VeriTable: Fast Equivalence Verification of Multiple Large Forwarding Tables

Garegin Grigoryan
grigorg@clarkson.edu
Clarkson University

Yaoqing Liu
liu@clarkson.edu
Clarkson University

Michael Leczinsky
leczinm@clarkson.edu
Clarkson University

Jun Li
lijun@cs.uoregon.edu
University of Oregon

Abstract—**Due to network practices such as traffic engineering and multi-homing, the number of routes—also known as IP prefixes—in the global forwarding tables has been increasing significantly in the last decade and continues growing in a super linear trend. One of the most promising solutions is to use smart Forwarding Information Base (FIB) aggregation algorithms to aggregate the prefixes and convert a large table into a small one. Doing so poses a research question, however, i.e., how can we quickly verify that the original table yields the same forwarding behaviors as the aggregated one? We answer this question in this paper, including addressing the challenges caused by the longest prefix matching (LPM) lookups. In particular, we propose the VeriTable algorithm that can employ a single tree/trie traversal to quickly check if multiple forwarding tables are forwarding equivalent, as well as if they could result in routing loops or black holes. The VeriTable algorithm significantly outperforms the state-of-the-art work for both IPv4 and IPv6 tables in every aspect, including the total running time, memory access times and memory consumption.**

I. Introduction

While every router on the Internet has a main Forwarding Information Base (FIB) (i.e. forwarding table) to direct traffic transmission, there are various scenarios where we need to verify if two or more forwarding tables residing in the same or different routers have the same forwarding behaviors. This identical forwarding behavior is also known as forwarding equivalence. The capability to conduct quick, simultaneous equivalence verification on multiple router forwarding tables is vitally important to ensure efficient and effective network operations. We will use two examples to demonstrate the fundamental significance of this research topic.

First of all, when router vendors develop, test and run their router software and hardware, they must verify that the Forwarding Information Base (FIB) table in the data plane is correctly derived from the Routing Information Base (RIB) table in the control plane. A typical carrier-grade router consists of three primary components: a control engine running various routing protocols, collecting routing information and selecting the best routes to a master forwarding table; many pieces of parallel forwarding engines, called line cards; and a switch fabric linking the control engine and forwarding engines [1]. Based on such distributed system design, routers can achieve better scalability and reliability. This also results in at least three copies of the same forwarding table within a single router. One copy is in the control plane, also known as the master forwarding table, which contains the best routes selected by the RIB. Another copy, mirrored from the master forwarding table, resides in the local memory of each line card. The third copy is maintained in each forwarding ASIC chip, which is in charge of fast IP routing lookup and packet forwarding. In theory, the three copies of forwarding tables should have exactly identical forwarding behaviors. However, in reality, this may not always be true. Thus, we are required to use a highly efficient forwarding table verification scheme for debugging and diagnosis purposes. Moreover, routes are frequently updated by neighbors and these changes need to be simultaneously reflected in all three copies of the forwarding table, which makes fast verification between the copies more challenging. For example, Cisco Express Forwarding (CEF) relies on real-time consistency checkers to discover prefix inconsistencies between RIB and FIB ([2], [3]), due to the asynchronous nature of the distribution mechanism for both databases.

Second, when Internet Service Providers (ISPs) use FIB aggregation techniques to reduce FIB size on a linecard, they must ensure that the aggregated FIB yields 100% forwarding equivalence as the original FIB ([4], [5], [6]). The basic idea is that multiple prefixes which share the same next hop or interface can be merged into one. The best routes that are derived from routing decision processes, e.g., BGP decision process, will be aggregated according to the distribution of their next hops before they are pushed to the FIB. The aggregated copy of the routes with a much smaller size will then be downloaded to the FIB. Unlike many other approaches that require either architectural or hardware changes ([7], [8]), FIB aggregation is promising because it is a software solution, local to single routers and does not require coordination between routers in a network or between different networks. This actually leads to an essential question, how can we quickly verify that the different FIB aggregation algorithms yield results which have the same semantical forwarding as the original FIB, particularly in the case where we need to handle many dynamic updates? Therefore, quick simultaneous equivalence verification on multiple forwarding tables is of great importance to verify the correctness of FIB aggregation algorithms' implementation. Although the real-time requirement of equivalence verification is not very high here, it yields great theoretical value to design advanced algorithms to reduce CPU running time.

More generally, service providers and network operators may want to periodically check if two or more forwarding tables in their network cover the same routing space. Ideally,

TABLE I: FIB Table Forwarding Equivalence

(a) FIB table 1

Prefix	Next hop
-	A
000	B
01	B
11	A
1011	A

(b) FIB table 2

Prefix	Next hop
-	B
001	A
1	A
100	A

all forwarding tables in the same domain are supposed to yield the same set of routes to enable reachability between customers and providers. Otherwise, data packets forwarded from one router may be dropped at the next-hop receiving router, also known as "blackholes". The occurrence of blackhole may stem from multiple reasons, such as misconfigurations, slow network convergence, protocol bugs and so forth. To this end, another essential question is that how we can quickly check if two or more forwarding tables cover the same routing space with consistent routes?

There are at least three challenges to overcome. An efficient algorithm must be able to:

(1) Verify forwarding equivalence over the entire IP address space, including 32-bit IPv4 and 128-bit IPv6, using the Longest Prefix Matching (LPM) rule in a super-fast manner. LPM rule refers to that the most specific routing entry and the corresponding next hop will be selected when there are multiple matches for the same packet. For example, in Table Ia, one packet destined to 01100011 (assume 8-bit address space) has two routing matches: "_" (0/0) with next hop A and 01 with next hop B. However, only the longest prefix 01 and the next hop B will be used to forward the packet out. When we refer to forwarding equivalence, the next hops, derived from LPM lookups against all participating forwarding tables should be identical for every IP address, thus we need to cover the entire IP address space (2^{32} addresses for IPv4 and 2^{128} addresses for IPv6) quickly to check if the condition is satisfied.

