TupleMerge: Fast Software Packet Processing for Online Packet Classification

James Daly*, Valerio Bruschi†‡, Leonardo Linguaglossa‡, Salvatore Pontarelli†§ Dario Rossi‡, Jerome Tollet¶, Eric Torng*, Andrew Yourtchenko¶ *Michigan State University, †University of Rome “Tor Vergata”, ‡Telecom ParisTech, §CNIT (National Inter-University Consortium for Telecommunications), ¶Cisco Systems

Abstract—**Packet classification is an important part of many networking devices, such as routers and firewalls. Software-Defined Networking (SDN) heavily relies on online packet classification which must efficiently process two different streams: incoming packets to classify and rules to update. This rules out many offline packet classification algorithms that do not support fast updates. We propose a novel online classification algorithm, TupleMerge (TM), derived from Tuple Space Search (TSS), the packet classifier used by Open vSwitch (OVS). TM improves upon TSS by combining hash tables which contain rules with similar characteristics. This greatly reduces classification time preserving similar performance in updates.**

We validate the effectiveness of TM using both simulation and deployment in an full-fledged software router, specifically within Vector Packet Processor (VPP). In our simulation results, which focus solely on the efficiency of the classification algorithm, we demonstrate that TM outperforms all other state of the art methods including TSS, PartitionSort (PS), and SAX-PAC. For example, TM is 34% faster at classifying packets and 30% faster at updating rules than PS. We then evaluate experimentally TM deployed within the VPP framework comparing TM against linear search and TSS, and also against TSS within the OVS framework. This validation of deployed implementations is important as SDN frameworks have several optimizations such as caches that may minimize the influence of a classification algorithm. Our experimental results clearly validate the effectiveness of TM. VPP TM classifies packets nearly two orders of magnitude faster than VPP TSS and at least one order of magnitude faster than OVS TSS.

I. Introduction

A. Motivation

Packet classification is a vital part of internet routing, firewalls, and other services. These services classify each packet in an incoming stream with a label, such as “forward on physical port 1” or “discard”, determined by the first rule in a multi-dimensional rulelist that matches the packet.

Packet classification has traditionally been viewed as an *offline problem* where we are given a static rulelist, and the aim is to build a static data structure that will classify a stream of packets as quickly as possible. With the advent of new networking paradigms such as software-defined networking (SDN) [1], exemplified by OpenFlow [2], and network function virtualization (NFV) [3], packet classification is now an *online problem* where the system receives a mixed stream of packets to classify and rules to update (insert or delete). Rather than creating a static data structure, we must create a *dynamic data structure* that supports both fast packet classification and fast rule update. We present a formal definition of the online packet classification problem in Section II.

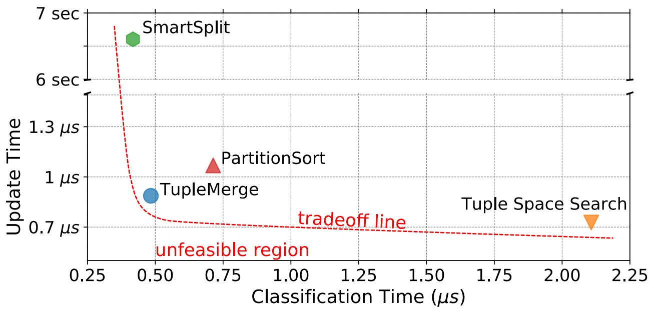


Fig. 1: State of the art of packet classification: tradeoff in the classification vs update times. Results are obtained via simulation (cfr. Sec. IV) and are averaged over ACL seeds with [1k,32k] rules.

All packet classifiers, offline and online, must have fast classification times because internet services have real-time constraints. They constantly receive new packets that go into a queue until they are processed. If they are not processed in a timely manner and the queue fills up, further packets are dropped which causes network congestion. Even before this point, delays in one location can cause congestion and problems elsewhere.

Online packet classifiers required by SDN introduce a new constraint for packet classifiers: fast updates. In SDN, a controller program dynamically configures the behavior of switches by inserting and deleting rules. These changes occur much more frequently than in traditional networks, so packet classification algorithms for SDN must support both fast updates and fast classification. In particular, OpenFlow based networks has update requirements that are not met by some current OpenFlow hardware switches, as reported in [4]. Achieving both fast classification and fast rule update is a research topic of growing interest, since delay in policy update can lead to thousands of mis-routed packets in high performance networks [5].

B. Summary and Limitations of Prior Art

One of the primary challenges in packet classification, among the typical metrics [6] used to evaluate packet classification performance, is simultaneously achieving *fast classification times* and *fast updates*. Because fast updates were not as important as fast classification for traditional offline packet classification, most methods, such as HyperCuts [7] and SmartSplit [8], essentially ignored updates to focus on minimizing classification time. Others such as Tuple Space Search (TSS) [9] sacrifice classification time for fast updates. The result is that high-speed classification methods are not competitive for updates while fast update methods are not competitive for classification time. We visually represent this tradeoff between classification time and update time in Fig. 1.

With the advent of SDN and the increased importance of fast updates, the best choice has been to use the fast update methods such as TSS; the fast classification methods are unusable because they provide no support for fast updates. For example, Open vSwitch [10] which uses TSS can handle up to 42,000 rule updates per second [4].

TSS, like all existing fast update methods, is a partitioning method that partitions the initial rule list into parts that can be efficiently processed. TSS groups rules based on the number of bits used for each field (a tuple) creating a hash table for all rules that share the same tuple. This leads to $O(d)$ searches, insertions, and deletions from each hash table where d is the number of dimensions. The main drawback is that the number of tuples can be large and each tuple may need to be searched leading to slow classification.

The current state of the art method that best balances fast classification and fast updates is PartitionSort (PS) [11]. Rather than partitioning rules based on tuples, PS partitions rules into sortable rulesets which can be stored using binary search trees that support $O(d + \log n)$ search, insertion, and deletion of rules where d is the number of dimensions and n is the number of rules. PS requires fewer partitions because sortable rulesets is less constraining than tuples; the drawback is that each sortable ruleset requires a little more time to process. In our simulations, we found that PS was 5.31 times faster than TSS at classifying packets on our largest classifiers, but 77% slower to update. Our objective is to improve upon PS in both classification time and update time.

C. Proposed Approach and Deployment Validation

Our approach, which we call TupleMerge (TM), improves upon TSS by relaxing the restrictions on which rules may be placed in the same table. We call our method TupleMerge because it merges tuples together to minimize the number of resulting tables. More precisely, TM defines compatibility between rules and tuples so that rules with similar but not identical tuples can be placed in the same table as long as each rule is compatible with that table's tuple. This significantly reduces the number of tables required and thus the overall classification times. In comparison to PS, TM may require more partitions, but each partition is represented using a hash table rather than a tree, so TM can still achieve faster packet classification and updates.

To fully validate the effectiveness of any packet processing algorithm, we must not only develop an efficient algorithm but also test the algorithm implementation inside a specific network appliance (such as a software router) as there are many implementation issues beyond algorithmic efficiency that impact the actual throughput. For example, SDN frameworks such as Open vSwitch (OVS) and Vector Packet Processor (VPP) can leverage optimizations such as caches that may minimize the influence of the packet classification algorithm on network throughput. We not only propose TM but also validate its true effectiveness by implementing it within the VPP framework and conducting rigorous experimental evaluations in actual network deployments.

D. Technical Challenges and Proposed Solutions

We face three key technical challenges in designing and validating our TM classifier. The first is *designing a flexible hashing scheme*. Since packets represent points, but rules represent volumes, it is difficult to use hashing schemes to match the two together. We address this by stretching rules so that they are all the same size by ignoring some bits in each rule. Hashing then lets us determine if the volume that the packet falls into contains a rule.

The second challenge is *designing a strategy for online partitioning*. This is challenging because we must achieve both fast updates and preserve fast classification speeds after many updates. We define a scheme for inserting new rules as well as a method for recognizing that an existing combination of rules does not work well together.

The third is *validating a deployed implementation of TM*. Validating a deployed implementation of any packet processing algorithm is extremely difficult and rarely done. Correctly integrating a new packet processing algorithm into a complex framework such as OVS or VPP is challenging enough; tuning the integration so that the algorithm works well with all the other components of the framework is extremely daunting. We propose to do exactly this; specifically, we integrate TM into the VPP framework and then conduct extensive experiments stimulating the framework with both synthetic and real traffic traces to validate the true effectiveness of TM deployed in the VPP framework.

E. Summary of Simulation and Experimental Results

We conduct extensive simulation-based comparisons between TupleMerge and other methods, in particular PartitionSort and Tuple Space Search. We find that TM outperforms PS, the current state-of-the-art, in both objectives. Specifically, it is 34% faster at classifying packets and 30% faster at updating rules than PS. Compared with TSS, TM is 7.4 times faster at classifying packets and is only 39% slower at updating rules. These simulation results suggest that TM may be the best choice for online packet classification.

We then validate the effectiveness of TM in deployment by conducting extensive experimental comparisons between different implementations in actual deployed network frameworks. Specifically, we contrast our VPP TM implementation against VPP TSS and OVS TSS implementations. Our results show that our deployed VPP TM is up to two orders of magnitude faster than both VPP TSS and OVS TSS under real traffic. Combining our simulation results with our deployment validation, we conclude that TM is extremely effective for online packet classification.

