp ← CLASSIFY_HEADER(S_{R,A}, H)
5:      node ← H.HEAD
6:      while (node.index ≤ i_tmp) or (node ≠ invalid) do
7:          i* ← SEARCH(node.inserted_rules, H)
8:          if i* ≠ invalid then
9:              return i*
10:         if node.delete_flag = 1 ∧ node.index = i_tmp then
11:             return RULESET_SEARCH(R_M, H, i_tmp)
12:         offset ← offset + LEN(node.inserted_rules)
13:         offset ← offset − (node.is_delete_node ? 1 : 0)
14:         node ← node.next
15:     if i* = invalid then
16:         i* ← i_tmp+ offset
17:     return i*
```

After several updates, the performance penalty induced by the growing size of $\mathcal{B}$ requires a rebuild of $\mathcal{S}_{\mathcal{R},\mathcal{A}}$ using $\mathcal{R}_M$ in order to restore matching performance to the level usually reached by $\mathcal{A}$. Depending on $\mathcal{A}$, a rebuild operation takes a considerable amount of time. Therefore, rebuilds should not be triggered too often. On the other hand, executing rebuilds too seldom may result in poor classification performance over time. Thus, we propose to use an *update threshold* δ, which determines how many update operations can be applied to the update buffer before the search data structure of $\mathcal{A}$ is rebuilt. Note that if $\delta = 0$, then SFL($\mathcal{A}$) behaves exactly like $\mathcal{A}$.

V. Evaluation

In this section, we evaluate the proposed SFL approach by comparing it to the existing classification algorithms linear search, Tuple Space Search [9], Bit Vector Search [5], HyperSplit [8], and RFC [10]. We choose this set of algorithms because they cover the most important flavors of classification algorithms, as defined in [4]: *exhaustive search* (linear search), *decomposition approaches* (Bit Vector search, RFC), *decision tree schemes* (HyperSplit), and *tuple space techniques* (Tuple Space Search). Furthermore, this algorithm selection exhibits widely different algorithmic ideas as well as distinctive update and classification performances, which makes them a suitable test base for the proposed algorithm-agnostic SFL approach.

In our evaluation, we first provide a complexity analysis of the relevant operations *update* (i. e., *insert* or *delete*) and *classify*. Second, we conduct a series of experiments in order to evaluate the classification and update performances using our implementations of each of the abovementioned algorithms as well as their SFL-enhanced variants.

A. Complexity Analysis

The SFL($\mathcal{A}$) algorithm can perform δ consecutive rule set updates in $\mathcal{O}(\delta + N)$ steps per update, as both the update buffer $\mathcal{B}$ and the master rule set $\mathcal{R}_M$ are modified. Here, N is the number of rules in $\mathcal{R}_M$, while δ is the update threshold of $\mathcal{B}$. As the $(\delta + 1)$-st update operation requires a rebuild of the search data structure of the algorithm $\mathcal{A}$ as well as the deletion of existing nodes in the update buffer, it takes the same number of steps as $\mathit{spawn}_{\mathcal{A}}(\mathcal{R}_M)$ plus $\mathcal{O}(\delta)$ steps. SFL($\mathcal{A}$)'s classification performance depends on the time needed by $\mathit{classify}(S_{\mathcal{R},\mathcal{A}}, H)$, since $\mathcal{A}$'s search structure is queried first. Subsequently, the update buffer $\mathcal{B}$ is searched, which takes at most δ steps if no update node with active delete flag is hit, and otherwise at most N steps due to the subsequent linear search in the master rule set $\mathcal{R}_M$. Note, however, that this linear search starts at the index of the deleted rule, which will reduce the cost of the linear search, if any rule but the first rule is deleted. Hence, classifying a packet header H using SFL($\mathcal{A}$) takes $\mathit{classify}(S_{\mathcal{R},\mathcal{A}}, H)$ plus $\mathcal{O}(N)$ steps in the worst case. Table I summarizes the runtime complexities for classification and update operations.

