Tuple Space Assisted Packet Classification With High Performance on Both Search and Update

Wenjun Li, *Member, IEEE*, Tong Yang, Ori Rottenstreich, Xianfeng Li, Gaogang Xie, Hui Li, Balajee Vamanan, Dagang Li, and Huiping Lin

***Abstract*— Software switches are being deployed in SDN to enable a wide spectrum of non-traditional applications. The popular Open vSwitch uses a variant of Tuple Space Search (TSS) for packet classifications. Although it has good performance on rule updates, it is less efficient than decision trees on lookups. In this paper, we propose a two-stage framework consisting of heterogeneous algorithms to adaptively exploit different characteristics of the rule sets at different scales. In the first stage, partial decision trees are constructed from several rule subsets grouped with respect to their *small fields*. This grouping eliminates rule replications at large scales, thereby enabling very efficient pre-cuttings. The second stage handles packet classification at small scales for *non-leaf terminal nodes*, where rule replications within each subspace may lead to inefficient cuttings. A salient fact is that small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. To exploit this favorable property, we employ a TSS-based algorithm for these subsets following tree constructions. Experimental results show that our work has comparable update performance to TSS in Open vSwitch, while achieving almost an order-of-magnitude improvement on classification performance over TSS.**

***Index Terms*— Packet classification, SDN, openflow, open vswitch, virtualization.**

Manuscript received April 15, 2019; accepted March 2, 2020. This work was supported in part by the National Keystone Research and Development Program of China under Grant 2017YFB0803204, in part by the PCL Future Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001, in part by the NSFC under Grant 61671001 and Grant 61725206, in part by the Research and Development Key Program of Guangdong under Grant 2019B010137001, and in part by the Shenzhen Research Program under Grant KQJSCX20180323174744219, Grant JCYJ20190808155607340, Grant JSGG20170824095858416, and Grant JCYJ20170306092030521. This article was presented in part at the IEEE INFOCOM, HI, USA, April 19, 2018. *(Corresponding author: Hui Li.)*

Wenjun Li is with the School of Electronic and Computer Engineering, Peking University, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: wenjunli@pku.edu.cn).

Tong Yang and Huiping Lin are with the Department of Computer Science and Technology, Peking University, Beijing 100871, China (e-mail: yangtongemail@gmail.com; phoenixrain@pku.edu.cn).

Ori Rottenstreich is with the Department of Computer Science, Technion–Israel Institute of Technology, Haifa 32000, Israel, and also with the Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 32000, Israel (e-mail: or@technion.ac.il).

Xianfeng Li is with the International Institute of Next Generation Internet, Macau University of Science and Technology, Taipa, Macau, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xifli@must.edu.mo).

Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xie@cnic.cn).

Hui Li and Dagang Li are with Shenzhen Graduate School, Peking University, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: lih64@pku.edu.cn; dgli@pku.edu.cn).

Balajee Vamanan is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA (e-mail: bvamanan@uic.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2020.2986935

I. INTRODUCTION

SOFTWARE virtual switches are becoming an important part of virtualized network infrastructures. Backed by SDN, virtual switches enable many non-traditional network functionalities like flexible resource partitioning and real-time migration. Despite their advantages on flexibility and low-cost, software switches have a performance concern. The prominent Open vSwitch enforces forwarding policies with OpenFlow [2] table lookups, which is essentially a multi-field packet classification problem [3], [4]. As an extensively studied bottleneck, packet classification in physical switches still relies on expensive TCAMs because algorithmic solutions implemented in software can hardly satisfy wire-speed forwarding in traditional network infrastructures [5]–[14]. With the advent of SDN and NFV, efficient algorithmic solutions using commodity memories such as DRAM/SRAM are becoming attractive again.

The first step towards meeting this revitalized demand is an understanding of the past research. Among existing algorithmic packet classification research, decision tree [15]–[25] and Tuple Space Search (TSS) [26]–[28] are two major approaches. In decision tree-based schemes, the geometric view of the packet classification problem is taken and a decision tree is built. They work by recursively partitioning the searching space into smaller subspaces until less than a predefined number of rules are contained by each subspace. In case a rule spans multiple subspaces, the problem of rule replication happens and a rule copy is needed for each overlapped subspace. This rule replication problem becomes especially serious during the cutting operations at small scales, where small rules across narrow spaces are to be separated from their overlapped large rules. Thus, decision tree-based schemes achieve fast lookup speed on packet classification, but cannot support fast updates due to the notorious rule replication problem.

Unlike traditional packet classification, OpenFlow has a much higher demand for updates, which further exacerbates the problem and makes decision tree algorithms inapplicable in this context [28], [29]. In contrast, TSS partitions rules into a set of hash tables (i.e., tuple space) with respect to their prefix length. Thus, rule replication never happens in TSS-based schemes, thereby enabling an average of one

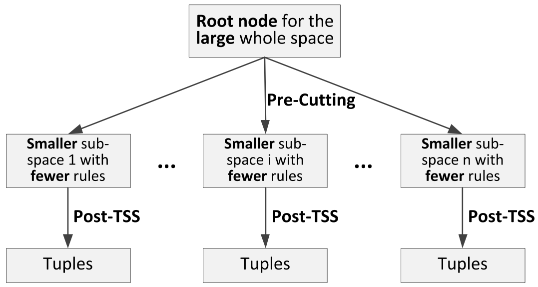


Fig. 1. The initial framework of CutTSS.

memory access for each rule update. As a result, the popular Open vSwitch implements a variant of TSS for its flow table lookups [28]. The primary reason is its good support for fast incremental rule updates, which is an important metric for SDN switches. Despite their advantages on fast updates, TSS-based schemes have a performance concern. For each incoming packet, TSS requires searching on every tuple, because the final matching is the one with the highest priority of matched rules from all tuples. This problem is especially serious for working on large space due to serious tuple expansion problem.

To achieve fast lookup and update at the same time, we propose CutTSS as shown in Figure 1, which fosters the strengths and circumvents the weaknesses of decision tree and TSS-based schemes. To the best of our knowledge, this is the first solution that can keep the advantages of these two schemes: fast lookup and fast update. First, we adopt cutting techniques to build decision trees at large scales, so that each packet can shrink its searching space in a few steps for fast lookups. Second, to improve the update performance, we introduce TSS-based schemes to assist decision tree construction at small scales. Overall, CutTSS exploits the strengths of both decision tree and TSS to circumvent their respective weaknesses.

To refine the framework of CutTSS, a two-stage framework consisting of heterogeneous algorithms is proposed. During the first stage, partial decision trees are constructed from several subsets grouped in their respective *small fields* (i.e., long prefix or narrow range), and leave some *non-leaf terminal nodes* (i.e., terminal nodes after pre-cuttings which contain rules more than the predefined number of rules for leaf nodes, as illustrated in Figure 6) for more efficient handling by TSS-based schemes. This grouping eliminates rule overlapping at large scales, thereby enabling very efficient pre-cuttings without any rule replications. The second stage handles packet classification at small scales for rules in *non-leaf terminal nodes*, where overlapping of rules within each subspace may become common that will lead to inefficient cuttings due to rule replications. Fortunately, a small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. Based on this property, we employ a TSS-based algorithm called PSTSS [28] for rules in these subsets to facilitate tree constructions. Therefore, by exploiting the benefits of decision tree and TSS techniques adaptively, CutTSS not only offers fast updates and linear memory, but also pushes the performance of algorithmic packet classification on par to hardware-based solutions. The main contributions of this paper include the following aspects:

- A scalable rule set partitioning algorithm based on the observation that most rules have at least one *small field* spanning across a narrow space, so the rule set can be efficiently partitioned into a few non-overlapping subsets.
- A set of novel cutting algorithms that exploit the global characteristics of the partitioned subset of rules, so that the rules can be partitioned into smaller subsets without rule replications.
- A two-stage framework combining decision tree and TSS techniques, which can adaptively exploit different characteristics of the rule sets at different scales.

We evaluate our algorithm using ClassBench [30], and the results show that CutTSS is able to produce a very small number of shorter trees with linear memory consumption even for rule sets up to 100k entries. Compared to the TSS algorithm in Open vSwitch, CutTSS achieves similar update performance, but outperforms TSS significantly on classification performance, achieving almost an order of magnitude improvement on average. Our implementation of CutTSS is publicly available on our website (http://www.wenjunli.com/CutTSS).

The rest of the paper is organized as follows. In Section II, we first briefly summarize the related work. After that, we make a set of observations and present the technical details of CutTSS in Section III. Section IV provides experimental results. Finally, conclusions are drawn in Section V.

II. Background and Related Work

In this section, we first review the background and some research efforts about the packet classification problem. After that, we briefly describe two major threads of algorithmic approaches: decision tree-based and tuple space-based packet classification. Finally, we give some summaries.

A. The Packet Classification Problem

The purpose of packet classification is to enable differentiated packet treatment according to a predefined packet classifier. A packet classifier is a set of rules, with each rule R consisting of a tuple of F field values (exact value, prefix or range) and an action (e.g., drop or permit) to be taken in case of a match. The rules in the classifier are often prioritized to resolve potential multiple match scenarios. Packet classification has been well studied for two decades, but most of them focused on high-speed lookups, with very little consideration on the performance of rule updates. However, unlike traditional packet classification, OpenFlow has a much higher demand for updates, making most of traditional algorithms inapplicable in the context of SDN. An example OpenFlow classifier is shown in Table I.

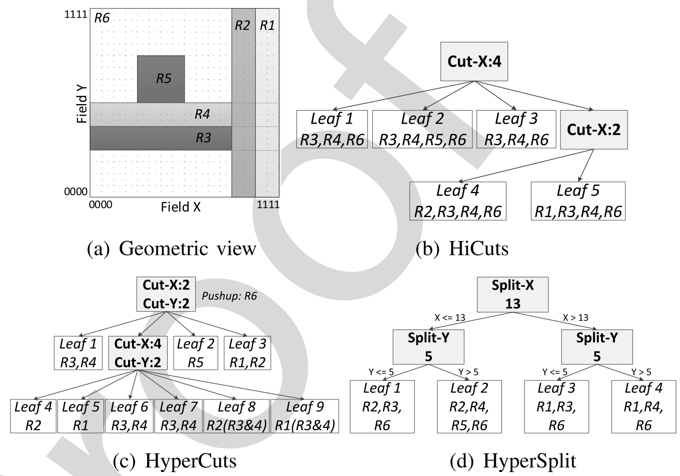
Packet classification is a hard problem with high complexity. From a geometric point of view, packet classification can be treated as a point location problem, which has been proved that the best bounds for locating a point are either $\Theta(log\ N)$ time with $\Theta(N^F)$ space, or $\Theta((logN)^{F-1})$ time with $\Theta(N)$ space for N non-overlapping hyper-rectangles in F-dimensional space [31]. Therefore, the worst-case mathematical complexity

TABLE I
AN EXAMPLE OPENFLOW 1.0 CLASSIFIER

Rule id	Ingress port	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	TCP/UDP Src Port	TCP/UDP Dst Port	*Action*
R_1	3	*	*	2048	*	*	206.159.213.125/32	101.152.182.8/30	0x06f/0xff	0	1024 : 65535	*	$action_1$
R_2	3	*	*	2048	*	*	15.25.70.8/30	*	*	0	*	0:1599	$action_2$
R_3	5	*	*	2048	*	*	*	18.152.125.32/30	0x11/0xff	1	1024 : 65535	1024 : 65535	$action_3$
R_4	5	*	*	2048	*	*	206.159.213.125/32	*	0x06/0xff	1	*	80	$action_4$
R_5	*	*	*	*	*	*	*	*	*	*	*	*	$action_5$

TABLE II
AN EXAMPLE OF 2-TUPLE CLASSIFIER

Rule id	Priority	Field X	Field Y	*Action*
R_1	6	111*	*	$action_1$
R_2	5	110*	*	$action_2$
R_3	4	*	010*	$action_3$
R_4	3	*	011*	$action_4$
R_5	2	01**	10**	$action_5$
R_6	1	*	*	$action_6$



(a) Geometric view (b) HiCuts (c) HyperCuts (d) HyperSplit

Fig. 2. Review on related decision trees ($binth = 4$).

of algorithmic packet classification is extremely high, which makes it impractical to achieve a wire-speed requirement within the capabilities of current memory technology. But fortunately, packet classification rules in real-life applications have some inherent characteristics that can be exploited to reduce the complexity. These inherent characteristics provide a good substrate for the exploration of practical algorithmic solutions [1], [15]–[18], [20], [21], [23], [26], [32]–[40]. Among them, decision tree and Tuple Space Search (TSS) are two major approaches. Next, we briefly summarize the related work on these two techniques. For the convenience of description, we use a small example of 2-tuple rule set shown in Table II for subsequent discussions. Figure 2(a) shows the geometric representation of the example rules given in Table II.

B. Decision Tree-Based Packet Classification

In decision tree-based schemes, the geometric view of the packet classification problem is taken and a decision tree is built. The root node covers the whole searching space containing all rules. They work by recursively partitioning the searching space into smaller subspaces until less than a predefined number of rules are contained by each subspace. In case a rule spans multiple subspaces, the undesirable rule replication happens (e.g., *R3*, *R4* and *R6* in Figure 2(b)). When a packet arrives, the decision tree is traversed to find a matching rule at a leaf node. According to the partitioning method on searching space, current decision trees can be categorized into two major approaches: *equal-sized cutting* and *equal-densed cutting (i.e., splitting)*.

1) Classical Decision Tree Schemes: Cutting based schemes, such as HiCuts [15] and HyperCuts [16], separate the searching space into many equal-sized subspaces using local optimizations. HiCuts cuts the searching space into many equal-sized subspaces recursively until the rules covered by each subspace is less than the pre-defined bucket size called *binth*. To reduce memory consumption, HiCuts uses some heuristics to select the cutting dimension and decides how many subspaces should be cut using a space optimization function. Figure 2(b) shows the decision tree generated by HiCuts, where the *Field X* is cut into four equal-sized subspaces (i.e., [0,3], [4,7], [8,11], [12,15]), and is further cut into two equal-sized subspaces (i.e., [12,13], [14,15]) to finish the decision tree construction. HyperCuts can be considered as an improved version of HiCuts, which is more flexible in that it allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. Besides, several optimization techniques are adopted in HyperCuts, such as *node merging*, *rule overlap*, *region compaction* and *pushing common rule subsets upwards*. But both HiCuts and HyperCuts have the same rule replication problem for rules spanning multiple subspaces, especially for large rule tables. Figure 2(c) shows the decision tree generated by HyperCuts.

In order to reduce the rule replications suffered from equal-sized cuttings, schemes based on splitting divide the searching space into unequal-sized subspaces containing a nearly equal number of rules. HyperSplit [17], a well-known splitting-based decision tree scheme, splits the searching space into two unequal-sized subspaces containing a nearly equal number of rules. Due to its simple binary separation in subspaces, the worst-case search performance of HyperSplit is explicit. However, even with the optimized binary space splitting, the memory consumption of HyperSplit still grows exponentially as the number of rules increases. Figure 2(d) shows the decision tree generated by HyperSplit, we can see that in each internal tree node, HyperSplit splits the selected field into two unequal-sized subspaces, with each subspace covering rules as balanced as possible.

2) Recent Decision Tree Schemes: EffiCuts [18], a well-known cutting based scheme, observed that real-life rules exhibit several inherent characteristics, and a good rule set partitioning can reduce rule replications dramatically. Thus, instead of building a single decision tree for all rules, EffiCuts separates rules into several subsets with each subset creating its own decision tree independently using a variant of HyperCuts. With all F fields considered, up to 2^F decision trees can be generated for F-tuple classifiers, resulting in a lot of overall memory accesses. In contrast, HybridCuts [20] separates rules based on a single field rather than all F fields in EffiCuts, thus, HybridCuts achieves a significant reduction in the number of subsets (i.e., from 2^F to $F+1$), which in turn reduces the overall memory accesses. However, due to the employment of HyperCuts, the worst-case search performance of HybridCuts is unbounded. Worse still, with the increase of the number of rule fields and the size of classifiers, the performance of HybridCuts drop dramatically due to the deteriorating rule replications. Instead of using the most significant bits for cuts, ByteCuts [23] introduces a new cutting scheme that uses any range of bits to build decision trees. However, ByteCuts can achieve high-speed construction of trees but not fast updates of rules.

In order to reduce rule replications suffered from splitting, ParaSplit [19] proposes a rule set partitioning algorithm to reduce rule set complexity, which significantly reduces the overall memory consumption in HyperSplit. However, ParaSplit employs a complex heuristic for rule set partitioning, which may require tens of thousands of iterations to reach an optimal partitioning. To achieve better scalability for different rule sets, SmartSplit [21] separates rules into at most four subsets to build balanced trees dynamically, achieving high-speed classification by leveraging the logarithmic search time of balanced search trees. By partitioning rules into several sortable subsets and building a MITree for each subset, PartitionSort [22] achieves logarithmic classification and update time for each subset simultaneously. Due to the stringent constraints on partitioning, PartitionSort requires much more trees than SmartSplit, resulting in slower classification. Instead of using a single cutting or splitting technique to build trees, CutSplit [1], the preliminary version of the proposed CutTSS, introduces a practical framework that can exploit the benefits of cutting and splitting techniques adaptively. However, due to the boring rule replications in its post-splitting stage, CutSplit can only achieve incremental updates by rebuilding sub-trees, consuming up to a few milliseconds in some cases, far more behind the wire-speed requirement of incremental updates.

C. Tuple Space-Based Packet Classification

In tuple space-based schemes, rules are partitioned into a set of hash tables (i.e., tuple space) based on easily computed rule characteristics. Thus, rules can be inserted and deleted from hash tables in amortized one memory access, resulting in faster updates. When a packet arrives, these partitioned hash tables are individually searched to find the best matching.

1) Classical Tuple Space Schemes: Tuple Space Search (TSS) [26], the basic tuple space-based packet classification, decomposes a classification query into a set of exact match queries in hash tables. TSS partitions rules into different hash tables based on a set of pre-computed tuples. Each tuple can be defined by concatenating the actual bits used in each field in order, so that a hash key can be created to map the rules of that tuple into its corresponding hash table. During classification or updates, those same bits are extracted from the packet or rule as a hash key for searches. For example, rules *R1* and *R2* shown in Table II should be placed in the same tuple space, because both of them use three and zero of the bits in their respective two fields. Thus, TSS builds four tuple spaces as shown in Table III for rules given in Table II. As an improvement, the Pruned Tuple Space Search (PTSS) algorithm [26] reduces the scope of the exhaustive search by performing a search on individual rule fields to find a subset of candidate tuple spaces. However, both TSS and PTSS have low classification speed, because the number of tuple space is large and each tuple space must be searched for every packet. This problem becomes more serious for classifiers with an increased number of fields such as OpenFlow classifiers.

TABLE III
TSS BUILDS 4 TUPLES FOR RULES GIVEN IN TABLE II

Tuple	*Rule id*	Rule Priority	Tuple Priority	Field X	Field Y	*Action*
(3, 0)	R_1	6	6	111*	*	$action_1$
	R_2	5		110*	*	$action_2$
(0, 3)	R_3	4	4	*	010*	$action_3$
	R_4	3		*	011*	$action_4$
(2, 2)	R_5	2	2	01**	10**	$action_5$
(0, 0)	R_6	1	1	*	*	$action_6$

2) Recent Tuple Space Schemes: TupleMerge [27], a recently proposed tuple space scheme, improves upon TSS by relaxing the restrictions on which rules may be placed in the same tuple space. By merging tuple spaces that contain rules with similar characteristics together, TupleMerge can reduce the number of candidate tuple spaces and thus the overall classification time. However, with more tuple spaces merged, its performance may be affected due to hash collisions. Priority Sorting Tuple Space Search (PSTSS) [28], which is used in Open vSwitch, improves the performance of TSS by sorting tuple spaces based on a pre-computed priority of each tuple space (i.e., *Tuple Priority* column in Table III). By searching tuple spaces in the descending order of priority, the search can terminate as soon as a match is found because it has the highest priority among all possible matched rules. Although PSTSS can improve average performance compared to TSS, its worst-case performance is still the same as TSS.

D. Summary of Prior Art

Clearly, decision tree-based packet classification has been actively investigated for two decades. But as far as we know, none of them can make an excellent trade-off among all key metrics. In particular, most of them can achieve high-speed packet classification but not fast updates, which seriously limit their scalability in the era of SDN. In contrast, tuple space-based schemes have been the *de-facto* choice in software switches, because they support fast updates with only linear memory consumption. However, these schemes still suffer from low classification performance especially for large classifiers, falling short of the needs of high-speed requirements in fast-growing networks.

III. CutTSS: Enjoying Both Worlds of Efficient Classication and Rule Update

In this section, we first introduce ideas behind the design of CutTSS. Then, we propose a scalable partitioning algorithm based on experimental observations, which can eliminate rule overlapping at large scales. To exploit these characteristics of partitioned subsets, a set of novel cuttings are designed to build partial trees without any rule replications in the first stage. After that, a two-stage framework consisting of heterogeneous algorithms is proposed to build decision trees for partitioned subsets. Finally, we give more insights on the effectiveness of CutTSS from both theoretical and experimental aspects.

A. Ideas & Framework

According to the above review and analyses given in CutSplit [1], we know that cutting techniques can separate searching space into smaller subspaces quickly for faster classification, but it suffers from serious rule replications. In contrast, TSS can completely avoid rule replications and support fast incremental updates, but it has longer classification time due to tuple expansions, especially for rule sets at large scales. Therefore, to foster the strengths and circumvent the weaknesses of decision tree and TSS schemes, the idea directly perceived is to combine the following two strategies:

- **Pre-Cutting at large scales**: Cutting-based partitioning on the searching space at large scales, which can reach any subspaces at small scales with very few steps.
- **Post-TSS at small scales**: TSS-based searching on subspaces at small scales, which can avoid inefficient cutting of decision trees.

Figure 1 shows the initial framework of the following CutTSS. However, in order to design scalable algorithms to meet the above design goals, an effective combination of these two ideas still faces several difficulties and challenges:

- **Low memory access**: Although partitioning can reduce rule overlapping significantly, it will increase the overall memory accesses. Thus, how to generate rule subsets as few as possible?
- **Low memory consumption**: Since cutting on rules overlapped at different scales will lead to serious rule replications, how to avoid rule replications during the first cutting stage?
- **Low update time**: Since many rules are overlapped and concentrated in some subspaces at small scales, which will lead to inefficient cuttings due to rule replications. How to avoid rule replications in these subspaces at small scales?

The answers to these questions are the key ideas in this paper. Our solution can be summarized in the following three steps:

- **Step 1: Partitioning based on very few small fields**: In order to eliminate rule overlapping at large scales and reduce the number of partitioned subsets, we separate rules into subsets based on their characteristics shared in very few *small fields*.
- **Step 2: Pre-cuttings by exploiting the global characteristics of the partitioned subsets**: After partitioning the rule set, we get a set of favorable fields for each partitioned subset, where a set of simpler cutting algorithms without prior optimizations can be applied for space partitioning.
- **Step 3: TSS-assisted cutting trees for fast updates**: Thanks to the clever partitioning and pre-cuttings without any rule replications, most of the rules can be separated into leaf nodes for the linear search, except for a small fraction of concentrated rules at small scales. For these *non-leaf terminal nodes*, we employ a TSS-based algorithm to facilitate the rest of tree constructions.

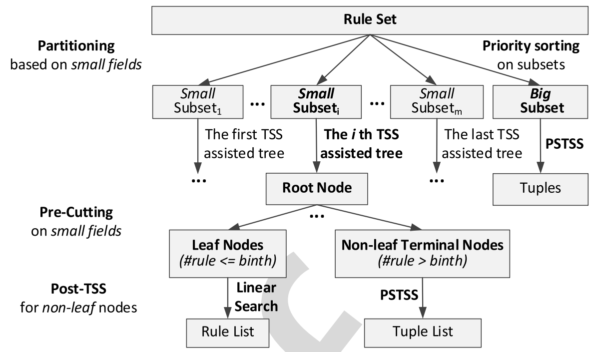


Fig. 3. The refined framework of CutTSS.

Based on these ideas, we give the refined framework of the proposed CutTSS shown in Figure 3. Overall, a complete packet classification framework with two heterogeneous stages exploiting favorable properties in their respective space scales is in place. Next, we give more details about CutTSS from the following three aspects: rule set partitioning, decision tree construction and decision tree operation.

B. Rule Set Partitioning Based on Small Fields

Classification rules in real-life applications have structural redundancies and several inherent characteristics that can be exploited to reduce the complexity. Thus, we use the publicly available ClassBench and OpenFlow-like rule tables for study to make observations on common characteristics of rule sets. It should be noted that the two OpenFlow-like rule tables are supported by the authors of ParaSplit [19], which were generated based on 216 real-life rules from enterprise customers. We first give a few definitions, then we present the key observations related to the following discussions on rule set partitioning.

1) Definitions: Given an N-field rule $R = (F_1, \ldots F_i, \ldots, F_N)$ and a threshold value vector $T = (T_1, \ldots T_i, \ldots, T_N)$, where $i \in \{1, 2, \ldots, N\}$, we first give some definitions for field F_i as follows:

- F_i is a ***big field***: the range length of field $F_i > T_i$;
- F_i is a ***small field***: the range length of field $F_i \leq T_i$.

Based on the above definitions for field F_i, we further give some definitions for R as follows:

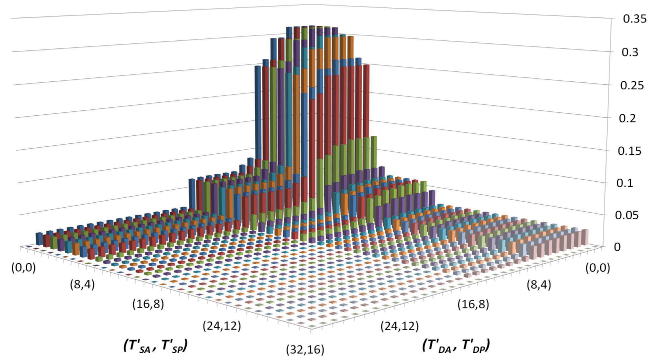


Fig. 4. The ratio of *big rules* for seed-ipc rule set.

TABLE IV
STATISTICAL RESULTS FOR 5-TUPLE & OPENFLOW-LIKE RULES

Rule Set(#rules)	Threshold Vector	*#Big Rules*	*#Small-k* Rules				
			k=1	*k=2*	*k=3*	*k=4*	*k>4*
seed-acl(752)		3	749	739	425	0	0
seed-fw(269)	$T = (2^{16}, 2^{16}, 2^{8}, 2^{8})$	4	265	218	17	2	0
seed-ipc(1550)		2	1548	1472	789	5	0
openflow-1(716)	$T = (2^{t_1}, 2^{t_2}...2^{t_i}...)$,	0	716	708	655	426	0
openflow-2(864)	t_i = half width of F_i	0	864	852	761	429	0

- R is a **big rule**: $\forall i \in \{1, 2, \ldots, N\}$, F_i is a *big field*;
- R is a ***k*-small rule**: R contains at least k *small fields*.

For a classical 5-tuple rule, since the protocol field is restricted to a small set of values (e.g., tcp, udp), we just consider the other four fields in this paper. Then the threshold value vector T for 5-tuple rules is simplified to a four-dimensional vector $T = (T_{SA}, T_{DA}, T_{SP}, T_{DP})$. For the sake of convenience in writing, we use a logarithmic vector *T'* to represent the threshold value vector T equivalently. For example, if we set threshold value vector $T = (2^{16}, 2^{16}, 2^{8}, 2^{8})$, then index logarithmic *T'* = (*16, 16, 8, 8*).

2) Observations: Based on the above definitions, we make some statistical experiments for several rule sets from ClassBench. There are three types of rule sets: ACL (Access Control List), FW (Firewall) and IPC (IP Chains). Figure 4 shows the ratio of *big rules* under different thresholds for seed IPC rule set. Due to length limitation, more results for ACL and FW rule sets are publicly available on our website (http://www.wenjunli.com/CutTSS). It is clear that the ratio of *big rules* is very low even under very demanding thresholds. For example, assume *T'* = (*16, 16, 8, 8*), the ratio of *big rules* for three types of rule set are all less than 0.01, that is to say, less than 1% rules are *big rules* under $T = (2^{16}, 2^{16}, 2^{8}, 2^{8})$. This indicates that the vast majority of the rules have at least one *small field* satisfying a threshold T. Essentially, this observation is consistent with previous observations. Otherwise, a large number of *big rules* may cause serious space overlapping which is contrary to previous observations that the number of rules or address prefixes matching a given packet is typically five or less.

Table IV shows statistical results for 5-tuple rule sets and OpenFlow-like rule tables. Clearly, this observation is still effective for OpenFlow-like rule tables: the vast majority of rules have at least one *small field*.

3) Rule Set Partitioning: Based on the above observations, we propose a partitioning algorithm to separate rules into several subsets. The purpose of our partitioning is to obtain a few subsets without duplicates among each other. For each subset, all contained rules should share a common characteristic for a set of rule fields: *small field*. Next, we introduce a simple heuristic as follows:

–Step1: Removal of big rules. Since the number of *big rules* is negligible, we can simply apply PSTSS for these rules.

–Step2: Selection of partitioning rules. We first count the distinct values for each field, then select a few fields with a large number of distinct values. The selection makes sure that for the vast majority of rules, there is at least one selected field with a small value. The rest rules without any small value in the selected fields will be treated as *big rules*.

–Step3: Fields-wise partitioning. Assume that M fields have been selected for F-tuple rule sets. We categorize rules based on field length (i.e., *big* or *small*) in all selected fields, leading to at most 2^M-1 subsets. This partitioning is different from EffiCuts from two perspectives: fewer fields and more flexible definition of *small/big field*, which enables much more flexible partitioning to generate fewer subsets.

–Step4: Selective subset merging. For subsets containing very few rules, we can merge these rules into other subsets that have fewer *small fields*. Due to the consideration of its relevance and space limitation, we do not elaborate on this algorithm in this paper. Note that our merging will not lead to rule replication in our decision trees, which is quite different from EffiCuts.

Take rules in Table I as an example, we first move *R5* into a *big rule* subset, then we calculate the number of distinct *small fields* in each field and pick *ip_src* & *ip_dst* as the two most distinct fields. Thus, we can partition the rule set into four subsets: *big_subset* = {*R5*}, $(small_{ip\_src}, small_{ip\_dst}) = \{R1\}$, $(big_{ip\_src}, small_{ip\_dst}) = \{R3\}$ and $(small_{ip\_src}, big_{ip\_dst}) = \{R2, R4\}$. Finally, we can merge $(small_{ip\_src}, small_{ip\_dst})$ with $(big_{ip\_src}, small_{ip\_dst})$ for a new subset $(arbitrary_{ip\_src}, small_{ip\_dst}) = \{R1, R3\}$, where *arbitrary field* contains both *small* and *big field*. Thus, three subsets are generated for the sample rule set: *big_subset* = {*R5*}, $(small_{ip\_src}, big_{ip\_dst}) = \{R2, R4\}$ and $(arbitrary_{ip\_src}, small_{ip\_dst}) = \{R1, R3\}$.

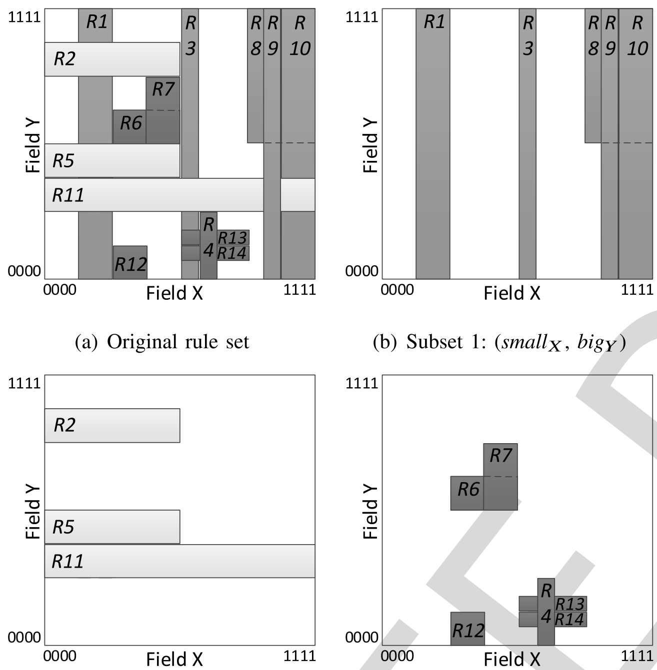
C. Decision Tree Construction: Pre-Cutting & Post-TSS

From Figure 3, we can see that a TSS-assisted tree will be built for each partitioned subset (except for the *big* subset). Thus, we will give more details about the tree building algorithm in CutTSS. For the convenience of description, we will use the rule set shown in Table V as a working example, where no *big rules* are included. Based on the above definitions of rule field, we can label each rule with a field vector as shown in Table V. Based on these field labels, the fourteen rules can be partitioned into three subsets as shown in Figure 5. For each subset, we then build the decision tree through the following two steps: pre-cutting and post-TSS.

1) Pre-Cutting: Fixed Cuttings on Small Fields: The rationale behind the above strategy of rule set partitioning is simple:

TABLE V
A NEW EXAMPLE OF 2-TUPLE CLASSIFIER (T_X = 4, T_Y = 4)

Rule id	Priority	Field X	Field Y	*Action*	*Field Label*
R_1	14	001*	*	$action_1$	<*small, big*>
R_2	13	0***	110*	$action_2$	<*big, small*>
R_3	12	1000	*	$action_3$	<*small, big*>
R_4	11	1001	00**	$action_4$	<*small, small*>
R_5	10	0***	011*	$action_5$	<*big, small*>
R_6	9	01**	100*	$action_6$	<*small, small*>
R_7	8	011*	10**	$action_7$	<*small, small*>
R_8	7	11**	1***	$action_8$	<*small, big*>
R_9	6	1101	*	$action_9$	<*small, big*>
R_{10}	5	111*	*	$action_{10}$	<*small, big*>
R_{11}	4	*	010*	$action_{11}$	<*big, small*>
R_{12}	3	010*	000*	$action_{12}$	<*small, small*>
R_{13}	2	10**	0010	$action_{13}$	<*small, small*>
R_{14}	1	10**	0001	$action_{14}$	<*small, small*>



(a) Original rule set (b) Subset 1: ($small_X$, big_Y)

(c) Subset 2: (big_X, $small_Y$) (d) Subset 3: ($small_X$, $small_Y$)

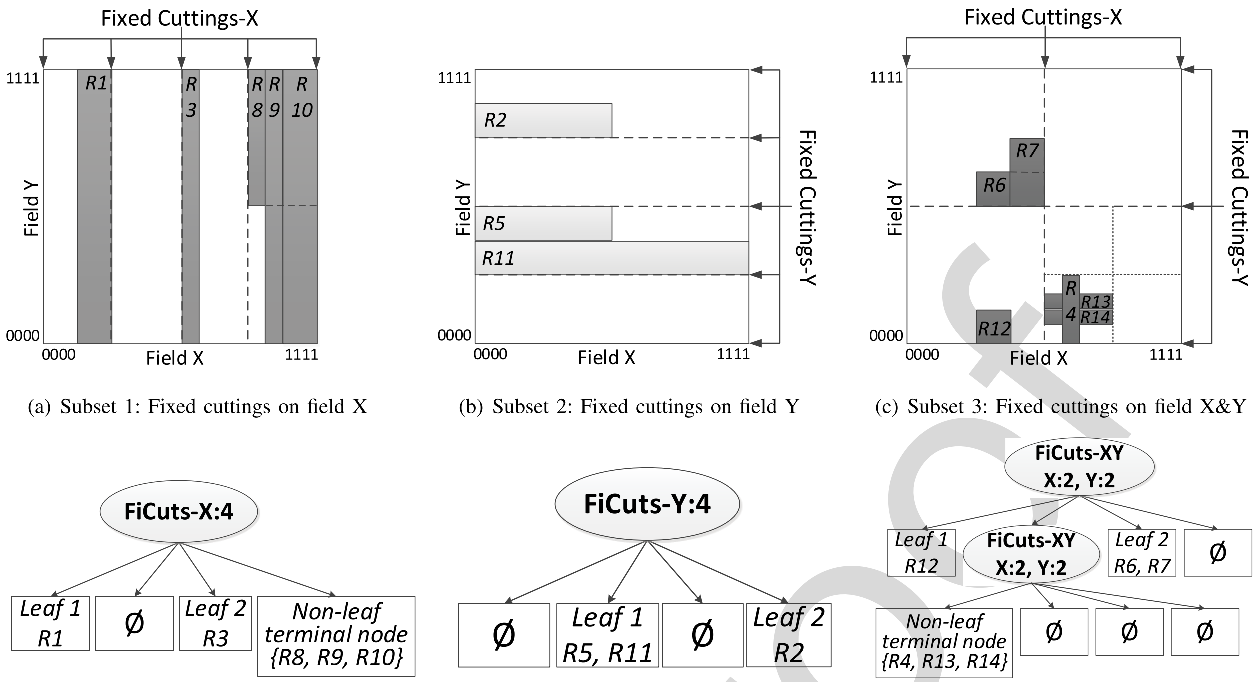
Fig. 5. Ruleset partitioning example (T_X = 4, T_Y = 4).

by grouping rules that are narrow in the same fields, rules that are large in these fields are excluded, and intensive rule replications caused by these excluded rules are eliminated, thereby enabling very efficient cuttings. What is more, this grouping can completely eliminate rule replications at large scales (i.e., larger than *small field*'s threshold) for prefix fields, because each prefix can never be overlapped with two shorter prefixes with the same prefix length. To exploit these favorable characteristics of partitioned subsets, we introduce a simple but effective cutting algorithm called Fixed Cuttings (FiCuts), which will be applied in the first stage partial tree construction.

FiCuts derives from HiCuts and HyperCuts, but with a better global view on the characteristics of the rule set. As shown in Figure 5(b), the rules in the subset are all *small* in *Field X*. This facilitates cuts along the *Field X* without any rule replications at large scales. Since the rules have been grouped into several subsets, with each subset sharing the same *small fields*, FiCuts can utilize this information to exploit efficient cuttings. Compared to HiCuts and HyperCuts, FiCuts has several differences in cutting details as follows: (1) Instead of changing cutting dimensions dynamically, FiCuts conducts cuttings on the subset along a set of fixed dimensions, where the rules are *small* in these fields; (2) Instead of deciding how many subspaces should be cut per step dynamically, the number of cuts per step in CutTSS is a fixed value (i.e., *MAXCUTS*, defined to control the number of empty tree node); (3) None of the prior optimization methods is required in FiCuts, making it simple enough to achieve fast lookups and updates; (4) As FiCuts is only designed for the construction of partial trees in the first stage, its cutting processes will stop not only in the leaf nodes containing rules less than the pre-defined *binth*, but also in the nodes located at small scales (i.e., cutting space smaller than the threshold of *small field*).

Thus, for the subsets containing one or more *small fields*, FiCuts will cut on that single or multiple *small fields* to build partial trees as illustrated in Figure 6. We can see from the partial trees that rule replication can be completely avoided and all rules are located in the bottom nodes of the tree (i.e., leaf nodes or *non-leaf terminal nodes*). Besides, as none of the prior optimization methods is adopted in equal-sized cutting processes, each node in the partial trees can be easily indexed by a string of bits, which can be used as an array key during lookups and updates. For each subset, FiCuts continues its cutting processes until the number of rules is less than the threshold for linear search or the cutting space is smaller than the threshold of *small field*. Take the subsets shown in Figure 6 as an example, FiCuts just works fine for the second subset: it builds the whole tree as illustrated in Figure 6(e), in which all rules are partitioned into leaf nodes. However, Figure 6(d) shows a different scenario, where pure FiCuts does not solve the problem completely, and only a partial tree can be constructed. When FiCuts reaches the rightmost cutting subspace in Figure 6(a), it is no longer effective by continuing cutting along *Field X*, because the cutting space in *Field X* is now smaller than T_X. Therefore, it is necessary to resort to other more effective methods to continue tree constructions at small scales.

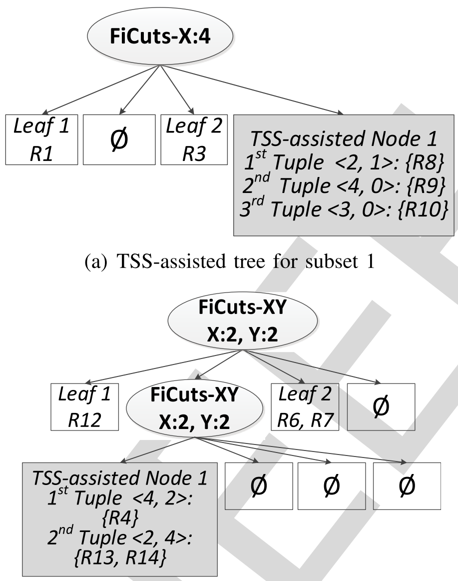
2) Post-TSS: Tuple Space Assisted Cutting Trees: After the first stage of pre-cuttings, two types of terminal nodes will be generated in the built partial trees: *leaf node* (i.e., #rules $\leq$ *binth*) and *non-leaf terminal node* (i.e., #rules $>$ *binth*). As a very limited number of rules are contained in leaf nodes, we can simply conduct a linear search on rules as in traditional decision trees. Thus, the second stage is mainly designed to handle packet classification on *non-leaf terminal node*. It is not difficult to see that the searching space has been separated into much smaller subspaces after pre-cuttings, where each subspace contains much fewer rules compared with the original rule set. On the other hand, a small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. Based on this property, we employ the PSTSS for rules in the *non-leaf terminal nodes* to facilitate tree constructions. Thus, for the two partial trees shown in Figure 6, we can build their complete trees without any rule replications as illustrated in Figure 7.



