VeriTable: Fast equivalence verification of multiple large forwarding tables

Yaoqing Liu*, Garegin Grigoryan, Jun Li, Guchuan Sun, Tony Tauber

Computer Science Department, Fairleigh Dickinson University, 1000 River Road, Teaneck, NJ 07666 United States

A R T I C L E   I N F O

Article history:
Received 1 January 2019
Revised 2 October 2019
Accepted 4 November 2019
Available online 4 December 2019

Keywords:
Routing table forwarding equivalence
Forwarding verification
Longest prefix matching
Routing scalability
Network diagnosis
Patricia trie algorithm

A B S T R A C T

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 Forwarding Information Base (FIB) aggregation algorithms with incremental handling of BGP updates. FIB aggregation compresses the forwarding tables by reducing the number of prefixes in an FIB. Obviously, FIB aggregation should preserve the forwarding behavior of the data plane, i.e., packets must be forwarded in the same directions as if the original FIB is applied. Failures at the control plane or an incorrect algorithm may violate this rule. Thus we pose a research question, how can we efficiently verify that the original table achieves the same forwarding behavior for a router as the aggregated one? This paper proposes VeriTable, an algorithm that addresses the problem of verification of the equivalence of forwarding tables and the challenges caused by the Longest Prefix Matching (LPM) lookups. VeriTable employs a single tree traversal to quickly check if multiple forwarding tables are equivalent, as well as if they result in network "blackholes”. VeriTable algorithm significantly outperforms the state-of-the-art work for both IPv4 and IPv6 tables in terms of the total running time, memory access times and memory consumption.

© 2019 Elsevier B.V. All rights reserved.

1. Introduction

While the amounts of the Internet traffic continues to increase, the Internet Service Providers' (ISP) task to provide fast, consistent and loop-free routing becomes more challenging. For example, the size of the global routing table, that is installed on backbone routers, exceeds 700,000 entries for IPv4 addresses [2] (see Fig. 1). The great amount of entries in the forwarding table leads to the problem known as the overflow of the Ternary Content Addressable Memory (TCAM). As TCAM is used for fast hardware-based selection of the Longest Prefix Match for each incoming packet, it has memory limitations, increased power consumption, and high operational costs [3–5]. To mitigate this problem, Forwarding Information Base (FIB) aggregation is considered as one of the most promising solutions [6]. The basic idea of FIB aggregation is that multiple prefix entries sharing a same next hop can be merged into a single entry. Unlike many other approaches, FIB aggregation requires neither architectural or hardware changes [7,8]. FIB aggregation is a software solution and can be applied locally on a single router. While FIB aggregation can compress forwarding tables by more than 50% [9], it is necessary to verify the correctness of such compression. More specifically, the forwarding behavior of a router should not change after the aggregated forwarding table replaces the original forwarding table. Moreover, the aggregation algorithm should correctly conduct incremental routing updates. Thus, the packet with any possible destination address will be forwarded or dropped regardless of what table was used, the original or the aggregated one.

This work is dedicated to the general problem of verifying the equivalence of forwarding tables, i.e. if a router's forwarding behavior is the same for each of the comparable tables. Besides a natural application as validating the correctness of FIB aggregation and the incremental updates, verifying the forwarding equivalence is necessary for testing a router’s hardware and software. Due to distributed system design of a traditional router, it contains at least three copies of the same forwarding table. The first copy, called the master forwarding table, is located at the control plane of the router, which is responsible for collecting, selecting and distributing routes. The master forwarding table, in its turn, is derived from a Routing Information Base (RIB). The second copy is located in the forwarding engines of a router. Finally, the third copy is located in forwarding chips with TCAM memory. All the copies of

---

* This paper is an extension of our previous publication "VeriTable: Fast Equivalence Verification of Multiple Large Forwarding Tables" [1] that appeared in INFOCOM 2018.

* Corresponding author.

E-mail address: yliu@fdu.edu (Y. Liu).

https://doi.org/10.1016/j.comnet.2019.106981

1389-1286/© 2019 Elsevier B.V. All rights reserved.
the forwarding tables on a router must be equivalent, which may not be the case when a router is incorrectly configured or after link failures [10]. In addition, the routing updates accepted by a router shall simultaneously be reflected in all of its forwarding tables. Thus, verifying the equivalence of multiple forwarding tables is necessary for debugging and diagnosing misconfiguration. An example of such software is Cisco Express Forwarding (CEF) real-time consistency checkers, that discover prefix inconsistencies between the RIB and FIB ([11,12]). Such inconsistencies may happen due to the asynchronous nature of the distribution mechanism for both databases. They include missing prefix or different next hops on a line card and in the RIB.

A relaxed version of the forwarding equivalence verification, when an algorithm needs to verify if certain routes are missing in at least one of the comparable forwarding tables, but exist in other forwarding table(s), will help network operators to prevent so-called network "black-holes". The existence of "black-holes" may lead to a silent disappearance of the traffic destined for a certain scope of IP addresses. Such failures may occur due to several reasons, such as misconfiguration of an individual router and slow network convergence [13].

There are at least four challenges necessary to overcome for designing an efficient algorithm, that verifies the equivalence of forwarding tables:

1. Verify forwarding equivalence over the entire IP address space, i.e. $2^{32}$ IPv4 addresses and $2^{128}$ IPv6 addresses. More specifically, the equivalence condition is satisfied only if, for each IP address, the Longest Prefix Matching operation in FIB results in a same next hop.
2. An efficient algorithm should be able to handle large forwarding tables for both IPv4 and IPv6 forwarding tables. It is estimated that millions of routing entries will be present in the global forwarding table in the next decade [14].
3. The algorithm should be fast enough for processing incremental updates. Typically, the number of such updates is 100 per second on average, however, it can reach the frequency of several thousand updates per second during the spikes.

4. In several cases, such as verification of a "black-hole-free" network or verifying the equivalence of all forwarding tables on the same device, the algorithm should be able to process multiple large forwarding tables simultaneously.

This work presents VeriTable, an algorithm that conquered all of the above-mentioned challenges and makes the following contributions:

1. It presents the design and the implementation of an algorithm that verifies multiple snapshots of arbitrary routing/forwarding tables simultaneously through a single PATRICIA tree [15] traversal.
2. For the first time, this work examines the forwarding equivalence over both real and large IPv4 and IPv6 forwarding tables; in addition, it for the first time demonstrates the results of aggregation of IPv6 forwarding tables.
3. VeriTable significantly outperforms existing work TaCo and Normalization. This work both demonstrates and evaluates these two algorithms, using IPv4 and IPv6 forwarding tables. According to the evaluation results, VeriTable is 2 and 5.6 times faster than TaCo 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, VeriTable is 1.6 and 4.5 times faster in terms of the total running time for IPv4 and IPv6, respectively.
4. In a relaxed version of VeriTable, it is able to quickly test if multiple forwarding tables cover the same routing space. We also extended the algorithm to quickly identify if there are loops and blackholes in a large network. The evaluation results are described in Section 4.

The rest of this paper is organized as follows. Section 2 presents the necessary background information on the Internet organization, a router’s architecture, the Longest Prefix Match rule and the problem of verifying the equivalence of forwarding tables. In addition, it presents two state-of-the-art solutions: TaCo and Normalization. Section 3 describes the theorem and property used in VeriTable algorithm, the design of VeriTable algorithm, its data structures

![Fig. 1. Active BGP entries in AS65000 (FIB) [2].](image-url)
and the workflow. Section 4 shows the evaluation of VeriTable over TaCo and Normalization. Finally, Section 6 concludes this work.

2. Background

2.1. The generic router architecture

Routers play a vital role in computer networking. First, they calculate, select and distribute the paths towards different destinations in the global network. Second, they direct the network traffic along selected paths hop by hop. A typical network router consists of two main components (see Fig. 2):

1. **Control plane.** Its duty is to run different routing protocols, such as the Border Gateway Protocol (BGP) [16], and to exchange routes towards other networks (i.e., their IP prefixes) with neighbor routers. In addition, for each destination prefix, the control plane runs BGP decision process, to pick the best routes among all collected. The destination IP prefixes and selected routes are stored in the Routing Information Base (RIB). Finally, each destination prefix and the next hop from the selected route is pushed in the data plane. The control plane usually runs on a cheap Dynamic Random Access Memory (DRAM).

2. **Data plane.** