Palmtrie: A Ternary Key Matching Algorithm for IP Packet Filtering Rules

Hirochika Asai
panda@wide.ad.jp
WIDE Project

ABSTRACT

Network security has become crucial to our society and industry. A firewall is an essential function in network operations for security enhancement. Network access control lists (ACLs) have been used to describe multi-layer security rules to determine whether a packet is passed or dropped by the firewall. An entry in ACLs contains so-called *don't care* bits, and consequently, ACL matching is generalized as a ternary matching problem. As ternary matching typically relies on dedicated hardware, high-performance ternary matching with commodity CPUs is challenging. We propose a practical algorithm for network ACL matching trie, named Palmtrie, that allows the multi-bit stride extension to achieve better performance. We evaluate the Palmtrie using synthetic ACLs that emulate an existing campus network and Internet backbone policies. The evaluation results demonstrate that the Palmtrie outperforms the existing ACL matching algorithms on extensive ACLs. For example, the lookup performance of the optimized Palmtrie achieves 4.76 times faster than the algorithm implemented in the widely used library (DPDK) for the scanning attack traffic on an ACL with one million entries. We also prove that the Palmtrie solves the problem with the build time of data structures in the existing algorithms.

CCS CONCEPTS

• **Networks** → **Packet classification**; • **Security and privacy** → *Firewalls.*

KEYWORDS

ternary matching, packet classification, firewalls, access control list

ACM Reference Format:
Hirochika Asai. 2020. Palmtrie: A Ternary Key Matching Algorithm for IP Packet Filtering Rules. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386367.3431289

1 INTRODUCTION

Network security has gained in importance as the Internet has become a vital infrastructure. Network security devices such as firewalls have widely been deployed to protect information assets from unauthorized network access. Firewalls are categorized into two types; stateful and stateless firewalls. Stateful firewalls manage the states of individual flows and apply an action to each packet acting on the managed state. Some stateful firewalls implement deep packet inspection. In contrast, stateless firewalls do not store any states of flows but apply an action to an individual packet using its header information. Thus, stateless firewalls are generally more scalable than stateful firewalls.

A network access control list (ACL) is commonly implemented on network devices as a fundamental stateless firewall function. An ACL is a set of multi-layer security rules using the header information of data communication protocols to filter packets through the device. ACL matching is generalized as a ternary matching problem. A challenge in ternary matching lies in the uniqueness of wild-card bits. In order to handle the wild-card bits, the existing packet classification algorithms [19, 30, 35] cut the multidimensional space of ACL rules with wild-card bits and then build a decision tree using these cuts. However, these algorithms have not achieved acceptable performance with commodity CPUs and have suffered from the slowness of build time.

We propose a practical algorithm for ACL matching trie, named *Palmtrie*. We design the Palmtrie to support a multi-bit stride that reduces the trie depth by branching using multi-bit chunks of a key for better lookup performance. The basic idea of the Palmtrie is devised from a Patricia trie [24, 29, 31]. However, the originality and the significance of the Palmtrie mostly lie in this multi-bit stride extension. The optimized version of the Palmtrie, Palmtrie$^+$, adopts the technique derived from Poptrie [3] for memory efficiency. We evaluate the lookup performance using synthetic ACLs that emulate an existing campus network and Internet backbone policies. The evaluation results highlight the effectiveness of the multi-bit stride extension of the Palmtrie. They also demonstrate that the Palmtrie$^+$ is memory efficient and runs 1.05–4.76 times faster than the algorithm implemented in the widely used library (DPDK [11]) on ACLs for the campus network policy of various sizes. It also outperforms the existing algorithms for Internet backbone policies on extensive ACLs.

ACLs have statically been configured in routers in general. BGP Flowspec [20], however, allows advertising filtering rules to neighbor routers. They would be dynamically updated [34]. Thus, the incremental update or rebuild performance also becomes essential. We also evaluate the build time of the Palmtrie$^+$ for various sizes of ACLs. The results show that the Palmtrie$^+$ provides exemplary performance in building its data structure while the existing algorithms require unacceptable time on extensive ACLs.

The rest of this paper is organized as follows. We look over related work in Section 2. We explain the ternary matching problem, then revisit Patricia trie, and finally describe the data structure

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431289

and algorithm of a Palmtrie in Section 3. The performance of the implemented Palmtrie for synthetic ACLs and traffic patterns is evaluated in Section 4. We discuss the evaluation results and the real-world deployment of the Palmtrie, including IPv6 support, in Section 5. We conclude this paper in Section 6.

2 RELATED WORK

Network devices generally equip content addressable memory (CAM) or ternary content addressable memory (TCAM) as a lookup table for various packet forwarding functions. CAM is a type of dedicated memory for fast exact matching. For example, a MAC address table utilizes it for Ethernet switching. TCAM is a unique type of memory for ternary matching whose keys of entries allow so-called *don't care* bits. Longest prefix matching leverages TCAM, particularly in IP routing. OpenFlow [23] and P4 [7] switches equip TCAM for their generalized packet forwarding engine. An ACL used as a stateless firewall function is another application of TCAM.

TCAM has been a standard technology for lookup tables in network devices for a long time [1, 21, 37, 39]. However, it has problems with its power consumption, heat, monetary cost, and scalability issues [5, 17]. Moreover, the emergence of network function virtualization (NFV) [13] gains importance in network algorithms running as software on commercial off-the-shelf (COTS) devices. Fast input/output (I/O) technologies such as DPDK [11] and netmap [27] with NFV have accelerated the research and development of high-performance packet forwarding engines on COTS devices.

Many network algorithms have been researched to implement various network functions on COTS devices instead of dedicated hardware such as CAM and TCAM. Hash tables such as Cuckoo Hashing [25] and Hopscotch Hashing [16] have been used for exact matching. For example, CuckooSwitch [40] adopts Cuckoo Hashing to implement the Ethernet switching function. Many research studies on longest prefix matching algorithms have also been conducted. Longest prefix matching is a particular case of the ternary matching problem that don't care bits follow each prefix, and the priority is the same as the prefix length. This constraint of longest prefix matching allows us to devise fast IP routing table lookup algorithms. Most of the longest prefix matching studies such as radix tree [9, 18, 29], Patricia trie [24, 29, 31], path-compressed trie [32], DIR-24-8-BASIC [14], SAIL [36], and Poptrie [3] leverage search tree data structures. These data structures and algorithms based on search trees rely on the constraint that longest prefix matching searches the deepest leaf from a search tree to result in the highest priority. Another approach of longest prefix matching, such as DXR [38], handles IP address prefixes as ranges. Fast exact matching and longest prefix matching algorithms have successfully been developed. However, the ternary matching problem, including ACL matching, is still challenging.

Linux `iptables` and FreeBSD `pf` adopt the sorted list to perform priority encoding in ACL matching. Linux Socket Filtering (LSF) [28] and the Berkeley Packet Filter (BPF) [22] use a domain-specific assembly language and implement a just-in-time (JIT) compiler to execute the comparison of each ACL entry efficiently. However, the computational complexity of the sorted list search is the order of n, where n is the number of entries. Therefore, it is not scalable to extensive ACLs.

Packet classification algorithms such as HiCuts [15], HyperCuts [30], EffiCuts [35], and NeuroCuts [19] have been proposed to use SRAM or DRAM instead of TCAM for ACL matching. To this end, they cut the multidimensional space of ACL rules and then build a decision tree with these cuts. As these algorithms do not rely on TCAM, they can run on commodity CPUs. However, they cannot achieve good lookup performance on commodity CPUs. Moreover, these algorithms are not generalized for ternary matching because they assume that each field in an ACL rule is exact, prefix, or range matching. Therefore, they do not support ternary matching fields such as TCP flags.

DPDK [11] implements a packet classification library. It employs a similar approach to the packet classification algorithms based on a decision tree [4, 15, 19, 30, 35] to convert an ACL to a single trie or multiple tries [26]. It also adopts performance optimization techniques for commodity CPUs such as the multi-bit (8-bit) stride traversal. However, a significant drawback of this approach is the build time of the data structure, especially for extensive ACLs. For example, it takes more than three hours to build the data structure for an ACL with 279 thousand entries, as we will see in Section 4.4.

3 PALMTRIE

We propose a Palmtrie to solve the ternary matching problem. We first explain the ternary matching problem and ACLs as the target application of the ternary matching problem. We then revisit Patricia tries and explain the basic idea of the Palmtrie that supports ternary matching by extending a Patricia trie. Based on the basic idea of Palmtrie, we extend the Palmtrie to improve the lookup performance with a multi-bit stride. Extending the Palmtrie to support a multi-bit stride is not straightforward as the other trie data structures because of don't care bits in keys. We then analyze the challenge in extending the Palmtrie to support the multi-bit stride and thoroughly describe the data structure and algorithm of the Palmtrie with the multi-bit stride extension.

3.1 Ternary Matching Problem and Access Control Lists

The ternary matching problem is to find an entry with the highest priority that matches a given query key in the table. Each entry of a ternary matching table holds a key, a value, and its priority. Unlike the exact matching problem, the ternary matching problem allows wild-card bits, called don't care bits, in the bit string of the key in a ternary matching table. In this paper, the bit `*` in a bit string represents a don't care bit. For example, a ternary matching table might contain a key of `011*1000` that matches any query keys of `01101000` and `01111000`. Note that query keys used to find matching entries in a table are binary bit strings, i.e., they are not allowed to contain any don't care bits. As mentioned above, a query key might match multiple entries. Therefore, the ternary matching problem selects an entry with the highest priority from the matching entries.