(a) Subset 1: Fixed cuttings on field X (b) Subset 2: Fixed cuttings on field Y (c) Subset 3: Fixed cuttings on field X&Y

(d) Partial tree for subset 1 (e) Whole tree for subset 2 (f) Partial tree for subset 3

Fig. 6. The first stage partial trees built by FiCuts (*MAXCUTS* = 4, *binth* = 2).



(a) TSS-assisted tree for subset 1

(b) TSS-assisted tree for subset 3

Fig. 7. The complete TSS-assisted decision trees in CutTSS.

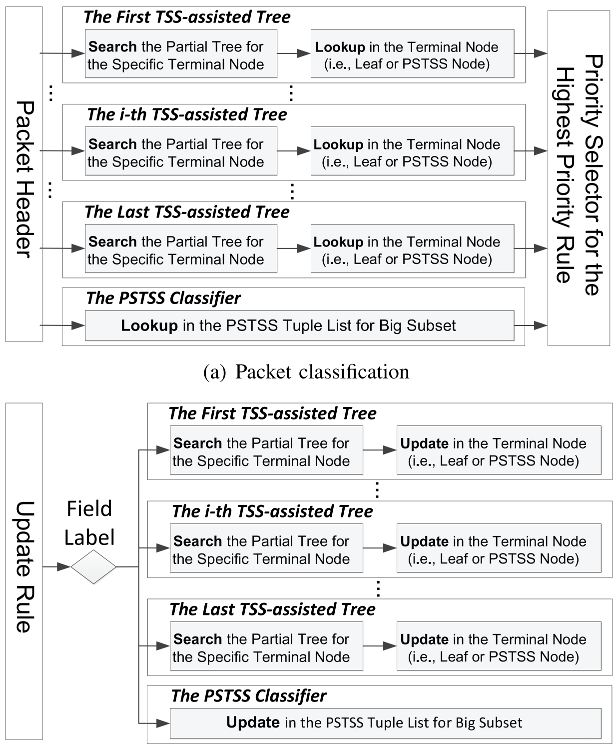
Up to now, three complete decision trees have been built for all rules given in Table V, as shown in Figure 6(e) and Figure 7. Overall, by exploiting the benefits of decision tree and TSS techniques adaptively, CutTSS can build TSS-assisted decision trees without any rule replications, thereby enabling fast updates and linear memory consumption.

3) Refined Optimizations: To further improve the performance, several optimizations have been adopted in our implementation as follows:

–Optimization 1: Priority sorting on partitioned subsets. For each incoming packet, CutTSS requires searching on every partitioned subset, even if a rule has been matched in an early subset. We improve on this by tracking the priority of partitioned subsets as that in PSTSS and PartitionSort algorithms, where the priority of each subset is the maximum priority of all the rules in it. By searching from greatest to least maximum priority on subsets, each lookup can terminate as soon as a rule is matched in an early subset.

–Optimization 2: Dynamic thresholds on terminal leaf nodes. For the terminal nodes after pre-cuttings, we adopt a dynamic threshold to distinguish leaf nodes and non-leaf nodes. The idea of this optimization is derived from the performance comparison for a lookup between linear search and TSS search. For example, the latest version of Open vSwtich (http://www.openvswitch.org) implements the PSTSS based on a variant of cuckoo hash [41], [42], where multiple hash lookups are required for each TSS lookup in Open vSwtich, which is much more complex and time-consuming than a linear search. Assume that each TSS lookup takes N times than a linear rule search, we can set the threshold as $N*M$, where M is the number of tuples in the terminal node.

–Optimization 3: Greedy thresholds on small fields. Essentially, *small field* is a relative concept of space scale. It is not difficult to see that narrower *small fields* may enable more effective pre-cuttings and less tuple spaces in *non-leaf terminal nodes*. However, narrower *small fields* may also lead to more rules in the *big subset* as illustrated in Figure 4, which may in turn increase the number of tuples in the *big subset*. To make a good trade-off, we select the thresholds on *small fields* by running a greedy algorithm during partitioning. The strategy of



(b) Rule update

Fig. 8. The framework of classification and update in CutTSS.

selecting thresholds in our implementation is simple: choose one that achieves the least average memory access.

D. Decision Tree Operation: Classification & Update

In this subsection, we complete the picture of CutTSS from the following aspects: packet classification and rule update.

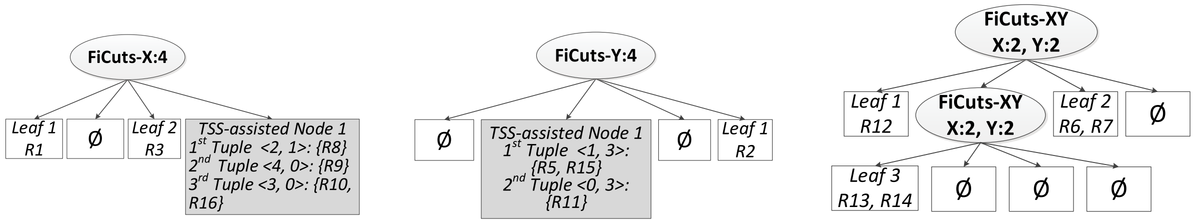
1) Packet Classification: For each incoming packet, CutTSS classifies the packet based on the framework shown in Figure 8(a). For each decision tree, CutTSS conducts classification in two steps: (1) Search the partial tree to find a terminal node; (2) Lookup for the best matching rule from the matched terminal node. Assuming that a 2-field incoming packet is $P_i =$ <1000, 0010>, we next give a working example for the rule set shown in Table V, where three decision trees are built as shown in Figure 6(e) and Figure 7: (1) For the decision tree shown in Figure 6(e), P_i can traverse this tree based on its first two bits in *Field Y* (i.e., 00). Thus, the first child node is found, and no rule is matched in this subset; (2) For the decision tree shown in Figure 7(a), P_i can traverse this tree based on its first two bits in *Field X* (i.e., 10). Thus, the third child node is matched, and R_3 is the best matching rule based on linear search; (3) For the decision tree shown in Figure 7(b), P_i can traverse this tree based on its first bit in *Field Y&X* (i.e., 0&1). Thus, the second child node is matched, and R_{13} is the best matching rule based on PSTSS search. Finally, R_3 with a higher priority will be the best matching rule for P_i.

2) Rule Update: For each updated rule, CutTSS updates the rule based on the framework shown in Figure 8(b). Unlike the above packet classification where all subsets have to be searched, CutTSS can perform each rule update just in a single subset, because the updated or inserted rule can only appear in a specific subset in CutTSS, depending on its field label vector. CutTSS performs rule updates in a tree in two steps: (1) Search the partial tree to find a terminal node; (2) Update (e.g., insert or delete) the rule pointed by the matched terminal node. When searching the partial tree for rule updates, the specific bits in each rule's *small fields* are used as a key for searching. Assuming that there are three update operations as follows: (1) Delete rule $R_4 =$ <1001, 00**>; (2) Insert rule $R_{15} =$ <1***, 010*>; (3) Insert rule $R_{16} =$ <110*, *>, we next give a working example for the rule set shown in Table V. By calculating the field label of R_4 (i.e., <*small*, *small*>), we known that R_4 may only appear in the decision tree shown in Figure 7(b), which is built for the subset shown in Figure 5(d). Then, R_4 can traverse this tree based on its first bit in *Field Y&X* (i.e., 0&1). Thus, the second child node is matched, and then R_4 will be updated in this terminal node. After removing R_4 from the PSTSS classifier, the number of rules in this node is reduced to the threshold of *binth*. Thus, we can replace this *non-leaf terminal node* with a new leaf node as shown in Figure 9(c). Similarly, we can first calculate the field label of R_{15} (i.e., <*big*, *small*>) and R_{16} (i.e., <*small*, *big*>), and then conduct updates as R_4 in the corresponding trees shown in Figure 6(e) and Figure 7(a), as illustrated in Figure 9.

E. Rationale Behind Effectiveness

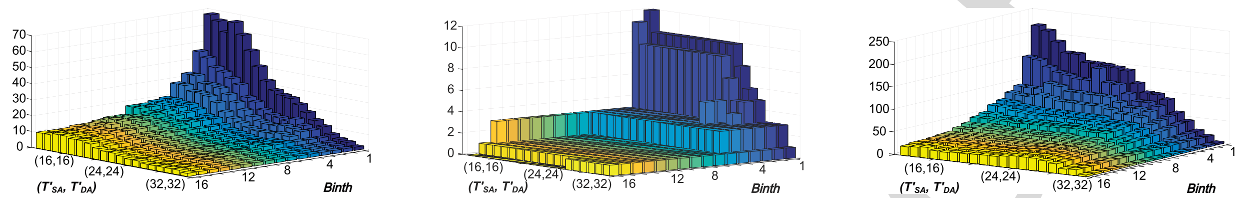
To reveal the rationale behind the effectiveness of CutTSS, we next give more insights from both theoretical and experimental aspects as follows.

1) Theoretical Analysis: Essentially, CutTSS is a two-stage tree framework built from the following two stages: (1) Coarse-grained pre-cutting with low memory consumption; (2) Fine-grained post-TSS with high performance. For the first-stage pre-cuttings in CutTSS, rule replications can be avoided completely, thereby enabling linear memory consumption for the partial trees. For the following tree constructions, CutTSS adopts PSTSS with a linear memory consumption to handle packet classification in *non-leaf terminal nodes*. Thus, for a F-dimensional subset containing N distinct rules, the memory consumption of CutTSS is $\Theta(N)$, which is the best theoretical bound proved in previous work as described in Section II(A). For each incoming packet or updated rule, CutTSS performs packet classification or rule update in two steps: (1) Search the partial tree based on the specific bits in each packet or rule in $\Theta(1)$ time; (2) Perform classification or update in the matched terminal node containging M rules (M≤N). Based on the above Section II(A), we can conclude that the worst-case time complexity of CutTSS is $\Theta((logM)^{F-1})$. Thus, compared to the theoretical worst-case time complexity (i.e., $\Theta((logN)^{F-1})$), CutTSS achieves $\Theta((\log_M N)^{F-1})$ times improvement. We then consider the average worst-case time complexity of CutTSS as follows: Assuming that all rules are evenly distributed, the width and the threshold value of the *small field* are

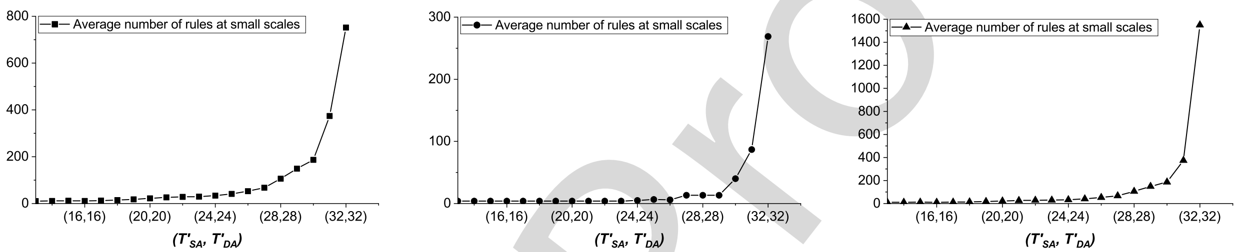


(a) Tree for subset 1: ($small_X$, big_Y) (b) Tree for subset 2: (big_X, $small_Y$) (c) Tree for subset 3: ($small_X$, $small_Y$)

Fig. 9. The new decision trees after three rule updates.



(a) Seed-acl(752): *#non-leaf terminal nodes* (b) Seed-fw(269): *#non-leaf terminal nodes* (c) Seed-ipc(1550): *#non-leaf terminal nodes*



(d) Seed-acl(752): Average number of rules (e) Seed-fw(269): Average number of rules (f) Seed-ipc(1550): Average number of rules

Fig. 10. Rule distribution density at different sized small scales.

2^W and 2^T, we can conclude that the average worst-case time complexity of CutTSS is $\Theta((logM_A)^{F-1})$, where $M_A = N{*}2^{(T-W)}$. Thus, from the perspective of theoretical analysis, the rational behind the effectiveness of CutTSS is essentially to perform the packet classification in a subspace at small scales that contains fewer rules. Although the theoretical bounds tell us that it is infeasible to design a single algorithm that can perform well in all cases, real-life classifiers have some inherent characteristics that can be exploited to reduce the complexity. Next, we give more insights from the aspect of experimental analysis.

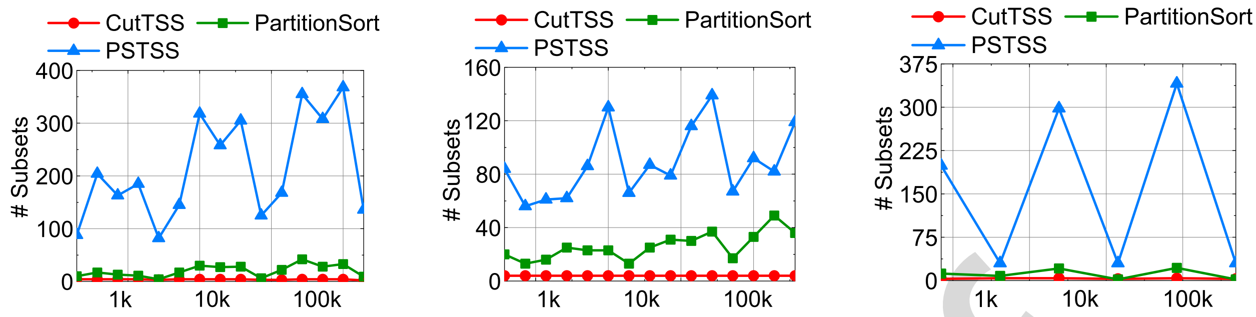
2) Experiential Analysis: We conduct experiential analysis based on the above three seed rule sets, to show more insights on the feature of rule distribution from two aspects: (1) Number of *non-leaf terminal nodes* at small scales; (2) Average number of rules at small scales. Take the subset shown in Figure 5(d) as an example, we can say that six rules are concentrated at three (over 4*4 = 16) distinct subspaces at small scales and the average number of rules is two. Among the three subspaces, only one of them contains rules more than *binth*, which will be handled by PSTSS in the tree. Based on this example, we now give more details about experiential analysis. Figure 10(a), (b) and (c) shows the number of subspaces containing rules more than *binth* at different sized small scales. We can see that although rules are distributed in many subspaces, the vast majority of them contain a small number of rules. In other words, the number of *non-leaf terminal nodes* in CutTSS is much smaller than the number of leaf nodes in the trees, thereby making CutTSS more like a traditional decision tree which can achieve high performance on classification inherently. That's why we call this tree as a TSS-assisted tree in CutTSS. Figure 10 (d), (e) and (f) shows the average number of rules over all subspaces that contain rules. We can see that even under very loose thresholds, the number of rules after the first stage pre-cutting is much smaller than the original rule set size, thereby enabling high performance on both search and update.

IV. Experimental Results

In this section, we present some experimental results of CutTSS. We start with an overview of our experimental methodology. After that, we evaluate our algorithm from the following key aspects: *tree construction*, *packet classification* and *rule update* respectively.

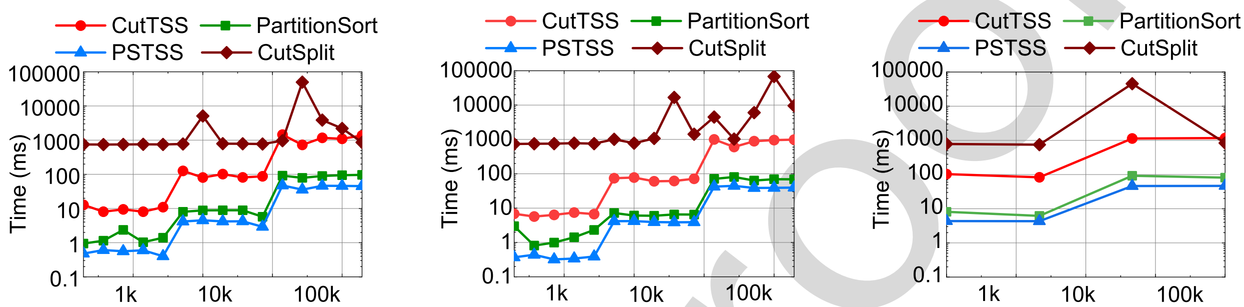
A. Experimental Methodology

We compare CutTSS with three algorithms: PSTSS, CutSplit and PartitionSort. Priority Sorting Tuple Space Search (PSTSS) is the algorithm with the fastest update performance, which is used in Open vSwitch for flow table lookups. CutSplit is the state-of-the-art decision tree with the



(a) ACL (b) FW (c) IPC

Fig. 11. Number of partitioned subsets.



(a) ACL (b) FW (c) IPC

Fig. 12. Construction time.

fastest classification performance. ParitionSort is the state-of-the-art splitting based tree with the best performance trade-off between classification and update. To facilitate fair comparison, we have made some modifications to the open-source code of the other three algorithms, and their performances are essentially not affected by our modification. We are very grateful to the authors of these algorithms, their open-source codes and selfless personal help enable us to make a fair and justifiable comparison. As a response, our implementation of CutTSS is also publicly available on our website (http://www.wenjunli.com/CutTSS).

1) Rule Sets: The rule sets used in our experiments are generated using ClassBench, whose size varies from 1k to 100k. There are three types of rule sets: ACL, FW and IPC. Each rule set is named by its type and size, e.g., FW_1k refers to the firewall rule set with about 1000 rules. For each size, we generate 12 rule sets respectively based on 12 seed parameter files (i.e, 5 ACL, 5 FW and 2 IPC) in ClassBench [30].

2) Simulation Environment: We measure classification time by classifying all packets in trace files generated by ClassBench when it constructs the corresponding rule set. In order to evaluate the actual lookup performance of classification algorithms, we conduct experiments by omitting caching in the fast path and consider only slow path classification for each incoming packet. To evaluate the performance of the incremental update, we measure update time as the time required to conduct one rule insertion or deletion. For each rule set, we shuffle rules randomly to generate a sequence of update operations, where half of the insertions are randomly mixed with half of the deletions.

3) Machine Environment: All experiments are run on a machine with AMD Radeon 5-2400G CPU@3.6GHz and 8G DRAM. The operating system is Ubuntu 16.04. To reduce the CPU jitter error, we take the average results by running ten times for each evaluation circularly.

B. Evaluation on Construction

1) Number of Subsets: Since the number of partitioned subsets in CutSplit is the same as in CutTSS, Figure 11 shows the number of subsets in CutTSS, PSTSS and PartitionSort. We find that CutTSS produces a relatively stable number of subsets regardless of the type and size of rule sets, averaging at 3.7 subsets across all of the rule sets. This favorable property makes CutTSS more suitable for concurrency. In contrast, the number of partitioned subsets in PSTSS and PartitionSort ranges from 2 to 368 with an average of 151.7 and 20.9 subsets respectively.

2) Construction Time: Figure 12 shows the construction time of CutTSS as well as PSTSS, PartitionSort and CutSplit. Clearly, PSTSS is the fastest one among them. In contrast, CutTSS takes a little more time than PSTSS because of its partial tree constructions in the pre-cutting stage. However, even for the rule sets up to 100k, CutTSS can still build decision trees in about one second, much faster than previous decision trees such as EffiCuts and SmartSplit that require almost ten minutes. We can also find that the construction time of CutTSS increases almost linearly with the rule set size, which makes it well suitable for larger classifiers.

3) Memory Consumption: Figure 13 shows the memory consumption of CutTSS as well as PSTSS, PartitionSort

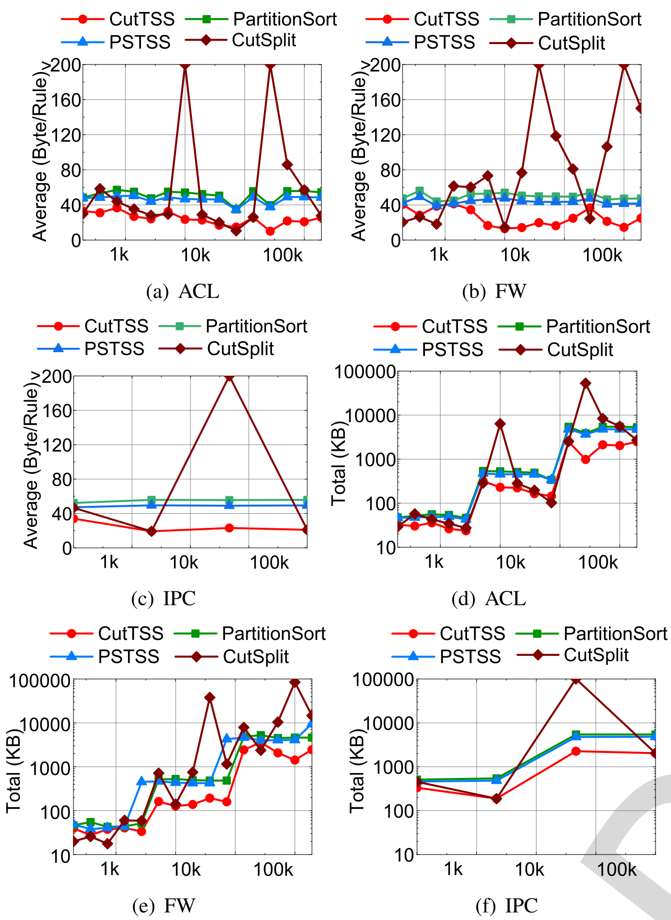


Fig. 13. Memory consumption.

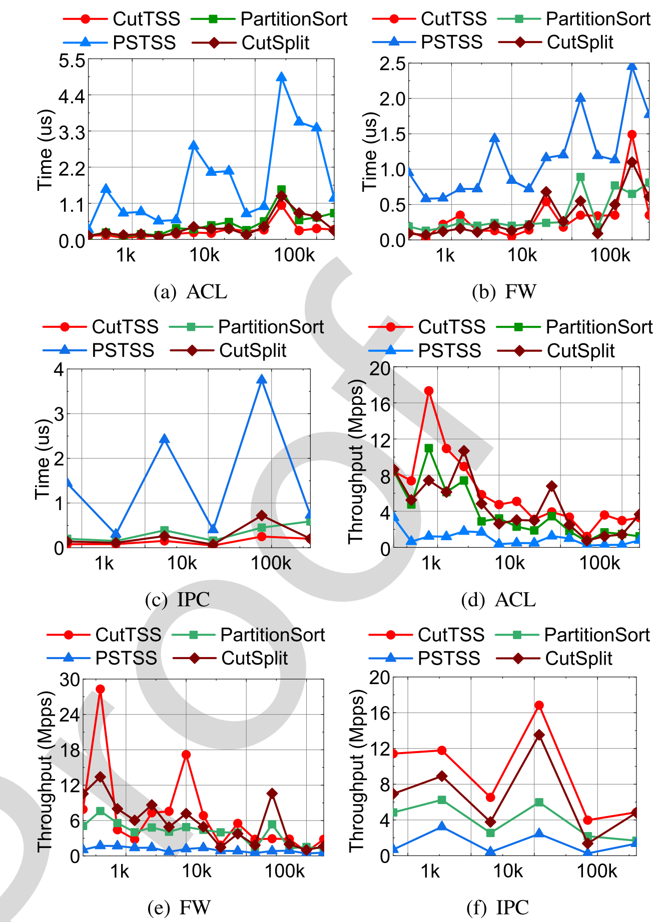


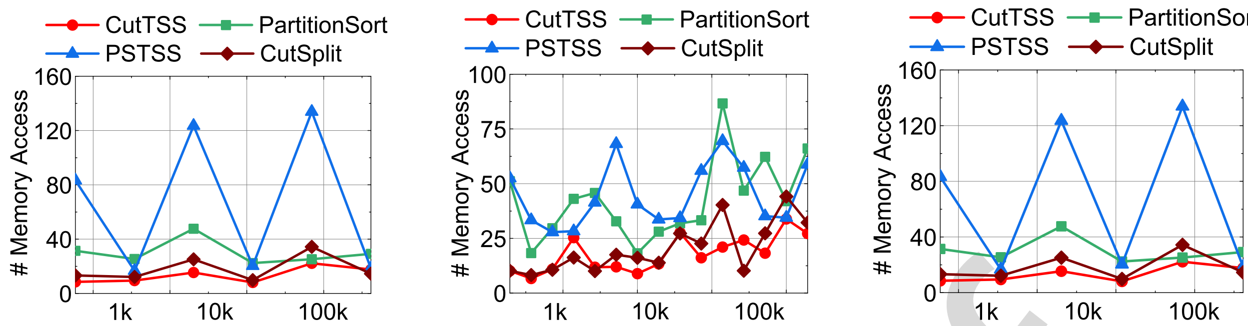
Fig. 14. Classification performance.

and CutSplit. Our experimental results show that our CutTSS requires less space than other algorithms, consuming an average of 25.8 Byte/Rule across all of the rule sets, while it requires 45.4 Byte/Rule, 50.9 Byte/Rule and 243.2 Byte/Rule in PSTSS, PartitionSort and CutSplit respectively. We can also find that, the memory consumption of CutTSS increases almost linearly with the rule set size, which makes it well suitable for larger classifiers.

C. Evaluation on Classification

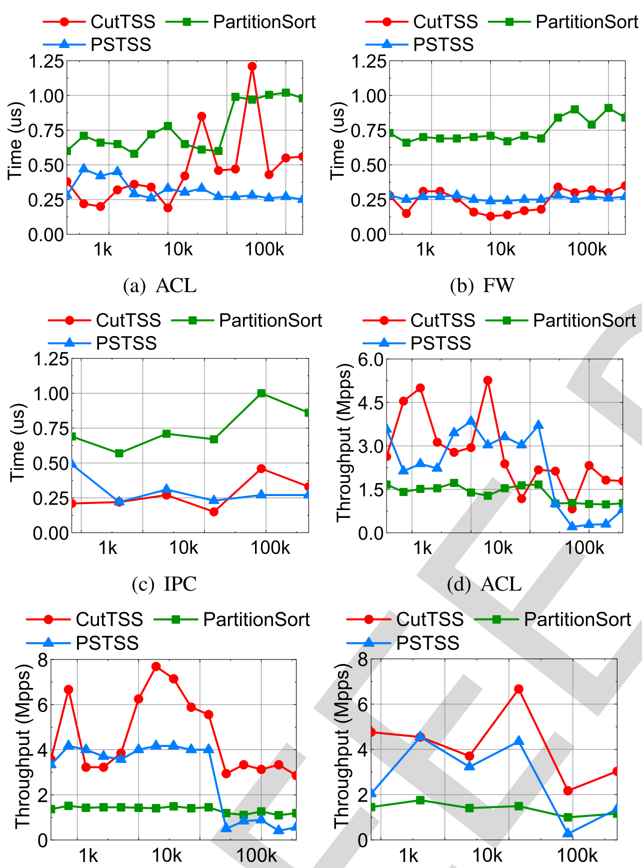
1) Average Classification Time: Figure 14 shows the average classification time and throughput of CutTSS as well as PSTSS, PartitionSort and CutSplit. In order to compare the performance of these algorithms, we first compute the average times for three different types of rules respectively, and then compute the ratio based on these average times. From Figure 14(a), (b) and (c), we can see that CutTSS requires less time to classify packets, with an average of 0.257 us, 0.318 us and 0.135 us for each type of rule set respectively, while PSTSS consumes an average of 1.765 us, 1.164 us and 1.506 us respectively. Thus, CutTSS achieves an average of 6.868 times, 3.661 times and 11.156 times speed-up on classification performance than PSTSS respectively, almost an order-of-magnitude improvement on classification time on average. Additionally, the experimental results show that CutTSS achieves 1.43 times and 1.89 times speed-up than CutSplit and PartitionSort respectively. It should be noted that, although there are much more subsets in PartitionSort, it can still achieve comparable performance to CutTSS. The reason is that, almost all the rules are concentrated in the first few subsets when ordered by maximum priority, so that most lookups in PartitionSort can terminate as soon as a rule is matched in the first few subsets.

2) Average Throughput: From Figure 14(d), (e) and (f), we can see that CutTSS achieves an average throughput of 6.013 Mpps (Million packets per second), 6.782 Mpps and 9.235 Mpps for each type of rule set respectively, while PSTSS achieves an average of 0.994 Mpps, 1.016 Mpps and 1.396 Mpps respectively. Thus, CutTSS achieves an average of 6.049 times, 6.675 times and 6.615 times improvement on throughput than PSTSS respectively. Compared to CutSplit and PartitionSort, CutTSS also achieves 1.304 times and 1.878 times improvement respectively across all of the rule sets. We can also see an interesting phenomenon in Figure 14 that the proposed CutTSS has much higher performance for a few rule sets, such as the second rule set and the seventh rule set in Figure 14(e). Actually, this phenomenon is caused by the characteristic of the seed parameter file in ClassBench. In Figure 14(e), the second, the seventh and the twelfth rule sets are generated based on the same seed parameter file, but



(a) ACL (b) FW (c) IPC

Fig. 15. Average memory access.



(a) ACL (b) FW (c) IPC (d) ACL (e) FW (f) IPC

Fig. 16. Update performance.

with different sizes. By checking the type of terminal nodes after pre-cuttings, we find that the ratio of *non-leaf terminal node* in these three rule sets is much less than that in other rule sets, meaning that the rules generated based on this specific seed file are more evenly distributed than others. Thus, most of the rules in these rule sets can be separated into leaf nodes and be searched with linear search as traditional decision trees. However, this phenomenon does not exist for the twelfth rule set in Figure 14(e), the reason is that, when the rule set contains more and more rules, there will be more and more tuples needed to be searched in *big subset*, which may become the performance hurdle of CutTSS.

3) Average Memory Access: Figure 15 shows the average memory access of CutTSS as well as PSTSS, PartitionSort and CutSplit. Note that we think traversing a tree node, a rule or a tuple as one memory access in our experiments. It is obvious that CutTSS is significantly better than other three algorithms. Compared to PSTSS, experimental results show that CutTSS achieves an average of 3.8 times reduction on the number of memory accesses. Compared to PartitionSort and CutSplit, CutTSS also achieves 2.3 times and 1.2 times improvement on average.

D. Evaluation on Incremental Update

Since CutSplit can not support fast incremental updates, we just evaluate update performance among CutTSS, PSTSS and PartitionSort. Figure 16 shows the average incremental update time and throughput of CutTSS as well as PSTSS and PartitionSort. From Figure 16(a), (b) and (c), we can see that CutTSS requires less time to update rules, achieving an average of 0.464 us, 0.246 us and 0.273 us for each type of rule set respectively, while PSTSS consumes an average of 0.314 us, 0.261 us and 0.301 us respectively. Additionally, our experimental results also show that, CutTSS achieves an average of 2.516 times speed-up on update time than PartitionSort across all of the rule sets. From Figure 16(d), (e) and (f), we can see that both CutTSS and PSTSS can achieve high throughput for updates, achieving at an average of 3.734 Mpps and 3.583 Mpps respectively. Thus, CutTSS has comparable update performance to PSTSS, which is used in Open vSwitch.

V. Conclusion

Open vSwitch implements a variant of TSS instead of decision tree-based algorithms despite their better performance on lookups, because the latter have poor support for fast incremental updating of rules, which is an important metric for SDN switches. However, TSS-based schemes can achieve fast updates but have a performance concern.

To achieve fast lookup and update at the same time, we propose CutTSS, a two-stage framework consisting of heterogeneous algorithms to adaptively exploit different characteristics of the rule sets at different scales. In the first stage, partial trees are constructed from rule subsets grouped with respect to their *small fields*. This grouping eliminates rule overlap at large scales, thereby enabling very efficient pre-cuttings without any rule replications. The second stage handles packet classification at small scales, where PSTSS is applied for these subsets to facilitate tree constructions. Overall, CutTSS exploits the strengths of both decision tree and TSS to circumvent their respective weaknesses. Experimental results show that CutTSS has comparable update performance to TSS in Open vSwitch, while achieving almost an order-of-magnitude improvement on classification performance over TSS.

References

[1] W. Li, X. Li, H. Li, and G. Xie, "CutSplit: A decision-tree combining cutting and splitting for scalable packet classification," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2645–2653.

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," in *Proc. ACM SIGCOMM*, 2008, pp. 69–74.

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

[4] H. J. Chao and B. Liu, *High Performance Switches Routers*. Hoboken, NJ, USA: Wiley, 2007.

[5] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*. Berlin, Germany: Springer-Verlag, 2010.

[6] H. Che, Z. Wang, K. Zheng, and B. Liu, "DRES: Dynamic range encoding scheme for TCAM coprocessors," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 902–915, Jul. 2008.

[7] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.

[8] B. Vamanan and T. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *Proc. ACM CoNEXT*, 2011, pp. 1–12.

[9] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1127–1140, Jun. 2013.

[10] O. Rottenstreich *et al.*, "Compressing forwarding tables," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1231–1239.

[11] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *Proc. ACM SIGCOMM*, 2014, pp. 15–26.

[12] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 39–50.

[13] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal In/Out TCAM encodings of ranges," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 555–568, Feb. 2016.

[14] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.

[15] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. IEEE Hot Interconnects*, May 1999, pp. 1–9.

[16] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2003, pp. 213–224.

[17] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE 28th Conf. Comput. Commun. (INFOCOM)*, Apr. 2009, pp. 648–656.

[18] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *Proc. ACM SIGCOMM*, 2010, pp. 207–218.

[19] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *Proc. IEEE 20th Annu. Symp. High-Perform. Interconnects*, Aug. 2012, pp. 1–8.

[20] W. Li and X. Li, "HybridCuts: A scheme combining decomposition and cutting for packet classification," in *Proc. IEEE 21st Annu. Symp. High-Perform. Interconnects*, Aug. 2013, pp. 41–48.

[21] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Oct. 2014, pp. 308–319.

[22] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted partitioning approach to high-speed and fast-update OpenFlow classification," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[23] J. Daly and E. Torng, "ByteCuts: Fast packet classification by interior bit extraction," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2654–2662.

[24] W. Li, T. Yang, Y.-K. Chang, T. Li, and H. Li, "TabTree: A TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–8.

[25] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2019, pp. 256–269.

[26] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, 1999, pp. 35–146.

[27] J. Daly and E. Torng, "TupleMerge: Building online packet classifiers by omitting bits," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2017, pp. 1–10.

[28] B. Pfaff *et al.*, "The design and implementation of open vswitch," in *Proc. USENIX NSDI*, 2015, pp. 117–130.

[29] M Kuźniar, P Perešíni, and D Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, 2015, pp. 347–359.

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

[31] M. H. Overmars and F. A. van der Stappen, "Range searching and point location among fat objects," *J. Algorithms*, vol. 21, no. 3, pp. 629–656, Nov. 1996.

[32] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, Oct. 1998.

[33] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 203–214, Oct. 1998.

[34] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.

[35] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE Conf. Comput. Commun. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, Mar. 2000, pp. 1193–1202.

[36] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, 2001, pp. 199–210.

[37] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 269–280.

[38] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards high-performance flow-level packet processing on multi-core network processors," in *Proc. 3rd ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Dec. 2007, pp. 17–26.

[39] C.-L. Hsieh and N. Weng, "Many-field packet classification for software-defined networking switches," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, Mar. 2016, pp. 13–24.

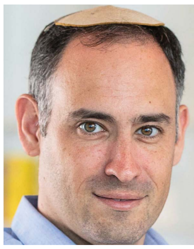
[40] T. Yang, A. X. Liu, Y. Shen, Q. Fu, D. Li, and X. Li, "Fast OpenFlow table lookup with fast update," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2636–2644.

[41] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[42] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2013, pp. 97–108.

Wenjun Li (Member, IEEE) received the B.Sc. degree from the University of Electronic Science and Technology of China in 2011 and the M.Sc. degree from Peking University in 2014, where he is currently pursuing the Ph.D. degree with the School of Electronics Engineering and Computer Science. From 2014 to 2015, he worked as a Researcher with the Network Research Department, Huawei Technologies Company, Ltd. His research interests focus on networking algorithms, such as packet classification, IP lookups, and sketches. He is a member of ACM and CCF.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS), China, from July 2013 to July 2014. He is currently an Associate Professor with the Department of Computer Science and Technology, Peking University. His research interests focus on networking algorithms, such as sketches, IP lookups, and Bloom filters. He published articles in SIGCOMM, SIGKDD, SIGMOD, JSAC, ToN, and so on.



Ori Rottenstreich received the B.Sc. degree in computer engineering *(summa cum laude)* and the Ph.D. degree from Technion in 2008 and 2014, respectively. He is currently an Assistant Professor with the Department of Computer Science and the Department of Electrical Engineering of the Technion, Haifa, Israel. From 2015 to 2017, he was a Post-Doctoral Research Fellow with the Department of Computer Science, Princeton University. His main research interest is computer networks.

Xianfeng Li received the B.S. degree from the School of Computer and Control Engineering, Beijing Institute of Technology, in 1995, and Ph.D. degree in computer science from the National University of Singapore, in 2005. He worked as an Associate Professor with the Peking University Shenzhen Graduate School. He is currently an Associate Professor with Macau University of Science and Technology (MUST). His research interests include software-defined networks, codesign of hardware and software, and the Internet of Things.

Gaogang Xie received the B.Sc. degree in physics and the M.S. and Ph.D. degrees in computer science from Hunan University. From 2002 to 2019, he was with the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently a Professor with the Computer Network Information Center (CNIC), CAS, and the University of Chinese Academy of Sciences. His research interests include Internet architecture, packet processing and forwarding, and Internet measurement.

Hui Li received the B.Eng. and M.S. degrees from the School of Information Engineering, Tsinghua University, Beijing, China, in 1986 and 1989, respectively, and Ph.D. degree from the Department of Information Engineering, The Chinese University of Hong Kong, in 2000. He was the Director of the Shenzhen Key Lab of Information theory and Future Internet architecture, the Director of the PKU Lab of China Environment for Network Innovations (CENI), National Major Research Infrastructure. He is currently a Full Professor of the Peking University Shenzhen Graduate School. He proposed the first co-governing future networking MIN based on blockchain technology and implemented its prototype on Operator's Network in the world, and this project MIN: Co-Governing MultiIdentifier Network Architecture and Its Prototype on Operator's Network was obtained the Award of World Leading Internet Scientific and Technological Achievements by the 6th World Internet Conference on 2019, Wuzhen, China. His research interests include network architecture, cyberspace security, distributed storage, and blockchain.

Balajee Vamanan received the Ph.D. degree from Purdue University in 2015. Prior to his graduate study, he worked in NVIDIA as a Design Engineer. He is currently an Assistant Professor with the Department of Computer Science, University of Illinois at Chicago (UIC). His research interests span various aspects of computer networks and computer systems.

Dagang Li received the bachelor's degree in telecommunications engineering from the Huazhong University of Science and Technology, Wuhan, China, and the Ph.D. degree from Katholieke Universiteit Leuven, Leuven, Belgium. He is currently an Assistant Professor with the School of Electronic and Computer Engineering, Peking University Shenzhen Graduate School, Shenzhen, China. His main research interest is computer networks.

Huiping Lin is currently pursuing the bachelor's degree with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. She is majoring in computer science and technology advised by Prof. T. Yang. Her research interests include computer networks, network measurement, hash algorithms and sketches. She has received the Merit Student Award and a Scholarship from Peking University.

Tuple Space Assisted Packet Classification With High Performance on Both Search and Update

Wenjun Li, *Member, IEEE*, Tong Yang, Ori Rottenstreich, Xianfeng Li, Gaogang Xie, Hui Li, Balajee Vamanan, Dagang Li, and Huiping Lin

***Abstract*— Software switches are being deployed in SDN to enable a wide spectrum of non-traditional applications. The popular Open vSwitch uses a variant of Tuple Space Search (TSS) for packet classifications. Although it has good performance on rule updates, it is less efficient than decision trees on lookups. In this paper, we propose a two-stage framework consisting of heterogeneous algorithms to adaptively exploit different characteristics of the rule sets at different scales. In the first stage, partial decision trees are constructed from several rule subsets grouped with respect to their *small fields*. This grouping eliminates rule replications at large scales, thereby enabling very efficient pre-cuttings. The second stage handles packet classification at small scales for *non-leaf terminal nodes*, where rule replications within each subspace may lead to inefficient cuttings. A salient fact is that small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. To exploit this favorable property, we employ a TSS-based algorithm for these subsets following tree constructions. Experimental results show that our work has comparable update performance to TSS in Open vSwitch, while achieving almost an order-of-magnitude improvement on classification performance over TSS.**

***Index Terms*— Packet classification, SDN, openflow, open vswitch, virtualization.**

Manuscript received April 15, 2019; accepted March 2, 2020. This work was supported in part by the National Keystone Research and Development Program of China under Grant 2017YFB0803204, in part by the PCL Future Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001, in part by the NSFC under Grant 61671001 and Grant 61725206, in part by the Research and Development Key Program of Guangdong under Grant 2019B010137001, and in part by the Shenzhen Research Program under Grant KQJSCX20180323174744219, Grant JCYJ20190808155607340, Grant JSGG20170824095858416, and Grant JCYJ20170306092030521. This article was presented in part at the IEEE INFOCOM, HI, USA, April 19, 2018. *(Corresponding author: Hui Li.)*

Wenjun Li is with the School of Electronic and Computer Engineering, Peking University, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: wenjunli@pku.edu.cn).