(2) Handle very large forwarding tables with a great number of routing entries. For instance, current IPv4 forwarding table size has been over 700,000 entries [9]. IPv6 forwarding tables are fast growing in a super-linear trend (more than 40,000 entries as of July 2017) [10]. It is estimated that millions of routing entries will be present in the global forwarding tables in the next decade [11]. Can we scale up our verification algorithm to handle large forwarding tables efficiently?

(3) Mutually verify the forwarding equivalence over multiple forwarding tables simultaneously. For example, Table I shows two forwarding tables with different prefixes and next hops, how can we quickly verify whether they yield forwarding equivalence or not under the aforementioned LPM rule? Can we scale out our algorithm to deal with many tables together but still yield good performance?

In this work, we conquered all of the challenges and made the following contributions: (1) We have designed and implemented a new approach to verify **multiple snapshots of arbitrary routing/forwarding tables** simultaneously through a single PATRICIA Trie [12] traversal; (2) It is the first time that we examined the forwarding equivalence over both real and large IPv4 and IPv6 forwarding tables; (3) The performance of our algorithm *VeriTable* significantly outperforms existing work *TaCo* and *Normalization*. For *TaCo*, *VeriTable* is **2 and 5.6 times faster** in terms of verification time for IPv4 and IPv6, respectively, while it only uses **36.1% and 9.3% of total memory** consumed by *TaCo* in a two-table scenario. For *Normalization* approach, *VeriTable* is **1.6 and 4.5 times** faster for their total running time for IPv4 and IPv6, respectively; and (4) In a relaxed version of our verification algorithm, we are able to quickly test if multiple forwarding tables cover the same routing space. If not, what are the route leaking points?

This paper has been structured as follows: first we introduce the state-of-the-art verification algorithm *TaCo* and *Normalization* between two tables in Section II; then we present our work *VeriTable* with a compact tree data structure. We depict the design of the algorithm in Section III. We evaluate both IPv4 and IPv6 forwarding tables in terms of running time, number of memory accesses as well as memory usage in the scenarios with two and more tables in Section IV. We describe related work in Section V. Finally, we conclude the paper with future work in Section VI.

II. State of the Art

To the best of our knowledge, there are two state-of-the-art algorithms designed for Forwarding Table Equivalence Verification: *TaCo* [13] and *Normalization* [14].

A. TaCo

TaCo verification algorithm bears the following features: (1) Uses separate Binary Trees (BTs) to store all entries for each forwarding/routing table; (2) Was designed to compare only two tables at once; (3) Has to leverage tree leaf pushing to obtain semantically equivalent trees; and (4) Needs to perform two types of comparisons: direct comparisons of the next hops for prefixes common between two tables and comparisons that involve LPM lookups of the IP addresses, extended from the remaining prefixes of each table. More specifically, *TaCo* needs to use four steps to complete equivalence verification for the entire routing space for Tables Ia and Ib: (a) Building a Binary Tree (BT) for each table, as shown in Figure 1; (b) BT Leaf Pushing, Figure 2 shows the resultant BTs; (c) Finding common prefixes and their next hops, then making comparisons; and (d) Extending non-common prefixes found in both BTs to IP addresses and making comparisons. Due to space constraint, we skip the detailed process.

Finally, when all comparisons end up with the equivalent results, *TaCo* theoretically proves that the two FIB tables yield semantic forwarding equivalence. As a result, *TaCo* undergoes many inefficient operations: (a) BT leaf pushing is time and memory consuming; (b) To find common prefixes for direct comparisons *TaCo* must traverse all trees and it is not a trivial task; (c) IP address extension and lookups for non-common prefixes are CPU-expensive; and (d) To compare n tables and find the entries that cause possible nonequivalence, it may require $(n-1)*n$ times of tree-to-tree comparisons (example: for 3 tables A, B, C there are 6 comparisons: A vs B, A

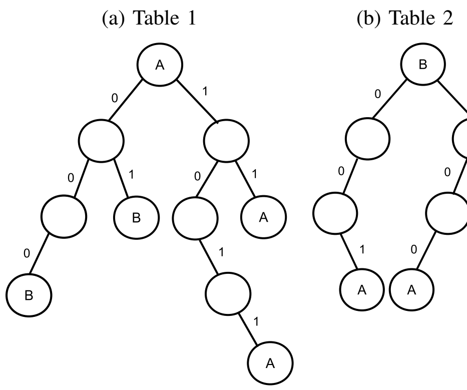


Fig. 1: Binary Prefix Trees

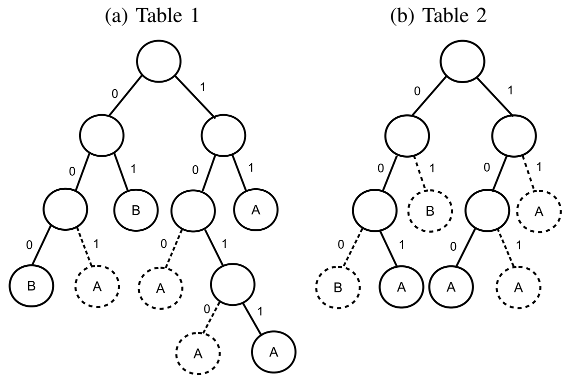


Fig. 2: Binary Prefix Trees After Leaf Pushing

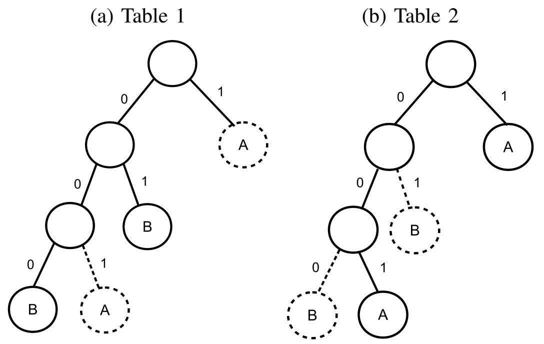


Fig. 3: Binary Prefix Trees After Normalization

vs C, B vs C, B vs A, C vs B, C vs A). For instance, it may require 90 tree-to-tree combinations to compare 10 tables mutually. On the contrary, our work-*VeriTable* eliminates all these expensive steps and accomplishes the verification over an entire IP routing space through a single tree/trie traversal. We describe it in detail in Section III.