II. Background

A. Problem statement

We now formally define the *Online Packet Classification* problem. We first need the following definitions. A packet field f is a set of nonnegative integers, usually $[0, 2^w - 1]$ for some integer w. A rule r over d fields is a mapping from a subset of those fields to some decision, represented $r = (s_1 \subseteq f_1 \times s_2 \subseteq f_2 \times \cdots \times s_d \subseteq f_d \to \mathit{decision})$. A packet $p = (p_1, p_2, \cdots p_d)$ matches r if $p_1 \in s_1 \wedge p_2 \in s_2 \wedge \cdots \wedge p_d \in s_d$. At any point in time, there is an active rulelist $[r_1, r_2, \cdots, r_n]$ sorted by order of priority. An example active rulelist is given in Table I. When we classify a packet p, we must return the first rule that matches it. Without loss of generality, we assume we start with an empty active rulelist [ ].

The input to the online packet classification problem is a sequence of requests where requests have two types: (i) rule updates (rule insertions or deletions) and (ii) packets to classify. The first request should be a rule insertion. When processing a request, either a rule update or packet classification, the online classifier must perform the request as quickly as possible, and we usually assume it has no knowledge of future requests, though it is possible that some requests might be queued up. The online classifier maintains a dynamic data structure representing the current active rules. When performing a rule update, it must balance the time required for the update against the cost of future packet classification searches. The goal is to come up with an algorithm and dynamic data structure that supports both fast packet classification and fast rule updates.

B. Packet classification algorithms

Although packet classification is a well-studied topic [12], [6], most prior work is on offline packet classification. This is not suitable for SDN, NFV, or for new applications such as Reflexive IP ACLs [13], since online packet classification where fast updates to the rule list is a strong requirement. We briefly review prior work dividing algorithms among the different classes of algorithms [12], focusing especially into *decision trees* and *partition-based*. We go into more detail on the partition-based methods as they can be used for online packet classification.

1) Decision Trees: Decision tree methods such as ByteCuts [14], HyperCuts [7], HiCuts [15], and HyperSplit [16] have emerged as the state-of-the-art for traditional offline packet classification because they achieve fast classification. These methods work by partitioning the search space and then distributing rules into the resulting partitions. The search space partitions, along with the corresponding rules, are represented by trie nodes. The main difficulty with these methods is rule replication that results because rule boundaries do not perfectly align with the partitions, so rules must be copied into multiple partitions (trie nodes). The resulting rule replication leads to both memory blowup and complex rule update. In particular, the best known way to update an arbitrary decision tree is to perform tree reconstruction which takes far too long making most decision tree methods unusable for online classification and thus out of the scope of this paper.

TABLE I: Example 2D rulelist

Rule	Source	Dest	Tuple
r_1	000	111	(3, 3)
r_2	010	100	(3, 3)
r_3	011	100	(3, 3)
r_4	010	11*	(3, 2)
r_5	10*	101	(2, 3)
r_6	110	***	(3, 0)
r_7	***	011	(0, 3)

This is even true for more recent decision tree methods such as EffiCuts [17] and SmartSplit [8] that mitigate some rule replication by performing rule partitioning (distinct from search space partitioning) in a manner similar to partition-based methods. Unfortunately, the rule partitioning they do is not enough to eliminate rule replication or provide a better alternative to tree reconstruction for performing rule update. One recent decision tree method, CutSplit [18], provides a rule update mechanism that avoids tree reconstruction in most cases. CutSplit achieves average update times of $50\mu s$ which is still one to two orders of magnitude larger than partition-based methods.

2) Partition-based: Since packet classification has become an "online" problem, different structures are needed that are able to more easily accommodate other operational points in the classification vs update time tradeoff. Partition-based approaches achieve this by splitting a ruleset into multiple smaller ones (based on rule characteristics that can easily be computed), so that fast exact-match techniques, like those based on hash tables, can be used to access the data structure for both lookup of existing rules and insertion of new rules. We compactly present the key characteristics of the best partition-based methods in Tab.II including TM.

Tuple Space Search (TSS) is one of the oldest offline packet classification methods that was left behind as faster methods were developed [9]; TSS has reemerged for online packet classification because it supports fast updates. In particular, Open vSwitch [21] uses TSS as its packet classifier because TSS offers constant-time updates and linear memory.

In TSS, each partition is defined by a mask tuple $\vec{m}$ which represents the actual bits used in each of the prefix or ternary fields of all rules in that partition. Since all the rules in a partition have the same $\vec{m}$, we can store these rules using a hash table where the masked bits $\vec{m}$ for each rule form the hash key. The main drawback of TSS is that the number of partitions/tables is large. This results in relatively slow packet classification as many tables must be searched. TM improves upon this by putting together rules from multiple TSS tables into a single hash table resulting in many fewer tables.

Srinivasan *et al.* suggest using a (usually 1-dimensional) preclassifier to identify which tables contain rules that might match the packet [9]. This allows fewer tables to be searched, potentially resulting in significant time savings. The tradeoff is that the preclassifier requires extra time and memory; for some rulelists, it may take more time to query the preclassifier than is saved by reducing the number of tables queried. In contrast, TM reduces the total number of tables and does not need a preclassifier.

Another option for reducing the number of tables is to

TABLE II: Mentioned state of the art packet classifiers in order of number of partitions, fewest to largest, comparing key metrics such as worst case lookup time per partition, worst case update time, and worst case storage.

	Linear Search (LS)	**SmartSplit (SS)** [8]	**SAX-PAC(3)** [19]	**Partition Sort (PS)** [11]	**SAX-PAC(2)** [19]	**Tuple Merge (TM)** [20]	**Tuple Space Search (TSS)** [9]
Partition Definition	single partition	fixed $O(1)$ partitions	non-intersecting rules ($\geq$ 3 fields)	sortable rules	non-intersecting rules ($<$ 2 fields)	merged tuples: subset of bits used in rules	tuples: bits used in rules
Partition Data Structure	sorted linked list	hypercuts or hypersplit decision trees	unspecified	binary search tree	binary search tree	hash table with collision limit c	hash table (no collisions)
WC lookup time per partition	$O(n)$ memory accesses	$O(log(n))$ tree height	unspecified	$O(d+log(n))$	$O(log(n))$	$O(1)$ hashed memory access + $O(c)$	$O(1)$ hashed memory access
WC update time	$O(n)$ but often much faster	no update mechanism given	update complexity not specified	$O(d+log(n))$	$O(log(n))$	$O(1)$ hashed memory access + $O(c)$	$O(1)$ hashed memory access
WC Storage	$O(n)$	superlinear in n	unspecified	$O(n)$	$O(n)$	$O(n)$	$O(n)$

reduce the number of tuples by ignoring certain fields (such as the port fields). This results in a drastic reduction in the number of tables, which is exponential in the number of fields. This may cause some false matches; removing fields reduces a rule's ability to exclude packets, but it cannot cause a matching rule to be missed. It may also cause multiple rules to match the same packet. We check the packet against the actual rule to solve both of these cases. We also exploit this optimization which, combined with other optimizations, results in far fewer tables than TSS.

Yet another strategy to avoid querying hash tables is to use Bloom filters [22]. Bloom filters are a very space-efficient probabilistic hash set that can answer whether an item is probably in the set (with the probability being related to memory usage) or definitely not in the set. Since Bloom filters are space efficient, they can be stored in cache and used to check whether a matching rule probably exists before performing a query to slower memory. This is a complementary optimization that TM can also leverage; in fact, it may be more efficient for TM as TM has fewer hash tables than TSS.

Recently, Yingchareonthawornchai *et al.* proposed a new partitioning method called PartitionSort (PS) [11] where they partition rules into a small number of sortable rulesets. They store each partition in a multi-dimensional tree that supports $O(d + \log n)$ searches and updates. They observe that the resulting number of partitions is significantly fewer than TSS and that the first few partitions contain the vast majority of rules, so high priority matches are likely to be found after searching only a few trees. As a result, PS classifies packets significantly faster than TSS with only a modest slowdown in update time. TupleMerge essentially achieves the best traits of both TSS and PS. TM, like TSS, uses hashing so each partition can be searched in $O(d)$ time rather than $O(d+\log n)$ time. TM, like PS, produces far fewer tables than TSS. PS's requirements for total ordering are more relaxed than TM, so PS requires fewer tables, on average, than TM, but this gain is typically offset by the extra $O(\log n)$ time required for searches and updates. Because of this, TM typically outperforms PS in both classification time and update time.

SAX-PAC is another partitioning scheme that requires the fewest partitions but in its most general form does not support fast classification [19]. SAX-PAC requires that the rules in a partition be non-overlapping; these rules can be ordered in any way since at most one rule will match any packet. Furthermore, specific fields can be ignored as long as the rules are still non-overlapping with only one extra false positive check. Unfortunately, while non-overlapping is a natural partitioning requirement to impose on rules, it does not provide any type of fast search guarantee unless we use only two fields. We thus identify two versions of SAX-PAC: SAX-PAC(2) which uses at most two fields and SAX-PAC(3) which uses at least three fields. SAX-PAC(2) supports rule update and fast classification, but PS requires fewer partitions than SAX-PAC(2) while still using binary search trees, so we compare against PS rather than SAX-PAC(2) in our experiments. SAX-PAC(3) unfortunately does support fast packet classification, so we do not compare against SAX-PAC(3). That said, we do use SAX-PAC(3) as a benchmark to determine how effective TM's partitioning scheme is in practice.

C. *Packet processing frameworks*

While a complete survey of all software processing frameworks (for which we refer the interested reader to [23]) is out of the scope of this paper, we briefly describe the two frameworks we use in this this paper: Open vSwitch (OVS) [21], [10] and Vector Packet Processor (VPP) [24], [25], [26]. Both leverage the Intel Data Plane Development Kit (DPDK)[27] to deal with the low-level networking operations (i.e. packet reception, forwarding). We describe both from the context of embedding online packet processing algorithms to implement ACLs.