Tong Yang and Huiping Lin are with the Department of Computer Science and Technology, Peking University, Beijing 100871, China (e-mail: yangtongemail@gmail.com; phoenixrain@pku.edu.cn).

Ori Rottenstreich is with the Department of Computer Science, Technion–Israel Institute of Technology, Haifa 32000, Israel, and also with the Department of Electrical Engineering, Technion–Israel Institute of Technology, Haifa 32000, Israel (e-mail: or@technion.ac.il).

Xianfeng Li is with the International Institute of Next Generation Internet, Macau University of Science and Technology, Taipa, Macau, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: xifli@must.edu.mo).

Gaogang Xie is with the Computer Network Information Center, Chinese Academy of Sciences, Beijing 100190, China (e-mail: xie@cnic.cn).

Hui Li and Dagang Li are with Shenzhen Graduate School, Peking University, Shenzhen 518055, China, and also with the Peng Cheng Laboratory, Shenzhen 518055, China (e-mail: lih64@pku.edu.cn; dgli@pku.edu.cn).

Balajee Vamanan is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA (e-mail: bvamanan@uic.edu).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2020.2986935

I. Introduction

SOFTWARE virtual switches are becoming an important part of virtualized network infrastructures. Backed by SDN, virtual switches enable many non-traditional network functionalities like flexible resource partitioning and real-time migration. Despite their advantages on flexibility and low-cost, software switches have a performance concern. The prominent Open vSwitch enforces forwarding policies with OpenFlow [2] table lookups, which is essentially a multi-field packet classification problem [3], [4]. As an extensively studied bottleneck, packet classification in physical switches still relies on expensive TCAMs because algorithmic solutions implemented in software can hardly satisfy wire-speed forwarding in traditional network infrastructures [5]–[14]. With the advent of SDN and NFV, efficient algorithmic solutions using commodity memories such as DRAM/SRAM are becoming attractive again.

The first step towards meeting this revitalized demand is an understanding of the past research. Among existing algorithmic packet classification research, decision tree [15]–[25] and Tuple Space Search (TSS) [26]–[28] are two major approaches. In decision tree-based schemes, the geometric view of the packet classification problem is taken and a decision tree is built. They work by recursively partitioning the searching space into smaller subspaces until less than a predefined number of rules are contained by each subspace. In case a rule spans multiple subspaces, the problem of rule replication happens and a rule copy is needed for each overlapped subspace. This rule replication problem becomes especially serious during the cutting operations at small scales, where small rules across narrow spaces are to be separated from their overlapped large rules. Thus, decision tree-based schemes achieve fast lookup speed on packet classification, but cannot support fast updates due to the notorious rule replication problem.

Unlike traditional packet classification, OpenFlow has a much higher demand for updates, which further exacerbates the problem and makes decision tree algorithms inapplicable in this context [28], [29]. In contrast, TSS partitions rules into a set of hash tables (i.e., tuple space) with respect to their prefix length. Thus, rule replication never happens in TSS-based schemes, thereby enabling an average of one

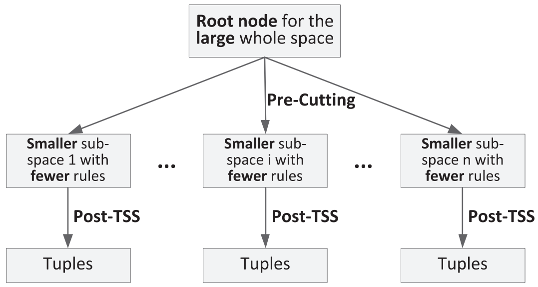


Fig. 1. The initial framework of CutTSS.

memory access for each rule update. As a result, the popular Open vSwitch implements a variant of TSS for its flow table lookups [28]. The primary reason is its good support for fast incremental rule updates, which is an important metric for SDN switches. Despite their advantages on fast updates, TSS-based schemes have a performance concern. For each incoming packet, TSS requires searching on every tuple, because the final matching is the one with the highest priority of matched rules from all tuples. This problem is especially serious for working on large space due to serious tuple expansion problem.

To achieve fast lookup and update at the same time, we propose CutTSS as shown in Figure 1, which fosters the strengths and circumvents the weaknesses of decision tree and TSS-based schemes. To the best of our knowledge, this is the first solution that can keep the advantages of these two schemes: fast lookup and fast update. First, we adopt cutting techniques to build decision trees at large scales, so that each packet can shrink its searching space in a few steps for fast lookups. Second, to improve the update performance, we introduce TSS-based schemes to assist decision tree construction at small scales. Overall, CutTSS exploits the strengths of both decision tree and TSS to circumvent their respective weaknesses.

To refine the framework of CutTSS, a two-stage framework consisting of heterogeneous algorithms is proposed. During the first stage, partial decision trees are constructed from several subsets grouped in their respective *small fields* (i.e., long prefix or narrow range), and leave some *non-leaf terminal nodes* (i.e., terminal nodes after pre-cuttings which contain rules more than the predefined number of rules for leaf nodes, as illustrated in Figure 6) for more efficient handling by TSS-based schemes. This grouping eliminates rule overlapping at large scales, thereby enabling very efficient pre-cuttings without any rule replications. The second stage handles packet classification at small scales for rules in *non-leaf terminal nodes*, where overlapping of rules within each subspace may become common that will lead to inefficient cuttings due to rule replications. Fortunately, a small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. Based on this property, we employ a TSS-based algorithm called PSTSS [28] for rules in these subsets to facilitate tree constructions. Therefore, by exploiting the benefits of decision tree and TSS techniques adaptively, CutTSS not only offers fast updates and linear memory, but also pushes the performance of algorithmic packet classification on par to hardware-based solutions. The main contributions of this paper include the following aspects:

- A scalable rule set partitioning algorithm based on the observation that most rules have at least one *small field* spanning across a narrow space, so the rule set can be efficiently partitioned into a few non-overlapping subsets.
- A set of novel cutting algorithms that exploit the global characteristics of the partitioned subset of rules, so that the rules can be partitioned into smaller subsets without rule replications.
- A two-stage framework combining decision tree and TSS techniques, which can adaptively exploit different characteristics of the rule sets at different scales.

We evaluate our algorithm using ClassBench [30], and the results show that CutTSS is able to produce a very small number of shorter trees with linear memory consumption even for rule sets up to 100k entries. Compared to the TSS algorithm in Open vSwitch, CutTSS achieves similar update performance, but outperforms TSS significantly on classification performance, achieving almost an order of magnitude improvement on average. Our implementation of CutTSS is publicly available on our website (http://www.wenjunli.com/CutTSS).

The rest of the paper is organized as follows. In Section II, we first briefly summarize the related work. After that, we make a set of observations and present the technical details of CutTSS in Section III. Section IV provides experimental results. Finally, conclusions are drawn in Section V.

II. Background and Related Work

In this section, we first review the background and some research efforts about the packet classification problem. After that, we briefly describe two major threads of algorithmic approaches: decision tree-based and tuple space-based packet classification. Finally, we give some summaries.

A. The Packet Classification Problem

The purpose of packet classification is to enable differentiated packet treatment according to a predefined packet classifier. A packet classifier is a set of rules, with each rule R consisting of a tuple of F field values (exact value, prefix or range) and an action (e.g., drop or permit) to be taken in case of a match. The rules in the classifier are often prioritized to resolve potential multiple match scenarios. Packet classification has been well studied for two decades, but most of them focused on high-speed lookups, with very little consideration on the performance of rule updates. However, unlike traditional packet classification, OpenFlow has a much higher demand for updates, making most of traditional algorithms inapplicable in the context of SDN. An example OpenFlow classifier is shown in Table I.

Packet classification is a hard problem with high complexity. From a geometric point of view, packet classification can be treated as a point location problem, which has been proved that the best bounds for locating a point are either $\Theta(log\ N)$ time with $\Theta(N^F)$ space, or $\Theta((logN)^{F-1})$ time with $\Theta(N)$ space for N non-overlapping hyper-rectangles in F-dimensional space [31]. Therefore, the worst-case mathematical complexity

TABLE I

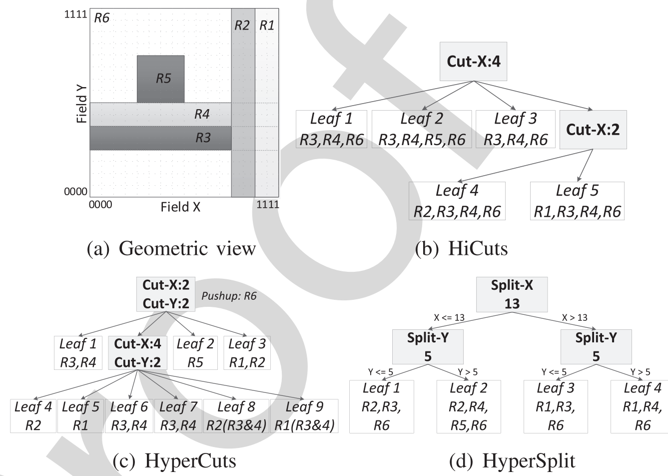
AN EXAMPLE OPENFLOW 1.0 CLASSIFIER

Rule id	Ingress port	Ether src	Ether dst	Ether type	VLAN id	VLAN priority	IP src	IP dst	IP proto	IP ToS bits	TCP/UDP Src Port	TCP/UDP Dst Port	*Action*
R_1	3	*	*	2048	*	*	206.159.213.125/32	101.152.182.8/30	0x06f/0xff	0	1024 : 65535	*	$action_1$
R_2	3	*	*	2048	*	*	15.25.70.8/30	*	*	0	*	0:1599	$action_2$
R_3	5	*	*	2048	*	*	*	18.152.125.32/30	0x11/0xff	1	1024 : 65535	1024 : 65535	$action_3$
R_4	5	*	*	2048	*	*	206.159.213.125/32	*	0x06/0xff	1	*	80	$action_4$
R_5	*	*	*	*	*	*	*	*	*	*	*	*	$action_5$

TABLE II

AN EXAMPLE OF 2-TUPLE CLASSIFIER

Rule id	Priority	Field X	Field Y	*Action*
R_1	6	111*	*	$action_1$
R_2	5	110*	*	$action_2$
R_3	4	*	010*	$action_3$
R_4	3	*	011*	$action_4$
R_5	2	01**	10**	$action_5$
R_6	1	*	*	$action_6$



(a) Geometric view (b) HiCuts

(c) HyperCuts (d) HyperSplit

Fig. 2. Review on related decision trees (*binth* = 4).

of algorithmic packet classification is extremely high, which makes it impractical to achieve a wire-speed requirement within the capabilities of current memory technology. But fortunately, packet classification rules in real-life applications have some inherent characteristics that can be exploited to reduce the complexity. These inherent characteristics provide a good substrate for the exploration of practical algorithmic solutions [1], [15]–[18], [20], [21], [23], [26], [32]–[40]. Among them, decision tree and Tuple Space Search (TSS) are two major approaches. Next, we briefly summarize the related work on these two techniques. For the convenience of description, we use a small example of 2-tuple rule set shown in Table II for subsequent discussions. Figure 2(a) shows the geometric representation of the example rules given in Table II.

B. Decision Tree-Based Packet Classification

In decision tree-based schemes, the geometric view of the packet classification problem is taken and a decision tree is built. The root node covers the whole searching space containing all rules. They work by recursively partitioning the searching space into smaller subspaces until less than a predefined number of rules are contained by each subspace. In case a rule spans multiple subspaces, the undesirable rule replication happens (e.g., *R3*, *R4* and *R6* in Figure 2(b)). When a packet arrives, the decision tree is traversed to find a matching rule at a leaf node. According to the partitioning method on searching space, current decision trees can be categorized into two major approaches: *equal-sized cutting* and *equal-densed cutting (i.e., splitting)*.

1) Classical Decision Tree Schemes: Cutting based schemes, such as HiCuts [15] and HyperCuts [16], separate the searching space into many equal-sized subspaces using local optimizations. HiCuts cuts the searching space into many equal-sized subspaces recursively until the rules covered by each subspace is less than the pre-defined bucket size called *binth*. To reduce memory consumption, HiCuts uses some heuristics to select the cutting dimension and decides how many subspaces should be cut using a space optimization function. Figure 2(b) shows the decision tree generated by HiCuts, where the *Field X* is cut into four equal-sized subspaces (i.e., [0,3], [4,7], [8,11], [12,15]), and is further cut into two equal-sized subspaces (i.e., [12,13], [14,15]) to finish the decision tree construction. HyperCuts can be considered as an improved version of HiCuts, which is more flexible in that it allows cutting on multiple fields per step, resulting in a fatter and shorter decision tree. Besides, several optimization techniques are adopted in HyperCuts, such as *node merging*, *rule overlap*, *region compaction* and *pushing common rule subsets upwards*. But both HiCuts and HyperCuts have the same rule replication problem for rules spanning multiple subspaces, especially for large rule tables. Figure 2(c) shows the decision tree generated by HyperCuts.

In order to reduce the rule replications suffered from equal-sized cuttings, schemes based on splitting divide the searching space into unequal-sized subspaces containing a nearly equal number of rules. HyperSplit [17], a well-known splitting-based decision tree scheme, splits the searching space into two unequal-sized subspaces containing a nearly equal number of rules. Due to its simple binary separation in subspaces, the worst-case search performance of HyperSplit is explicit. However, even with the optimized binary space splitting, the memory consumption of HyperSplit still grows exponentially as the number of rules increases. Figure 2(d) shows the decision tree generated by HyperSplit, we can see that in each internal tree node, HyperSplit splits the selected field into two unequal-sized subspaces, with each subspace covering rules as balanced as possible.

2) Recent Decision Tree Schemes: EffiCuts [18], a well-known cutting based scheme, observed that real-life rules exhibit several inherent characteristics, and a good rule set partitioning can reduce rule replications dramatically. Thus, instead of building a single decision tree for all rules, EffiCuts separates rules into several subsets with each subset creating its own decision tree independently using a variant of HyperCuts. With all F fields considered, up to 2^F decision trees can be generated for F-tuple classifiers, resulting in a lot of overall memory accesses. In contrast, HybridCuts [20] separates rules based on a single field rather than all F fields in EffiCuts, thus, HybridCuts achieves a significant reduction in the number of subsets (i.e., from 2^F to $F+1$), which in turn reduces the overall memory accesses. However, due to the employment of HyperCuts, the worst-case search performance of HybridCuts is unbounded. Worse still, with the increase of the number of rule fields and the size of classifiers, the performance of HybridCuts drop dramatically due to the deteriorating rule replications. Instead of using the most significant bits for cuts, ByteCuts [23] introduces a new cutting scheme that uses any range of bits to build decision trees. However, ByteCuts can achieve high-speed construction of trees but not fast updates of rules.

In order to reduce rule replications suffered from splitting, ParaSplit [19] proposes a rule set partitioning algorithm to reduce rule set complexity, which significantly reduces the overall memory consumption in HyperSplit. However, ParaSplit employs a complex heuristic for rule set partitioning, which may require tens of thousands of iterations to reach an optimal partitioning. To achieve better scalability for different rule sets, SmartSplit [21] separates rules into at most four subsets to build balanced trees dynamically, achieving high-speed classification by leveraging the logarithmic search time of balanced search trees. By partitioning rules into several sortable subsets and building a MITree for each subset, PartitionSort [22] achieves logarithmic classification and update time for each subset simultaneously. Due to the stringent constraints on partitioning, PartitionSort requires much more trees than SmartSplit, resulting in slower classification. Instead of using a single cutting or splitting technique to build trees, CutSplit [1], the preliminary version of the proposed CutTSS, introduces a practical framework that can exploit the benefits of cutting and splitting techniques adaptively. However, due to the boring rule replications in its post-splitting stage, CutSplit can only achieve incremental updates by rebuilding sub-trees, consuming up to a few milliseconds in some cases, far more behind the wire-speed requirement of incremental updates.

C. Tuple Space-Based Packet Classification

In tuple space-based schemes, rules are partitioned into a set of hash tables (i.e., tuple space) based on easily computed rule characteristics. Thus, rules can be inserted and deleted from hash tables in amortized one memory access, resulting in faster updates. When a packet arrives, these partitioned hash tables are individually searched to find the best matching.

1) Classical Tuple Space Schemes: Tuple Space Search (TSS) [26], the basic tuple space-based packet classification, decomposes a classification query into a set of exact match queries in hash tables. TSS partitions rules into different hash

TABLE III

TSS BUILDS 4 TUPLES FOR RULES GIVEN IN TABLE II

Tuple	*Rule id*	Rule Priority	Tuple Priority	Field X	Field Y	*Action*
(3, 0)	R_1	6	6	111*	*	$action_1$
	R_2	5		110*	*	$action_2$
(0, 3)	R_3	4	4	*	010*	$action_3$
	R_4	3		*	011*	$action_4$
(2, 2)	R_5	2	2	01**	10**	$action_5$
(0, 0)	R_6	1	1	*	*	$action_6$

tables based on a set of pre-computed tuples. Each tuple can be defined by concatenating the actual bits used in each field in order, so that a hash key can be created to map the rules of that tuple into its corresponding hash table. During classification or updates, those same bits are extracted from the packet or rule as a hash key for searches. For example, rules *R1* and *R2* shown in Table II should be placed in the same tuple space, because both of them use three and zero of the bits in their respective two fields. Thus, TSS builds four tuple spaces as shown in Table III for rules given in Table II. As an improvement, the Pruned Tuple Space Search (PTSS) algorithm [26] reduces the scope of the exhaustive search by performing a search on individual rule fields to find a subset of candidate tuple spaces. However, both TSS and PTSS have low classification speed, because the number of tuple space is large and each tuple space must be searched for every packet. This problem becomes more serious for classifiers with an increased number of fields such as OpenFlow classifiers.

2) Recent Tuple Space Schemes: TupleMerge [27], a recently proposed tuple space scheme, improves upon TSS by relaxing the restrictions on which rules may be placed in the same tuple space. By merging tuple spaces that contain rules with similar characteristics together, TupleMerge can reduce the number of candidate tuple spaces and thus the overall classification time. However, with more tuple spaces merged, its performance may be affected due to hash collisions. Priority Sorting Tuple Space Search (PSTSS) [28], which is used in Open vSwitch, improves the performance of TSS by sorting tuple spaces based on a pre-computed priority of each tuple space (i.e., *Tuple Priority* column in Table III). By searching tuple spaces in the descending order of priority, the search can terminate as soon as a match is found because it has the highest priority among all possible matched rules. Although PSTSS can improve average performance compared to TSS, its worst-case performance is still the same as TSS.

D. Summary of Prior Art

Clearly, decision tree-based packet classification has been actively investigated for two decades. But as far as we know, none of them can make an excellent trade-off among all key metrics. In particular, most of them can achieve high-speed packet classification but not fast updates, which seriously limit their scalability in the era of SDN. In contrast, tuple space-based schemes have been the *de-facto* choice in software switches, because they support fast updates with only linear memory consumption. However, these schemes still suffer from low classification performance especially for large classifiers, falling short of the needs of high-speed requirements in fast-growing networks.

III. CutTSS: Enjoying Both Worlds of Efficient Classication and Rule Update

In this section, we first introduce ideas behind the design of CutTSS. Then, we propose a scalable partitioning algorithm based on experimental observations, which can eliminate rule overlapping at large scales. To exploit these characteristics of partitioned subsets, a set of novel cuttings are designed to build partial trees without any rule replications in the first stage. After that, a two-stage framework consisting of heterogeneous algorithms is proposed to build decision trees for partitioned subsets. Finally, we give more insights on the effectiveness of CutTSS from both theoretical and experimental aspects.

A. Ideas & Framework

According to the above review and analyses given in CutSplit [1], we know that cutting techniques can separate searching space into smaller subspaces quickly for faster classification, but it suffers from serious rule replications. In contrast, TSS can completely avoid rule replications and support fast incremental updates, but it has longer classification time due to tuple expansions, especially for rule sets at large scales. Therefore, to foster the strengths and circumvent the weaknesses of decision tree and TSS schemes, the idea directly perceived is to combine the following two strategies:

- **Pre-Cutting at large scales**: Cutting-based partitioning on the searching space at large scales, which can reach any subspaces at small scales with very few steps.
- **Post-TSS at small scales**: TSS-based searching on subspaces at small scales, which can avoid inefficient cutting of decision trees.