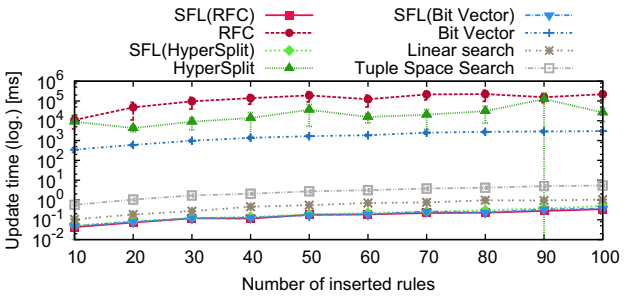
B. Isolated Operations Benchmark Scenario

In our first set of measurements, we investigated the rule insertion, rule deletion, and classification performances of the SFL approach in an isolated scenario. All experiments were conducted on a machine with an Intel Xeon E5-1660v3 CPU with eight physical cores and 128 GB of main memory, running Ubuntu 14.04 LTS. The classification algorithms were implemented in C and compiled using `gcc 4.8.4`. For rule set insertions, we used the *ClassBench* benchmark tool [29] in order to generate ten different five-dimensional rule sets of $2{,}000 + k$ rules, where k represents the number of inserted rules ($k \in \{10, 20, \ldots, 100\}$). The used dimensions were IPv4 source and destination addresses, the transport layer protocol, as well as source and destination ports. For each rule set of size $2{,}000 + k$, we then removed k rules at randomly chosen positions. That is, the initial rule sets and search data structures in our experiment stored $2{,}000$ rules. Subsequently, we measured for each k, how long it takes to execute k rule insertions of the previously removed rules into the search data structure. We also conducted this experiment for k rule deletions with ten different rule sets with $2{,}000$ rules for each k. The results for insertion and deletion times are shown in Figures 5a and 5b. It can be seen that the classic variants of RFC, HyperSplit, and Bit Vector search require large amounts of time to process k insertions and deletions. In contrast, the linear search and Tuple Space Search approaches can process updates several orders of magnitudes faster, as they do not have to rebuild their search structures, but instead perform incremental updates. Also, we observe that the SFL($\mathcal{A}$) algorithms ($\mathcal{A} \in \{$RFC, HyperSplit, Bit Vector$\}$) require the least amount of time, as the updates can be quickly inserted into the update buffer. Note that we did not perform a rebuild operation for the SFL algorithm versions in these experiments.

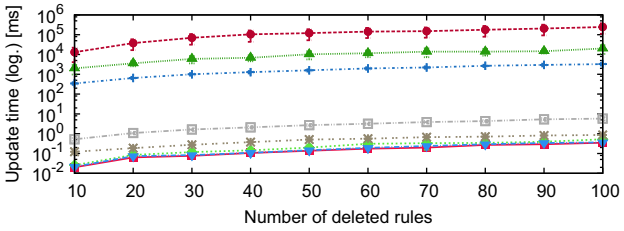
Next, we measured the classification performance of the algorithms after k updates by classifying ClassBench-generated traces of $100{,}000$ packet headers, which are uniformly distributed over the installed rule sets. The results are shown in Figures 5c and 5d. It can be seen that non-SFL algorithms with long update times clearly beat linear search and Tuple Space Search in terms of classification performance, with RFC

TABLE I: Runtime complexities of classification algorithms

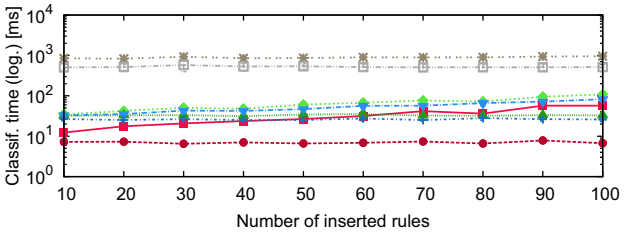
$N = \lvert\mathcal{R}_M\rvert$	d = #dimensions	W = CPU word width