B. Tree Normalization

Rétvári *et al.* in [14] show that a unique form of a BT can be obtained through *Normalization*, a procedure that eliminates brother leaves with identical labels (e. g. next hop values) from a leaf-pushed BT. Indeed, if a recursive substitution is applied to the BTs in Figure 2, BT (a) and (b) will be identical. It is possible to prove that the set of tables with the same forwarding behavior have identical normalized BTs. More specifically, *Normalization* verification approach has three steps involved: leaf pushing, tree compression, and side-by-side verification. Leaf pushing refers to formalizing a normal binary tree to a full binary tree, where a node is either a leaf node or a node with two children nodes. The leaf nodes' next hops are derived from their ancestors who have a next hop value originally. Tree compression involves compressing two brother leaf nodes that have the same values to their parent node. This is a recursive process until no brother leaf nodes have the same next hops. The final verification process will traverse every leaf node on both BTs to verify the forwarding equivalence side by side. Figure 3 shows the normalized BTs after the first two steps. As we can observe, normalizing BTs to a unique form involves several inefficient operations including time-consuming leaf pushing and complicated leaf compression. Section IV presents the evaluation results between our algorithm and both *TaCo* and *Normalization* approaches.

III. Design of VeriTable

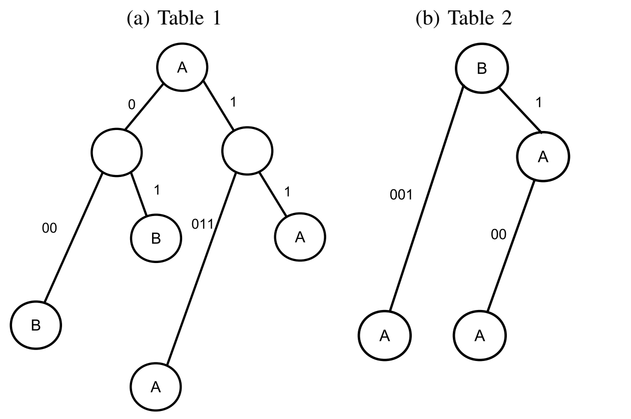
In this section, we introduce the main data structures used in our work first, then describe the terms, design steps and the algorithms of *VeriTable*. Along with the presentation, we use an example to show how the verification scheme works.

A. PATRICIA Trie

Instead of using a BT to store a forwarding table, we use a PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric) Trie [12] data structure, which is based on a radix tree using a radix of two. PATRICIA Trie (PT) is a compressed binary tree and can be quickly built and perform fast IP address prefix matching. For instance, Figure 4 demonstrates the corresponding PTs for FIB Table Ia and FIB Table Ib. The most distinguished part for a PT is that the length difference between a parent prefix and its child prefix can be equal to and greater than 1. This is different than a BT, where the length difference must be 1. As a result, as shown in the example, PTs only require 7 and 4 nodes, but BTs require 10 and 7 nodes for the two tables, respectively. While the differences for small tables are not significant, however, they are significant for large forwarding tables with hundreds of thousands of entries. An exemplar IPv4 forwarding table with 575,137 entries needs 1,620,965 nodes for a BT, but only needs 1,052,392 nodes for a PT. We have detailed comparison in terms of running time and memory consumption in Evaluation Section IV. These features enable a PT to use less space and do faster search, but results in more complicated operations in terms of node creations and deletions, e.g., what if a new node with prefix *100* needs to be added in Figure 4a? It turns out that we have to use an additional glue node to accomplish this task. There are a number of other complex cases, but out of the scope of this work.

B. Design

Our design consists of two primary tasks: Building and initializing a joint PT, and verifying forwarding equivalence in a post-order traversal over the joint PT.



Hollow nodes denote *GLUE* nodes helping build the trie structure, other non-hollow nodes are called *REAL* nodes, whose prefixes were from one of the forwarding tables

Fig. 4: PATRICIA Tries (PTs)

TABLE II: Trie Node's Attributes

Name	Data Type	Description
$parent$	Node Pointer	Points to a node's parent node
l	Node Pointer	Points to a node's left child node if exists
r	Node Pointer	Points to a node's right child node if exists
$prefix$	String	Binary string
$length$	Integer	The length of the prefix, 0-32 for IPv4 or 0-128 for IPv6
$nexthop$	Integer Array	Next hops of this prefix in $T_1 ... T_n$, size n
$type$	Integer	Indicates if a node is a $GLUE$ or $REAL$

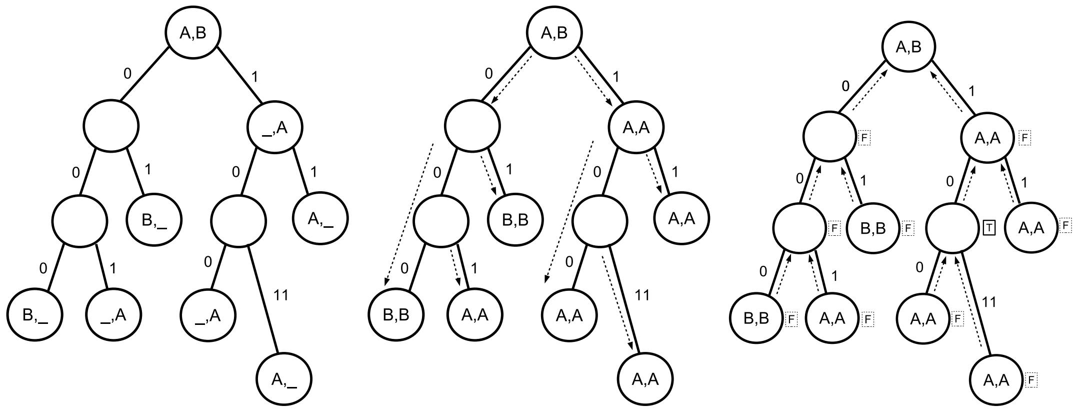
1) Building a Joint PT: Building a joint PT for all routing/forwarding tables. Rather than building multiple BTs for each individual table and comparing them in an one-to-one peering manner, as *TaCo* and *Normalization* do, **we build an accumulated PT using all tables one upon another**. When building the trie, we use a number of fields on each node to help make various decisions. At the beginning, we take the first table as input and initiate all necessary fields to construct a PT accordingly. Afterwards, during the joining process with other tables, the nodes with the same prefixes will be merged. Regarding next hops, we use an integer array to store hops for the same prefix which is located at the same node. The size of the array is the same as the number of tables for comparison. The next hops cannot be merged because they may be different for the same prefix in different tables and also will be used for comparisons, thus they will be placed at the corresponding nth element in the array starting from 0, where n is the index number of the input FIB table (we assume only one next hop for each distinct prefix in an FIB table in this work). For instance, the next hop A of prefix *001* in FIB Table 2 will be assigned as the second element in the *Next Hop* Array on the node with prefix *001*. If there is no next hop for a prefix in a particular table, the element value in the array will be initialized as "_" by default, or called *"empty"* next hop (we used "-1" in our implementation). If there is at least one "*non-empty*" next hop in the array, we mark the *Node Type* value as *REAL*, indicating this node contains a real prefix and next hop derived from one of the tables. Otherwise, we call it a *GLUE* node. Algorithm 1 elaborates the detailed workflow to build a joint PT for multiple tables. Table II describes a trie node's attributes in our data structure. Figure 5a shows the resultant joint PT for FIB Table Ia and Ib.