1) Open vSwitch (OVS): Briefly, OVS is a multi-layer software switch released as open-source in the context of the Linux Foundation's Open vSwitch project [28] and it is specifically designed to tackle the OpenFlow classification problem in software. As described in [29], OVS operates as a daemon which controls a group of network interfaces: the packet forwarding between the interfaces is blocked until a rule enables a forwarding action. We now highlight three key properties about OVS that are relevant for our experiments. First, OVS uses a generic table of match/action rules meaning that its Classifier may include exact match rules (for L2 switching) and/or longest prefix matching rules (for L3 routing). Second, differently from VPP, OVS processes packets one at a time which means that once a packet is classified, the packet classification process moves on to process the next packet. Third, OVS has a three-tiered cache to speed up packet classification [30].

2) Vector Packet Processor (VPP): was recently released as open-source software as part of the Linux Foundation's Fast Data IO (FD.io) project [31]. Before its release, VPP was present in high-end Cisco routers. VPP is designed to take advantage of general-purpose CPU architectures and implements a full network stack in user-space. We now highlight three key VPP properties that are relevant for our experiments. First, VPP has separate *nodes* in a *forwarding graph* where each node implements a specific forwarding function such as L2 exact matches, L3 routing via longest prefix matches, and ACL matching. Second, VPP processes packets in batches of typically 256 packets. Combining the above two properties, VPP packet classification moves the batch of packets from node to node in the forwarding graph processing the appropriate packets in the batch at the appropriate node using the corresponding table and algorithm from that node. Thus, unlike in OVS, once packets from a batch are dropped in an earlier node in the forwarding graph, those void packet slots in the batch are not used by the next node in the forwarding graph. Third, the current version of VPP does not come with any form of caching to speed up packet processing. But VPP does leverage prefetching to maximize usage of multiple CPU pipelines and minimize memory access stalls.

3) ACLs in VPP vs OVS frameworks: Currently, OVS includes the TSS packet classification algorithm whereas VPP includes a naïve linear search (LS) algorithm in version 17.04 and TSS in version 17.10 or newer.

In this work, we further extend VPP by deploying the TM algorithm in VPP 17.10. We can then explore (i) the impact of an *algorithm* (e.g., comparing VPP TM vs VPP TSS) and (ii) the impact of the *framework* it is deployed in (e.g., comparing VPP TM vs OVS TSS). We present such experimental comparisons in Section V.

III. Algorithm design

A. Definitions

Let L be a list of n unique rules. Each rule r is a d-tuple of range, prefix, or ternary fields, where each field defines a subset of some finite discrete set (usually $[0, 2^w - 1]$ for some w). A range $s = [l, h]$ is a discrete interval between l and h, inclusive. A prefix p of length $0 \leq w_p \leq w$ is represented by a bit string $\{0,1\}^{w_p} *^{w-w_p}$, where * represents that it matches both 0 and 1. Thus, each prefix represents a range where l and h can only take on certain values. Specifically, a prefix $p = n*^{w-w_p}$, with $0 \leq n < 2^{w_p}$ corresponds to a range $[n \cdot 2^{w-w_p}, n \cdot 2^{w-w_p} + (2^{w-w_p} - 1)]$.

A ternary field is a different generalization of a prefix in that the 0s, 1s, and *s can be mixed in any order. They have an associated bit mask of length w where a 1 represents that the ternary field has a 0 or a 1 in that position while a 0 represents a * in that position. Thus a prefix has a bit mask of $1^{w_p} 0^{w-w_p}$.

A rule contains a tuple of d fields. If all of the fields are the same type, we say that the rule is also of that type. So a rule containing only ternary fields is a ternary rule. A rule containing multiple types of fields is a mixed rule. A rule list likewise inherits the descriptor of the rules it contains. Note that since prefixes generalize to either ranges or ternary fields, we may consider such a mixed range-prefix rule list as purely a range rule list (or likewise for ternary fields) if we do not handle prefixes specially.

Two fields conflict if the subsets defined by them are not disjoint. Thus, two prefixes p_a and p_b conflict if either their first w_a or w_b bits are the same. Two d-dimensional rules r_a and r_b conflict if $r_a(i)$ conflicts with $r_b(i)$ for any $1 \leq i \leq d$.

A prefix or ternary rule r has a mask tuple $\vec{m}_r$ associated with it. This tuple is the associated prefix length or ternary mask of each of its fields. For example, the rule (00*, 010) will have a mask tuple of (2, 3). The bits in r associated with the tuple are its key $k_{t,r}$. We say that $\vec{m}_1 \subseteq \vec{m}_2$ if $\vec{m}_2$ has a 1 in every position where $\vec{m}_1$ does; that is each of the bits used by $\vec{m}_1$ is also used by $\vec{m}_2$. For example, given the rules in Table I $\vec{m}_{r_4} \subseteq \vec{m}_{r_1}$ as each field in $(3, 2)$ is smaller than the corresponding fields in $(3, 3)$.

In this paper, we use the mask tuple to quickly match packets to rules using hashing. Since range fields do not have masks, this does not yield a suitable solution for us. We transform range fields into prefix fields by expanding them to the longest containing prefix; that is the longest prefix that contains both l and h. This changes the semantics slightly, yielding some false positive matches, so we validate such results to ensure correctness.

For example, consider the one-dimensional rule represented by the range [9, 10] being matched against the packets 7, 8, 10, and 12. Since [9, 10] cannot be represented by a prefix, we widen it to [8, 11], which can be represented by the prefix 10**. This prefix will match 8 and 10 but reject 7 and 12. We then validate against the original range, rejecting 8 as a false positive, but accepting 10.

B. TupleMerge in a nutshell

There are two key differences that make TupleMerge faster at packet classification than TSS. First, TM exploits the observation that not all bits are always required to identify which rules are potential matches. In TSS, rules are separated so that rules that use exactly the same tuple will end up in the same hash table and every other rule will end up in a different table. Many rules have similar but not identical tuples which TSS places in separate tables. TM allows these rules to be placed into the same table, reducing the number of tables required; this leads to faster classification.

Second, TM has methods for reducing the number of hash collisions. It is possible for two keys to be hashed to the same value. Additionally, both TSS and TM may produce the same hash key from different rules because of information that is omitted. For TSS, this is because most implementations will omit port fields (which are ranges) and sometimes other fields as well; the exact field selection is usually chosen by the designer ahead of time and is the same for all tables. For TM this occurs because it has decided to omit some bits from the tuple. In either case, this can create false matches where a rule is incorrectly determined to match a rule (but it cannot cause a matching rule to be missed). TSS has only one option when this happens; it must sequentially search all of the colliding rules until it finds a true match. TM uses the number of collisions to control the number and specificity of its tables; if there are too many collisions (determined by parameter c) it adds more specific tables to reduce the number of collisions so that no table will have more than c rules collide on any given packet.

We now explain how TM allows rules with different tuples to be placed in the same hash table. Like TSS, each TM hash table T has an associated tuple $\vec{m}_T$. In TM unlike TSS, a rule r may be placed in T if $\vec{m}_T \subseteq \vec{m}_r$; in TSS, r may be placed in T only if $\vec{m}_T = \vec{m}_r$. We create a hash key $k_{T,r}$ from $\vec{m}_T$ and r by extracting the bits specified by $\vec{m}_T$ from r. The key issue is identifying what tuples $\vec{m}_T$ should be constructed for each ruleset, particularly as we construct the classifier in an online fashion. We describe this in more detail in our Rule Insertion section below.

Consider for instance the classifier in Table I: r_2 has tuple $(3,3)$ and r_4 has tuple $(3,2)$. We could place both r_1 and r_4 into the same table if the table used $(3,2)$ for its tuple. If we wanted to use $(3,3)$ for the tuple, we would find that r_4 is incompatible since $3 > 2$; this would require splitting r_4 into two rules which we do not do because this complicates updating. Another possibility is to use $(3,1)$ for the tuple; both rules would have a hash key of $(010, 1**)$ which creates a hash value collision in the table. This tuple would be allowed if the number of collisions in this hash value does not exceed c. TM maintains a list $\mathcal{T}$ of tables created along with their tuples where the tables are sorted in order of the maximum priority rule contained and a mapping $F : L \to \mathcal{T}$ of rules to tables. During packet classification, only $\mathcal{T}$ is used. The mapping F supports fast deletions of rules. Both require space that is linear in n.

C. TupleMerge structures

1) Packet Classification: TupleMerge classifies packets just like TSS classifies packets; we search for a packet in each table in $T \in \mathcal{T}$ in order by extracting the bits specified by $\vec{m}_T$ from the packet and hashing those bits to see if a matching rule is found. As with TSS, once a high priority matching rule is found, later tables with lower priority rules do not need to be searched. While the table construction algorithm changes, the packet classifier can remain the same. This is a potential implementation advantage that allows TM to be easily integrated into existing implementations.

2) Tuple Selection: We now describe how TupleMerge chooses which tuples to use for tables. For TSS, no choice is necessary as it simply uses the tuples defined by the used bits for each rule. For TM, we have the opportunity to use fewer bits to define tuples which can reduce the number of tables, at the cost of increasing the number of collisions. We strive to balance between using too few bits and having too many collisions and using too many bits and having too many tables. One key observation is that if two rules overlap, having them collide in the same table is a better option than creating separate tables for the two rules. For this reason, we maintain a relatively high collision limit parameter c.

Consider the situation when we have to create a new table T for a given rule r. This occurs for the first rule inserted and for later rules if it is incompatible with all existing tables. In this event, we need to determine $\vec{m}_T$ for a new table. Setting $\vec{m}_T = \vec{m}_r$ is not a good strategy; if another similar, but slightly less specific, rule appears we will be unable to add it to T and will thus have to create another new table. We thus consider two factors: is the rule more strongly aligned with source or destination addresses (usually the two most important fields) and how much slack needs to be given to allow for other rules. If the source and destination addresses are close together (within 4 bits for our experiments), we use both of them. Otherwise, we drop the smaller (less specific) address and its associated port field from consideration; r is predominantly aligned with one of the two fields and should be grouped with other similar rules. This is similar to TSS dropping port fields, but since it is based on observable rule characteristics it is more likely to keep important fields and discard less useful ones.