To better understand the ternary matching problem, we show an example of a ternary matching table in Table 1. A higher number represents a higher priority in this paper. Here, we consider the case to look up the ternary matching table for the query key of `01110101`; the query key of `01110101` matches two entries with

Table 1: An example of a ternary matching table. Each entry stores a key, a value, and its priority. Entry IDs are for the explanation.

Entry ID	Key	Value	Priority
1	011*1000	1	6
2	1*0***10	2	8
3	0001****	3	9
4	10110011	4	3
5	0*1101**	5	7
6	1110****	6	4
7	010010**	7	5
8	01110***	8	2
9	1*******	9	1

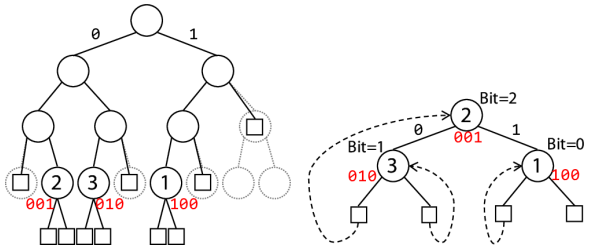
Table 2: An example of an ACL. This ACL permits outgoing packets from `192.0.2.0/24` incoming ICMP, DNS over UDP, and established TCP packets to `192.0.2.0/24`.

```
permit ip 192.0.2.0/24 0.0.0.0/0
permit icmp 0.0.0.0/0 192.0.2.0/24
permit udp 0.0.0.0/0 eq 53 192.0.2.0/24
permit tcp 0.0.0.0/0 192.0.2.0/24 established
deny ip 0.0.0.0/0 192.0.2.0/24
```

keys of `0*1101**` and `01110***`. Then, we compare their priority to select an entry with the highest priority. In these two entries, the entry with the key of `0*1101**` (i.e., Entry 5) has the highest priority, and thus, is finally returned as the lookup result.

Searching for entries from an ACL is one of the applications of the ternary matching problem. An ACL describes multi-layer security rules with actions to be applied to each packet. In general, ACL entries are written up by the following layer 2–4 header information; the destination and source Ethernet addresses, EtherType, IEEE 802.1Q (VLAN) tag information, source and destination IP addresses, a protocol number, source and destination TCP/UDP port numbers, and TCP flags. IP addresses are generally specified by prefix notation. Port numbers may be represented in ranges. TCP flags are represented as a ternary bit string, consisting of `0`, 1, and `*`. We exclude layer 2 rules for simplicity and only use layer 3 and 4 rules in the rest of this paper.

Table 2 shows an example of an ACL. This ACL assumes that `192.0.2.0/24` is the internal network and describes the rules to pass all outgoing packets, incoming ICMP, DNS responses whose source UDP port is 53, and established TCP packets. Any incoming packets other than the above rules are dropped. The entries in an ACL are sorted by priority, i.e., higher priority at the top of the list. An entry consists of a sequence of action (`permit` or `deny`), a protocol name (any protocols over IP (`ip`), ICMP (`icmp`), UDP (`udp`), or TCP (`tcp`)), a source IP address prefix, an optional source port number range following a range keyword (`eq` for *equal to*, etc.) for layer 4 protocols, a destination IP prefix, an optional destination port number range following a range keyword, and an optional keyword `established` for TCP. The keyword of `established` denotes TCP packets with ACK or RST flags in the TCP header. This



(a) Radix Tree (b) Patricia Trie

Figure 1: Radix tree and Patricia trie for three keys; 1) `100`, 2) `001`, and 3) `010`. The Patricia trie compresses its data structure by eliminating unary branching nodes from the radix tree. Instead, the Patricia trie adds a bit index to each node. Bit indices are denoted by *Bit=* in the figure. Solid circles and squares represent nodes with a value and pointers, respectively. Red numbers are the values stored in each node.

means that an ACL entry with the keyword of `established` is converted into two ternary matching entries of the TCP flags field, i.e., ****1**** (ACK) and ******1** (RST). In the same way, a port range is also converted into multiple entries. The link to the tool to convert ACL entries to ternary matching entries is referred to from the Palmtrie source code.

A naive approach to implement ACL matching is searching for a matching entry from the list sorted by the priority. However, the computational complexity is of the order of n, where n is the number of entries in the list. Therefore, it is not efficient for extensive ACLs.

3.2 Patricia Trie (Revisited)

Search trees have widely been used for finding an entry corresponding to a query key. A radix tree and a Patricia trie are fundamental search tree data structures. These search trees are called tries. In a trie, any descendent nodes of a node have a common prefix of the node. Therefore, a trie is also known as a prefix tree.

In this subsection, we revisit a Patricia trie [24, 29, 31] as the proposed algorithm and data structure, Palmtrie, is based on a Patricia trie. A Patricia trie is a fundamental trie data structure to search a matching data entry corresponding to a given key. A Patricia trie compresses its data structure by eliminating unary branching nodes from a radix tree [9, 18, 29].

A radix tree is a simple binary tree. Each branch of a radix tree adds one bit to the prefix of the parent node. Therefore, the prefix of a node at a depth of d in the tree[1] is the d-bit value of the node. The lookup procedure of the radix tree is just traversing the tree by examining the d-th most significant bit in the given key. Patricia trie eliminates unary branching nodes from the radix tree. To this end, a Patricia trie adds a bit index attribute, *bit*, to each node. The value *bit* is used as the bit index to extract a bit from the given key to determine the branch direction (i.e., left or right). It means that the bit string of a key from the most significant bit to the bit index represents the prefix of the node. The number of bits of the key minus the value *bit* minus 1 is equivalent to the depth of the radix

[1]With zero-based numbering. The depth of the root node is defined to be 0.

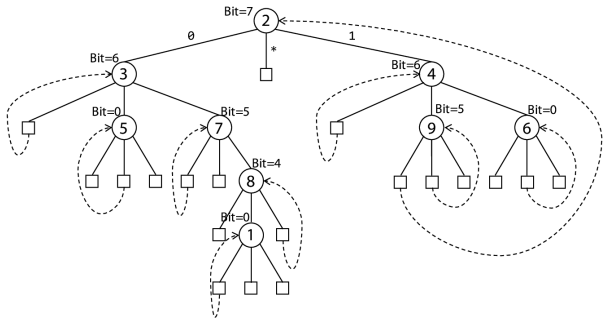


Figure 2: An example of a Palmtrie for the dataset in Table 1. A Palmtrie adds a center branch to each node of the Patricia trie for a don't care bit. Circles and squares represent nodes with a value and pointers, respectively.

tree. In other words, let the bit length of keys and the bit index of a node *bit* be L and b, respectively, the most significant $(L - b)$ bits are the prefix in a Patricia trie. For example, the radix tree and the Patricia trie for keys `100`, `001`, and `010` are illustrated in Figure 1. The Patricia trie is memory efficient as the number of nodes in the Patricia trie equals the number of keys.

The lookup procedure of the Patricia trie is recursively defined. It traverses the trie by examining the bit in the given key corresponding to the bit index of the node; goes down to the left for `0` or to the right for `1`. It terminates the traversal if the bit index of the node is greater than or equal to the current bit index. It finally compares the key of the resulting node with the given key then returns the node if these keys match each other.

For inserting a new node with a given key, the insertion algorithm traverses descendent nodes in the trie to find the position to insert a new node with the key. It checks if the prefix of the descendent node matches the given key during the traversal. If it does not match, it replaces the descendent node with a new node and lets the original descendent node be a descendent node of the new node. The bit index of the new node is set to the most significant different bit between these nodes. If it reaches an empty pointer (NULL), it replaces the pointer to NULL with a new node. The bit index of the new node is set to 0.

3.3 Basic Idea of Palmtrie

Patricia trie is not limited to binary branches but can construct a multiway tree such as branching by alphabetic characters. Therefore, Patricia trie can store keys containing don't care bits using ternary branches without changing the construction procedure of the data structure. Only the search procedure for ternary matching is different from the original algorithm.

We name the proposed trie that solves the ternary matching problem, *Palmtrie*. A node of the basic Palmtrie has three pointers going down to *left*, *right*, and *center* descendent nodes, a bit index referred to as *bit* in figures and algorithms, a *key*, a *value*, and its *priority*. The branches to left, right, and center descendent nodes correspond to `0`, `1`, and `*`, respectively. Here, we look into an example of a Palmtrie for better understanding. Figure 2 illustrates an example of a Palmtrie for the dataset shown in Table 1. The

Algorithm 1 lookup(N, *key*, *bit*); the lookup procedure of Palmtries for the query *key* traversing from the node N. The function match(*key*, N) returns a boolean value by checking if the given *key* matches the ternary string key of N. The function extract(*key*, *off*, *len*) extracts bits of length *len* from the *key*, starting at the offset *off*,. The function max(x, y) returns the node with the largest priority from the given nodes x and y. A NULL node is treated as the lowest priority. The symbol of ← represents the variable assignment operator.

```
1:  if N = NULL then
2:      return NULL;
3:  end if
4:  if bit ≤ N.bit then
5:      if match(key, N) then
6:          return N;
7:      else
8:          return NULL;
9:      end if
10: end if
11: c ← lookup(N.center, key, N.bit);
12: if extract(key, N.bit, 1) then
13:     lr ← lookup(N.right, key, N.bit);
14: else
15:     lr ← lookup(N.left, key, N.bit);
16: end if
17: return max(lr, c);
```

insertion and deletion algorithms of a Palmtrie are the same as those of the original Patricia trie. The lookup procedure is different from the original Patricia trie due to the uniqueness of a don't care bit that matches both `0` and `1`. Note that a don't care bit is handled as a ternary value that does not match `0` or `1` in the insertion and deletion procedures.

The *lookup()* function is recursively defined in Algorithm 1. It is called with three arguments to perform a lookup; the root node for the first argument N, the query key for the second argument *key*, and $L - 1$ for the third argument *bit*. The algorithm goes down to the *center* node for a don't care bit of the stored keys (Line 11) as it matches any of `0` and `1`. It is one of the differences from Patricia tries. In addition, it also performs exact matching for the *left* or the *right* branches (Lines 12–16). With this recursive procedure, it performs the comparison of the query key and the key stored in the entry (Line 5) when the descendent node pointer goes back up to the trie (Line 4). As multiple entries might match the query key, it also performs priority encoding to select an entry with the highest priority from the matching entries (Line 17). This priority encoding is another difference from Patricia tries.

For the further explanation of Palmtrie variants, we refer to this Palmtrie as the *basic Palmtrie* or *Palmtrie (basic)*. We also define two terms as follows; *exact matching branch* and *don't care branch*. The former denotes the traversal to the descendent node uniquely identified from the bit or the chunk extracted from the query key by the bit index. The latter denotes the traversal to a branch containing one or more don't care bits in the extracted bit or chunk of bits. In Algorithm 1, Line 11 and Lines 12–16 are don't care branch and exact matching branch, respectively.

Table 3: Computational complexity of lookup, insertion and deletion procedures of the sorted list and the Palmtrie.

Algorithm	Lookup	Insertion / Deletion
Sorted List	O(n)	O(n) or O($\log_2 n$)
Palmtrie	O($n^{\log_3 2}$)	O($\log_3 n$)

Here, we look through the lookup procedure for the query key of `01110101` to the Palmtrie in Figure 2. We begin the traversal from the root node, Node 2. It first traverses to the don't care branch of the root node. The don't care branch reaches NULL, and then it terminates the traversal. It also traverses to the left descendent node, Node 3, for the exact matching branch corresponding to the 7th bit of the query key, `0`. It goes down from Node 3 to Node 5 and gets Node 5 as a candidate result at the don't care branch from Node 5. It then compares the query key of `01110101` with the key stored in Node 5 of `0*1101**`. As these keys match, Node 5 is returned as a candidate result for the don't care branch from Node 3.

Moving back from the stack to the exact matching branch from Node 3, it traverses to the right descendent node, Node 7, for the exact matching branch corresponding to the 6th bit of the query key, 1. As the center descendent node of Node 7 is NULL, it terminates the traversal to the don't care branch. For the exact matching branch from Node 7, it goes down to the right descendent node, Node 8, as the 5th bit of the query key is 1. Node 1 is traversed for the don't care branch from Node 8. As the bit index of Node 1 is 0, it finds NULL at the right descendent node for the exact matching branch corresponding to the 0th bit of the query key 1. From Node 8, it also traverses the right descendent node for the exact matching branch corresponding to the 4th bit of the query key, 1, and finally obtains Node 8 as another candidate result. The key stored in Node 8 of `011*1000` matches the query key of `01110101`, and consequently, Node 8 is returned as a candidate result for the exact matching branch from Node 3. Comparing the priority of two candidate results from Node 3, Nodes 5 and 8, the priority of Node 5 is higher than that of Node 8. Accordingly, we obtain Node 5 as a result.

Here, we discuss the computational complexity of the basic Palmtrie lookup algorithm. Let the height (i.e., maximum search depth) of a Palmtrie be d, and the number of steps to traverse the left or right triangle for the exact matching branch or the center triangle for the don't care branch be c_d, c_d is defined by the following recurrence formula: $c_d = 2c_{d-1}$. By solving this formula, we obtain $c_d = c_0 \cdot 2^d$. Assuming a dense Palmtrie, the height of the Palmtrie is considered $d = \log_3 n$, where n denotes the number of entries because the Palmtrie is a ternary tree. Thus, the computational complexity of the Palmtrie lookup algorithm is calculated as O($2^{\log_3 n}$), which equals to O($n^{\log_3 2}$) $\approx$ O($n^{0.63}$). In the worst case of a sparse and unbalanced trie, the computational complexity becomes O(n) but is bound to O(L^2).

Table 3 summarizes the theoretical computational complexity of the sorted list and the Palmtrie. The complexity of the Palmtrie, O($n^{\log_3 2}$), is significantly lower than that of the sorted list, O(n). Thus, the Palmtrie is considered more scalable compared to the sorted list. We will evaluate the lookup performance for various datasets and traffic patterns in Section 4.

3.4 Multi-bit Stride Extension

The recursive lookup procedure of the basic Palmtrie is executed bit by bit for a query key. Therefore, it requires many comparison and memory load operations, up to the bit length of keys for a query, to traverse the tree. It leads to the slowness of the lookup algorithm. We introduce a multi-bit stride that is a common performance improvement technique. A multi-bit stride reduces the depth of the trie and the number of steps to search down the trie. For example, Tree BitMap [12] for longest prefix matching adopts a 4-bit stride.

A simple approach to construct a Palmtrie with a multi-bit stride is to extend a ternary trie to a 3^k-way trie where k is the stride size. However, this multi-bit stride extension of Palmtries creates many descendent nodes and leads to a scalability issue. The number of descendent nodes of a Palmtrie for a k-bit stride is 3^k while that of a binary trie is 2^k. Thus, a Palmtrie creates $(3/2)^k$ times more descendent nodes than a binary trie for a k-bit stride. Moreover, $(3^k - 2^k)$ nodes of the descendent nodes of a k-bit stride Palmtrie contain one or more don't care bits, and consequently, need additional traversal to these nodes for don't care branches.

Let $\binom{n}{r}$ be the binomial coefficient, i.e., the number of combinations of n things taken r at a time, $\binom{k}{i}$ descendent nodes need to be traversed for don't care branches when i bits of the chunk to the descendent nodes are don't care bits. This means that $\sum_{i=0}^{k} \binom{k}{i}$ descendent nodes including the exact matching branch are traversed. It adds to the computational complexity of the Palmtrie lookup procedure. Moreover, it is not memory efficient because many of these descendent nodes are possibly empty (i.e., NULL). Thus, the multi-bit stride extension to Palmtries is challenging.

For example, assuming $k = 3$, the number of descendent nodes of this naive Palmtrie node is 27. The 8 of the 27 descendent nodes are exact matching branches, and thus, one of them is uniquely selected from the query key chunk of the 3-bit stride. The rest 19 of the 27 descendent nodes include one or more don't care bits in the key chunks of the 3-bit stride. Therefore, we need to check and traverse these 19 descendent nodes for the query key chunk of the 3-bit stride. We can exclude the 11 nodes of these 19 descendent nodes by matching the binary bits of the key chunk, but the rest 7 descendent nodes still need to be traversed for don't care branches.

We focus on the most significant don't care bit in the chunk of keys to reduce the number of descendent nodes and simplify the traversal for don't care branches. We combine don't care branches that contain one or more don't care bits at any positions other than the least significant bit in the k-bit chunk of a key into a don't care branch of a subtree that allows only one don't care bit at the least significant bit. This means that the stride size for the combined don't care branches is less than k. For example of $k = 3$, we combine `0*0`, `0**`, and `0*1` into `0*`, `1*0`, `1**`, and `1*1` into `1*`, and `*00`, `*01`, `*10`, `*11`, `*0*`, `*1*`, `**0`, `**1`, and `***` into `*`. Otherwise, we keep four don't care branches that have a don't care bit at the least significant bit of the 3-bit chunk.

Figure 3 depicts part of a k-bit stride Palmtrie where $k = 3$. Eight filled black nodes are the descendent nodes of the root for the exact matching branch. Seven filled red nodes are the descendent nodes for the don't care branch for those including don't care bits. On traversing the trie, one filled black node is selected from a k-bit chunk of the query key. Besides, k filled red nodes are also

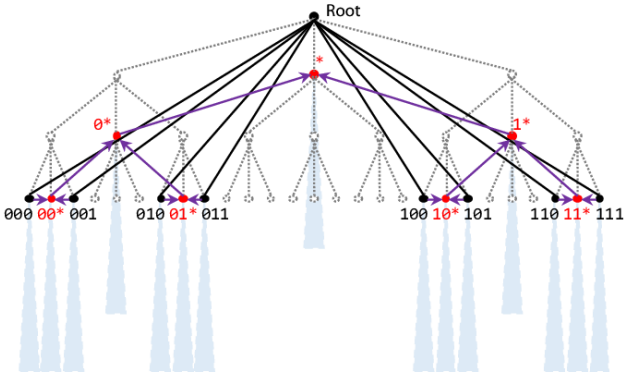


Figure 3: Part of a 3-bit stride Palmtrie. Solid black edges are the exact matching branches of k-bit stride Palmtrie. Dotted circles and edges are the nodes and the edges of the basic Palmtrie, respectively. Eight filled black nodes at the bottom and seven filled red nodes are the descendent nodes of the root for the exact matching branch and the don't care branch, respectively. Light blue triangles are subtrees.

traversed for the don't care branch. The search for gray dotted circles in subtree triangles below filled red nodes are combined into the don't care branch to these filled red nodes. Purple solid arrows in this figure represent the search procedure for the don't care branches. By applying the right bit shift operation to the value of a chunk, the prefix excluding the least significant * bit of the don't care branch is obtained.