Algorithm 1 Building a Joint PT T

```
1: procedure $BuildJointPT(T_{1..n})$
2:     Initialize a PT $T$ with its head node
3:     Add prefix $0/0$ on its head node.
4:     Set default next hop values in the Next Hops array.
5:     for each table $T_i \in T_{1..n}$ do
6:         for each entry $e$ in $T_i$ do
7:             Find a node $n$ in $T$ such as $n.prefix$ is a
                longest match for $e.prefix$ in $T$
8:             if $n.prefix = e.prefix$ then
9:                 $n.nexthop_i \leftarrow e.nexthop$
10:                $n.type \leftarrow REAL$
11:            else
12:                Generate new node $n'$
13:                $n'.prefix \leftarrow e.prefix$
14:                $n'.nexthop_i \leftarrow e.nexthop$
15:                $n'.type \leftarrow REAL$
16:                Assume $n$ has a child $n_c$
17:                if the overlapping portion of $n_c$ and $n'$ is
    longer than $n.length$ but shorter than $n'.length$ bits then
18:                    Generate a glue node $g$
19:                    $n'.parent \leftarrow g$
20:                    $n_c.parent \leftarrow g$
21:                    $g.parent \leftarrow n$
22:                    $g.type \leftarrow GLUE$
23:                    Set $g$ as a child of $n$
24:                    Set $n'$ and $n_c$ as children of $g$
25:                else
26:                    $n'.parent \leftarrow n$
27:                    $n_c.parent \leftarrow n'$
28:                    Set $n_c$ as a child of $n'$
29:                    Set $n'$ as a child of $n$
30:                end if
31:            end if
32:        end for
33:    end for
34: end procedure
```

There are a few advantages for the design of a joint PT: (a) Many common prefixes among different tables will share the same trie node and prefix, which can **considerably reduce memory consumption and computational time for new node creations**; (b) Common prefixes and uncommon prefixes will be automatically gathered and identified in one single tree after the combination; and (c) The design will greatly speed up subsequent comparisons of next hops between multiple tables without traversing multiple tries.

2) Post-Order Equivalence Verification: After building the joint PT and initializing all necessary fields, we start the



(a) Initial Joint PT (b) Joint PT after top-down process (c) Joint PT after bottom-up verification

In Figure a, for *REAL* nodes, the nth element denotes the next hop value of the corresponding prefix from the nth forwarding table. "_" indicates that no such prefix and next hop exist in the forwarding table. In Figure b, after each top-down step, the fields with previous "_" value will be filled with new next hop values derived from the corresponding *Next Hop* array elements of its nearest *REAL* node. In Figure c, F denotes $False$ and T denotes $True$ for the *LEAK* flag. *GLUE* nodes will carry T over to its parent recursively until finding a *REAL* node.

Fig. 5: VeriTable Algorithm

verification process, which only needs one post-order PT traversal and includes two steps to accomplish the forwarding equivalence verification as follows:

(A) Top-down inheriting next hops. First, we follow a simple but very important rule: ***According to the LPM rule, the real next hop value for a prefix that has an "empty" next hop on the joint PT should be inherited from its closest REAL ancestor, whose next hop exists and is "non-empty"***. For example, to search the LPM matching next hop for prefix *000* in the second table using Figure 5a, the next hop value should return B, which was derived from the second next hop B of its nearest *REAL* ancestor – the root node. The top-down process will help each specific prefix on a *REAL* node in the joint PT to inherit a next hop from its closest *REAL* ancestor if the prefix contains an "empty" next hop. More specifically, when moving down, we compare the *Next Hops* array in the *REAL* ancestor node with the array in the *REAL* child node. If there are elements in the child array with "empty" next hops, then our algorithm fills them out with the same values as the parent. If there are "non-empty" next hops present in the child node, then we keep them. Note that all *GLUE* nodes (hollow nodes in Figure 5a) will be skipped during this process, because they are merely ancillary nodes helping build up the trie structure and do not carry any next hop information. After this step, every *REAL* node will have a new *Next Hops* array without any *"empty"* next hops. **The instantiated next hops will facilitate the verification process without additional retrievals of next hops from their distant ancestors.** Figure 5b shows the results after the top-down step. If there is not a default route 0/0 in the original forwarding tables, we make up one with next hop value 0 and node type *REAL* for calculation convenience.

(B) Bottom-up verify LPM next hops. In fact, this process is interwoven with the top-down process in our recursive post-order verification program. While the program moves downward, the top-down operations will be executed. While it moves upward, a series of operations will be conducted as follows. First of all, a leaf node at the bottom may be encountered, where the *Next Hops* array will be checked linearly, element by element. If there are any discrepancies, then we can immediately conclude that the forwarding tables yield different forwarding behaviors, because the LPM prefixes end up with different next hops. In other words, they are not forwarding equivalent. If all next hops share the same value, we move upward to its **directly connected** parent node, where we check the prefix length difference from the recently visited child node.