We then look at the absolute lengths of the addresses. If the address is long, we are more likely to try to add shorter lengths and likewise the reverse. We thus remove a few bits from both address fields with more bits removed from longer addresses. For 32 bit addresses, we remove 4 bits, 3 for more than 24, 2 for more than 16, and so on (so 8 and fewer bits don't have any removed). We only do this for prefix fields like addresses; both range fields (like ports) and exact match fields (like protocol) should remain as they are.

3) Rule Insertion: Now consider the case where we need to insert a new rule r into TM's classifier. We first try to find an existing table that is compatible with r. We search the tables $T \in \mathcal{T}$ in order of max priority rule. For each table T, r is compatible with T if $\vec{m}_T \subseteq \vec{m}_r$. If no compatible table is found, we create a new table for r as described above.

If a compatible table is found, we perform a hash probe to check whether c rules with the same key already exist within T. If not, we add r to T and add the mapping $r \to T$ to F. Otherwise, this is a signal that we ignored too many bits in $\vec{m}_T$ and we need to split the table into two tables. We select all of the colliding rules ℓ and find their maximum common tuple $\vec{m}_\ell$. Normally $\vec{m}_\ell$ is specific enough to hash ℓ with few or no collisions. We then create a new table T_2 with tuple $\vec{m}_\ell$ and transfer all compatible rules from T to T_2. If $\vec{m}_\ell$ is not specific enough, we find the field with the biggest difference between the minimum and maximum tuple lengths for all of the rules in ℓ and set that field to be the average of those two

TABLE III: TM builds 3 tables

Rule	Source	Dest	Nominal Tuple	Used Tuple
r_1	000	111	(3, 3)	(3, 2)
r_2	010	100	(3, 3)	(3, 2)
r_3	011	100	(3, 3)	(3, 2)
r_4	010	11*	(3, 2)	(3, 2)
r_5	10*	101	(2, 3)	(2, 0)
r_6	110	***	(3, 0)	(2, 0)
r_7	***	011	(0, 3)	(0, 3)

TABLE IV: An alternate TM scheme builds 2 tables

Rule	Source	Dest	Nominal Tuple	Used Tuple
r_1	000	111	(3, 3)	(3, 0)
r_3	011	100	(3, 3)	(3, 0)
r_4	010	11*	(3, 2)	(3, 0)
r_6	110	***	(3, 0)	(3, 0)
r_2	010	100	(3, 3)	(0, 3)
r_5	10*	101	(2, 3)	(0, 3)
r_7	***	011	(0, 3)	(0, 3)

values. We then transfer all compatible rules as before. This guarantees that some rules from ℓ will move and that T_2 will have a smaller number of collisions than T did.

4) Rule Deletion: Deleting a rule r from TupleMerge is straightforward. We can find which table T contains r with a single lookup from F. We then remove r from T and the mapping $r \rightarrow T$ from F. This requires $O(1)$ hash updates.

5) Offline Table Selection: We now consider the case where we periodically rebuild the tables. We restrict ourselves to fast algorithms that take roughly a second, a reasonable amount of time given that OpenFlow hardware switches can require roughly half a second for the data plane to catch up to the control plane update command. In this scenario, we try to place as many high-priority rules in early tables to minimize the number of tables searched. Specifically, if a high priority rule is matched in an early table, tables that contain only lower priority rules do not need to be searched.

Let L_i be the first i rules in L. Let $\vec{m}_{L_i}$ be the maximum common tuple of these rules; that is it is the tuple that contains all of the bits that they have in common. We now create a counter H of hashes, ℓ_i, a proposed table, and $\bar{\ell}_i$, the rules not in ℓ_i. For each rule $r \in L$, we compute $h(r)$. If $H(h(r)) < c$, where c is an argument representing a maximum number of collisions, we add r to ℓ_i and increment $H(h(r))$. Otherwise we add r to $\bar{\ell}_i$. Each different $\vec{m}_{L_i}$ will yield a ℓ_i and $\bar{\ell}_i$. To maximize the probability that we do not need to search future tables, we select the ℓ_i that minimizes the maximum priority of $\bar{\ell}_i$ (the first missing rule is as late as possible), breaking ties by maximum $|\ell_i|$, thus reducing the number of tables required. We make a new table $T = (\ell_i, m_{L_i})$ and append it to $\mathcal{T}$, and repeat on the remaining rules $L' = \bar{\ell}_i$ until L' is empty.

For example, consider the rules in the classifier in Table I. Tuple $(3, 2)$ can contain 4 rules without any collisions: r_1, r_2, r_3, and r_4. Tuple $(3, 3)$ can only contain the first three rules, while $(2, 2)$ has a collision between r_2 and r_3, making them worse choices. We create a table containing rules r_1, r_2, r_3 and r_4 and repeat on the remaining three rules. The results from this process can be seen in the classifier in Table III.

Other strategies exist that may yield fewer tables. For example, the classifier in Table IV contains only two tables. However, unless the incoming packet matches rule r_1, both tables will need to be searched. The classifier in Table III will need to search only 1 table if any of r_1, r_2, r_3, or r_4 match, which may provide better results in practice even if the worst case is not as good.

D. TupleMerge implementation in VPP

We faced two main challenges in implementing TM in VPP: (i) updating the data structure to support both IPv4 and IPv6 and (ii) accommodating the batch processing workflow of VPP [26]. Due to lack of space, we can only provide a survey of the implementation choices of TM for VPP; the complete source code is available at [32].

Besides dealing with these two challenges, we also needed to add flow caching to VPP as it did not previously provide any caching mechanism. Since our TM in VPP differs from our TM in simulation, we will compare the performance of TM in the two settings. These results can be seen in Section V-B.

1) TM changes: In the simulation implementation the 5-tuple data structure uses ranges (defined as intervals between a low and a high value), so each field requires two variables representing the lower and upper bound. Instead, in our VPP implementation the 5-tuple is instead represented by a bitmask. This representation can be more easily adapted to work with hash tables; furthermore, most of the packet fields are accessed via prefix match (a subset of bitmask match). As in the simulation implementation, we handle the ranges for port fields by using the smallest bit mask (prefix) that includes the range; this requires checking matching packets against potentially a linked list of actual ranges that correspond to the bit mask to ensure they are true matches. We further extend the TM structure to handle IPv6 packets and improve the 'relax method' to deal with extra IPv6 bytes.

Finally, unlike the simulation implementation that uses a separate hash table for each partition, we use a single hash table by simply adding an ID field for each table. That is, a query is formed by combining the key with this ID so that only the correct "virtual table" within the single hash table is queried. This simplifies the management of tables in part by eliminating the issue of having different sized tables.

2) Implementing Caching in VPP: As mentioned before, OVS uses a three-tiered cache (i.e. EMC [30]) during the packet classification stage whereas VPP did not have any cache for packet classification. Since real traffic exhibits *spatial skew* that a cache can exploit to improve packet classification performance, we needed to add a front-end cache system to VPP so that we could fairly compare different packet classification algorithms.

In our cache, we store the most frequently seen flows (i.e., the heavy hitters) as opposed to the most frequently seen rules because rules must be stored with some context such as the matching flow or else erroneous behavior may result. For example, suppose we start with an empty cache and there are two rules r_1 and r_2 where r_1 has higher priority than r_2 and r_1 is more specific than r_2. Further suppose that some packet from flow f_2 arrives that matches r_2 but not r_1 and that and we simply store the matching rule r_2 in the cache without the context that it matched a packet from flow f_2.

If we then process a packet from flow f_1 that matches both r_1 and r_2, if we search the cache, we will find rule r_2 and erroneously classify this packet using r_2's action which may be different than r_1's action. Since the cache stores flows, it maintains a representation of the flow (plus the action to apply) instead of the matched rule. Note that when a rule is added or removed, the results given by the cache could be inconsistent with respect to the new ruleset. We address this issue by flushing the entire cache whenever rules are added or removed to ensure correct operation.

Our ACL plugin processes a packet using the following sequence of actions. First, it checks to see if the packet belongs to an already in-progress cached flow. If so, it applies the resulting action. If not, it will start the TM packet classification mechanism. This creates an opportunity to tradeoff ACL computation for cache space.

IV. Simulation results: Algorithmic comparison

We now evaluate the performance of the TM algorithm and compare it with the other classification algorithms presented in Section II. We make our simulator available as open-source software at [33]. Note this simulation evaluation only compares the algorithms in isolation and does not necessarily represent how the algorithms will perform when deployed in an actual network. We measure that in Section V.

A. Simulation Setup

We ran our experiments on rulelists created with ClassBench [34], [35]. ClassBench comes with 12 seed files that can be used to generate rulelists with different properties. While the seeds are divided into three categories (5 access control lists (ACL), 5 firewalls (FW), and 2 IP-chain (IPC)), different seeds in the same category can have very different properties.

We generate rulelists of nine different sizes, from 1k up to 256k rules. For each of the 12 ClassBench seeds, we generate 5 rulelists of each size for 540 rulelists total.

We use these rulelists in three types of scenarios. For an offline scenarios, we construct a static data structure from all the rules and then use the data structure to classify packets. For an online scenarios, we have two protocols. In the first protocol, we create a dynamic data structure where the rules are fed to the online algorithm one at a time in a random order where each algorithm receives the rules in the same order. The resulting classifier is then used to classify packets. It classifies 1,000,000 packets which are generated using the ClassBench method for generating packets given a rulelist. In the second protocol which we use to measure update time, half of the rules are selected at random to be in the initial active rulelist. We create an initial data structure by inserting these rules one at a time into the data structure in a random order where each algorithm receives the rules in the same order. A list of 500,000 insertions and 500,000 deletions is then created and shuffled. Each insertion inserts one of the excluded rules and each deletion removes one of the included rules, each chosen at random from the available rules. This sequence is the same for each algorithm. In this protocol, we do not use the resulting data structure to classify packets.