In other words, the stride size for traversing to don't care branches is changed according to the position of the most significant don't care bit in a k-bit key chunk. Note that the stride size to get a query key chunk is the fixed length of k, although part of the chunk is used for don't care branches. This flexibility of the stride size leads to an alignment mismatch to the key length. Therefore, a bit index for the least significant chunk of a key allows a negative value greater than $-k$. A bit indexed by a negative value is treated as 0 in extracting a chunk.

It requires to modify the insertion algorithm. As the prefix of a subtree for a don't care branch ends with *, the bit position of * can be the bit index for don't care branches. It means that the * bit must be the root of a subtree. Therefore, a key is first split between the position of * and one less significant bit position. Besides, the binary string part of the key is split by k-bit chunks. In this way, the key to insert is split into chunks, and the insertion algorithm traverses the trie to find the position to insert by checking the prefix of subtrees. Other than the key split method, the insertion algorithm follows that of Patricia tries described in Section 3.2.

Figure 4 illustrates a k-bit stride Palmtrie for the dataset in Table 1. Red solid arrows in Figure 4 represent the search procedure to don't care branches. We look at the lookup procedure for the query key of `01110101` to the Palmtrie shown in Figure 4. As the bit index of the root node, Node 2, is 5, it extracts 3 bits from the 7th to 5th bits, `011`. It first traverses don't care branches from the root node for `01*`, `0*`, and `*`. A don't care branch for `0*` has a descendent node of Node 5. The query bit chunk of `101` is extracted as the bit index of the node is 2. Then, it traverses don't care branches from the node. The descendent node for * points to Node 5 itself, and then it is returned as a candidate result. The exact matching branch from the root node goes down to Node 8. The bit index of the node is 2, and then the query bit chunk is `101`. The descendent node for * points to Node 1. The descendent node for `10*` points to Node 8 itself. Thus, Node 8 is a candidate result. Comparing the priority of two candidate results, Nodes 5 and 8, Node 5 is returned as it has a higher priority. Moving back to Node 1, the bit index of Node 1 is -1, then the chunk of `010` is extracted. All of the descendent nodes for don't care branches and the exact matching branch for `010` are NULL, then none of the descendent nodes of Node 1 is a candidate result. In this way, it finally returns Node 5 as a result.

3.5 Practical Optimization Techniques

Practical optimization techniques must be taken into account to achieve good performance. This subsection describes three practical optimization techniques for Palmtries; 1) efficient descendent node indexing, 2) a CPU cache efficient search procedure, and 3) low-priority subtree skipping.

As described in the previous subsection, a k-bit stride Palmtrie node has 2^k descendent nodes for the exact matching branch and $2^k - 1$ descendent nodes for the don't care branch. To efficiently perform the traversal to exact matching branches and don't care branches, they are required to be indexed by a chunk or any values easily computed from the chunk. Figure 5 illustrates the data structure of the pointers to descendent nodes of Palmtrie. The pointers organize two contiguous arrays; one is for exact matching branches, and the other is for don't care branches. The array for exact matching branches is uniquely indexed by the chunk of a key. The array for don't care branches is indexed so that the index of the array for the prefix is defined by the following equation: $2^l + p - 1$, where l and p are the prefix length of binary digits and the prefix of binary digit part, respectively. This allows efficient don't care branch traversal. Tree BitMap [12] employs a similar indexing technique.

The second practical optimization takes into account the memory locality. The lookup procedure of a Palmtrie is a depth-first search, as shown in Algorithm 1. It first searches the don't care branch and then goes down to the exact matching branch. In Lines 12–16 of Algorithm 1, N.bit, N.left, and N.right refer to the member variables of N. However, the data of N is possibly evicted from the CPU cache during the traversal to the center in Line 11. It might cause cache misses in Lines 12–16 for the exact matching branch. To efficiently utilize the CPU cache, we push a descendent node and N.bit to the self-managed stacks so that N is not referred to again for the exact matching branch.

The other optimization is low-priority subtree skipping. This optimization adds the highest priority of the subtree, `max_priority`, to each node when building the data structure. The lookup procedure skips the traversal to the subtree if the priority of the current candidate node is higher than this `max_priority` of the subtree.

Algorithm 2 is the lookup algorithm of the Palmtrie with the multi-bit stride extension and these practical optimization techniques. In this algorithm, Lines 14 and 16 are traversals to the don't care branches with efficient descendent node indexing. Lines 2, 4, 12, and 18 are the second optimization for efficient CPU cache

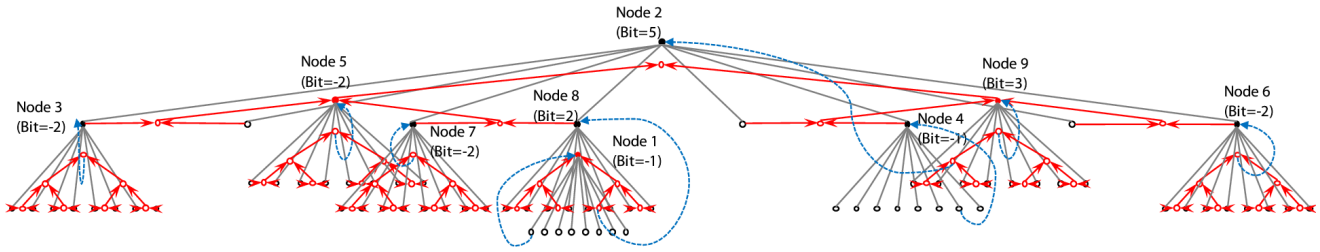


Figure 4: An example of a k-bit stride Palmtrie ($k = 3$) for the dataset in Table 1. Each filled circle denotes a node with a value. The value and the bit of interest are denoted on a node. Red solid arrows represent links for don't care branches. Blue dotted arrows represent backtrack links pointing to resulting nodes whose keys are compared with a query key.

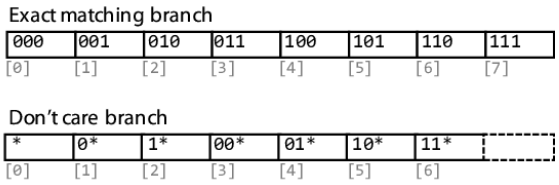


Figure 5: Two contiguous arrays of pointers to descendent nodes ($k = 3$). The array for the exact matching branch contains 2^k elements, and one of them is traversed. The array for the don't care branch contains $2^k - 1$ elements, and k elements of them are traversed.

utilization using the self-managed stacks. Line 5 is low-priority subtree skipping. Hereafter, we call this variant *Palmtrie*$_k$, where k denotes the stride size.

3.6 Palmtrie$^+$: Lookup Optimization with Population Count

Each node of Palmtrie$_k$ requires $(2^{k+1}-1)$ pointers for exact matching branches and don't care branches. However, as we can see in Figure 4, nodes of Palmtrie$_k$ have many NULL pointers, each of which typically requires 4 or 8 bytes. These pointers significantly increase the memory footprint of the data structure and lead to inefficient CPU cache utilization. For example, when the stride size k is 8, and the pointer size is 8 bytes, each node allocates more than 4 kilobytes of memory.

To cope with this, we leverage sets of a bitmap and a contiguous array of descendent nodes [3, 12]. They remove the NULL pointers and compress the pointers with a representative pointer to the descendent node array. Each bit in the bitmap indicates whether the corresponding descendent node is a non-NULL value by a bit of 1 or NULL by a bit of 0. The bitmap is also used to locate the descendent node corresponding to an index with the population count operation. Poptrie [3] leverages the *popcnt* CPU instruction to speed up the population count operation.

Derived from Poptrie, we use an array of descendent nodes instead of pointers to compress the data structure. A contiguous array can be compressed with a bitmap in the same way as Poptrie. A descendent node in the compressed array is indexed by the bitmap and the population count operation: A descendent node corresponding

Algorithm 2 lookup(N, *key*); the lookup procedure of Palmtrie$_k$ for the query *key* traversing from the node N. The member variables of a node, descendants and ternaries, are the contiguous arrays for the exact matching branch and the don't care branch, respectively. p and b are stack variables. « and » represent left bit shift and right bit shift operations, respectively.

```
1:  r ← NULL; p ← []; b ← [];
2:  p[0] ← N; b[0] ← L - k; n ← 1;
3:  while n > 0 do
4:      n ← n - 1; x ← p[n]; bit ← b[n];
5:      if r.priority ≤ x.max_priority then
6:          if bit ≤ x.bit && match(key, x) then
7:              r ← max(r, x);
8:          else
9:              i ← extract(key, x.bit, k);
10:             c ← x.descendants[i];
11:             if c ≠ NULL then
12:                 p[n] ← c; b[n] ← x.bit; n ← n + 1;
13:             end if
14:             o ← (i » 1) | (1 « (k - 1));
15:             for i ∈ {0, . . . , k − 1} do
16:                 t ← x.ternaries[(o » i) - 1];
17:                 if t ≠ NULL then
18:                     p[n] ← t; b[n] ← x.bit; n ← n + 1;
19:                 end if
20:             end for
21:         end if
22:     end if
23: end while
24: return r;
```

to the i-th element of the original array is indexed by the number of 1s in the least significant i bits of the bitmap for the contiguous array of nodes. The use of a contiguous array for descendent nodes instead of pointers disallows any nodes to point back up to the trie. Therefore, we push the nodes with keys and values to the leaves of the trie. Instead, the internal nodes only retain descendent nodes for exact matching and don't care branches. It is similar to the relationship between B-tree [6, 18] and B+ tree [18]. Thus, we name this *Palmtrie*$^+$.