Since we use a PT as our data structure, two cases may occur: $d = 1$ and $d > 1$, where d denotes the length difference between the parent node and the child node. The first case $d = 1$ for all children nodes implies that the parent node has no extra routing space to cover between itself and the children nodes. On the contrary, the second case $d > 1$ indicates the parent node covers more routing space than that of all children nodes. If $d > 1$ happens at any time, we will set a *LEAK* flag variable at the parent node to indicate that all of the children nodes are not able to cover the same routing space as the parent, which will lead to "leaking" certain routing space to check for verification. Therefore, in this case, the parent itself needs to be checked by the LPM rule to make sure the "leaking" routing space is checked as well. If there is no child for a given parent, we consider it as $d > 1$. As long as there is one *LEAK* flag initiated at a parent node, the flag will be carried over up to the nearest *REAL* node, which can be the parent node itself or a further ancestor. The verification process of forwarding equivalence will be conducted on the *Next Hops* array of this *REAL* node. Once the process passes over a *REAL* node, the flag will be cleared so that the "leaking" routing space will not be double checked. Intuitively, we check the forwarding equivalence over the routing space covered by leaf nodes first, then over the remaining "leaking" routing space covered by internal *REAL* nodes. Figure 5c demonstrates the bottom-up *LEAK* flag setting and carried-over process. For example, $d = 2$ between parent *10* and its child *1011*, so the *LEAK* flag on node *10* will be set to $True$ first. Since node *10* is a *GLUE* node, the *LEAK* flag will be carried over to its nearest *REAL* ancestor node *1* with the *Next Hops* array *(A,A)*, where the $leaking$ routing space will be checked and accordingly the *LEAK* flag will be cleared to $False$ to avoid future duplicate checks.

In Algorithm 2, we show the pseudocode of our recursive function *VeriTable* that quickly verifies whether the multiple forwarding tables are equivalent or not. If not, the corresponding LPM prefixes and next hops will be printed out. The algorithm consists of both a top-down next hop inheritance process and a bottom-up LPM matching and verification process. We have mathematically proved the correctness of the algorithm but do not present the proof here due to limited space.

Algorithm 2 Forwarding Equivalence Verification. The initial value of $ancestor$ is $NULL$, and the initial value of $node$ is $T \rightarrow root$. For simplicity, we assume the root node is $REAL$.

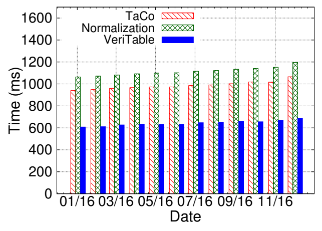
```
1: procedure VeriTable($ancestor$, $node$)
2:     if $node.type = REAL$ then
3:         $ancestor$ = $node$ ▷ The closest ancestor node for a REAL node is the node istelf
4:     end if
5:     $l \leftarrow node.l$
6:     $r \leftarrow node.r$
7:     if $l \neq NULL$ then
8:         if $l.type = REAL$ then
9:             $InheritNextHops(ancestor, l)$ ▷ A REAL child node inherits next hops from the closest REAL ancestor to initialize “empty” next hops
10:        end if
11:        $LeftFlag \leftarrow VeriTable(ancestor, l)$ ▷ LeftFlag and RightFlag signify the existing leaks at the branches
12:    end if
13:    if $r \neq NULL$ then
14:        if $r.type = REAL$ then
15:            $InheritNextHops(ancestor, r)$
16:        end if
17:        $RightFlag \leftarrow VeriTable(ancestor, r)$
18:    end if
19:    if $l = NULL \wedge r = NULL$ then
20:        $CompareNextHops(node)$ ▷ The leaf nodes’ next hops are always compared; a verified node always returns the false LeakFlag.
21:        $LeakFlag \leftarrow False$
22:        return $LeakFlag$
23:    end if
24:    if $l \neq NULL \wedge l.length - node.length > 1$ then
25:        $LeakFlag \leftarrow True$
26:    else if $r \neq NULL \wedge r.length - node.length > 1$ then
27:        $LeakFlag \leftarrow True$
28:    else if $l = NULL \vee r = NULL$ then
29:        $LeakFlag \leftarrow True$
30:    else if $LeftFlag = True \vee RightFlag = True$ then
31:        $LeakFlag \leftarrow True$
32:    end if
33:    if $LeakFlag = True \wedge node.type = REAL$ then
34:        $CompareNextHops(node)$
35:        $LeakFlag \leftarrow False$
36:    end if
       return $LeakFlag$
37: end procedure
```

IV. Evaluation

All experiments are run on a machine with Intel Xeon Processor E5-2603 v3 1.60GHz and 64GB memory. Datasets are provided by the RouteViews project of the University of Oregon (Eugene, Oregon USA) [15]. We collected 12 IPv4 RIBs and 12 IPv6 RIBs on the first day of each month in 2016, and used AS numbers as next hops to convert them into 24 routing/forwarding tables. By the end of 2016, there were about 633K IPv4 routes and 35K IPv6 routes in the global forwarding tables. We then applied an optimal FIB aggregation algorithm to these tables to obtain the aggregated forwarding tables. IPv4 yields a better aggregation ratio (about 25%) than IPv6 (about 60%), because IPv4 has a larger number of prefixes. The original and aggregated tables were semantically equivalent and used to evaluate the performance of our proposed *VeriTable* vs the state-of-the-art *TaCo* and *Normalization* (see description in Section II) verification algorithms in a two-table scenario. We use the following metrics for the evaluations: tree/trie building time, verification time, number of node accesses, and memory consumption.