Algorithm	Operation	Runtime Complexity
SFL($\mathcal{A}$)	*spawn*	$\mathit{spawn}_{\mathcal{A}}$
	update ($\leq \delta$ ops)	$\mathcal{O}(\delta + N)$
	update ($(\delta+1)$-st op)	$\mathit{spawn}_{\mathcal{A}} + \mathcal{O}(\delta)$
	classify	$\mathit{classify}_{\mathcal{A}} + \mathcal{O}(N)$
Tuple Space Search	*spawn* / *update*	$\mathcal{O}(N)$
	classify	$\mathcal{O}(N)$
Linear search	*spawn* / *update*	$\mathcal{O}(N)$
	classify	$\mathcal{O}(N)$
RFC	*spawn* / *update*	$\mathcal{O}(N^d)$
	classify	$\mathcal{O}(d)$
HyperSplit	*spawn* / *update*	$\mathcal{O}(N^d)$
	classify	$\mathcal{O}(d \cdot log\ (2 \cdot N))$
Bit Vector	*spawn* / *update*	$\mathcal{O}(d \cdot N^2)$
	classify	$\mathcal{O}(d \cdot log\ N + \frac{d \cdot N}{W})$



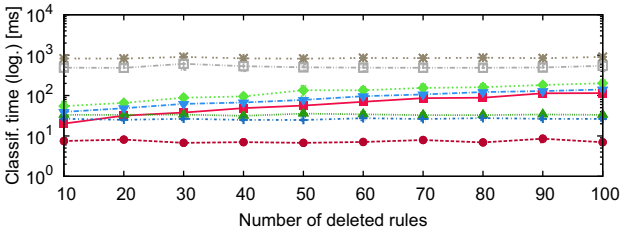
(a) Mean update durations for rule insertions, 90% conf. ivls.



(b) Mean update durations for rule deletions, 90% conf. ivls.



(c) Mean classification times after rule insertions, 90% conf. ivls.



(d) Mean classification times after rule deletions, 90% conf. ivls.

Fig. 5: Benchmark results for deletions and insertions, 10 rule sets for each data point, starting with 2,000 rules.

delivering the best performance. Furthermore, the plots exhibit that the SFL variants of RFC, HyperSplit, and Bit Vector are also considerably faster than linear search and Tuple Space Search. However, we observe that with an increasing number of updates the performance of the SFL algorithms degrades. This was to be expected, as the update buffer, which is searched linearly for each packet header, becomes larger. Also, this confirms that rebuilds of the efficient search structure are needed after a certain number of updates.

C. Timing Behaviour Benchmark Scenario

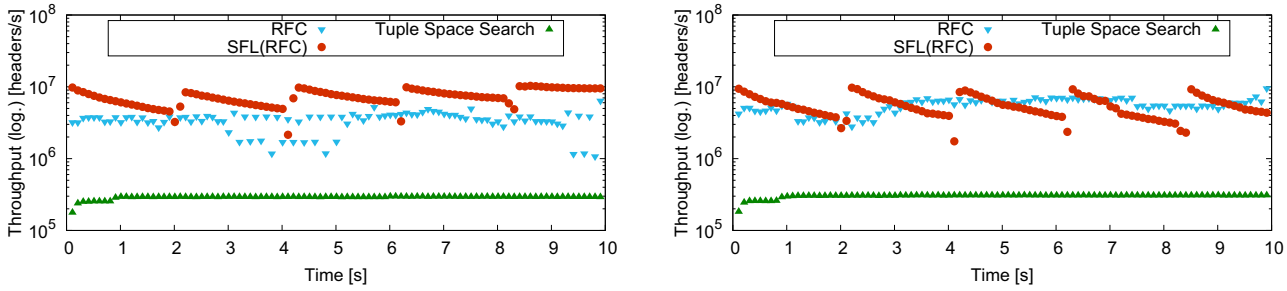
In the second set of experiments, we studied the behaviour of the SFL approach and the other algorithms in a system simulation over a time period of 10 seconds. This experiment was conducted by a simulator which runs four threads: one thread drives the classification algorithm, which is constantly fed with packet headers by a second thread as fast as possible. The third thread applies either 10, 100, 300, or 600 rule insertions or deletions to the installed rule set in fixed intervals. Finally, the last thread counts the number of processed packet headers every 100 milliseconds. At the start of the experiments, we installed a ClassBench-generated rule set with 2,000 rules. As in the previous experiments, the used traces were also ClassBench-generated and uniformly distributed over the rule sets. For each number of updates and operation type combination, we executed ten measurements with different rule sets and traces. The δ parameter was set to 20.