Algorithm 3 shows the lookup algorithm of Palmtrie$_k^+$, where k is the stride size. A node of a Palmtrie$_k^+$ is a *union* data structure, as illustrated in Figure 6. The bit index attribute of a node is used to determine whether the node is an internal node or a leaf; $-\infty$ for a

Algorithm 3 lookup(T, *key*); the lookup procedure of Palmtrie$_k^+$ for the query *key* traversing from the root T.root. T is the trie data structure that maintains nodes.

```
1:  r ← NULL; p ← []; b ← [];
2:  p[0] ← T.root; b[0] ← L - k; n ← 1;
3:  while n > 0 do
4:      n ← n - 1; x ← p[n]; bit ← b[n];
5:      if r.priority ≤ x.max_priority then
6:          if x.bit = −∞ && match(key, x) then
7:              r ← max(r, x);
8:          else
9:              i ← extract(key, x.bit, k);
10:             if (1 « i) & x.bitmap_c then
11:                 j ← popcnt(((1 « i) - 1) & x.bitmap_c);
12:                 c ← T.nodes[x.offset_c + j];
13:                 if c ≠ NULL then
14:                     p[n] ← c; b[n] ← x.bit; n ← n + 1;
15:                 end if
16:             end if
17:             o ← (i » 1) | (1 « (k - 1));
18:             for i ∈ {0, . . . , k − 1} do
19:                 h ← (o » i) - 1;
20:                 if (1 « h) & x.bitmap_c then
21:                     j ← popcnt(((1 « h) - 1) & x.bitmap_t);
22:                     t ← T.nodes[x.offset_t + j];
23:                     p[n] ← t; b[n] ← x.bit; n ← n + 1;
24:                 end if
25:             end for
26:         end if
27:     end if
28: end while
29: return r;
```

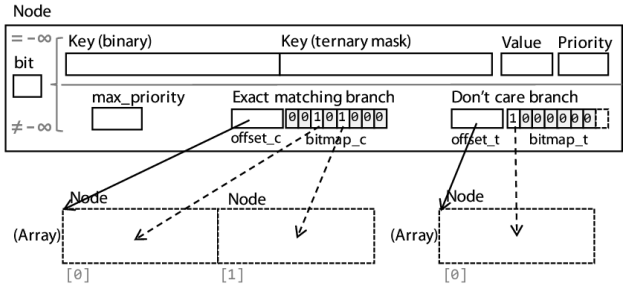


Figure 6: The data structure of a node of Palmtrie$^+$. A node of Palmtrie$^+$ is a union data structure identified by the `bit` index attribute; $-\infty$ for a leaf and the other value greater than $-k$ for an internal node. An internal node has two sets of a bitmap and an offset to point to descendent node arrays for exact matching branches and don't care branches. The bitmap indexes the corresponding descendent node in the same way as Poptrie.

leaf. Note that we use $-k$ instead of $-\infty$ in the implementation as commodity CPUs do not support $-\infty$ for an integer value, and the `bit` index attribute for an internal node must be greater than $-k$. Attributes offset_c and offset_t of a node are offsets for exact matching branches and don't care branches, respectively. These offsets are used as indices to locate the position of the array in a contiguous array of nodes maintained by the Palmtrie$_k^+$. Attributes bitmap_c and bitmap_t of a node represent bitmaps for exact matching branches and don't care branches, respectively.

As Palmtrie$_k^+$ pushes the nodes with keys and values to the leaves of the trie and forms contiguous arrays for descendent nodes, Palmtrie$_k^+$ does not support the incremental update. However, it is easy to build Palmtrie$_k^+$ from Palmtrie$_k$ because the tree structure of Palmtrie$_k^+$ is the same as that of Palmtrie$_k$. Therefore, the update procedure of Palmtrie$_k^+$ first performs the incremental update of Palmtrie$_k$ and then builds Palmtrie$_k^+$ from Palmtrie$_k$. To be more precise, the insertion and deletion of an ACL entry require rebuilding the data structure of Palmtrie$_k^+$ from Palmtrie$_k$, although the update for an existing key can be incrementally performed.

4 EVALUATION

We implemented the lookup and update algorithms of Palmtries. The code is available at https://github.com/pixos/palmtrie. For the evaluation, the key length L is set to 128 bits to suffice for IPv4 layer 3–4 ACL rules. A key consists of two-bit strings, *data* and *mask*, which represent binary digits and don't care bits, respectively. Therefore, 256 bits (32 bytes) are used in total for a key. We allocate 8 bytes and 4 bytes for a value and a priority, respectively. To compare with the existing algorithms for ACL matching, we also implemented the sorted list and employed the `examples/l3fwd-acl` program from DPDK version 18.11 (DPDK-ACL) and EffiCuts [35][2]. To validate the correctness of the code, we have run tests that compare the lookup results of Palmtries with those of the sorted list and have confirmed they match with each other.

We use a computer equipped with Intel(R) Core i7 6700K (4.0 GHz, 8 MB cache) and 32 GB memory (four 8 GB DDR4-2133 modules) with Ubuntu 16.04.6 LTS (Linux kernel 4.4.0-146) for the evaluation. The performance measurement program runs on a single CPU core. It counts the number of lookups while repeatedly calling the lookup function of each algorithm with a traffic pattern for 300 seconds. This program reports the lookup counts every 10 seconds to analyze the performance statistically. Thus, we obtain 30 samples of 10-second intervals for each measurement. In figures of the lookup performance evaluation, we plot the average lookup rate in mega lookups per second (Mlps), and an error bar represents the standard deviation of the 30 samples.

4.1 ACL Datasets and Traffic Patterns

We use two types of ACLs that emulate different policies for the evaluation; 1) a synthetic campus network consisting of multiple departments and laboratories, and 2) ClassBench [33]. The former datasets are emulating firewall rules of an existing campus network. The latter datasets include various address spaces, and thus, are targeting Internet backbone routers.

For the campus network policy, an ACL is generated by splitting `10.0.0.0/8` into equal size of 2^q prefixes for $q \in \{0, \ldots, 16\}$. Let a split prefix be $\mathbb{P}$, the ACL permits all outbound traffic from $\mathbb{P}$, and inbound ICMP packets to $\mathbb{P}$. It also permits inbound DNS and NTP responses and established TCP traffic. The first `/27` region of

[2]We use the code at https://github.com/kun2012/compressedcut to measure the performance of EffiCuts.

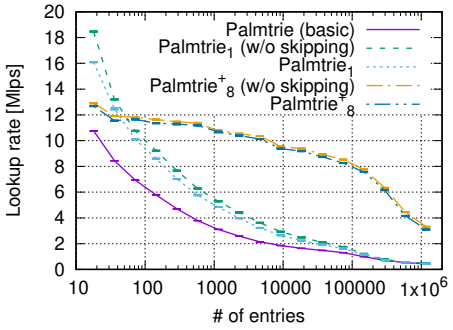


Figure 7: Lookup performance of Palmtrie (basic), Palmtrie$_1$ and Palmtrie$^+_8$ without low-priority subtree skipping (w/o skipping), Palmtrie$_1$, and Palmtrie$^+_8$ on various sizes of ACLs for the uniform traffic.

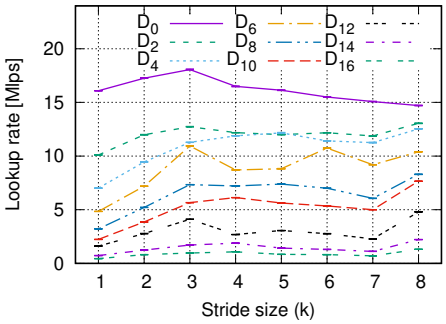


Figure 8: Lookup performance of Palmtrie$_k$ for the uniform traffic where $k = 1, \cdots, 8$. Palmtrie$_8$ achieves the best performance on D_2, D_4, D_8, D_{10}, D_{12}, D_{14}, and D_{16}.

$\mathbb{P}$ is set to the demilitarized zone (DMZ), i.e., passing any traffic. The second /27 region of $\mathbb{P}$ is set for services to permit incoming DNS, HTTP, HTTPS, QUIC, SMTP, POP3, IMAP, IMAPS, and POP3 traffic. We define the dataset for q as D_q. The number of entries in the ACL of D_q is $17 \cdot 2^q$. As the entries for each split prefix $\mathbb{P}$ include one entry with the `established` keyword, the number of ternary matching entries is $18 \cdot 2^q$.

We generate two synthetic patterns of traffic for the evaluation with the campus network ACLs; 1) uniform and 2) reverse-byte order scanning. The uniform traffic pattern is generated so that the pattern uniformly and randomly results in each entry. It is one of the most challenging traffic patterns because we cannot leverage the entry caching technique due to randomness. The reverse-byte order scanning traffic is based on a real scanning attack pattern observed on the Internet [10]. It consists of incoming TCP SYN packets with the destination port of 5060 (SIP). The reverse-byte order of destination addresses is sequential within the range of `10.0.0.0/8`; i.e., the sequence of $\cdots$, `10.255.0.0`, `10.0.1.0`, `10.1.1.0`, $\cdots$. Random source addresses and port numbers are assigned. The reverse-byte order scanning is a more realistic traffic pattern than the uniform pattern where ACLs are used.