Figure 1 shows the initial framework of the following CutTSS. However, in order to design scalable algorithms to meet the above design goals, an effective combination of these two ideas still faces several difficulties and challenges:

- **Low memory access**: Although partitioning can reduce rule overlapping significantly, it will increase the overall memory accesses. Thus, how to generate rule subsets as few as possible?
- **Low memory consumption**: Since cutting on rules overlapped at different scales will lead to serious rule replications, how to avoid rule replications during the first cutting stage?
- **Low update time**: Since many rules are overlapped and concentrated in some subspaces at small scales, which will lead to inefficient cuttings due to rule replications. How to avoid rule replications in these subspaces at small scales?

The answers to these questions are the key ideas in this paper. Our solution can be summarized in the following three steps:

- **Step 1: Partitioning based on very few small fields**: In order to eliminate rule overlapping at large scales and reduce the number of partitioned subsets, we separate rules into subsets based on their characteristics shared in very few *small fields*.
- **Step 2: Pre-cuttings by exploiting the global characteristics of the partitioned subsets**: After partitioning the rule set, we get a set of favorable fields for each partitioned subset, where a set of simpler cutting algorithms without prior optimizations can be applied for space partitioning.
- **Step 3: TSS-assisted cutting trees for fast updates**: Thanks to the clever partitioning and pre-cuttings without any rule replications, most of the rules can be separated into leaf nodes for the linear search, except for a small fraction of concentrated rules at small scales. For these *non-leaf terminal nodes*, we employ a TSS-based algorithm to facilitate the rest of tree constructions.

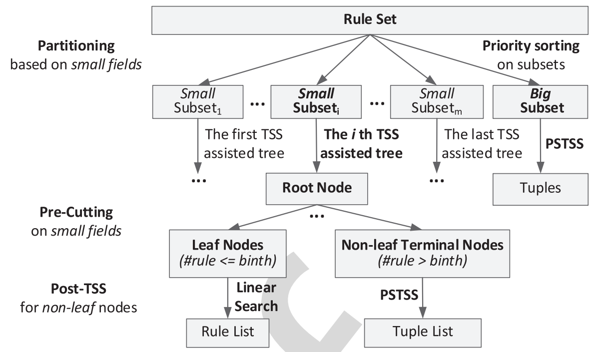


Fig. 3. The refined framework of CutTSS.

Based on these ideas, we give the refined framework of the proposed CutTSS shown in Figure 3. Overall, a complete packet classification framework with two heterogeneous stages exploiting favorable properties in their respective space scales is in place. Next, we give more details about CutTSS from the following three aspects: rule set partitioning, decision tree construction and decision tree operation.

B. Rule Set Partitioning Based on Small Fields

Classification rules in real-life applications have structural redundancies and several inherent characteristics that can be exploited to reduce the complexity. Thus, we use the publicly available ClassBench and OpenFlow-like rule tables for study to make observations on common characteristics of rule sets. It should be noted that the two OpenFlow-like rule tables are supported by the authors of ParaSplit [19], which were generated based on 216 real-life rules from enterprise customers. We first give a few definitions, then we present the key observations related to the following discussions on rule set partitioning.

1) Definitions: Given an N-field rule $R = (F_1, \ldots F_i, \ldots, F_N)$ and a threshold value vector $T = (T_1, \ldots T_i, \ldots, T_N)$, where $i \in \{1, 2, \ldots, N\}$, we first give some definitions for field F_i as follows:

- F_i is a ***big field***: the range length of field $F_i > T_i$;
- F_i is a ***small field***: the range length of field $F_i \leq T_i$.

Based on the above definitions for field F_i, we further give some definitions for R as follows:

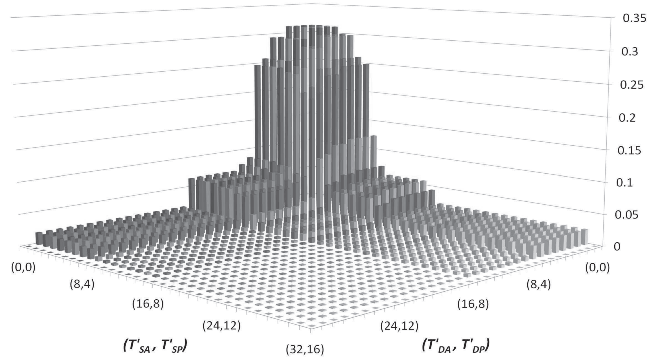


Fig. 4. The ratio of *big rules* for seed-ipc rule set.

TABLE IV
STATISTICAL RESULTS FOR 5-TUPLE & OPENFLOW-LIKE RULES

Rule Set(#rules)	Threshold Vector	#*Big Rules*	#*Small-k* Rules				
			k=1	*k=2*	*k=3*	*k=4*	*k>4*
seed-acl(752)	T = (2^{16}, 2^{16}, 2^8, 2^8)	3	749	739	425	0	0
seed-fw(269)		4	265	218	17	2	0
seed-ipc(1550)		2	1548	1472	789	5	0
openflow-1(716)	T = (2^{t_1}, 2^{t_2}...2^{t_i}...),	0	716	708	655	426	0
openflow-2(864)	t_i = half width of F_i	0	864	852	761	429	0

- R is a **big rule**: $\forall i \in \{1, 2, \ldots, N\}$, F_i is a *big field*;
- R is a ***k*-small rule**: R contains at least k *small fields*.

For a classical 5-tuple rule, since the protocol field is restricted to a small set of values (e.g., tcp, udp), we just consider the other four fields in this paper. Then the threshold value vector T for 5-tuple rules is simplified to a four-dimensional vector $T = (T_{SA}, T_{DA}, T_{SP}, T_{DP})$. For the sake of convenience in writing, we use a logarithmic vector T' to represent the threshold value vector T equivalently. For example, if we set threshold value vector $T = (2^{16}, 2^{16}, 2^8, 2^8)$, then index logarithmic $T' = (16, 16, 8, 8)$.

2) Observations: Based on the above definitions, we make some statistical experiments for several rule sets from ClassBench. There are three types of rule sets: ACL (Access Control List), FW (Firewall) and IPC (IP Chains). Figure 4 shows the ratio of *big rules* under different thresholds for seed IPC rule set. Due to length limitation, more results for ACL and FW rule sets are publicly available on our website (http://www.wenjunli.com/CutTSS). It is clear that the ratio of *big rules* is very low even under very demanding thresholds. For example, assume $T' = (16, 16, 8, 8)$, the ratio of *big rules* for three types of rule set are all less than 0.01, that is to say, less than 1% rules are *big rules* under $T = (2^{16}, 2^{16}, 2^8, 2^8)$. This indicates that the vast majority of the rules have at least one *small field* satisfying a threshold T. Essentially, this observation is consistent with previous observations. Otherwise, a large number of *big rules* may cause serious space overlapping which is contrary to previous observations that the number of rules or address prefixes matching a given packet is typically five or less.

Table IV shows statistical results for 5-tuple rule sets and OpenFlow-like rule tables. Clearly, this observation is still effective for OpenFlow-like rule tables: the vast majority of rules have at least one *small field*.

3) Rule Set Partitioning: Based on the above observations, we propose a partitioning algorithm to separate rules into several subsets. The purpose of our partitioning is to obtain a few subsets without duplicates among each other. For each subset, all contained rules should share a common characteristic for a set of rule fields: *small field*. Next, we introduce a simple heuristic as follows:

–Step1: Removal of big rules. Since the number of *big rules* is negligible, we can simply apply PSTSS for these rules.

–Step2: Selection of partitioning rules. We first count the distinct values for each field, then select a few fields with a large number of distinct values. The selection makes sure that for the vast majority of rules, there is at least one selected field with a small value. The rest rules without any small value in the selected fields will be treated as *big rules*.

–Step3: Fields-wise partitioning. Assume that M fields have been selected for F-tuple rule sets. We categorize rules based on field length (i.e., *big* or *small*) in all selected fields, leading to at most 2^M-1 subsets. This partitioning is different from EffiCuts from two perspectives: fewer fields and more flexible definition of *small/big field*, which enables much more flexible partitioning to generate fewer subsets.

–Step4: Selective subset merging. For subsets containing very few rules, we can merge these rules into other subsets that have fewer *small fields*. Due to the consideration of its relevance and space limitation, we do not elaborate on this algorithm in this paper. Note that our merging will not lead to rule replication in our decision trees, which is quite different from EffiCuts.

Take rules in Table I as an example, we first move *R5* into a *big rule* subset, then we calculate the number of distinct *small fields* in each field and pick *ip_src* & *ip_dst* as the two most distinct fields. Thus, we can partition the rule set into four subsets: *big_subset* = {*R5*}, ($small_{ip\_src}$, $small_{ip\_dst}$) = {*R1*}, (big_{ip_src}, $small_{ip\_dst}$) = {*R3*} and ($small_{ip\_src}$, big_{ip_dst}) = {*R2, R4*}. Finally, we can merge ($small_{ip\_src}$, $small_{ip\_dst}$) with (big_{ip_src}, $small_{ip\_dst}$) for a new subset ($arbitrary_{ip\_src}$, $small_{ip\_dst}$) = {*R1, R3*}, where *arbitrary field* contains both *small* and *big field*. Thus, three subsets are generated for the sample rule set: *big_subset* = {*R5*}, ($small_{ip\_src}$, big_{ip_dst}) = {*R2, R4*} and ($arbitrary_{ip\_src}$, $small_{ip\_dst}$) = {*R1, R3*}.

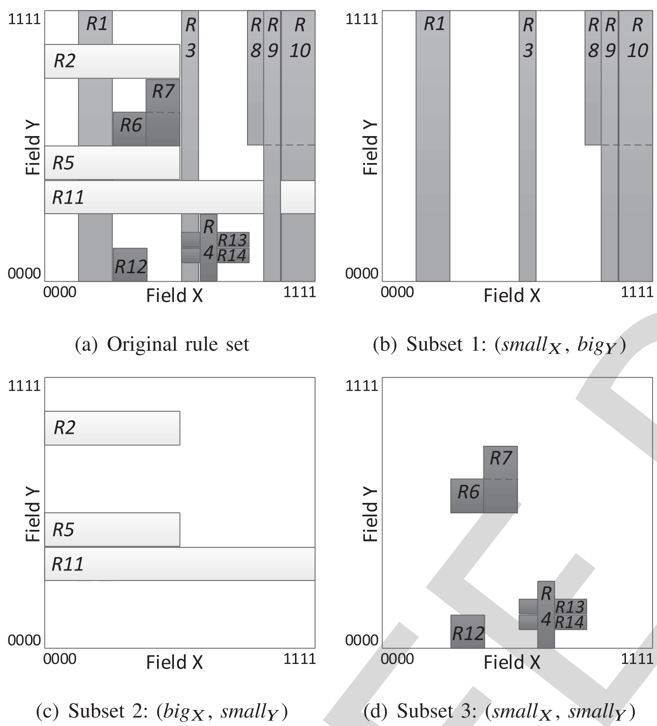
C. Decision Tree Construction: Pre-Cutting & Post-TSS

From Figure 3, we can see that a TSS-assisted tree will be built for each partitioned subset (except for the *big* subset). Thus, we will give more details about the tree building algorithm in CutTSS. For the convenience of description, we will use the rule set shown in Table V as a working example, where no *big rules* are included. Based on the above definitions of rule field, we can label each rule with a field vector as shown in Table V. Based on these field labels, the fourteen rules can be partitioned into three subsets as shown in Figure 5. For each subset, we then build the decision tree through the following two steps: pre-cutting and post-TSS.

1) Pre-Cutting: Fixed Cuttings on Small Fields: The rationale behind the above strategy of rule set partitioning is simple:

TABLE V
A NEW EXAMPLE OF 2-TUPLE CLASSIFIER ($T_X = 4$, $T_Y = 4$)

Rule id	Priority	Field X	Field Y	*Action*	*Field Label*
R_1	14	001*	*	$action_1$	*<small, big>*
R_2	13	0***	110*	$action_2$	*<big, small>*
R_3	12	1000	*	$action_3$	*<small, big>*
R_4	11	1001	00**	$action_4$	*<small, small>*
R_5	10	0***	011*	$action_5$	*<big, small>*
R_6	9	01**	100*	$action_6$	*<small, small>*
R_7	8	011*	10**	$action_7$	*<small, small>*
R_8	7	11**	1***	$action_8$	*<small, big>*
R_9	6	1101	*	$action_9$	*<small, big>*
R_{10}	5	111*	*	$action_{10}$	*<small, big>*
R_{11}	4	*	010*	$action_{11}$	*<big, small>*
R_{12}	3	010*	000*	$action_{12}$	*<small, small>*
R_{13}	2	10**	0010	$action_{13}$	*<small, small>*
R_{14}	1	10**	0001	$action_{14}$	*<small, small>*



(a) Original rule set (b) Subset 1: ($small_X$, big_Y)

(c) Subset 2: (big_X, $small_Y$) (d) Subset 3: ($small_X$, $small_Y$)

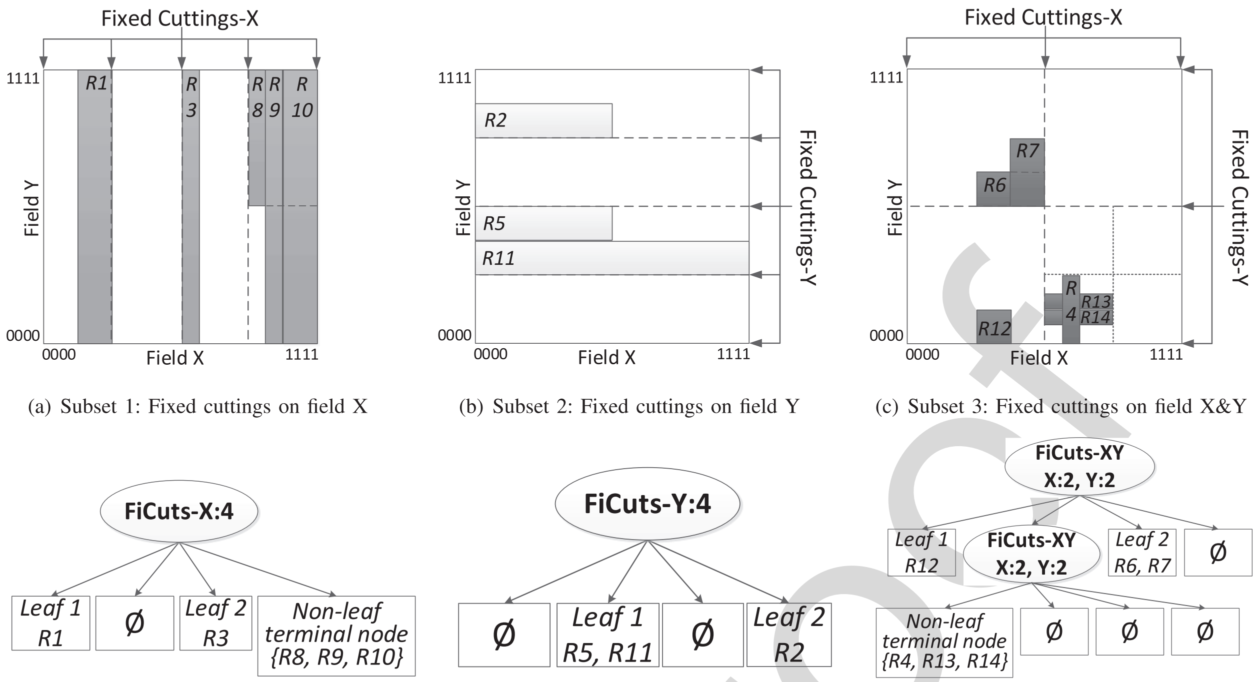
Fig. 5. Ruleset partitioning example ($T_X = 4$, $T_Y = 4$).

by grouping rules that are narrow in the same fields, rules that are large in these fields are excluded, and intensive rule replications caused by these excluded rules are eliminated, thereby enabling very efficient cuttings. What is more, this grouping can completely eliminate rule replications at large scales (i.e., larger than *small field*'s threshold) for prefix fields, because each prefix can never be overlapped with two shorter prefixes with the same prefix length. To exploit these favorable characteristics of partitioned subsets, we introduce a simple but effective cutting algorithm called Fixed Cuttings (FiCuts), which will be applied in the first stage partial tree construction.

FiCuts derives from HiCuts and HyperCuts, but with a better global view on the characteristics of the rule set. As shown in Figure 5(b), the rules in the subset are all *small* in *Field X*. This facilitates cuts along the *Field X* without any rule replications at large scales. Since the rules have been grouped into several subsets, with each subset sharing the same *small fields*, FiCuts can utilize this information to exploit efficient cuttings. Compared to HiCuts and HyperCuts, FiCuts has several differences in cutting details as follows: (1) Instead of changing cutting dimensions dynamically, FiCuts conducts cuttings on the subset along a set of fixed dimensions, where the rules are *small* in these fields; (2) Instead of deciding how many subspaces should be cut per step dynamically, the number of cuts per step in CutTSS is a fixed value (i.e., *MAXCUTS*, defined to control the number of empty tree node); (3) None of the prior optimization methods is required in FiCuts, making it simple enough to achieve fast lookups and updates; (4) As FiCuts is only designed for the construction of partial trees in the first stage, its cutting processes will stop not only in the leaf nodes containing rules less than the pre-defined *binth*, but also in the nodes located at small scales (i.e., cutting space smaller than the threshold of *small field*).

Thus, for the subsets containing one or more *small fields*, FiCuts will cut on that single or multiple *small fields* to build partial trees as illustrated in Figure 6. We can see from the partial trees that rule replication can be completely avoided and all rules are located in the bottom nodes of the tree (i.e., leaf nodes or *non-leaf terminal nodes*). Besides, as none of the prior optimization methods is adopted in equal-sized cutting processes, each node in the partial trees can be easily indexed by a string of bits, which can be used as an array key during lookups and updates. For each subset, FiCuts continues its cutting processes until the number of rules is less than the threshold for linear search or the cutting space is smaller than the threshold of *small field*. Take the subsets shown in Figure 6 as an example, FiCuts just works fine for the second subset: it builds the whole tree as illustrated in Figure 6(e), in which all rules are partitioned into leaf nodes. However, Figure 6(d) shows a different scenario, where pure FiCuts does not solve the problem completely, and only a partial tree can be constructed. When FiCuts reaches the rightmost cutting subspace in Figure 6(a), it is no longer effective by continuing cutting along *Field X*, because the cutting space in *Field X* is now smaller than T_X. Therefore, it is necessary to resort to other more effective methods to continue tree constructions at small scales.

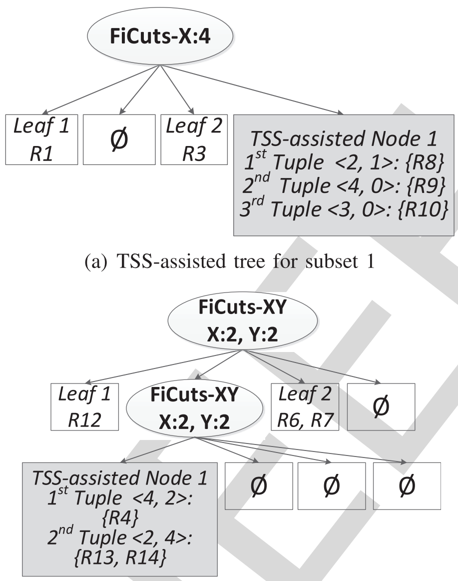
2) Post-TSS: Tuple Space Assisted Cutting Trees: After the first stage of pre-cuttings, two types of terminal nodes will be generated in the built partial trees: *leaf node* (i.e., #rules $\leq$ *binth*) and *non-leaf terminal node* (i.e., #rules $>$ *binth*). As a very limited number of rules are contained in leaf nodes, we can simply conduct a linear search on rules as in traditional decision trees. Thus, the second stage is mainly designed to handle packet classification on *non-leaf terminal node*. It is not difficult to see that the searching space has been separated into much smaller subspaces after pre-cuttings, where each subspace contains much fewer rules compared with the original rule set. On the other hand, a small space means long address prefixes or less nesting levels of ranges, both indicating a very limited tuple space. Based on this property, we employ the PSTSS for rules in the *non-leaf terminal nodes* to facilitate tree constructions. Thus, for the two partial trees shown in Figure 6, we can build their complete trees without any rule replications as illustrated in Figure 7.



(a) Subset 1: Fixed cuttings on field X (b) Subset 2: Fixed cuttings on field Y (c) Subset 3: Fixed cuttings on field X&Y

(d) Partial tree for subset 1 (e) Whole tree for subset 2 (f) Partial tree for subset 3

Fig. 6. The first stage partial trees built by FiCuts (*MAXCUTS* = 4, *binth* = 2).



(a) TSS-assisted tree for subset 1

(b) TSS-assisted tree for subset 3

Fig. 7. The complete TSS-assisted decision trees in CutTSS.