Figures 6a and 6b show an excerpt of this evaluation series, namely the classification performance trend of classic RFC, SFL(RFC), and Tuple Space Search over the 10-second time span for 100 insertions and deletions, respectively. We observe that Tuple Space Search is able to provide near-constant throughput of processed headers, despite the update operations that are issued every 100 ms. This can be explained by the algorithm's support for quick incremental updates. In contrast, the non-SFL RFC algorithm exhibits a heterogeneous performance: due to the fact that it has to rebuild its data structure with every update operation, it can seldom demonstrate its excellent classification performance; linear list search is used as a fallback search structure while the rebuild is active. The SFL(RFC) algorithm, in contrast, can deliver a high throughput most of the time, as only every 21st update leads to a complete rebuild, while the first 20 updates are inserted into the update buffer. Furthermore, the plots exhibit that each buffer update leads to a small performance penalty, as the linear component in the SFL(RFC) structure grows. However, its overall performance is clearly superior to both Tuple Space Search and classic RFC.

The complete results for all algorithms, update frequencies, and operation types are shown in Table II. The table shows the mean number of processed headers in millions as well as the mean number of updates that could be processed over the 10-second measurement durations. All mean values are shown with their respective 90% confidence intervals. It can be seen that in most cases, the SFL($\mathcal{A}$) algorithms can provide a significantly better throughput than both their classic counterparts as well as the incrementally updateable algorithms linear search and Tuple Space Search. When it comes to update performance, we notice that the SFL variants are always equal or superior to their classic counterparts. However, we also point out that the SFL(HyperSplit) and SFL(RFC) algorithms are inferior to linear search and Tuple Space Search in this respect, because every 21st update is still expensive. In contrast, SFL(Bit Vector) provides the same update performance as the updateable approaches, with more than a tenfold increase in throughput.

D. Influence of the δ parameter

Finally, we examined how the choice of δ influences the achievable classification throughput as well as the number of feasible updates during a certain time period. To this end,



(a) 100 rule insertions, issued periodically every 100 ms.

(b) 100 rule deletions, issued periodically every 100 ms.

Fig. 6: Throughput results over 10 s, starting with 2,000 rules. Data points show the classification throughput [headers/s].

TABLE II: Measurement results for 10 second evaluation runs with continuously issued updates, 2,000 rules

		Insert operations				Delete operations			
		10 ops	100 ops	300 ops	600 ops	10 ops	100 ops	300 ops	600 ops
Linear search	**HP**	1.30 ± 0.01	1.20 ± 0.01	1.00 ± 0.01	0.80 ± 0.01	1.30 ± 0.01	1.30 ± 0.01	1.20 ± 0.01	1.10 ± 0.01
	UP	10.00 ± 0.00	100.00 ± 0.00	299.70 ± 0.24	599.70 ± 0.24	10.00 ± 0.00	100.00 ± 0.00	299.90 ± 0.16	599.70 ± 0.24
Tuple Space Search	**HP**	2.70 ± 0.71	2.40 ± 0.62	2.40 ± 0.64	2.30 ± 0.62	2.70 ± 0.70	2.60 ± 0.67	2.60 ± 0.66	2.70 ± 0.72
	UP	10.00 ± 0.00	100.00 ± 0.00	299.70 ± 0.24	557.20 ± 49.16	10.00 ± 0.00	100.00 ± 0.00	300.00 ± 0.00	567.40 ± 45.37
HyperSplit	**HP**	27.70 ± 3.71	4.50 ± 1.71	2.00 ± 0.25	1.90 ± 0.31	26.60 ± 4.64	3.90 ± 1.81	2.20 ± 0.42	2.30 ± 0.40
	UP	9.90 ± 0.16	75.60 ± 12.49	72.20 ± 19.02	81.10 ± 18.17	9.40 ± 0.93	72.90 ± 15.62	85.30 ± 24.91	85.00 ± 22.99