A. Tree/Trie Building Time

TaCo, *Normalization* and *VeriTable* need to build their data structures using forwarding table entries before the verification process. *TaCo* and *Normalization* need to build two separate BTs while *VeriTable* only needs to build one joint PT. Figure 6 shows the building time for both IPv4 and IPv6. Our algorithm *VeriTable* outperforms *TaCo* and *Normalization* in both cases. In Figure 6a for IPv4, *TaCo* uses minimum 939.38ms and maximum 1065.41ms with an average 986.27ms to build two BTs. For *Normalization*, it is 1063.42ms, 1194.95ms and 1113.96ms respectively. Our *VeriTable* uses minimum 608.44ms and maximum 685.02ms with an average 642.27ms to build a joint PT. *VeriTable* only uses 65.11% of the building time of *TaCo* and 57.65% of the building time of *Normalization* for IPv4 tables. In the scenario of IPv6 in Figure 6b, *TaCo* uses minimum 137.94ms and maximum 186.73ms with an average 168.10ms to build two BTs; for *Normalization* these numbers are 162.40ms, 225.75ms and 197.25ms. Our *VeriTable* uses minimum 36.39ms and maximum 49.99ms with an average 45.06ms to build a joint PT. *VeriTable* only uses 26.78% and 22.84% of the building time of *TaCo* and *Normalization* respectively for IPv6 tables. Although IPv6 has much larger address space than IPv4, *VeriTable* yields much less building time under IPv6 than that of IPv4, which we attribute to the small FIB size and the usage of a compact


(a) IPv4 Building Time

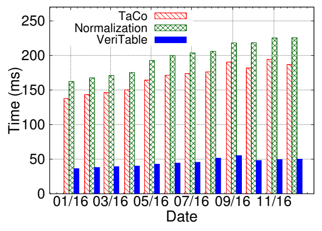

(b) IPv6 Building Time

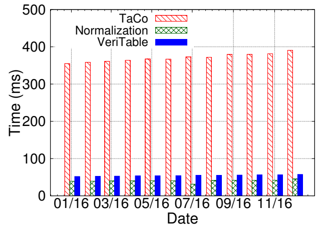
Fig. 6: IPv4 and IPv6 Tree/Trie Building Time

data structure – a joint PT in our algorithm. Note the slower *Normalization* building time due to the operation of tree compression performed by that algorithm.

B. Verification Time

A valid verification algorithm needs to cover the whole routing space (2^{32} IP addresses for IPv4 and 2^{128} IP addresses for IPv6) to check if two tables bear the same forwarding behaviors. The verification time to go through this process is one of the most important metrics that reflects whether the algorithm runs efficiently or not. Figure 7 shows the running time of *TaCo*, *Normalization* and *VeriTable* for both IPv4 and IPv6, respectively. Our *VeriTable* significantly outperforms *TaCo* in both cases. *TaCo* takes minimum 355.06ms and maximum 390.60ms with an average 370.73ms to finish the whole verification process. Our *VeriTable* takes minimum 51.91ms and maximum 57.48ms with an average 54.63ms to verify the entire IPv4 routing space. *VeriTable* only takes 14.73% of the verification time of *TaCo* for verification over two IPv4 tables. **Taking building time into consideration, *VeriTable* is about 2 times faster than *TaCo* for IPv4 verification (1356ms VS 696ms).** *Normalization* verification time for IPv4 tables is slightly faster than that of *VeriTable* (which is not the case for IPv6 tables). This is achieved due to the compression that shrinks the size of the BTs for verification process. However, *Normalization* has much longer building time than *VeriTable*. **Overall, considering both building and verification time, *VeriTable* is faster than *Normalization* by 40% (696.90ms VS 1154.08ms) for IPv4 verification**.

Figure 7b shows the IPv6 scenario (note the Y-axis is a log scale). *TaCo* takes minimum 75.17ms and maximum 103.18ms with an average 92.79ms to finish the whole verification process. For *Normalization* it is 11.47ms, 15.58ms, 13.87ms respectively. Our *VeriTable* takes minimum 1.44ms and maximum 1.97ms with an average 1.75ms to verify the entire IPv6 routing space. *VeriTable* only takes 1.8% and 12.6% of the verification time of *TaCo* and *Normalization* respectively for verification over two IPv6 tables. **Considering both building and verification time, *VeriTable* is 5.6 times faster than *TaCo* (261ms VS 47ms) and 4.5 times faster than *Normalization* (211ms VS 47ms) for IPv6 verification**. The fundamental cause for such a large performance gap is due to the single trie traversal used in *VeriTable* over a joint PT with intelligent selection of certain prefixes for comparisons without tree normalization, see Section III-B2 in detail. Note, that the leaf pushing operation over IPv6 forwarding table causes a significant inflation of the BTs which explains much


(a) IPv4 Verification Time

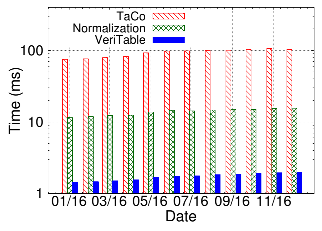
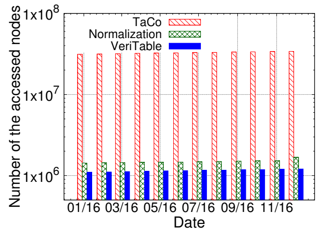

(b) IPv6 Verification Time

Fig. 7: IPv4 and IPv6 Verification Time


(a) IPv4 Node Accesses

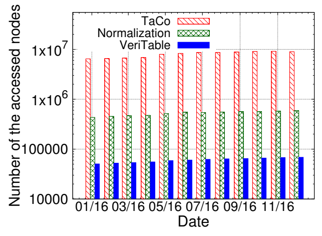

(b) IPv6 Node Accesses

Fig. 8: IPv4 and IPv6 Number of Node Accesses

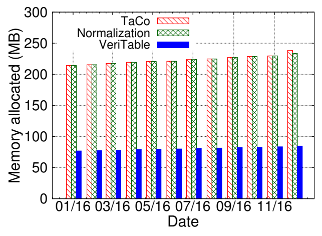
slower speed of *TaCo* and *Normalization* verification for IPv6 tables than for IPv4 tables.