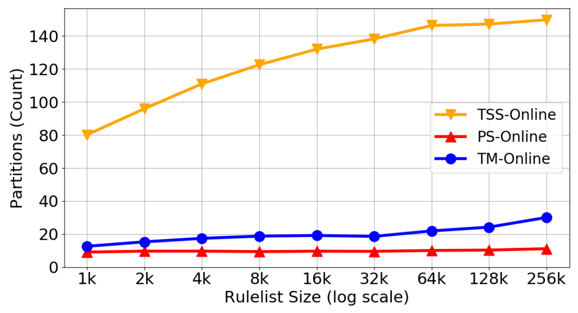


Fig. 2: Average # of partitions required for each rulelist size

For a given rulelist L and a given algorithm A, we measure four things. First is classification time, denoted $CT(A, L)$, which is the total time required to classify all 1,000,000 packets divided by 1,000,000. Second is the number of partitions $P(A, L)$ in the resulting data structure. This is useful as a static estimator of each data structure's classification speed. Third is update time, denoted $UT(A, L)$, which is the total time required to perform all 1,000,000 rule updates divided by 1,000,000. Note we do not include the time to create the initial data structure with half the rules. Finally is memory, denoted $M(A, L)$, which is the total memory required by the data structure.

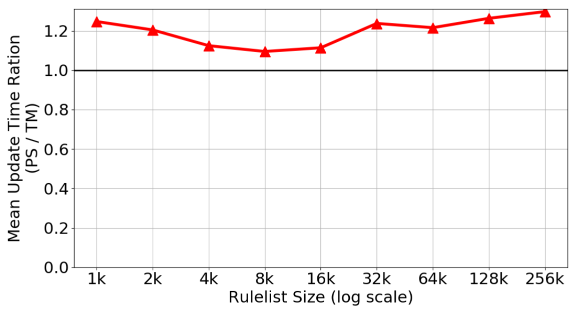
For each metric, we average the results across all rulelists of a given size or seed. To better compare an algorithm A, typically PS, against TM, we compute the relative classification time of A and TM on a rulelist L, denoted $RCT(A, TM, L)$, as the ratio $CT(A, L)/CT(TM, L)$; higher values are better for TM. Likewise, we compute the relative update time of A and TM for rulelist L, denoted $RUT(A, TM, L)$, to be the ratio $UT(A, L)/UT(TM, L)$. We average these relative time metrics across all rulelists of a given size or seed. We note that for the two relative time metrics, the range of ratios is relatively small whereas for the raw metrics, some values are much larger than others and thus have a much larger impact on the final average. This leads to some apparent discrepancies between the two averages.

These simulation were run on a machine with an Intel Xeon CPU at 2.8 GHz, 4 cores, and 6 GB of RAM running Ubuntu 16.04. We verified correctness of classification results by ensuring each classifier returns the same answer for each test packet.

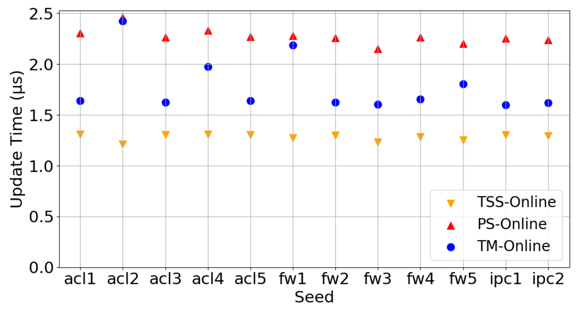
B. Comparison with PartitionSort (PS)

We compare TupleMerge against PS [11] as it represents the state-of-the-art online packet classifier. *Briefly, in our simulation results, online TM outperforms online PS in both classification speed and update time.* Focusing on the relative metrics averaged over all seeds and all sizes, TM performs rule updates 30% faster than PS, and TM classifies packets 34.2% faster than PS. PS also requires 2.71 times the amount of memory that TM does, though both require only linear memory in the number of rules.

We now examine these results in more detail. The number of tables required for online TM, online PS, and TSS are shown in Figure 2. Relative update time results averaged over each rulelist size are shown in Figure 3a, and raw update time results for online TM, online PS, and TSS for each seed for the

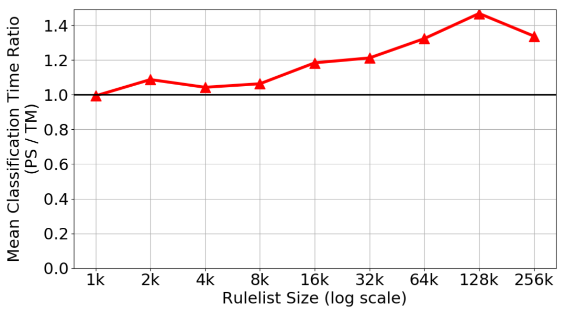


(a) Relative Update Time between PS and TM

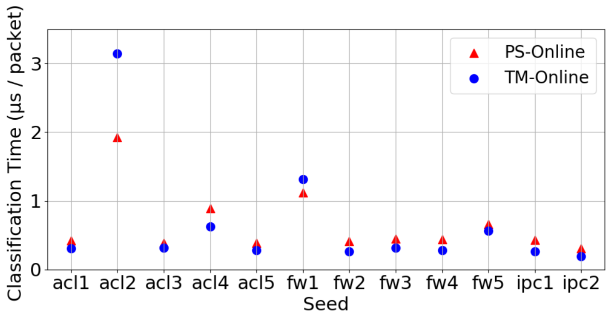


(b) Absolute Update Time on 256k rules

Fig. 3: Online Update Time



(a) Relative Online Classification Time between PS and TM



(b) Absolute Online Classification Time on 256k rules

Fig. 4: Online Classification Time

256k rulelists are shown in Figure 3b. Relative classification time results averaged over each rulelist size are shown in Figure 4a, and raw classification time results for online TM and online PS for each seed for the 256k rulelists are shown in Figure 4b.

Looking more closely at update time, we see that TM consistently outperforms PS in every way. Specifically, TM has a faster update time for all rulelist sizes and for all 12 seeds. For the largest rulelists, TM outperforms PS for all 12 seeds and has an average update time of only $1.78\mu s$ whereas PS has an average update time of $2.27\mu s$.

Looking more closely at classification time, TM again outperforms PS for all rulelist sizes where the gap between TM and PS generally increases with rulelist size, but we do observe that PS does outperform TM for some seeds. On the largest rulelists, TM takes $0.64\mu s$ on average to classify a packet, while PS takes $0.65\mu s$. TM classifies faster on 10 of the 12 seeds, but the two remaining seeds (ACL2 and FW1) are the slowest for both PS and TM which skews the average classification times to be closer than the average relative classification time. Focusing on the other 10 seeds, TM takes $0.15\mu s$ to $0.76\mu s$ (average $0.34\mu s$) to classify a packet and PS takes $0.26\mu s$ to $1.00\mu s$ (average $0.47\mu s$) to classify a packet. These results can be seen in Figure 4b.

These results can be explained by the number of partitions required and the time required to search each partition. We plot the average number of partitions required by TM and PS for each rulelist size in Figure 2. As expected, TM typically requires a few more partitions than PS. Focusing on the 256k rulelists, TM requires an average of 30 partitions whereas PS requires an average of 11 partitions. TM typically achieves a smaller classification time because TM uses hash tables ($O(d)$ time) whereas PS uses trees ($O(d+\log n)$ time). However, for the worst case seeds, ACL2 and FW1, TM requires 131 and 53 partitions, respectively, whereas PS requires only 24 and 17 partitions, respectively. For the remaining ten seeds, TM requires at most 38 partitions.

The number of partitions has more effect on classification time than rule update time. For rule insertion, once we find an available partition, we do not need to search later partitions. For rule deletion, both schemes store a pointer to the partition containing each rule.

For memory, for the 256k rulelists, TM requires only 4.47 MiB on average whereas PS requires 12.19 MiB on average.

C. *Comparison with Tuple Space Search (TSS)*

We now compare TM with TSS [9]. *Briefly, TM is, on average, 7.43 times faster at classifying packets than TSS, and TM is, on average, 39% slower at rule updates than TSS.*

For our 256k rulelists, TM takes on average $0.64\mu s$ to classify a packet whereas TSS takes on average $2.93\mu s$ to classify a packet. For our 256k rulelists, TM takes on average $1.78\mu s$ to update a rule whereas TSS takes on average $1.28\mu s$ to update a rule. We thus see that for only a small penalty in rule update, TM achieves a significant gain in classification time. This improvement in classification time is largely explained by the fact that TM requires many fewer tables than TSS, as can be seen in Figure 2. Besides improving classification time, TM also requires 1.93 times less memory on average than TSS.