SFL(HyperSplit)	**HP**	35.00 ± 0.61	28.70 ± 1.28	19.00 ± 2.83	11.20 ± 1.78	32.10 ± 0.50	23.00 ± 2.16	18.50 ± 1.22	12.70 ± 1.63
	UP	10.00 ± 0.00	100.00 ± 0.00	273.90 ± 40.09	560.10 ± 31.38	10.00 ± 0.00	99.90 ± 0.16	300.00 ± 0.00	586.00 ± 21.44
RFC	**HP**	50.60 ± 22.93	16.00 ± 9.03	5.50 ± 2.92	4.50 ± 2.15	47.30 ± 21.82	13.00 ± 10.03	5.70 ± 2.89	5.30 ± 2.63
	UP	8.30 ± 1.39	44.10 ± 23.72	75.80 ± 52.84	64.10 ± 41.28	8.20 ± 1.48	45.20 ± 22.36	77.80 ± 51.68	74.20 ± 48.93
SFL(RFC)	**HP**	81.60 ± 2.52	43.60 ± 15.22	30.50 ± 15.11	21.80 ± 11.21	67.80 ± 2.56	29.80 ± 9.84	22.10 ± 10.13	17.50 ± 8.44
	UP	10.00 ± 0.00	75.70 ± 12.78	159.80 ± 60.53	288.20 ± 132.77	10.00 ± 0.00	72.90 ± 15.77	172.40 ± 57.63	288.60 ± 132.77
Bit vector	**HP**	34.60 ± 1.31	17.80 ± 3.16	8.00 ± 3.08	5.40 ± 0.88	34.40 ± 1.48	17.80 ± 2.82	9.00 ± 3.29	7.10 ± 1.51
	UP	10.00 ± 0.00	100.00 ± 0.00	244.20 ± 16.07	264.30 ± 45.13	10.00 ± 0.00	100.00 ± 0.00	270.90 ± 8.68	324.90 ± 68.94
SFL(Bit vector)	**HP**	32.40 ± 0.87	31.00 ± 1.04	27.90 ± 1.34	23.00 ± 1.61	30.70 ± 1.06	25.60 ± 0.96	24.00 ± 1.24	22.20 ± 1.15
	UP	10.00 ± 0.00	100.00 ± 0.00	299.70 ± 0.24	599.70 ± 0.24	10.00 ± 0.00	99.90 ± 0.16	299.90 ± 0.16	599.70 ± 0.24

HP: Mean number of processed headers in millions with 90% confidence intervals **UP**: Mean number of processed updates with 90% confidence intervals

we used the same experimental setup as in Section V-C, but varied the δ parameter from 0 to 100 in steps of 5. Figure 7 shows the total number of classified headers after 10 and 25 seconds, with 100 issued updates (either insertions or deletions) evenly distributed over the measurement time span. The figures reveal that for $\delta \leq 10$, the SFL variants of Bit vector search, HyperSplit, and RFC always perform better than their non-enhanced counterparts. However, if δ is chosen too large, the throughput finally starts to decrease and can even become worse than the original algorithm due to the search overhead introduced by the update buffer. This characteristic is particularly well visible for the Bit vector algorithm in Figures 7a and 7d and can be explained by its relatively short update times, in comparison to HyperSplit and RFC.

The number of updates that could be carried out during the measurement periods is shown in Figure 8. Figures 8a and 8d indicate that both variants of the Bit vector search can process all 100 issued updates. In contrast, SFL(HyperSplit) and SFL(RFC) require δ parameters of 10 and 50, respectively, to handle all issued updates, as shown in Figures 8b, 8c, 8e, and 8f. As expected, the figures also show that the SFL variants of HyperSplit and RFC can always process more updates than their non-enhanced counterparts for $\delta > 0$.

VI. Conclusion

We proposed and evaluated the SFL approach, a technique to increase the update responsiveness of fast packet classification algorithms. The idea behind SFL is to provide an update buffer that lazily propagates rule set changes into the core search data structure of the used classification algorithm. Due to its algorithm-agnostic design, SFL can be used with any classification algorithm that computes the index of the most highly prioritized matching rule. Furthermore, SFL is tunable: through the δ parameter, it can be configured to optimize the search data structure more or less aggressively. Our evaluation results show that the SFL variants of the fast Bit Vector, HyperSplit, and RFC algorithms perform significantly better than their non-SFL counterparts in both update and classification performance in highly dynamic environments. In particular, the SFL(Bit Vector) algorithm yields the same update responsiveness as linear search and Tuple Space Search at a tenfold classification performance increase.