C. Number of Node Accesses

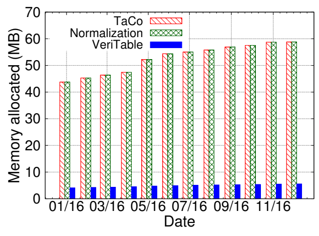
Node accesses, similarly to memory accesses, refer to how many tree/trie nodes will be visited during verification. The total number of node accesses is the primary factor to determine the verification time of an algorithm. Figure 8 shows the number of total node accesses for both IPv4 and IPv6 scenarios. Due to the novel design of *VeriTable*, we are able to control the total number of node accesses to a significantly low level. For example, node accesses range from 1.1 to 1.2 million for 580K and 630K comparisons, which is **less than 2 node accesses** per comparison for IPv4, and it yields similar results for IPv6. On the contrary, *TaCo* and *Normalization* requires larger number of node accesses per comparison. For instance, *TaCo* bears **35 node accesses** per comparison, on average, for IPv4 and **47 node accesses** per comparison, on average, in IPv6. *Normalization* has **4 node accesses** per comparison in both cases. There are two main reasons for the gaps between *VeriTable* and *TaCo* and *Normalization*: (a) *VeriTable* uses a joint PT but *TaCo* and *Normalization* uses separate BTs. In a BT, it only goes one level down for each search step while multiple levels down in a PT; and (b) *VeriTable* conducts only one post-order PT traversal. *TaCo* conducts many repeated node accesses over a BT, including searching for a match from the BT root, using simply longest prefix matching process for each comparison. Due to the the unique form of a normalized BT, *Normalization* requires no mutual IP address lookups and thus conducts significantly less node accesses than *TaCo*.

D. Memory Consumptions

Memory consumption is another important metric to evaluate the performance of algorithms. Figure 9 shows the comparisons between *TaCo*, *Normalization* and *VeriTable* for both IPv4 and IPv6 in terms of their total memory consumptions. In both scenarios, *VeriTable* outperforms *TaCo* and



(a) IPv4 Memory Consumption



(b) IPv6 Memory Consumption

Fig. 9: IPv4 and IPv6 Memory Consumption

Normalization significantly. ***VeriTable* only consumes around 38% (80.86*MB*)** of total memory space than that of *TaCo* and *Normalization* (223*MB*) on average for the same set of IPv4 forwarding tables. In the IPv6 case, *VeriTable* bears even more outstanding results, **which only consumes 9.3% (4.9*MB*)** of total memory space than that of *TaCo* (53*MB*) and *Normalization* on average. Overall, thanks to the new design of our verification algorithm, *VeriTable* outperforms *TaCo* and *Normalization* in all aspects, including total running time, number of node accesses and memory consumption.

The differences in memory consumption by *VeriTable*, *Normalization* and *TaCo* are caused by the unique combined trie data structure used in *VeriTable*. A node in *Normalization* and *TaCo* holds a single next hop instead of an array of next hops, because *TaCo* and *Normalization* build separate BTs for each forwarding table. Moreover, those BTs inflate after leaf pushing.

E. *Performance for Multiple Tables*

We also evaluated the performance of *VeriTable* to check the forwarding equivalence and differences over multiple forwarding tables simultaneously. In the experiments, we intentionally added 2000 distinct errors when a new forwarding table was added. Then we verified that the same number of errors will be detected by *VeriTable* algorithm. Starting from 2 tables, we gradually checked up to 10 tables simultaneously. The evaluation results have been shown in Table III. There are two primary observations. First, *VeriTable* is able to check the whole address space very quickly over 10 large forwarding tables (336.41ms) with relatively small memory consumptions (165MB). Second, the building time, verification time, node accesses, and memory consumptions grow much slower than the total number of forwarding entries. This indicates that *VeriTable* can scale quite well for equivalence checking of a large number of tables. On the contrary, *TaCo* and *Normalization* naturally was not designed to compare multiple forwarding tables. In theory, *TaCo* may need $n * (n - 1)$ table-to-table comparisons to find the exact entries that cause differences, which is equal to 90 comparisons for this 10-table scenario. On the other hand, *Normalization* needs additional decompression steps to find such entries. We skip evaluation of *TaCo* and *Normalization* for multiple tables due to the high complexity.

F. *"Blackholes" Detection*

A relaxed version with minor changes of our *VeriTable* algorithm is able to quickly detect the routing space differences between multiple FIBs. More specifically, after building the joint PT for multiple FIBs, *VeriTable* goes through the same verification process recursively. When traversing each *Next Hops* array, it checks if there is a scenario where the array contains at least one default next hop (the next hop on default route 0/0) and at least one non-default next hop. If yes, it indicates that at least one FIB misses some routing space while another FIB covers it, which may lead to routing "blackholes". In our experiments, we used data from RouteViews [15] project, where 10 forwarding tables that contain the largest number of entries were collected and then merged to a super forwarding table with 691,998 entries. Subsequently, we one-to-one compared the routing spaces of the 10 individual forwarding tables with the super forwarding table. The results of these comparisons (see Table IV in detail) show that none of these 10 forwarding tables fully cover the routing space of the merged one. The Leaking Routes in Table IV were calculated by the number of subtrees in the joint PT under which an individual forwarding table "leaks" certain routes but the merged super forwarding table covers them. These facts imply that the potential routing blackholes may take place between routers in the same domain or between different domains. To this end, our *VeriTable* verification algorithm can identify these potential blackholes efficiently. On the contrary, *TaCo* and *Normalization* may not be easily used to detect block holes and loops because different FIBs may result in different shapes of BTs (even normalized), which makes it hard for comparison.