We also employ ClassBench [33] to generate Internet backbone ACLs and traffic patterns. We generate 18 sets of ACL rules and traces using three parameter files; `acl1_seed` (ACL), `fw2_seed` (FW), and `ipc2_seed` (IPC). Note that we do not use `fw1_seed` and `ipc1_seed` because EffiCuts with the default parameters failed to build the data structure for the ACLs generated with these parameter files. We generate six sets for each parameter file with the number of rules of 1K, 10K, 50K, 100K, 200K, and 500K. The prefix and suffix of the dataset name denote the parameter file and the number of rules, respectively; e.g., FW100K is a 100K-entry ACL generated with `fw2_seed`.

4.2 Effect of Optimizations in Palmtrie

We first evaluate the effect of the practical optimization techniques introduced in Section 3.5. Here, we compare the lookup performance between Palmtrie (basic) and Palmtrie$_1$. Figure 7 shows the lookup performance for the uniform traffic. It demonstrates that Palmtrie$_1$ improves the lookup performance by 4.32%–49.6% compared to Palmtrie (basic). The improvement is remarkable for ACLs with a small number of entries. It is because the cache efficiency yielded by self-managed stacks is more critical on smaller ACLs. This figure also shows the disadvantage of low-priority subtree skipping. The 6.6% degradation of Palmtrie$^+_8$ for the uniform traffic on the one-million-entry ACL (D$_{16}$) is much higher than expected from the branch misprediction added by low-priority subtree skipping (Line 5 in Algorithms 2 and 3). We suspect that traversing to the skipped subtrees avoids cache eviction for frequently visited nodes. Although we observed this small lookup performance degradation for the campus network ACLs, we confirmed that low-priority subtree skipping contributed to the lookup performance for the ClassBench datasets. For example, Palmtrie$^+_8$ runs 1.32–5.40 times faster than that without low-priority subtree skipping. Therefore, we adopt low-priority subtree skipping in the Palmtrie.

We then evaluate the effect of the multi-bit stride extension. Figure 8 summarizes the lookup performance of Palmtries$_k$ at various factors of k for the uniform traffic. This figure shows that the highest branching order with the stride size of $k = 8$ achieves the best performance on extensive ACLs (e.g., D_8, D_{10}, D_{12}, D_{14}, and D_{16}) while the lower branching order does on tiny ACLs (D_0). It is attributed to the cache efficiency that memory locality is more effective on smaller ACLs than the effect of depth reduction by the higher branching order. These results also suggest aligning the stride size to 8 bits because the size of elements of an ACL entry such as IP addresses and port numbers is byte-aligned. The 6-bit stride also achieves good performance because the register size for indexing with a bitmap fits the 64-bit CPU architecture. Hereafter, we evaluate $k = 6$ and $k = 8$ for the evaluation in the rest of the paper from these results.

We also evaluate the memory utilization of Palmtrie and its variants in this subsection. As described in Section 3.6, Palmtrie$_k$ requires a significant amount of memory when the stride size of k increases. Thus, Palmtrie$^+$ adopts a technique to reduce the memory footprint. Figure 9 shows the memory utilization of Palmtrie$_1$, Palmtrie$_6$, Palmtrie$_8$, Palmtrie$^+_6$, and Palmtrie$^+_8$, for the campus network ACLs. This figure demonstrates that the memory utilization of Palmtrie and its variants is proportional to the number of ACL entries. However, Palmtrie$_6$ and Palmtrie$_8$ consumes 12 and 47

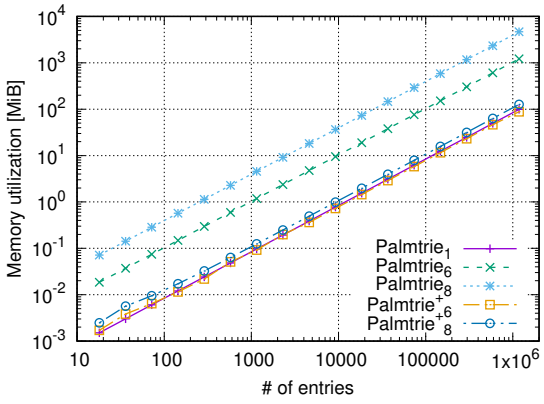


Figure 9: Memory utilization (1 MiB = 2^{20} bytes). Palmtrie$_k$ requires a large amount of memory. Palmtrie^+_k reduces the memory utilization, which is the same order as Palmtrie$_1$.

times more memory than Palmtrie$_1$, respectively. In contrast, the memory utilization of Palmtrie$^+_6$ and Palmtrie$^+_8$ is almost the same as that of Palmtrie$_1$. It proves that the technique introduced in Palmtrie$^+$ successfully solves the problem with excessive memory utilization caused by the multi-bit stride extension.

4.3 Lookup Performance

This subsection evaluates the lookup performance of Palmtries comparing to the existing algorithms on two different datasets; the synthetic campus network ACLs and ClassBench datasets.

We first evaluate the lookup performance on the campus network ACLs. Figure 10 shows the lookup performance of Palmtrie$_6$, Palmtrie$_8$, Palmtrie$^+_6$, Palmtrie$^+_8$, and two conventional algorithms, the sorted list, and DPDK-ACL, for the uniform traffic and the reverse-byte order scanning traffic on various ACLs. The lookup performance of the sorted list is significantly degraded on the ACLs with a large number of entries because its computational complexity for lookup is $O(n)$. Palmtrie$^+_8$ runs 9.52×10^3 times and 2.39×10^4 times faster than the sorted list on D_{16} for the uniform traffic and the reverse-byte order scanning traffic, respectively. However, the sorted list outperforms Palmtrie$^+_8$ on D_0, D_1, and D_2. This is attributed to predictive sequential memory access in the sorted list. It might take advantage of hardware prefetching of CPU caches.

Figure 10 demonstrates that Palmtrie$^+_8$ outperforms DPDK-ACL on all ACLs for the uniform traffic and the reverse-byte order scanning traffic. The average lookup performance of Palmtrie$^+_8$ is 1.05–2.58 times and 1.17–4.76 times faster than DPDK-ACL for the uniform traffic and the reverse-byte order scanning traffic, respectively. The characteristics of each algorithm for the reverse-byte order scanning traffic are similar to that for the uniform traffic. However, these results highlight the advantage of Palmtrie$^+$ compared to DPDK-ACL as the reverse-byte order scanning traffic is a more realistic traffic pattern compared to the uniform traffic.

These results practically suggest to use the sorted lists for very small ACLs (e.g., less than 50 entries), the Palmtrie with lower branching order such as Palmtrie$^+_6$ for medium size of ACLs (e.g., less than 1000 entries), and the Palmtrie with higher branching order such as Palmtrie$^+_8$ for large size of ACLs. However, as we can see that Palmtrie$^+_6$ runs faster than Palmtrie$^+_8$ on D_{16} for the reverse-byte order scanning traffic, we need further research for an optimal branching order.

Table 4: Lookup performance comparisons for ClassBench datasets. The lookup rate is listed in Mlps.

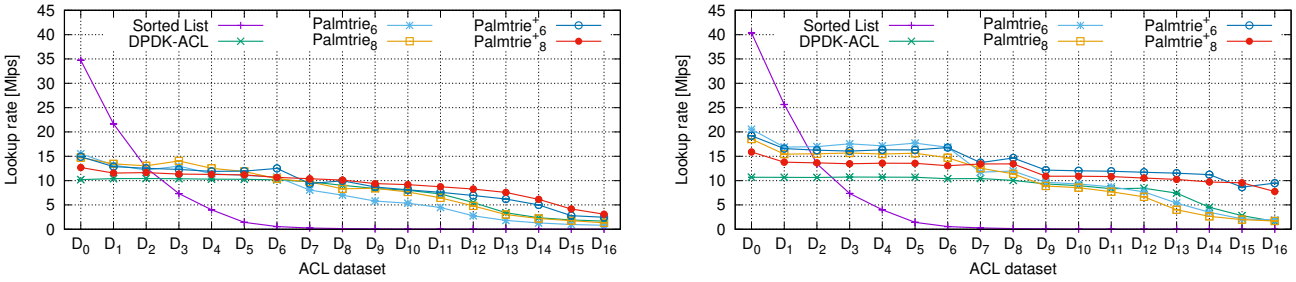
Dataset	EffiCuts	DPDK-ACL	Palmtrie$^+_8$
ACL1K	0.247	11.8	10.4
ACL10K	0.223	10.7	9.43
ACL50K	0.202	2.44	4.57
ACL100K	0.202	1.83	3.57
ACL200K	N/A	1.67	5.08
ACL500K	N/A	1.66	5.35
FW1K	0.171	10.0	6.33
FW10K	0.162	9.35	3.74
FW50K	0.173	4.51	3.33
FW100K	0.166	2.93	2.72
FW200K	0.156	2.30	2.46
FW500K	0.146	3.00	7.25
IPC1K	0.388	11.4	11.3
IPC10K	0.381	8.96	9.56
IPC50K	0.381	4.32	9.29
IPC100K	0.347	2.75	6.43
IPC200K	0.310	2.30	5.77
IPC500K	0.248	2.65	7.83

We also evaluate the lookup performance with another type of synthetic datasets using ClassBench. In this subsection, we evaluate the performance of EffiCuts [35], DPDK-ACL, and Palmtrie$^+_8$. However, the implementation of EffiCuts does not support protocol flags (e.g., TCP flags). Therefore, we exclude TCP flags from the rule-sets to fairly compare these algorithms.

Table 4 summarizes the lookup performance for ClassBench datasets. EffiCuts failed to build the data structure for ACL200K and ACL500K. The results show that Palmtrie$^+_8$ significantly outperforms EffiCuts. It runs 15.8–49.7 times faster than EffiCuts. The results also show that Palmtrie$^+_8$ runs 1.07–3.22 times faster than DPDK-ACL for 200K and 500K datasets. However, Palmtrie$^+_8$ is slower than DPDK-ACL on ACL1K, ACL10K, FW1K, FW10K, FW50K, FW100K, and IPC1K. This is because DPDK-ACL builds an optimized trie for each dimension of ACL rules such as IP address prefixes and port ranges, although Palmtrie$^+_8$ is a general ternary matching data structure. We will further investigate lookup performance optimizations for Palmtries that do not sacrifice the build time, such as a software pipelining technique [2], in the future.

4.4 Update Performance

Incremental updates are also vital for ACL matching algorithms, as previously mentioned. As described in Section 3.6, the update procedure of Palmtrie^+_k is divided into two; 1) incremental updates of Palmtrie$_k$ and 2) optimization and rebuild of Palmtrie^+_k (compilation part). Moreover, the existing algorithms, EffiCuts and DPDK-ACL, do not support incremental updates. Therefore, we evaluate the



(a) Uniform traffic

(b) Reverse-byte order scanning traffic

Figure 10: Lookup performance comparison for the uniform traffic and the reverse-byte order scanning traffic on various sizes of campus network ACLs. The performance of Palmtrie$^+$ runs faster than the sorted list and DPDK-ACL on extensive ACLs. The sorted list outperforms the other algorithms on tiny ACLs.

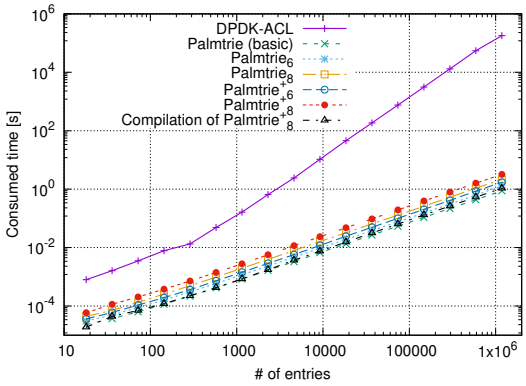


Figure 11: Build time of DPDK-ACL, Palmtrie (basic), Palmtrie$_6$, Palmtrie$_8$, Palmtrie$^+_6$, and Palmtrie$^+_8$, for the synthetic campus network ACLs. The build time of Palmtrie and its variants is less than 5 seconds while the build time of DPDK-ACL for extensive ACLs is not acceptable.

time to build the data structure in this paper. For Palmtrie^+_k, we also measure the time of the compilation part, and we then discuss the incremental update performance from the build time and the compilation time.

The time to build each data structure for the campus network ACLs is shown in Figure 11. It reveals that the build time of DPDK-ACL for extensive ACLs is unacceptable. It takes more than ten seconds for ACLs with no fewer entries than 9216. Moreover, it takes more than three hours for large tables such as D_{14}, D_{15}, and D_{16}, whose number of entries is more than 294 thousand. It is entirely unacceptable even for the static configuration.

The build time of Palmtrie (basic) and Palmtrie$_k$ is the summation of incremental updates for inserting all entries. A linear function approximates it in Figure 11. This means that the insertion time of Palmtrie (basic) and Palmtrie$_k$ is not dominated by the number of existing entries in the data structure but by the number of entries to insert. Palmtries$_6$ and Palmtries$_8$ take 1.29 and 2.14 seconds for inserting all entries of D_{16}, respectively. Therefore, we conjecture that Palmtries$_6$ and Palmtries$_8$ can update an entry with a microsecond order as they support incremental updates.

Palmtrie^+_k does not support incremental updates, and consequently, Palmtrie^+_k requires the compilation from Palmtrie$_k$. Therefore, the build time, including the compilation time, is more critical. Figure 11 also shows the compilation time of Palmtrie$^+_8$ from Palmtrie$_8$. This figure demonstrates that the compilation time is also proportional to the number of existing entries in the data structure. Looking at the ACL with one million entries (D_{16}), the total build time of Palmtrie$^+_8$ is 3.21 seconds. The time to build Palmtrie$_8$ by inserting all the entries is 2.14 seconds, and the compilation of Palmtrie$^+_8$ from Palmtrie$_8$ takes 1.07 seconds. It would be acceptable for most cases to perform the insertion (and deletion) in bulk. However, it indicates that the insertion of one entry to a one-million-entry ACL takes more than one second when the update is performed one by one. Therefore, the applications of Palmtrie$^+$ need to take into account the compilation frequency.

We also evaluate the update performance for ClassBench datasets. Table 5 summarizes the update performance of EffiCuts, DPDK-ACL, Palmtrie$^+_8$, and the compilation time of Palmtrie$^+_8$ from Palmtrie$_8$ for ClassBench datasets. The update performance of Palmtrie$^+_8$ in this table is consistent with that for the campus network policy ACLs; The build time and the compilation time are proportional to the number of entries. This table shows that the build time of Palmtrie$^+_8$ is notably shorter than that of EffiCuts and DPDK-ACL. It is attributed to the simpleness of the data structure of Palmtrie$_k$ and Palmtrie^+_k, while EffiCuts and DPDK-ACL require complex data structure optimizations.

5 DISCUSSION

IPv6 support and performance evaluation: This paper has focused on IPv4 layer 3–4 ACLs. However, the penetration of IPv6 gains the importance of IPv6 network security. Therefore, it is crucial to support IPv6 in ACL matching. The data structure of the Palmtrie is not specific to IPv4. Hence, it is easy to extend it to IPv6 from the viewpoint of the data structure. One modification to support IPv6 is extending the key length to suffice IPv6 source and

Table 5: Update performance comparisons for ClassBench datasets. The build time is listed in seconds. The consumed time for the compilation part of Palmtrie$^+_8$ is parenthesized.

Dataset	EffiCuts	DPDK-ACL	Palmtrie$^+_8$	
ACL1K	0.0231	0.0129	0.00568	(0.00171)
ACL10K	1.05	0.253	0.0355	(0.0116)
ACL50K	43.1	2.12	0.180	(0.0597)
ACL100K	169	4.72	0.362	(0.118)
ACL200K	N/A	8.45	0.707	(0.231)
ACL500K	N/A	21.5	1.76	(0.587)
FW1K	0.0221	0.816	0.00326	(0.000997)
FW10K	0.957	0.590	0.0270	(0.00847)
FW50K	20.1	9.40	0.127	(0.0394)
FW100K	77.3	3.29	0.254	(0.0786)
FW200K	301	6.60	0.507	(0.161)
FW500K	1690	148	1.27	(0.432)
IPC1K	0.0131	0.0274	0.00246	(0.000847)
IPC10K	1.88	5.14	0.0287	(0.00857)
IPC50K	42.2	1.94	0.136	(0.0420)
IPC100K	166	5.26	0.268	(0.0852)
IPC200K	664	9.68	0.535	(0.178)
IPC500K	4170	25.5	1.38	(0.470)

destination addresses and the other attributes such as port numbers and protocol flags. This paper has used a 128-bit key length ($L = 128$) for the Palmtrie for IPv4 layer 3–4 ACL rules. To be a more generic data structure for ACLs, e.g., IPv6 layer 2–4 ACLs, a 512-bit key length is sufficient. However, a longer key length degrades the lookup performance for the following two reasons: 1) It increases the memory footprint of Palmtrie$_k$ nodes and Palmtrie^+_k leaves to store the key. 2) The key comparison, i.e., the match() function in Algorithms 1–3, consumes more CPU cycles. For example, when changing the key length from 128 to 512 bits, the memory utilization of Palmtrie$^+_8$ is increased by 66.7%, and the lookup performance of Palmtrie$^+_8$ for ClassBench datasets slows down by 5.48%–30.1%. Thus, the Palmtrie is feasible to support IPv6, but performance degradation is expected. We will tackle the challenge of the Palmtrie with the longer key length.

Another challenge on the IPv6 ACL matching study lies in the performance evaluation. We lack public IPv6 datasets on ACLs and traffic. Moreover, ACL rules and traffic patterns on IPv6 have not been well researched. ClassBench used in this paper does not support IPv6 dataset generation. To accelerate the research on IPv6 ACL matching, we need more public datasets and models on IPv6 ACL rules and traffic patterns.

Real-world deployment considerations: The evaluation results reveal that Palmtrie$^+_8$ does not always achieve the best lookup performance. For example, the sorted list outperforms Palmtries on tiny ACLs. Therefore, Section 4.3 practically suggests using the sorted lists for tiny ACLs rather than Palmtries. As shown in Section 4.4, the build time of Palmtrie$_8$ is shorter than one millisecond for the campus network policy ACLs no larger than D_5 (576 entries). Thus, the time to switch the internal data structure and algorithm between the sorted list and the Palmtrie$_8$ on a threshold around 100 entries is negligible. We can also switch the data structure between Palmtrie$_6$ and Palmtrie$_8$ in a sub-second order on ACLs with no more than 10 thousand entries. Hence, we can dynamically choose the best data structure and algorithm from the sorted and Palmtrie variants for an ACL, although we need to avoid the flapping of data structure switching at a threshold. However, switching between DPDK-ACL and Palmtrie variants is challenging because of the unacceptable build time of DPDK-ACL for extensive ACLs. Instead of using DPDK-ACL, we will look at the datasets where DPDK-ACL performs better and analyze the optimization techniques of DPDK-ACL that contribute to the performance, and then look for further optimizations of the Palmtrie to improve the performance.