Acknowledgement

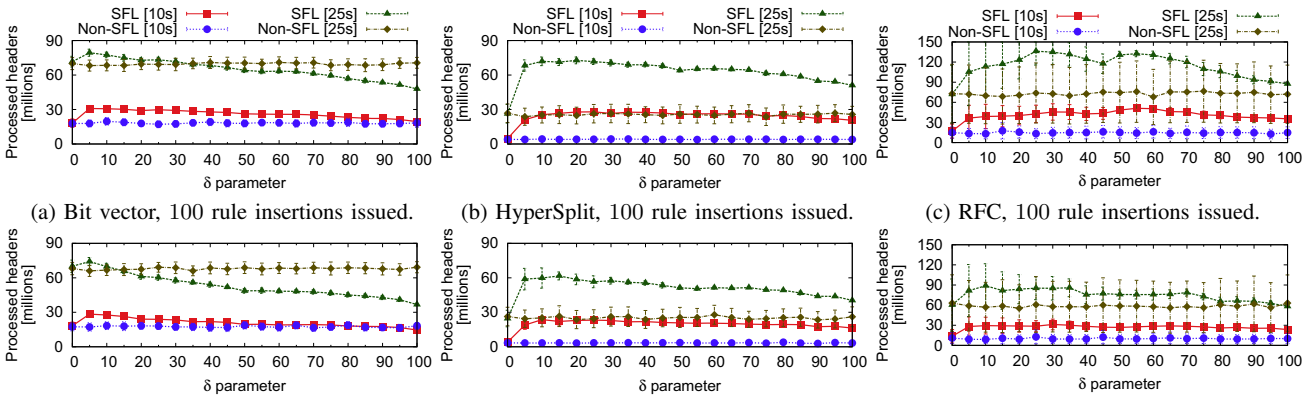
We would like to express our gratitude to the BMWi (German Federal Ministry of Economics and Energy), who funded this research in the context of the HARDFIRE project.

References

[1] F. Baboescu, S. Singh, and G. Varghese, "Packet classification for core routers: Is there an alternative to CAMs?" in *INFOCOM '03*, Mar. 2003, pp. 53–63.

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

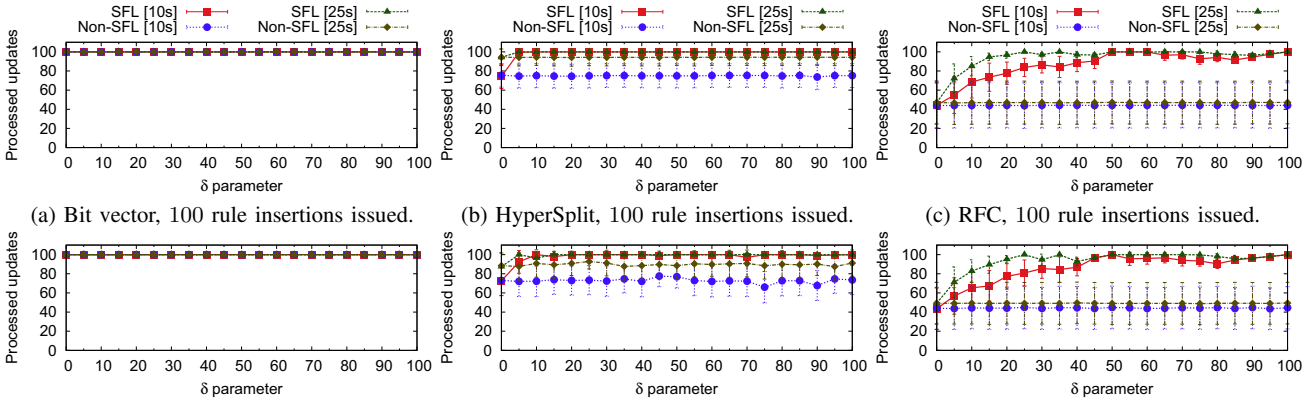
[3] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network: The Magazine of Global Internetworking*, vol. 15, no. 2, pp. 24–32, Mar. 2001.



(a) Bit vector, 100 rule insertions issued. (b) HyperSplit, 100 rule insertions issued. (c) RFC, 100 rule insertions issued.