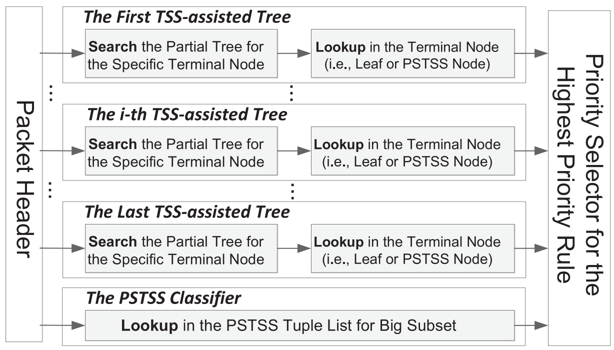
Up to now, three complete decision trees have been built for all rules given in Table V, as shown in Figure 6(e) and Figure 7. Overall, by exploiting the benefits of decision tree and TSS techniques adaptively, CutTSS can build TSS-assisted decision trees without any rule replications, thereby enabling fast updates and linear memory consumption.

3) Refined Optimizations: To further improve the performance, several optimizations have been adopted in our implementation as follows:

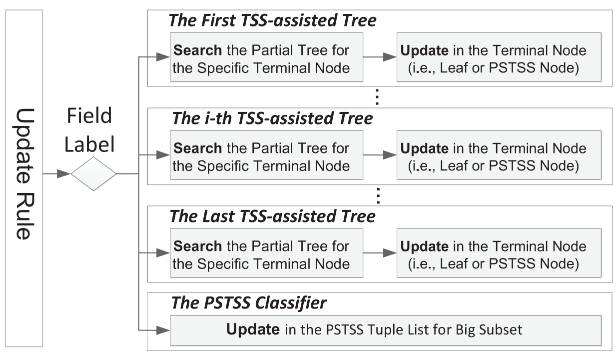
–Optimization 1: Priority sorting on partitioned subsets. For each incoming packet, CutTSS requires searching on every partitioned subset, even if a rule has been matched in an early subset. We improve on this by tracking the priority of partitioned subsets as that in PSTSS and PartitionSort algorithms, where the priority of each subset is the maximum priority of all the rules in it. By searching from greatest to least maximum priority on subsets, each lookup can terminate as soon as a rule is matched in an early subset.

–Optimization 2: Dynamic thresholds on terminal leaf nodes. For the terminal nodes after pre-cuttings, we adopt a dynamic threshold to distinguish leaf nodes and non-leaf nodes. The idea of this optimization is derived from the performance comparison for a lookup between linear search and TSS search. For example, the latest version of Open vSwtich (http://www.openvswitch.org) implements the PSTSS based on a variant of cuckoo hash [41], [42], where multiple hash lookups are required for each TSS lookup in Open vSwtich, which is much more complex and time-consuming than a linear search. Assume that each TSS lookup takes N times than a linear rule search, we can set the threshold as $N*M$, where M is the number of tuples in the terminal node.

–Optimization 3: Greedy thresholds on small fields. Essentially, *small field* is a relative concept of space scale. It is not difficult to see that narrower *small fields* may enable more effective pre-cuttings and less tuple spaces in *non-leaf terminal nodes*. However, narrower *small fields* may also lead to more rules in the *big subset* as illustrated in Figure 4, which may in turn increase the number of tuples in the *big subset*. To make a good trade-off, we select the thresholds on *small fields* by running a greedy algorithm during partitioning. The strategy of



(a) Packet classification



(b) Rule update

Fig. 8. The framework of classification and update in CutTSS.

selecting thresholds in our implementation is simple: choose one that achieves the least average memory access.

D. Decision Tree Operation: Classification & Update

In this subsection, we complete the picture of CutTSS from the following aspects: packet classification and rule update.

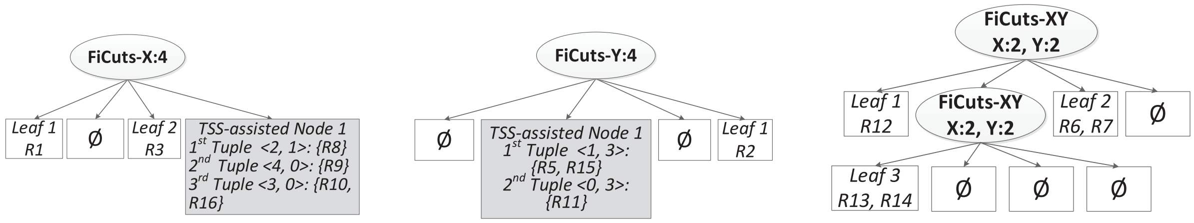
1) Packet Classification: For each incoming packet, CutTSS classifies the packet based on the framework shown in Figure 8(a). For each decision tree, CutTSS conducts classification in two steps: (1) Search the partial tree to find a terminal node; (2) Lookup for the best matching rule from the matched terminal node. Assuming that a 2-field incoming packet is $P_i = $ <1000, 0010>, we next give a working example for the rule set shown in Table V, where three decision trees are built as shown in Figure 6(e) and Figure 7: (1) For the decision tree shown in Figure 6(e), P_i can traverse this tree based on its first two bits in *Field Y* (i.e., 00). Thus, the first child node is found, and no rule is matched in this subset; (2) For the decision tree shown in Figure 7(a), P_i can traverse this tree based on its first two bits in *Field X* (i.e., 10). Thus, the third child node is matched, and R_3 is the best matching rule based on linear search; (3) For the decision tree shown in Figure 7(b), P_i can traverse this tree based on its first bit in *Field Y&X* (i.e., 0&1). Thus, the second child node is matched, and R_{13} is the best matching rule based on PSTSS search. Finally, R_3 with a higher priority will be the best matching rule for P_i.

2) Rule Update: For each updated rule, CutTSS updates the rule based on the framework shown in Figure 8(b). Unlike the above packet classification where all subsets have to be searched, CutTSS can perform each rule update just in a single subset, because the updated or inserted rule can only appear in a specific subset in CutTSS, depending on its field label vector. CutTSS performs rule updates in a tree in two steps: (1) Search the partial tree to find a terminal node; (2) Update (e.g., insert or delete) the rule pointed by the matched terminal node. When searching the partial tree for rule updates, the specific bits in each rule's *small fields* are used as a key for searching. Assuming that there are three update operations as follows: (1) Delete rule $R_4 = $ <1001, 00**>; (2) Insert rule $R_{15} = $ <1***, 010*>; (3) Insert rule $R_{16} = $ <110*, *>, we next give a working example for the rule set shown in Table V. By calculating the field label of R_4 (i.e., <*small*, *small*>), we known that R_4 may only appear in the decision tree shown in Figure 7(b), which is built for the subset shown in Figure 5(d). Then, R_4 can traverse this tree based on its first bit in *Field Y&X* (i.e., 0&1). Thus, the second child node is matched, and then R_4 will be updated in this terminal node. After removing R_4 from the PSTSS classifier, the number of rules in this node is reduced to the threshold of *binth*. Thus, we can replace this *non-leaf terminal node* with a new leaf node as shown in Figure 9(c). Similarly, we can first calculate the field label of R_{15} (i.e., <*big*, *small*>) and R_{16} (i.e., <*small*, *big*>), and then conduct updates as R_4 in the corresponding trees shown in Figure 6(e) and Figure 7(a), as illustrated in Figure 9.

E. Rationale Behind Effectiveness

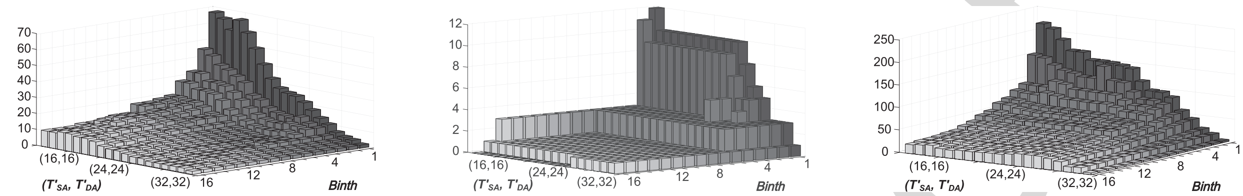
To reveal the rationale behind the effectiveness of CutTSS, we next give more insights from both theoretical and experimental aspects as follows.

1) Theoretical Analysis: Essentially, CutTSS is a two-stage tree framework built from the following two stages: (1) Coarse-grained pre-cutting with low memory consumption; (2) Fine-grained post-TSS with high performance. For the first-stage pre-cuttings in CutTSS, rule replications can be avoided completely, thereby enabling linear memory consumption for the partial trees. For the following tree constructions, CutTSS adopts PSTSS with a linear memory consumption to handle packet classification in *non-leaf terminal nodes*. Thus, for a F-dimensional subset containing N distinct rules, the memory consumption of CutTSS is $\Theta(N)$, which is the best theoretical bound proved in previous work as described in Section II(A). For each incoming packet or updated rule, CutTSS performs packet classification or rule update in two steps: (1) Search the partial tree based on the specific bits in each packet or rule in $\Theta(1)$ time; (2) Perform classification or update in the matched terminal node containging M rules (M$\leq$N). Based on the above Section II(A), we can conclude that the worst-case time complexity of CutTSS is $\Theta((logM)^{F-1})$. Thus, compared to the theoretical worst-case time complexity (i.e., $\Theta((logN)^{F-1})$), CutTSS achieves $\Theta((\log_M N)^{F-1})$ times improvement. We then consider the average worst-case time complexity of CutTSS as follows: Assuming that all rules are evenly distributed, the width and the threshold value of the *small field* are

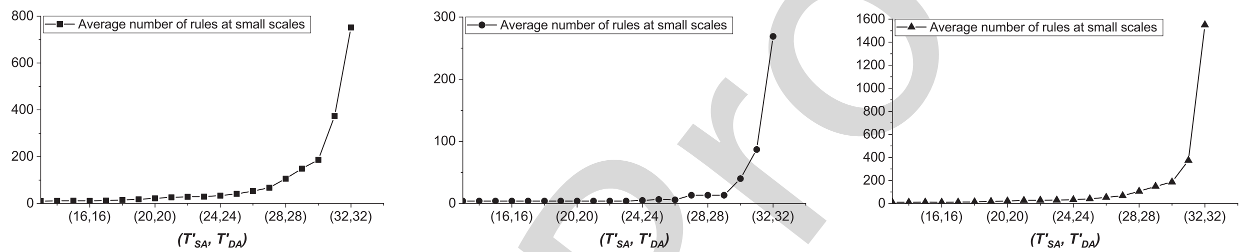


(a) Tree for subset 1: ($small_X$, big_Y) (b) Tree for subset 2: (big_X, $small_Y$) (c) Tree for subset 3: ($small_X$, $small_Y$)

Fig. 9. The new decision trees after three rule updates.



(a) Seed-acl(752): *#non-leaf terminal nodes* (b) Seed-fw(269): *#non-leaf terminal nodes* (c) Seed-ipc(1550): *#non-leaf terminal nodes*



(d) Seed-acl(752): Average number of rules (e) Seed-fw(269): Average number of rules (f) Seed-ipc(1550): Average number of rules

Fig. 10. Rule distribution density at different sized small scales.

2^W and 2^T, we can conclude that the average worst-case time complexity of CutTSS is $\Theta((logM_A)^{F-1})$, where M_A = N*$2^{(T-W)}$. Thus, from the perspective of theoretical analysis, the rational behind the effectiveness of CutTSS is essentially to perform the packet classification in a subspace at small scales that contains fewer rules. Although the theoretical bounds tell us that it is infeasible to design a single algorithm that can perform well in all cases, real-life classifiers have some inherent characteristics that can be exploited to reduce the complexity. Next, we give more insights from the aspect of experimental analysis.

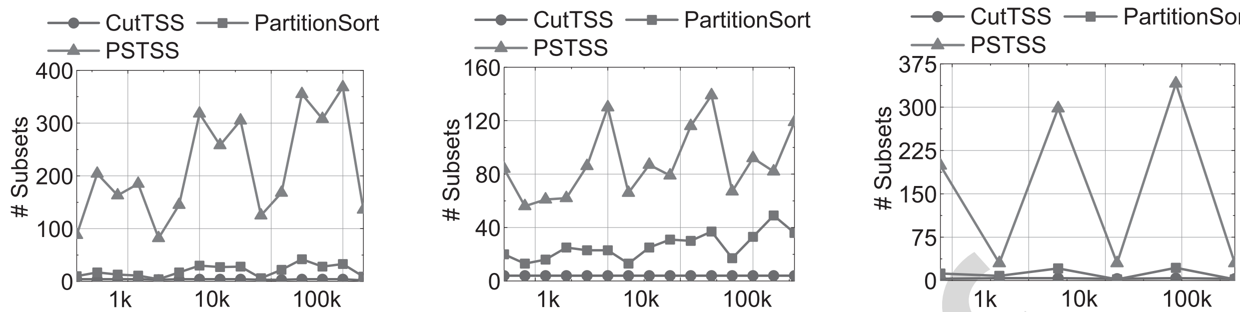
2) Experiential Analysis: We conduct experiential analysis based on the above three seed rule sets, to show more insights on the feature of rule distribution from two aspects: (1) Number of *non-leaf terminal nodes* at small scales; (2) Average number of rules at small scales. Take the subset shown in Figure 5(d) as an example, we can say that six rules are concentrated at three (over 4*4 = 16) distinct subspaces at small scales and the average number of rules is two. Among the three subspaces, only one of them contains rules more than *binth*, which will be handled by PSTSS in the tree. Based on this example, we now give more details about experiential analysis. Figure 10(a), (b) and (c) shows the number of subspaces containing rules more than *binth* at different sized small scales. We can see that although rules are distributed in many subspaces, the vast majority of them contain a small number of rules. In other words, the number of *non-leaf terminal nodes* in CutTSS is much smaller than the number of leaf nodes in the trees, thereby making CutTSS more like a traditional decision tree which can achieve high performance on classification inherently. That's why we call this tree as a TSS-assisted tree in CutTSS. Figure 10 (d), (e) and (f) shows the average number of rules over all subspaces that contain rules. We can see that even under very loose thresholds, the number of rules after the first stage pre-cutting is much smaller than the original rule set size, thereby enabling high performance on both search and update.

IV. Experimental Results

In this section, we present some experimental results of CutTSS. We start with an overview of our experimental methodology. After that, we evaluate our algorithm from the following key aspects: *tree construction*, *packet classification* and *rule update* respectively.

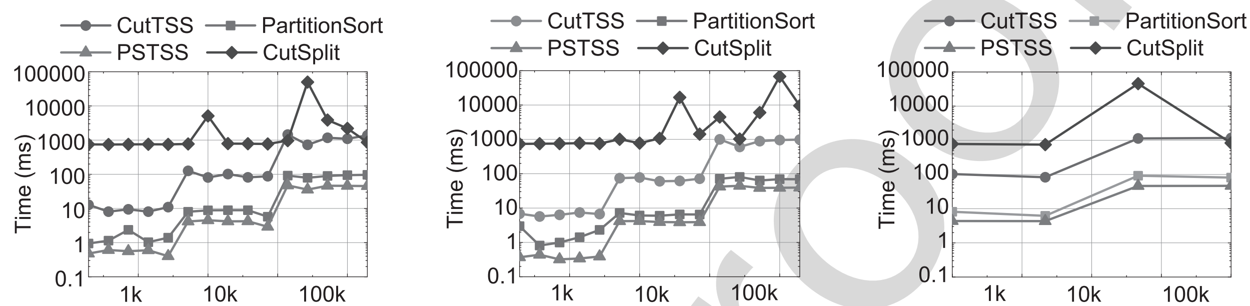
A. Experimental Methodology

We compare CutTSS with three algorithms: PSTSS, CutSplit and PartitionSort. Priority Sorting Tuple Space Search (PSTSS) is the algorithm with the fastest update performance, which is used in Open vSwitch for flow table lookups. CutSplit is the state-of-the-art decision tree with the



(a) ACL (b) FW (c) IPC

Fig. 11. Number of partitioned subsets.



(a) ACL (b) FW (c) IPC

Fig. 12. Construction time.

fastest classification performance. ParitionSort is the state-of-the-art splitting based tree with the best performance trade-off between classification and update. To facilitate fair comparison, we have made some modifications to the open-source code of the other three algorithms, and their performances are essentially not affected by our modification. We are very grateful to the authors of these algorithms, their open-source codes and selfless personal help enable us to make a fair and justifiable comparison. As a response, our implementation of CutTSS is also publicly available on our website (http://www.wenjunli.com/CutTSS).

1) Rule Sets: The rule sets used in our experiments are generated using ClassBench, whose size varies from 1k to 100k. There are three types of rule sets: ACL, FW and IPC. Each rule set is named by its type and size, e.g., FW_1k refers to the firewall rule set with about 1000 rules. For each size, we generate 12 rule sets respectively based on 12 seed parameter files (i.e, 5 ACL, 5 FW and 2 IPC) in ClassBench [30].

2) Simulation Environment: We measure classification time by classifying all packets in trace files generated by ClassBench when it constructs the corresponding rule set. In order to evaluate the actual lookup performance of classification algorithms, we conduct experiments by omitting caching in the fast path and consider only slow path classification for each incoming packet. To evaluate the performance of the incremental update, we measure update time as the time required to conduct one rule insertion or deletion. For each rule set, we shuffle rules randomly to generate a sequence of update operations, where half of the insertions are randomly mixed with half of the deletions.

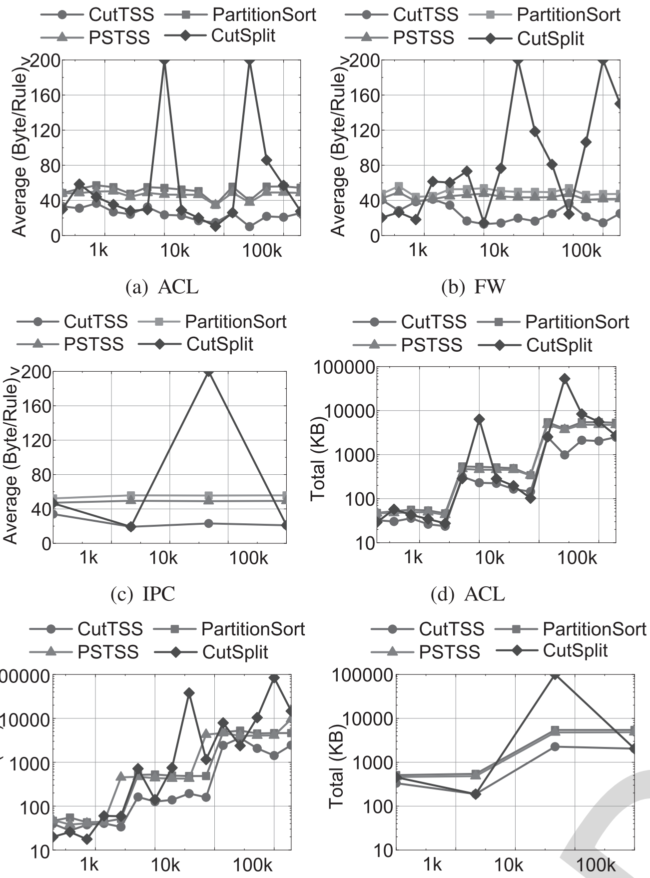
3) Machine Environment: All experiments are run on a machine with AMD Radeon 5-2400G CPU@3.6GHz and 8G DRAM. The operating system is Ubuntu 16.04. To reduce the CPU jitter error, we take the average results by running ten times for each evaluation circularly.

B. Evaluation on Construction

1) Number of Subsets: Since the number of partitioned subsets in CutSplit is the same as in CutTSS, Figure 11 shows the number of subsets in CutTSS, PSTSS and PartitionSort. We find that CutTSS produces a relatively stable number of subsets regardless of the type and size of rule sets, averaging at 3.7 subsets across all of the rule sets. This favorable property makes CutTSS more suitable for concurrency. In contrast, the number of partitioned subsets in PSTSS and PartitionSort ranges from 2 to 368 with an average of 151.7 and 20.9 subsets respectively.