Comparisons with decision tree-based packet classification algorithms: As presented in Section 4.3, EffiCuts is relatively slow on commodity CPUs. The decision tree-based packet classification algorithms such as EffiCuts and NeuroCuts are designed to be executed on hardware using SRAM or DRAM. The hardware implementation allows complex logic execution on tree traversals and pipelined memory access. Therefore, the performance of these algorithms has been evaluated by the tree depth or the memory access count instead of lookup rates.

We have not compared the Palmtrie$^+$ with NeuroCuts [19] because the public code of NeuroCuts is in Python, and consequently, it is hard to measure lookup rates. Therefore, we do not directly compare the performance with NeuroCuts in this paper. According to the paper [19], NeuroCuts improves the median performance by only 52% over EffiCuts. As Palmtrie$^+_8$ runs 15.8–49.7 times faster than EffiCuts, we conjecture that Palmtrie^+_k outperforms NeuroCuts as well as EffiCuts,

6 CONCLUDING REMARKS

We proposed a practical algorithm for ACLs named Palmtrie to solve the ternary matching problem. The basic Palmtrie achieved the computational complexity of O($n^{\log_3 2}$) for lookup. A challenge on Palmtries was to support a multi-bit stride to improve the lookup performance by reducing the height of the trie. We devised the multi-bit stride extension and proposed the Palmtrie$^+$ with various optimization techniques. We demonstrated that the Palmtrie outperformed the existing algorithms, the sorted list, EffiCuts, and DPDK-ACL, on extensive ACLs, while the naive sorted list runs faster than the Palmtrie on tiny ACLs. The Palmtrie$^+_8$ achieved 1.05–4.76 times faster lookup performance than DPDK-ACL on the campus network ACLs. It also achieved good build time (e.g., less than 5 seconds for an ACL with one million entries) while DPDK-ACL did not. A Palmtrie is proven to be a scalable and fast algorithm that solves the ternary matching problem. We expect various applications of the Palmtrie, such as flow monitoring [8].

ACKNOWLEDGMENTS

This work has been motivated by the need for real-world network operations. I am grateful to Shin Miyakawa, Yasuhiro Ohara, and Masakazu Asama for their generous support in the early stage of this work. I also thank the anonymous reviewers and shepherd for their invaluable comments on our paper.

REFERENCES

[1] B. Agrawal and T. Sherwood. 2008. Ternary CAM Power and Delay Model: Extensions and Uses. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 16, 5 (May 2008), 554–564. https://doi.org/10.1109/TVLSI.2008.917538

[2] Hirochika Asai. 2019. Deep Pipelining: Efficient Pipelining of Network Function Chains with Coroutines. In *2019 IEEE Conference on Network Softwarization (NetSoft)*. 324–332. https://doi.org/10.1109/NETSOFT.2019.8806673

[3] Hirochika Asai and Yasuhiro Ohara. 2015. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. ACM, New York, NY, USA, 57–70. https://doi.org/10.1145/2785956.2787474

[4] F. Baboescu, Sumeet Singh, and G. Varghese. 2003. Packet classification for core routers: is there an alternative to CAMs?. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, Vol. 1. 53–63 vol.1.

[5] Masanori Bando, Yi-Li Lin, and H. Jonathan Chao. 2012. FlashTrie: Beyond 100-Gb/s IP Route Lookup Using Hash-based Prefix-compressed Trie. *IEEE/ACM Trans. Netw.* 20, 4 (2012), 1262–1275. https://doi.org/10.1109/TNET.2012.2188643

[6] R. Bayer and E. M. Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.* 1, 3 (Sept. 1972), 173–189. https://doi.org/10.1007/BF00288683

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

[8] B. Claise, B. Trammell, and P. Aitken. 2013. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011 (INTERNET STANDARD). http://www.ietf.org/rfc/rfc7011.txt

[9] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[10] Alberto Dainotti, Alistair King, kc Claffy, Ferdinando Papale, and Antonio Pescapè. 2012. Analysis of a "/0" Stealth Scan from a Botnet. In *Proceedings of the 2012 Internet Measurement Conference* (Boston, Massachusetts, USA) *(IMC '12)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/2398776.2398778

[11] DPDK Project. 2018. DPDK: Data Plane Development Kit. http://dpdk.org.

[12] Will Eatherton, George Varghese, and Zubin Dittia. 2004. Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates. *ACM SIGCOMM Comput. Commun. Rev.* 34, 2 (2004), 97–122. https://doi.org/10.1145/997150.997160

[13] ETSI NFV Industry Specification Group. 2014. Network Functions Virtualization – White Paper #3. *ETSI* (2014).

[14] P. Gupta, S. Lin, and N. McKeown. 1998. Routing lookups in hardware at memory access speeds. In *INFOCOM '98. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Vol. 3. 1240–1247 vol.3. https://doi.org/10.1109/INFCOM.1998.662938

[15] Pankaj Gupta and Nick McKeown. 1999. Packet classification using hierarchical intelligent cuttings. In *Hot Interconnects VII*, Vol. 40.

[16] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. *Distributed Computing: 22nd International Symposium, DISC 2008, Arcachon, France, September 22-24, 2008. Proceedings.* Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Hopscotch Hashing, 350–364. https://doi.org/10.1007/978-3-540-87779-0_24

[17] Weirong Jiang, Qingbo Wang, and V.K. Prasanna. 2008. Beyond TCAMs: An SRAM-Based Parallel Multi-Pipeline Architecture for Terabit IP Lookup. In *IEEE INFOCOM*. 1786–1794. https://doi.org/10.1109/INFOCOM.2008.241

[18] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison Wesley Longman Publishing Co., Inc.

[19] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. 2019. Neural Packet Classification. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 256–269. https://doi.org/10.1145/3341302.3342221

[20] P. Marques, N. Sheth, R. Raszuk, B. Greene, J. Mauch, and D. McPherson. 2009. Dissemination of Flow Specification Rules. RFC 5575 (Proposed Standard). http://www.ietf.org/rfc/rfc5575.txt Updated by RFC 7674.

[21] A.J. McAuley and P. Francis. 1993. Fast Routing Table Lookup using CAMs. In *IEEE INFOCOM*, Vol. 3. 1382–1391. https://doi.org/10.1109/INFCOM.1993.253403

[22] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture.. In *USENIX winter*, Vol. 46.

[23] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.* 38, 2 (March 2008), 69–74. https://doi.org/10.1145/1355734.1355746

[24] Donald R. Morrison. 1968. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM* 15, 4 (Oct. 1968), 514–534. https://doi.org/10.1145/321479.321481

[25] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[26] Jiang Qian, Susan Hinrichs, and Klara Nahrstedt. 2001. ACLA: A framework for access control list (ACL) analysis and optimization. In *Communications and Multimedia Security Issues of the New Century*. Springer, 197–211.

[27] Luigi Rizzo. 2012. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association, 9.

[28] Jay Schulist, Daniel Borkmann, and Alexei Starovoitov. [n.d.]. Linux Socket Filtering aka Berkeley Packet Filter (BPF). https://www.kernel.org/doc/Documentation/networking/filter.txt.

[29] Robert Sedgewick. 1997. *Algorithms in C, Parts 1–4: Fundamentals, Data Structures, Sorting, Searching* (3rd ed.). Addison-Wesley Professional.

[30] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet Classification Using Multidimensional Cutting. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (Karlsruhe, Germany) *(SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 213–224. https://doi.org/10.1145/863955.863980

[31] Keith Sklower. 1991. A Tree-Based Packet Routing Table for Berkeley Unix. In *USENIX Winter Conference*. 93–104.

[32] V. Srinivasan and George Varghese. 1998. Faster IP Lookups Using Controlled Prefix Expansion. In *Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems* (Madison, Wisconsin, USA) *(SIGMETRICS '98/PERFORMANCE '98)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/277851.277863

[33] D. E. Taylor and J. S. Turner. 2007. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking* 15, 3 (2007), 499–511.

[34] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, and et al. 2019. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 214–226. https://doi.org/10.1145/3341302.3342088

[35] Balajee Vamanan, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2010. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 207–218. https://doi.org/10.1145/1851182.1851208

[36] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. 2014. Guarantee IP Lookup Performance with FIB Explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) *(SIGCOMM '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2619239.2626297

[37] F. Zane, G. Narlikar, and A. Basu. 2003. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *IEEE INFOCOM*, Vol. 1. 42–52. https://doi.org/10.1109/INFCOM.2003.1208657

[38] Marko Zec, Luigi Rizzo, and Miljenko Mikuc. 2012. DXR: Towards a Billion Routing Lookups Per Second in Software. *SIGCOMM Comput. Commun. Rev.* 42, 5 (Sept. 2012), 29–36. https://doi.org/10.1145/2378956.2378961

[39] Kai Zheng, Chengchen Hu, Hongbin Lu, and Bin Liu. 2006. A TCAM-based Distributed Parallel IP Lookup Scheme and Performance Analysis. *IEEE/ACM Trans. Netw.* 14, 4 (Aug. 2006), 863–875. https://doi.org/10.1109/TNET.2006.880171

[40] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2013. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies* (Santa Barbara, California, USA) *(CoNEXT '13)*. ACM, New York, NY, USA, 97–108. https://doi.org/10.1145/2535372.2535379

A ARTIFACTS

As described in Section 4, the code of the Palmtrie is available at https://github.com/pixos/palmtrie. This repository does not include datasets for the evaluation but those for testing.

The archive file of the code and the datasets to reproduce the key results in this paper, `palmtrie-conext.tar.gz`, is also available as an auxiliary material of this paper on the ACM Digital Library page. README.md in this archive describes the instructions to build the software and conduct the evaluation.