Given the significant improvement in classification time and even memory, TM clearly is a better choice than TSS as an online packet classification algorithm. At the same time, TSS is the de facto standard algorithm in deployed networks. It

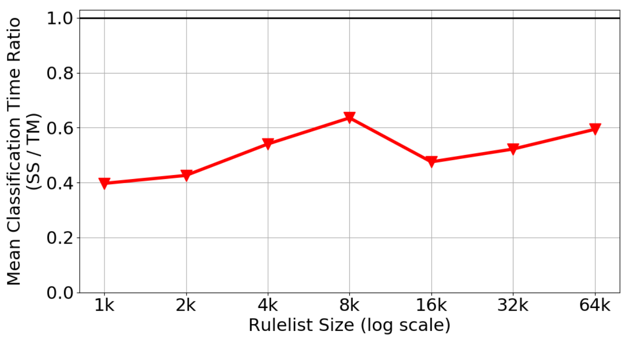


Fig. 5: Relative classification time between SS and TM

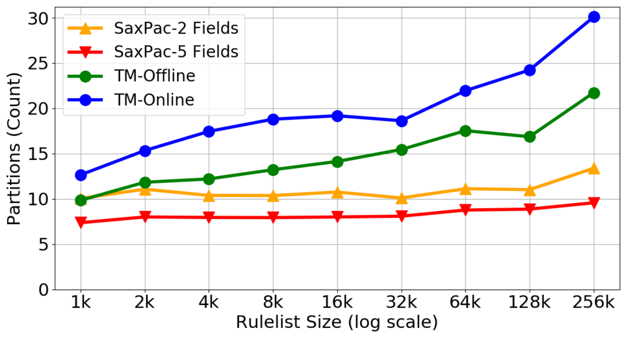


Fig. 6: Number of partitions produced by Sax-Pac vs TM

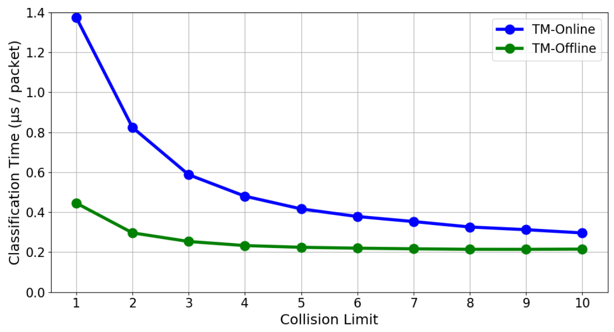


Fig. 7: TM search time vs small collision limits c

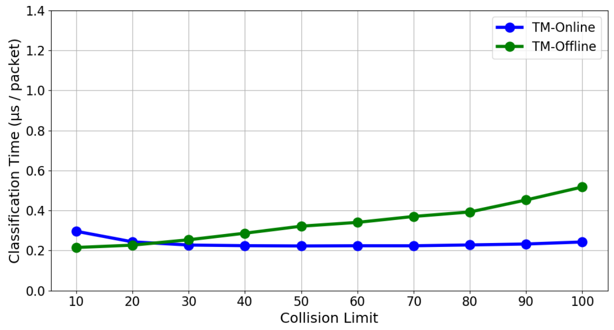


Fig. 8: TM search time vs larger collision limits c

remains to be seen, in Section V, whether TM continues to outperform deployed TSS.

D. Comparison with SmartSplit (SS)

We now compare offline TupleMerge with SS, the current state-of-the-art offline packet classification algorithm, to assess how much we give up in classification time to achieve fast updates [8]. SS selectively builds either HyperCuts [7] or HyperSplit [16] trees depending on expected performance and memory requirements. We compare these algorithms on classification time, construction time, and memory usage. The largest rulelists we use are the 64k rulelists because of SS's slow construction time.

SS is significantly faster than TM (or the other methods studied). For our 64k rulelists, on average, SS classifies packets in $0.12\mu s$ whereas offline TM classifies packets in $0.24\mu s$. The relative classification time required by SS and TM can be seen in Figure 5.

TM requires orders of magnitude less time to build a classifier than SS. While TM can usually generate a classifier in seconds even for very large rulesets, SS can take hours, even for reasonably sized rulesets.

Finally, *SS has very unpredictable memory requirements.* For smaller rulelists, it produces a single HyperCuts tree, maximizing classification speed. As the number of rules increases, it switches over to multiple trees and/or HyperSplit trees, both of which reduce rule replication and thus memory required. This doesn't fix the underlying problem though. Eventually there are enough problematic rules that this is not enough.

E. Comparison with SAX-PAC

SAX-PAC [19] partitions the rulelist into non-overlapping or order-independent sets. As stated earlier, we do not directly compare against SAX-PAC (2) since PS requires fewer partitions than SAX-PAC (2). We do compare against SAX-PAC (3) to measure how well our table selection scheme performs.

TM requires a comparable number of partitions to SAX-PAC. When SAX-PAC uses all of the fields, TM requires on average 2.1 times more partitions than SAX-PAC. Compared against SAX-PAC (2), TM requires only 68% more tables. This shows that the number of partitions required by TM is not too much larger than the theoretical minimum. These results can be seen in Figure 6.

F. TupleMerge Collision Limit

We now measure how the collision limit c influences TM's effectiveness. A higher c allows to produce fewer tables, but each table may take longer to search. We report the average classification time as a function of c in Figures 7 and 8.

We make the following observations. First, online TM needs a larger c than offline TM. The optimal c values are 40 and 8 for online TM and offline TM, respectively. Second, for the same c value, offline TM tends to have more entries with collision limits close to c than online TM leading offline TM to be slower than online TM for $c > 25$ in the average case. Third, smaller rule lists generally need smaller c. Finally, using a fixed value of c will not significantly increase classification time due to flatness of the graph near the optimal c. The maximum increases in average classification time were 7.8% and 4.4% for online TM ($c = 40$, fw1 seed) and offline TM ($c = 8$, fw5 seed), respectively.

Since the choice of the c value is a critical aspect of our algorithm, we performed additional simulations using ad-hoc rulesets generated mixing ClassBench ACL and FW rulesets. In these additional experiments, using the fixed c instead of the optimal c increased the classification time by at most 2.4% for the online classifier and 2.8% for the offline classifier.

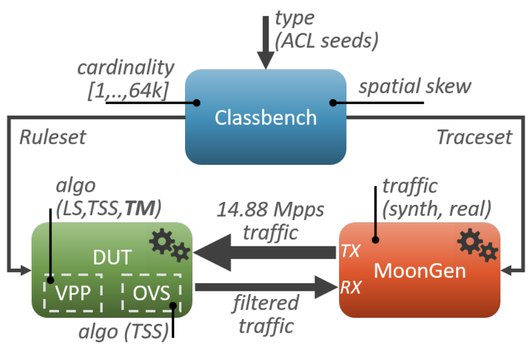


Fig. 9: Synoptic of experimental methodology

V. Experimental results: Frameworks comparison

As mentioned before, an accurate evaluation of a complete system, e.g. when the packet classification algorithm is embedded in the software router, is missing from the literature. Indeed, although the simulation results from the previous section provide a fair comparison of algorithm performance, these results are only valid as a reference in a controlled environment. We now provide an accurate evaluation of our VPP TM implementation and the current state of the art OVS TSS implementation, and we provide access to our open source code [32].

A. Experimental Methods

We first outline our methodology for getting realistic yet repeatable benchmark results. We illustrate in Fig. 9 the key components that we use: namely, ClassBench, MoonGen and the VPP/OVS Device Under Test (DUT).

1) Testbed: Our testbed setup consists of a COTS server with 2× Intel Xeon Processor E52690, each with 12 physical cores running at 2.60 GHz in hyper-threading and 576KB (30MB) L1 (L3) cache. The server is equipped with 2× Intel X520 NICs with dual-port 10Gbps full duplex links directly connected with SFP+ interfaces. In this paper, we stress-test the DUT packet classification performance using a single-core: a complementary viewpoint, that we leave for future work, would be to leverage Receive Side Scaling (RSS) to find the minimum number of threads able to sustain the input rate.

2) Ruleset generation (ClassBench): We generate rulesets using ClassBench [34], [35]. Consistent with our simulations and much prior work [20], [11], [8], [19], we use the twelve different ClassBench default seeds (5×ACL, 5×FW and 2×IPC). For each seed, we generate rulesets ranging in size from 2 rules up to 64K rules. We further make each rule including the default rule have the `ACCEPT` action so that no flows are dropped due to classification; thus we stress the DUT as much as possible.

3) Synthetic workload generation (ClassBench and MoonGen): We generate synthetic workloads using ClassBench and MoonGen. We first use ClassBench to generate tracesets, that are sequences of synthetic packet headers, that exercise a given ruleset. To provide conservative results, we generate uniform random synthetic tracesets which we feed to the workload generator. We then develop a MoonGen [36] application that takes a ClassBench traceset as input and generates a 10 Gbps stream of 64 Bytes packets as output. We ensure each packet payload has the minimum size of 64 Bytes to stress test the packet classifier. Considering bytes of Ethernet preamble and 9.6ns of inter-frame gap, this leads to a worst-case rate of 14.88 Mpps.

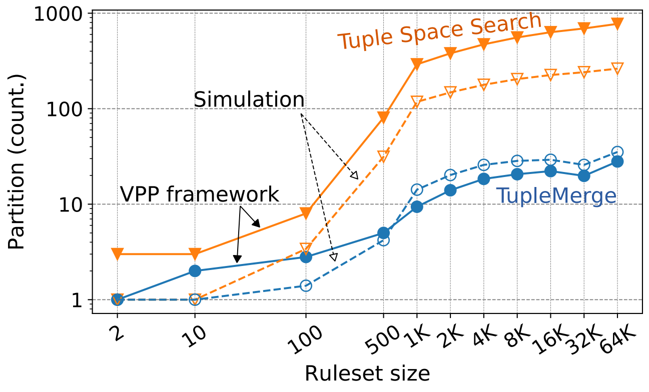


Fig. 10: Number of partitions required by TM and TSS: Simulation vs VPP 17.10 Experiments.

4) Real workload generation (CAIDA): We also use a real traffic trace obtained from CAIDA [37] capping the original packets to 64B of size to obtain a worst-case traffic rate[1]. Since this is a real trace, we can test how OVS and VPP caches can exploit the spatial skew. At the same time, since the packets in a CAIDA trace have no relationship with the ClassBench seeds, most packets will not match any rule meaning all tables will have to be searched and only the *default* rule will be matched. This is a worst case scenario for both TM and TSS, but since TM has fewer tables the impact is less significant for our algorithm.