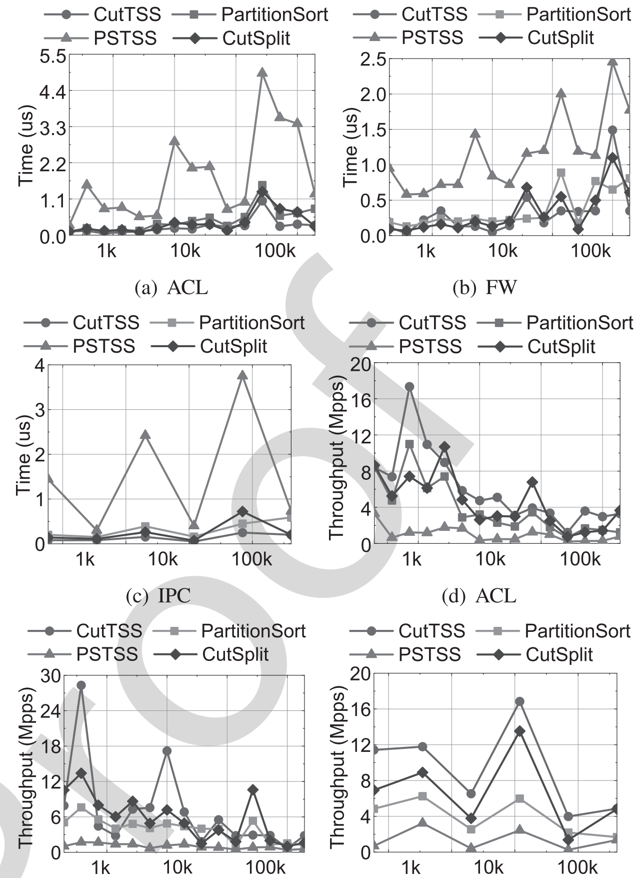
2) Construction Time: Figure 12 shows the construction time of CutTSS as well as PSTSS, PartitionSort and CutSplit. Clearly, PSTSS is the fastest one among them. In contrast, CutTSS takes a little more time than PSTSS because of its partial tree constructions in the pre-cutting stage. However, even for the rule sets up to 100k, CutTSS can still build decision trees in about one second, much faster than previous decision trees such as EffiCuts and SmartSplit that require almost ten minutes. We can also find that the construction time of CutTSS increases almost linearly with the rule set size, which makes it well suitable for larger classifiers.

3) Memory Consumption: Figure 13 shows the memory consumption of CutTSS as well as PSTSS, PartitionSort



(a) ACL (b) FW (c) IPC (d) ACL (e) FW (f) IPC

Fig. 13. Memory consumption.



(a) ACL (b) FW (c) IPC (d) ACL (e) FW (f) IPC

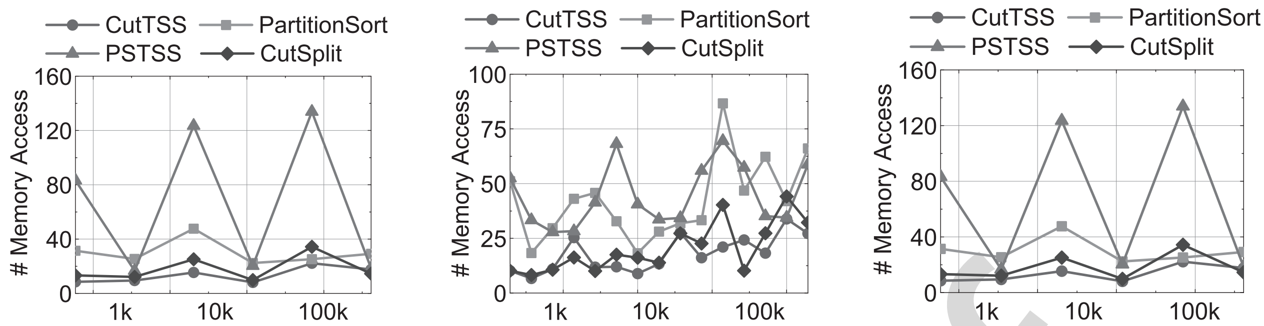
Fig. 14. Classification performance.

and CutSplit. Our experimental results show that our CutTSS requires less space than other algorithms, consuming an average of 25.8 Byte/Rule across all of the rule sets, while it requires 45.4 Byte/Rule, 50.9 Byte/Rule and 243.2 Byte/Rule in PSTSS, PartitionSort and CutSplit respectively. We can also find that, the memory consumption of CutTSS increases almost linearly with the rule set size, which makes it well suitable for larger classifiers.

C. Evaluation on Classification

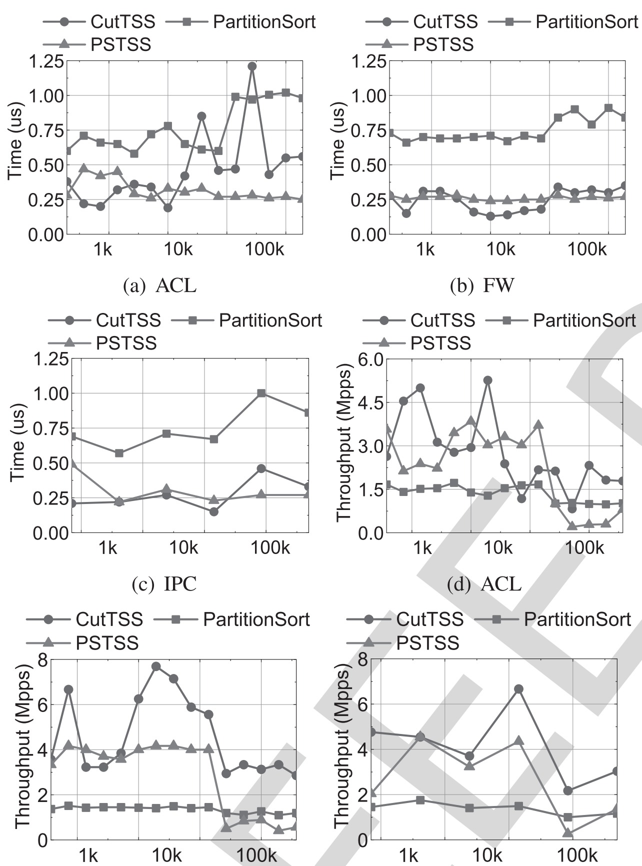
1) Average Classification Time: Figure 14 shows the average classification time and throughput of CutTSS as well as PSTSS, PartitionSort and CutSplit. In order to compare the performance of these algorithms, we first compute the average times for three different types of rules respectively, and then compute the ratio based on these average times. From Figure 14(a), (b) and (c), we can see that CutTSS requires less time to classify packets, with an average of 0.257 us, 0.318 us and 0.135 us for each type of rule set respectively, while PSTSS consumes an average of 1.765 us, 1.164 us and 1.506 us respectively. Thus, CutTSS achieves an average of 6.868 times, 3.661 times and 11.156 times speed-up on classification performance than PSTSS respectively, almost an order-of-magnitude improvement on classification time on average. Additionally, the experimental results show that CutTSS achieves 1.43 times and 1.89 times speed-up than CutSplit and PartitionSort respectively. It should be noted that, although there are much more subsets in PartitionSort, it can still achieve comparable performance to CutTSS. The reason is that, almost all the rules are concentrated in the first few subsets when ordered by maximum priority, so that most lookups in PartitionSort can terminate as soon as a rule is matched in the first few subsets.

2) Average Throughput: From Figure 14(d), (e) and (f), we can see that CutTSS achieves an average throughput of 6.013 Mpps (Million packets per second), 6.782 Mpps and 9.235 Mpps for each type of rule set respectively, while PSTSS achieves an average of 0.994 Mpps, 1.016 Mpps and 1.396 Mpps respectively. Thus, CutTSS achieves an average of 6.049 times, 6.675 times and 6.615 times improvement on throughput than PSTSS respectively. Compared to CutSplit and PartitionSort, CutTSS also achieves 1.304 times and 1.878 times improvement respectively across all of the rule sets. We can also see an interesting phenomenon in Figure 14 that the proposed CutTSS has much higher performance for a few rule sets, such as the second rule set and the seventh rule set in Figure 14(e). Actually, this phenomenon is caused by the characteristic of the seed parameter file in ClassBench. In Figure 14(e), the second, the seventh and the twelfth rule sets are generated based on the same seed parameter file, but



(a) ACL (b) FW (c) IPC

Fig. 15. Average memory access.



(a) ACL (b) FW (c) IPC (d) ACL (e) FW (f) IPC

Fig. 16. Update performance.

with different sizes. By checking the type of terminal nodes after pre-cuttings, we find that the ratio of *non-leaf terminal node* in these three rule sets is much less than that in other rule sets, meaning that the rules generated based on this specific seed file are more evenly distributed than others. Thus, most of the rules in these rule sets can be separated into leaf nodes and be searched with linear search as traditional decision trees. However, this phenomenon does not exist for the twelfth rule set in Figure 14(e), the reason is that, when the rule set contains more and more rules, there will be more and more tuples needed to be searched in *big subset*, which may become the performance hurdle of CutTSS.

3) Average Memory Access: Figure 15 shows the average memory access of CutTSS as well as PSTSS, PartitionSort and CutSplit. Note that we think traversing a tree node, a rule or a tuple as one memory access in our experiments. It is obvious that CutTSS is significantly better than other three algorithms. Compared to PSTSS, experimental results show that CutTSS achieves an average of 3.8 times reduction on the number of memory accesses. Compared to PartitionSort and CutSplit, CutTSS also achieves 2.3 times and 1.2 times improvement on average.

D. Evaluation on Incremental Update

Since CutSplit can not support fast incremental updates, we just evaluate update performance among CutTSS, PSTSS and PartitionSort. Figure 16 shows the average incremental update time and throughput of CutTSS as well as PSTSS and PartitionSort. From Figure 16(a), (b) and (c), we can see that CutTSS requires less time to update rules, achieving an average of 0.464 us, 0.246 us and 0.273 us for each type of rule set respectively, while PSTSS consumes an average of 0.314 us, 0.261 us and 0.301 us respectively. Additionally, our experimental results also show that, CutTSS achieves an average of 2.516 times speed-up on update time than PartitionSort across all of the rule sets. From Figure 16(d), (e) and (f), we can see that both CutTSS and PSTSS can achieve high throughput for updates, achieving at an average of 3.734 Mpps and 3.583 Mpps respectively. Thus, CutTSS has comparable update performance to PSTSS, which is used in Open vSwitch.

V. Conclusion

Open vSwitch implements a variant of TSS instead of decision tree-based algorithms despite their better performance on lookups, because the latter have poor support for fast incremental updating of rules, which is an important metric for SDN switches. However, TSS-based schemes can achieve fast updates but have a performance concern.

To achieve fast lookup and update at the same time, we propose CutTSS, a two-stage framework consisting of heterogeneous algorithms to adaptively exploit different characteristics of the rule sets at different scales. In the first stage, partial trees are constructed from rule subsets grouped with respect to their *small fields*. This grouping eliminates rule overlap at large scales, thereby enabling very efficient pre-cuttings without any rule replications. The second stage handles packet classification at small scales, where PSTSS is applied for these subsets to facilitate tree constructions. Overall, CutTSS exploits the strengths of both decision tree and TSS to circumvent their respective weaknesses. Experimental results show that CutTSS has comparable update performance to TSS in Open vSwitch, while achieving almost an order-of-magnitude improvement on classification performance over TSS.

References

[1] W. Li, X. Li, H. Li, and G. Xie, "CutSplit: A decision-tree combining cutting and splitting for scalable packet classification," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2645–2653.

[2] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," in *Proc. ACM SIGCOMM*, 2008, pp. 69–74.

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

[4] H. J. Chao and B. Liu, *High Performance Switches Routers*. Hoboken, NJ, USA: Wiley, 2007.

[5] C. R. Meiners, A. X. Liu, and E. Torng, *Hardware Based Packet Classification for High Speed Internet Routers*. Berlin, Germany: Springer-Verlag, 2010.

[6] H. Che, Z. Wang, K. Zheng, and B. Liu, "DRES: Dynamic range encoding scheme for TCAM coprocessors," *IEEE Trans. Comput.*, vol. 57, no. 7, pp. 902–915, Jul. 2008.

[7] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 18, no. 2, pp. 490–500, Apr. 2010.

[8] B. Vamanan and T. Vijaykumar, "TreeCAM: Decoupling updates and lookups in packet classification," in *Proc. ACM CoNEXT*, 2011, pp. 1–12.

[9] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1127–1140, Jun. 2013.

[10] O. Rottenstreich *et al.*, "Compressing forwarding tables," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1231–1239.

[11] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster, "SAX-PAC (scalable and expressive packet classification)," in *Proc. ACM SIGCOMM*, 2014, pp. 15–26.

[12] O. Rottenstreich and J. Tapolcai, "Lossy compression of packet classifiers," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, May 2015, pp. 39–50.

[13] O. Rottenstreich, I. Keslassy, A. Hassidim, H. Kaplan, and E. Porat, "Optimal In/Out TCAM encodings of ranges," *IEEE/ACM Trans. Netw.*, vol. 24, no. 1, pp. 555–568, Feb. 2016.

[14] O. Rottenstreich and J. Tapolcai, "Optimal rule caching and lossy compression for longest prefix matching," *IEEE/ACM Trans. Netw.*, vol. 25, no. 2, pp. 864–878, Apr. 2017.

[15] P. Gupta and N. McKeown, "Packet classification using hierarchical intelligent cuttings," in *Proc. IEEE Hot Interconnects*, May 1999, pp. 1–9.

[16] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet classification using multidimensional cutting," in *Proc. Conf. Appl., Technol., Archit., Protocols Comput. Commun. (SIGCOMM)*, 2003, pp. 213–224.

[17] Y. Qi, L. Xu, B. Yang, Y. Xue, and J. Li, "Packet classification algorithms: From theory to practice," in *Proc. IEEE 28th Conf. Comput. Commun. (INFOCOM)*, Apr. 2009, pp. 648–656.

[18] B. Vamanan, G. Voskuilen, and T. Vijaykumar, "EffiCuts: Optimizing packet classification for memory and throughput," in *Proc. ACM SIGCOMM*, 2010, pp. 207–218.

[19] J. Fong, X. Wang, Y. Qi, J. Li, and W. Jiang, "ParaSplit: A scalable architecture on FPGA for terabit packet classification," in *Proc. IEEE 20th Annu. Symp. High-Perform. Interconnects*, Aug. 2012, pp. 1–8.

[20] W. Li and X. Li, "HybridCuts: A scheme combining decomposition and cutting for packet classification," in *Proc. IEEE 21st Annu. Symp. High-Perform. Interconnects*, Aug. 2013, pp. 41–48.

[21] P. He, G. Xie, K. Salamatian, and L. Mathy, "Meta-algorithms for software-based packet classification," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Oct. 2014, pp. 308–319.

[22] S. Yingchareonthawornchai, J. Daly, A. X. Liu, and E. Torng, "A sorted partitioning approach to high-speed and fast-update OpenFlow classification," in *Proc. IEEE 24th Int. Conf. Netw. Protocols (ICNP)*, Nov. 2016, pp. 1–10.

[23] J. Daly and E. Torng, "ByteCuts: Fast packet classification by interior bit extraction," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2654–2662.

[24] W. Li, T. Yang, Y.-K. Chang, T. Li, and H. Li, "TabTree: A TSS-assisted bit-selecting tree scheme for packet classification with balanced rule mapping," in *Proc. ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Sep. 2019, pp. 1–8.

[25] E. Liang, H. Zhu, X. Jin, and I. Stoica, "Neural packet classification," in *Proc. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2019, pp. 256–269.

[26] V. Srinivasan, S. Suri, and G. Varghese, "Packet classification using tuple space search," in *Proc. ACM SIGCOMM*, 1999, pp. 35–146.

[27] J. Daly and E. Torng, "TupleMerge: Building online packet classifiers by omitting bits," in *Proc. 26th Int. Conf. Comput. Commun. Netw. (ICCCN)*, Jul. 2017, pp. 1–10.

[28] B. Pfaff *et al.*, "The design and implementation of open vswitch," in *Proc. USENIX NSDI*, 2015, pp. 117–130.

[29] M Kuźniar, P Perešíni, and D Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, 2015, pp. 347–359.

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

[31] M. H. Overmars and F. A. van der Stappen, "Range searching and point location among fat objects," *J. Algorithms*, vol. 21, no. 3, pp. 629–656, Nov. 1996.

[32] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, "Fast and scalable layer four switching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 191–202, Oct. 1998.

[33] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 203–214, Oct. 1998.

[34] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proc. ACM SIGCOMM*, 1999, pp. 147–160.

[35] A. Feldman and S. Muthukrishnan, "Tradeoffs for packet classification," in *Proc. IEEE Conf. Comput. Commun. 19th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, Mar. 2000, pp. 1193–1202.

[36] F. Baboescu and G. Varghese, "Scalable packet classification," in *Proc. ACM SIGCOMM*, 2001, pp. 199–210.

[37] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducing of field labels," in *Proc. IEEE INFOCOM*, Mar. 2005, pp. 269–280.

[38] Y. Qi, B. Xu, F. He, B. Yang, J. Yu, and J. Li, "Towards high-performance flow-level packet processing on multi-core network processors," in *Proc. 3rd ACM/IEEE Symp. Archit. Netw. Commun. Syst. (ANCS)*, Dec. 2007, pp. 17–26.

[39] C.-L. Hsieh and N. Weng, "Many-field packet classification for software-defined networking switches," in *Proc. Symp. Archit. Netw. Commun. Syst. (ANCS)*, Mar. 2016, pp. 13–24.

[40] T. Yang, A. X. Liu, Y. Shen, Q. Fu, D. Li, and X. Li, "Fast OpenFlow table lookup with fast update," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 2636–2644.

[41] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[42] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," in *Proc. 9th ACM Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2013, pp. 97–108.

Wenjun Li (Member, IEEE) received the B.Sc. degree from the University of Electronic Science and Technology of China in 2011 and the M.Sc. degree from Peking University in 2014, where he is currently pursuing the Ph.D. degree with the School of Electronics Engineering and Computer Science. From 2014 to 2015, he worked as a Researcher with the Network Research Department, Huawei Technologies Company, Ltd. His research interests focus on networking algorithms, such as packet classification, IP lookups, and sketches. He is a member of ACM and CCF.

Tong Yang received the Ph.D. degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS), China, from July 2013 to July 2014. He is currently an Associate Professor with the Department of Computer Science and Technology, Peking University. His research interests focus on networking algorithms, such as sketches, IP lookups, and Bloom filters. He published articles in SIGCOMM, SIGKDD, SIGMOD, JSAC, ToN, and so on.

Ori Rottenstreich received the B.Sc. degree in computer engineering *(summa cum laude)* and the Ph.D. degree from Technion in 2008 and 2014, respectively. He is currently an Assistant Professor with the Department of Computer Science and the Department of Electrical Engineering of the Technion, Haifa, Israel. From 2015 to 2017, he was a Post-Doctoral Research Fellow with the Department of Computer Science, Princeton University. His main research interest is computer networks.

Xianfeng Li received the B.S. degree from the School of Computer and Control Engineering, Beijing Institute of Technology, in 1995, and Ph.D. degree in computer science from the National University of Singapore, in 2005. He worked as an Associate Professor with the Peking University Shenzhen Graduate School. He is currently an Associate Professor with Macau University of Science and Technology (MUST). His research interests include software-defined networks, codesign of hardware and software, and the Internet of Things.

Gaogang Xie received the B.Sc. degree in physics and the M.S. and Ph.D. degrees in computer science from Hunan University. From 2002 to 2019, he was with the Institute of Computing Technology, Chinese Academy of Sciences (CAS). He is currently a Professor with the Computer Network Information Center (CNIC), CAS, and the University of Chinese Academy of Sciences. His research interests include Internet architecture, packet processing and forwarding, and Internet measurement.

Hui Li received the B.Eng. and M.S. degrees from the School of Information Engineering, Tsinghua University, Beijing, China, in 1986 and 1989, respectively, and Ph.D. degree from the Department of Information Engineering, The Chinese University of Hong Kong, in 2000. He was the Director of the Shenzhen Key Lab of Information theory and Future Internet architecture, the Director of the PKU Lab of China Environment for Network Innovations (CENI), National Major Research Infrastructure. He is currently a Full Professor of the Peking University Shenzhen Graduate School. He proposed the first co-governing future networking MIN based on blockchain technology and implemented its prototype on Operator's Network in the world, and this project MIN: Co-Governing MultiIdentifier Network Architecture and Its Prototype on Operator's Network was obtained the Award of World Leading Internet Scientific and Technological Achievements by the 6th World Internet Conference on 2019, Wuzhen, China. His research interests include network architecture, cyberspace security, distributed storage, and blockchain.

Balajee Vamanan received the Ph.D. degree from Purdue University in 2015. Prior to his graduate study, he worked in NVIDIA as a Design Engineer. He is currently an Assistant Professor with the Department of Computer Science, University of Illinois at Chicago (UIC). His research interests span various aspects of computer networks and computer systems.

Dagang Li received the bachelor's degree in telecommunications engineering from the Huazhong University of Science and Technology, Wuhan, China, and the Ph.D. degree from Katholieke Universiteit Leuven, Leuven, Belgium. He is currently an Assistant Professor with the School of Electronic and Computer Engineering, Peking University Shenzhen Graduate School, Shenzhen, China. His main research interest is computer networks.

Huiping Lin is currently pursuing the bachelor's degree with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China. She is majoring in computer science and technology advised by Prof. T. Yang. Her research interests include computer networks, network measurement, hash algorithms and sketches. She has received the Merit Student Award and a Scholarship from Peking University.