V. Related Work

TaCo algorithm, proposed by Tariq et al. [13], is designed to verify forwarding equivalence between two forwarding tables. *TaCo* builds two separate binary trees for two tables and performs tree normalization and leaf-pushing operations. Section II elaborates the algorithm in detail. *VeriTable* is very different from *TaCo*. *VeriTable* builds a single joint PATRICIA Trie for multiple tables and leverages novel ways to avoid duplicate tree traversals and node accesses, thus outperforms *TaCo* in all aspects as shown in Section IV. Inconsistency of forwarding tables within one network may lead to different types of problems, such as blackholes, looping of IP packets, packet losses and violations of forwarding policies. Network properties that must be preserved to avoid misconfiguration of a network can be defined as a set of invariants. Mai et al. introduced *Anteater* in [16], that converts the current network state and the set of invariants into instances of boolean satisfiability problem (SAT) and resolves them using heuristics-based SAT-solvers. Zeng et al. introduced *Libra* in [17], and they used MapReduce [18] to analyze rules from forwarding tables on a network in parallel. Due to the distributed model of MapReduce, *Libra* analyzes the forwarding tables significantly faster than *Anteater*. *VeriFlow* [19], proposed by Khurshid et al., leverages software-defined networking to collect forwarding rules and then slice the network into *Equivalence classes* (*ECs*). Kazemian et al. introduced *NetPlumber* in [20],

TABLE III: Results of Comparing 10 IPv4 FIB Tables Simultaneously

Number of tables	Total number of entries	Building time(ms)	Verification time (ms)	Number of comparisons	Node access times	Number of errors	Memory (MB)
2	1230512	962	82	586942	1133115	4000	84
3	1845768	1326	108	1175884	1137115	6000	94
4	2461024	1684	135	1766826	1141115	8000	104
5	3076280	2060	172	2359768	1145115	10000	114
6	3691536	2471	194	2954710	1149115	12000	124
7	4306792	2869	213	3551652	1153115	14000	134
8	4922048	3248	224	4150594	1157115	16000	145
9	5537304	3630	322	4751536	1161115	18000	155
10	6152560	4007	337	5354478	1165115	20000	165

TABLE IV: One-to-one Comparison of Individual Forwarding Tables with the Merged Super Table

Table size	Router IP	ASN	BGP peers	Leaking Routes
673083	203.189.128.233	23673	204	489
667062	202.73.40.45	18106	1201	507
658390	103.247.3.45	58511	1599	566
657232	198.129.33.85	292	153	495
655528	64.71.137.241	6939	6241	667
655166	140.192.8.16	20130	2	879
646912	85.114.0.217	8492	1504	796
646892	195.208.112.161	3277	4	772
641724	202.232.0.3	2497	294	1061
641414	216.221.157.162	3257	316	1239

a real-time network analyzer based on *Header Space Analysis* protocol-agnostic framework, described in [21]. *NetPlumber* is compatible with both SDN and conventional networks. It incrementally verifies the network configuration upon every policy change in a quick manner. Different from the network-wide verification methods above, *VeriTable* aims to investigate whether multiple static forwarding tables yield the same forwarding behaviors-given any IP packet with a destination address or they cover the same routing space.

VI. Conclusion

We designed and developed *VeriTable*, which can quickly determine if multiple routing or forwarding tables yield the same or different forwarding behaviors, and our evaluation results using real forwarding tables significantly outperform its counterparts. The novel algorithms and compact data structures can offer benefit not only in forwarding equivalence verification scenarios, but also in many other scenarios where we use Longest Prefix Matching rule for lookups, e.g., checking if route updates in control plane are consistent with the ones in forwarding plane. Moreover, the principles used in this paper can be applied to network-wide abnormality diagnosis of network problems, such as scalable and efficient forwarding loop detection and avoidance in the data plane of a network. In addition, VeriTable can be extended to handle incremental updates applied to the forwarding tables in a network. Our future work will explore these directions.

References

[1] H. Chao and B. Liu, *High Performance Switches and Routers*, ser. Wiley - IEEE. Wiley, 2007, pp. 5–9. [Online]. Available: https://books.google.com/books?id=kzstoFdvZ2sC

[2] "Troubleshooting Prefix Inconsistencies with Cisco Express Forwarding," http://www.cisco.com/c/en/us/support/docs/ip/express-forwarding-cef/14540-cefincon.html.

[3] "Configuring CEF Consistency Checkers," http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/ipswitch_cef/configuration/12-4/isw-cef-12-4-book/isw-cef-checkers.html.

[4] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill, "Constructing optimal IP routing tables," vol. 1, pp. 88–97, 1999.

[5] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis, "SMALTA: practical and near-optimal FIB aggregation," in *Proceedings of the Seventh Conference on emerging Networking EXperiments and Technologies*. ACM, 2011, p. 29.

[6] Y. Liu, B. Zhang, and L. Wang, "FIFA: Fast incremental FIB aggregation," in *INFOCOM, 2013 Proceedings IEEE*. IEEE, 2013, pp. 1–9.

[7] D. McPherson, S. Amante, and L. Zhang, "The Intra-domain BGP Scaling Problem," *RIPE 58, Amsterdam*, 2009.

[8] D. Saucez, L. Iannone, O. Bonaventure, and D. Farinacci, "Designing a deployable internet: The locator/identifier separation protocol," *IEEE Internet Computing*, vol. 16, no. 6, pp. 14–21, 2012.

[9] CIDR, "The CIDR report." [Online]. Available: http://www.cidr-report.org/

[10] "AS131072 IPv6 BGP Table Data," http://bgp.potaroo.net/v6/as2.0/index.html.

[11] Geoff Huston, "BGP in 2016." [Online]. Available: https://blog.apnic.net/2017/01/27/bgp-in-2016/

[12] D. R. Morrison, "PATRICIA practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 514–534, 1968.

[13] A. Tariq, S. Jawad, and Z. A. Uzmi, "TaCo: Semantic Equivalence of IP Prefix Tables," in *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE, 2011, pp. 1–6.

[14] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, and Z. Heszberger, "Compressing IP forwarding tables: towards entropy bounds and beyond," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 111–122.

[15] Advanced Network Technology Center and University of Oregon, "The RouteViews project." [Online]. Available: http://www.routeviews.org/

[16] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 290–301.

[17] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat, "Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks," in *NSDI*, vol. 14, 2014, pp. 87–99.

[18] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[19] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.

[20] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real Time Network Policy Checking Using Header Space Analysis." in *NSDI*, 2013, pp. 99–111.

[21] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks." in *NSDI*, vol. 12, 2012, pp. 113–126.