(d) Bit vector, 100 rule deletions issued. (e) HyperSplit, 100 rule deletions issued. (f) RFC, 100 rule deletions issued.

Fig. 7: Mean number of processed headers after 10 and 25 seconds with varying δ, 100 updates issued, 90% conf. ivls.



(a) Bit vector, 100 rule insertions issued. (b) HyperSplit, 100 rule insertions issued. (c) RFC, 100 rule insertions issued.

(d) Bit vector, 100 rule deletions issued. (e) HyperSplit, 100 rule deletions issued. (f) RFC, 100 rule deletions issued.

Fig. 8: Mean number of processed updates after 10 and 25 seconds with varying δ, 100 updates issued, 90% conf. ivls.

[4] D. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238–275, Sept. 2005.

[5] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," in *SIGCOMM '98*, Aug. 1998, pp. 203–214.

[6] F. Baboescu and G. Varghese, "Scalable packet classification," in *SIGCOMM '01*, Aug. 2001, pp. 199–210.

[7] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *HOTI '99*, Aug. 1999, pp. 34–41.

[8] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *INFOCOM '09*, Apr. 2009, pp. 648–656.

[9] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *SIGCOMM '99*, Aug. 1999, pp. 135–146.

[10] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *SIGCOMM '99*, Aug. 1999, pp. 147–160.

[11] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," in *SIGCOMM '98*, Aug. 1998, pp. 191–202.

[12] A. Liu and M. G. Gouda, "Complete redundancy detection in firewalls," in *DBSec '05*, Aug. 2005, pp. 193–206.

[13] A. Liu, E. Torng, and C. Meiners, "Firewall compressor: An algorithm for minimizing firewall policies," in *INFOCOM '08*, Apr. 2008, pp. 691–699.

[14] J. Daly, A. Liu, and E. Torng, "A difference resolution approach to compressing access control lists," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 610–623, Feb 2016.

[15] S. Hager, S. Selent, and B. Scheuermann, "Trees in the list: Accelerating list-based packet classification through controlled rule set expansion," in *CoNEXT '14*, Dec. 2014, pp. 101–107.

[16] S. Hager, P. John, A. Fiessler, and B. Scheuermann, "Minflate: Combining rule set minimization with jump-based expansion for fast packet classification," in *ANCS '16*, Mar. 2016, pp. 115–116.

[17] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *ASIACCS '06*, Mar. 2006, pp. 332–342.

[18] T. Ganegedara and V. Prasanna, "StrideBV: Single chip 400g+ packet classification," in *ReConFig '12*, Dec. 2012, pp. 1–6.

[19] B. Vamanan and T. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *CoNEXT '11*, Dec. 2011, pp. 1–12.

[20] S. Hager, F. Winkler, B. Scheuermann, and K. Reinhardt, "MPFC: Massively parallel firewall circuits," in *LCN '14*, Sept. 2014, pp. 305–313.

[21] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multi-layer packet classification with graphics processing units," in *CoNEXT '14*, Dec. 2014, pp. 109–120.

[22] S. Zhou, S. Singapura, and V. Prasanna, "High-performance packet classification on GPU," in *HPEC '14*, Sept. 2014, pp. 1–6.

[23] A. Fiessler, S. Hager, B. Scheuermann, and A. Moore, "HyPaFilter: A versatile hybrid FPGA packet filter," in *ANCS '16*, Mar. 2016, pp. 25–36.

[24] D. Shah and P. Gupta, "Fast incremental updates on ternary-CAMs for routing lookups and packet classification," in *HOTI '00*, Aug. 2000.

[25] B. Pfaff *et al.*, "The design and implementation of Open vSwitch," in *NSDI '15*, May 2015, pp. 117–130.

[26] "The netfilter.org project," www.netfilter.org, last access: August 9, 2016.

[27] "IPFW firewall," www.freebsd.org/cgi/man.cgi?ipfw, last access: August 9, 2016.

[28] Y. Qu and V. Prasanna, "High-performance and dynamically updatable packet classification engine on FPGA," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 1, pp. 197–209, Jan 2016.

[29] D. Taylor and J. Turner, "ClassBench: a packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, June 2007.