B. VPP-TM validation: Simulation vs Experiments

We validate the fundamental correctness of our TM and TSS VPP implementations by comparing the number of partitions generated by them for a given ruleset with the number of partitions generated by our TM and TSS simulation implementations on the same ruleset. We report this validation in Fig. 10 as a function of the cardinality of the ruleset.

We observe that the actual VPP TSS implementation requires, on average, 170% more partitions than the original TSS algorithm due to IP fragmentation. In fact, an IP fragmented packet does not provide all the fields of the 5-tuple and therefore requires a different mask. This likely could be eliminated by improving the TSS implementation in VPP, but we do not carry this out since VPP TSS is not the focus of our paper. On the other hand, our VPP TM implementation actually *reduces the number of partitions* required with respect to the simulation implementation for rulesets larger than 1K rules. In fact, the simulation implementation uses *ranges* whereas the VPP implementation uses *bitmasks*: as such, the VPP implementation can exploit various bit-level "tricks" to relax and compact different bitmasks resulting in fewer

[1]The experiments used the equinix-sanjose.dirA.20120119-132400.UTC .anon.pcap trace.

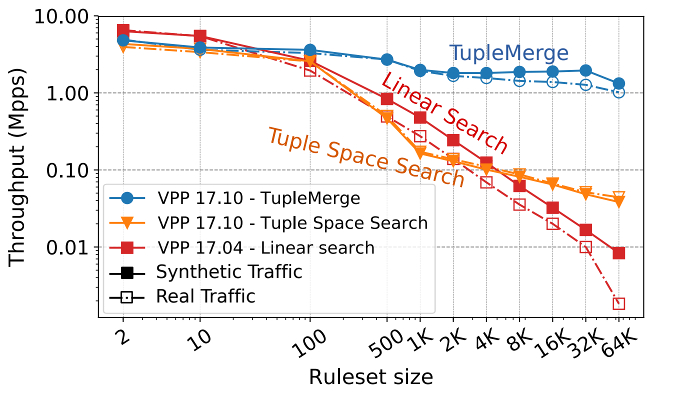


Fig. 11: Experimental throughput stressing the VPP framework with a 14.88 Mpps real vs synthetic traffic.

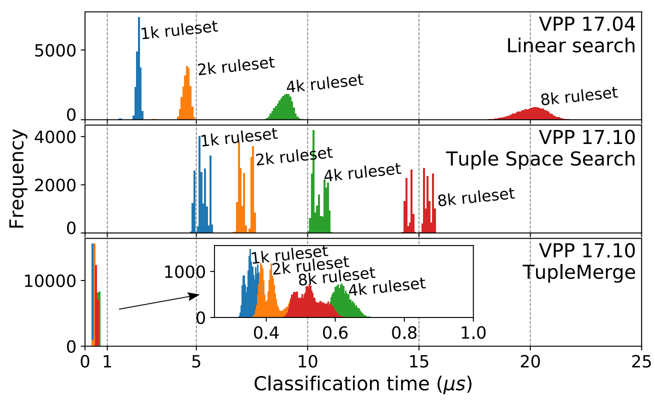


Fig. 12: Distribution of the experimental classification time taken by the different VPP implementations to classify a packet. Results are made with ACL_1 seed.

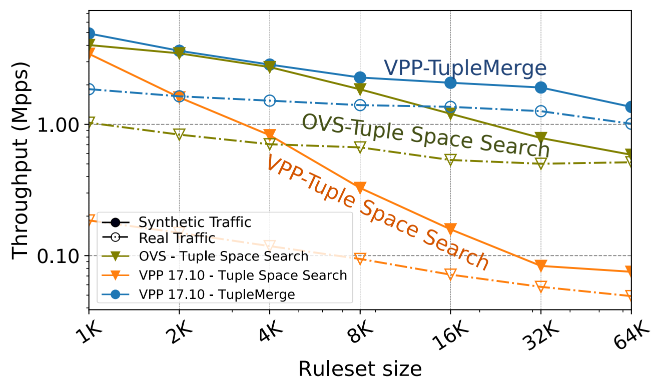


Fig. 13: Experimental throughput stressing VPP and OVS frameworks with a 14.88 Mpps real vs synthetic traffic. For both frameworks a frontend cache is enabled

partitions and thus validating the decision to use bit masks. Moreover, by putting all the partitions in one hash table, we simplify memory allocation which, in turn, leads to more efficient resolution of hash table collisions. This results in fewer memory accesses for each hash table lookup leading to faster processing.

C. Algorithm comparison: VPP implementations of LS, TSS and TM

We now compare the VPP LS, TSS, and TM implementations, noting that LS is only available with VPP 17.04 and TSS and TM are available with VPP 17.10. In these experiments, we do not use the VPP flow cache so that we can focus on the packet classification algorithm's performance. We acknowledge that VPP 17.04 may contribute a little towards the relatively poor performance of LS, but we feel safe in suggesting it is a relatively small factor and that naïve LS is the main cause of the performance difference.

For each ruleset, we stress test our three algorithms using both a synthetic workload and our real traffic workload under a worst-case input rate (a stream of 14.88 millions of packets per second) that is processed on a single core. We remind the reader that for the synthetic workload, *all rules are matched with equal probability*, whereas for the real traffic workload, the default rule is the main rule matched. We report average throughput results for all rulesets of a given size for our synthetic workload and our real traffic workload; confidence intervals are tight but not shown to avoid cluttering the figure.

We report average throughput results in Fig. 11; this figure has 6 lines (3 algorithms for 2 workloads). We make five key observations. First, TM clearly outperforms both TSS and TM once ruleset size is at least 500 rules. Second, the performance gap increases as ruleset size increases reaching one order of magnitude for rulesets of size 4k and nearly two orders of magnitude for rulesets of size 64k rules. Third, there is relatively little difference in performance for our synthetic traffic versus our real traffic, particularly for TM. Fourth, TM performance only slowly degrades as ruleset size increases. TM maintains a throughput of at least 1 Mpps for all ruleset sizes and workloads. Finally, surprisingly, LS matches or outperforms TSS until rulesets have at least 8k rules.

We next inspect the distribution of the per-packet classification time to determine whether or not classification time is stable. Per-packet classification times are reported in Fig. 12 for LS (top), TSS (middle), and TM (bottom), for the ACL_1 seed ruleset sizes 1k, 2k, 4k, and 8k (results are qualitatively similar across seeds). We make two key observations. First, TM classification times, all well under $1\mu s$, are significantly faster than those for LS, which range from $2\mu s$ to over $20\mu s$ and TSS, which range from $5\mu s$ to over $15\mu s$. Second, while the packet classification time increases significantly as ruleset size increases, the packet classification time for TM are quite tight in the range of $0.2\mu s$ to $0.7\mu s$. Thus, TM is much more predictable and stable across ruleset sizes.

D. Effect of VPP front-end cache

We now measure the effect of the VPP front-end cache on the performance of VPP TSS and VPP TM. Specifically, we run the synthetic workloads and the real workload on VPP TSS and VPP TM with front-end cache and VPP TSS and VPP TM without front-end cache with rulesets ranging in size from 1k rules to 64k rules. We report the average multiplicative throughput gain achieved with the front-end cache in Table V; a value of two means that VPP TM with cache has twice the throughput of VPP TM without cache.

TABLE V: Throughput gain of front-end cache with respect to VPP TSS and VPP TM without cache.

Ruleset size	1K	2K	4K	8K	16K	32K	64k	AVG
TM, Syn	2.48	2.01	1.58	1.22	1.1	0.98	1.03	1.49
TM, Real	0.96	0.98	0.97	0.98	0.98	0.99	0.99	0.98
TSS, Syn	21.07	12.38	8.19	4.02	2.45	1.73	1.97	7.4
TSS, Real	1.08	1.08	1.07	1.08	1.08	1.13	1.11	1.09

TABLE VI: Example Source Port Range to Prefix Expansion for the range [1:17].

Rule	Source Port - Prefix	Range representation
r_1	0000 0001	[1:1]
r_1	0000 001*	[2:3]
r_1	0000 01**	[4:7]
r_1	0000 1***	[8:15]
r_1	0001 000*	[16:17]

We make three observations. First, the cache helps both algorithms with the synthetic workload more than with the real workload. This is expected because in the synthetic workload, all rules are matched with equal probability whereas in the real traffic workload, the default rule is the main rule matched. Thus, for the real traffic workload, almost all partitions must be searched and the cache can actually cause a slowdown because of the extra lookup. We see this play out for VPP TM; that is, VPP TM without front-end cache is actually faster on the real workload than VPP TM with front-end cache. Second, the cache helps VPP TSS more than VPP TM largely because VPP TSS is so much slower at classifying packets than VPP TM. Finally, the cache benefit decreases as ruleset size increases which again is to be expected as there will be fewer cache hits as ruleset size increases.

E. Framework comparison: VPP vs OVS

We now compare our VPP TM implementation against the state of the art, represented by OVS TSS. In these experiments, we use OVS's three-tiered cache and the front-end cache we developed for VPP. Thus, these VPP results will generally be faster than the results from section V-C which did not use the VPP front-end cache. Finally, we note that while the VPP front-end cache helps most of the time, the OVS three-tiered cache is at the kernel level and is more effective than the VPP front-end cache.

However, OVS is designed for OpenFlow compliant rules, it is more limited than VPP, particularly in the context of port ranges. Specifically, OVS can only specify fields using prefixes, represented as a $\langle value, mask\rangle$ pair, and thus a single port range must be expanded to multiple prefixes (equivalently $\langle value, mask\rangle$ pairs) [38]. An example of such an expansion is given by the classifier in Table VI. For our rulesets, the average increase in ruleset size due to this range to prefix expansion would be +133% for the five ACL seeds, +248% for the five FW seeds, and +18% for the two IPC seeds. Rather than subjecting OVS TSS to this expansion, we make the following modification to the rules in each ruleset. For VPP, we use each rule as is. For OVS TSS, we modify each rule so that each port range other than all ports is a single port number. This modification clearly favors OVS TSS.

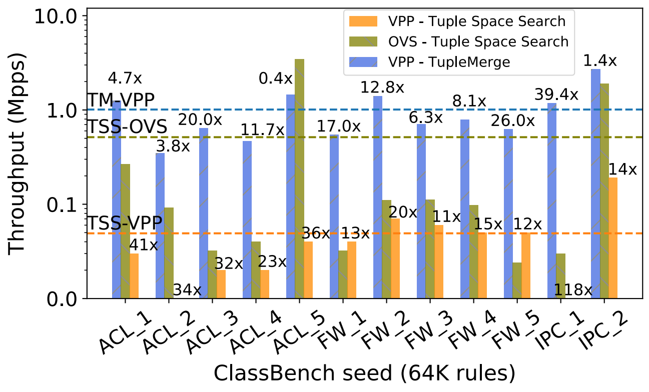


Fig. 14: Experimental throughput of VPP TM, VPP TSS and OVS TSS for different ClassBench seeds (rulesets size equal to 64k) with real traffic; dashed lines represent average throughput over all twelve seeds

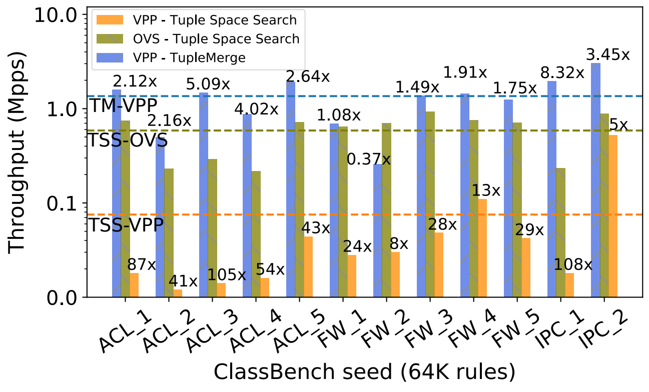


Fig. 15: Experimental throughput of VPP TM, VPP TSS and OVS TSS for different ClassBench seeds (rulesets size equal to 64k) with synthetic traffic; dashed lines represent average throughput over all twelve seeds

We report our average classification throughput results in Fig. 13; these results consist of 6 lines as we have three framework and algorithm combinations (VPP TSS, VPP TM, and OVS TSS) and two workloads (real, synthetic). We make the following observations. First, VPP TM significantly outperforms OVS TSS for the real workload on all ruleset sizes and the synthetic workload on rulesets of size at least 16k rules; for smaller rulesets, VPP TM still outperforms OVS TSS on the synthetic workloads, but the gap is small. One reason the gap is small for small rulesets and synthetic traffic is that OVS's kernel level cache is more effective than VPP's front-end cache. Second, VPP TM experiences a relatively small loss in throughput as ruleset size increases. Third, all the framework and algorithm combinations work better on the synthetic workload than the real workload. This again is explained by the fact that in the synthetic workload, all rules are matched with equal probability whereas in the real traffic workload, the default rule is the main rule matched. In general, for the real traffic, all the partitions must be searched, and TM's superior performance is most evident in this scenario. Finally, OVS TSS outperforms VPP TSS for two reasons. First, due to some implementation issues, VPP TSS produces more partitions than OVS TSS. Second, OVS's kernel level cache is more effective than VPP's front-end cache.

We next explore the results for the largest ruleset size of 64k rules in more detail showing all twelve ClassBench seeds separately. We plot the results with real traffic, Fig. 14, and synthetic traffic, Fig. 15. In both figures, the bars show the throughput achieved by each framework and algorithm combination (VPP TM, VPP TSS, and OVS TSS), and the dashed lines indicate the average throughput for each framework and algorithm combination over all twelve seeds. We see that VPP TM outperforms OVS TSS on almost every seed except ACL5 for the real traffic and FW2 for the synthetic traffic. Overall, VPP has an average speedup of $12.63\times$ for real traffic and $2.86\times$ for synthetic traffic. VPP TM outperforms VPP TSS on all seeds for both types of traffic with an average speedup of $31.20\times$ for real traffic and $45.86\times$ for synthetic traffic.

VI. Conclusion

We provide the following contributions. First, we provide a flexible hashing scheme that uses far fewer tables than Tuple Space Search and thus classifies packets much more quickly. Second, we show how to effectively insert and delete rules while retaining fast classification speeds. Together, we produce TupleMerge, a new online packet classification method that surpasses the previous state-of-the-art method, PartitionSort, on classification speed, update time, and memory.

We further integrate TupleMerge into the widely used open source VPP framework producing VPP TM. Our experiments show that VPP TM achieves consistently superior performance to the previous state of the art, namely VPP TSS and OVS TSS. Specifically, VPP TM outperforms VPP TSS by two orders of magnitude in terms of classification speed when we ignore the front-end cache. Likewise, when using caches, VPP TM classifies packets $12.63\times$ faster, on average, than OVS TSS, even when OVS TSS is given rules where port ranges are reduced to single ports.

Finally, we also provide an open source implementation and test environment [32] that can be used by anyone to carry out their own realistic yet reproducible experiments with VPP LS, VPP TS, VPP TM, and OVS TSS.

Acknowledgments

This work benefited from support of NewNet@Paris, Cisco's Chair "Networks for the future" at Telecom ParisTech. This work is partially supported by the EU Commission in the frame of the H2020 project 5G-PICTURE (grant #762057).

References

[1] A. Hakiri, A. Gokhale, P. Berthou, D. C. Schmidt, and T. Gayraud, "Software-defined networking: Challenges and research opportunities for future internet," *Computer Networks*, vol. 75, pp. 453–471, 2014.

[2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM CCR*, 2008.

[3] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[4] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about sdn flow tables," in *Passive and Active Measurement*, 2015.

[5] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *ACM SIGCOMM HotSDN*, 2013.

[6] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, 2001.

[7] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *ACM SIGCOMM '03*, 2003.

[8] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *IEEE ICNP*, 2014.

[9] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *ACM SIGCOMM CCR*, 1999.

[10] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch," in *NSDI*, 2015, pp. 117–130.

[11] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted partitioning approach to high-speed and fast-update openflow classification," in *IEEE ICNP*, 2016.

[12] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Computing Surveys (CSUR)*, vol. 37, no. 3, pp. 238–275, 2005.

[13] R. Deal, *Cisco router firewall security*. Cisco Press, 2004.

[14] J. Daly and E. Torng, "Bytecuts: Fast packet classification by interior bit extraction," in *IEEE INFOCOM*, 2018, pp. 2654–2662.

[15] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Hot Interconnects VII*, vol. 40, 1999.

[16] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *IEEE INFOCOM*, 2009.

[17] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "Efficuts: optimizing packet classification for memory and throughput," in *ACM SIGCOMM CCR*, 2010.

[18] W. Li, X. Li, H. Li, and G. Xie, "Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification," in *IEEE INFOCOM*, 2018, pp. 2645–2653.

[19] K. Kogan, S. I. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "Exploiting order independence for scalable and expressive packet classification," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 1251–1264, April 2016.

[20] J. Daly and E. Torng, "Tuplemerge: Building online packet classifiers by omitting bits," in *IEEE ICCCN*, 2017.

[21] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer," in *Hotnets*, 2009.

[22] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended bloom filter: An aid to network processing," in *SIGCOMM '05*, 2005.

[23] D. Cerović, V. Del Piccolo, A. Amamou, K. Haddadou, and G. Pujolle, "Fast packet processing: A survey," *IEEE Communications Surveys & Tutorials*, 2018.

[24] D. R. Barach and E. Dresselhaus, "Vectorized software packet forwarding," Jun. 14 2011, uS Patent 7,961,636.

[25] "Vpp white paper," Tech. Rep., 2017. [Online]. Available: https://fd.io/wp-content/uploads/sites/34/2017/07/FDioVPPwhitepaperJuly2017.pdf

[26] L. Linguaglossa, D. Rossi, D. Barach, D. Marjon, and P. Pfiester, "High-speed software data plane via vectorized packet processing," *IEEE Communication Magazine*, 2018.

[27] Data plane development kit. [Online]. Available: https://dpdk.org

[28] T. L. Foundation. (2016) Open vswitch project. [Online]. Available: https://www.openvswitch.org

[29] R. Giller, "Open vswitch with dpdk, overview," Tech. Rep., September 2016. [Online]. Available: https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview

[30] A. Fischetti and B. Bodireddy, "Ovs-dpdk datapath classifier," Tech. Rep., October 2016. [Online]. Available: https://software.intel.com/en-us/articles/ovs-dpdk-datapath-classifier

[31] Fd.io project. [Online]. Available: https://fd.io

[32] [Online]. Available: https://github.com/TeamRossi/VPP-ACL

[33] [Online]. Available: https://github.com/drjdaly/tuplemerge

[34] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Transactions on Networking (TON)*, vol. 15, no. 3, pp. 499–511, 2007.

[35] J. Matoušek, G. Antichi, A. Lučanskỳ, A. W. Moore, and J. Kořenek, "Classbench-ng: Recasting classbench after a decade of network evolution," in *ACM/IEEE ANCS*, 2017, pp. 204–216.

[36] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *ACM IMC*, 2015.

[37] CAIDA. [Online]. Available: https://www.caida.org

[38] H. Liu, "Efficient mapping of range classifier into ternary-cam," in *High Performance Interconnects*. IEEE, 2002.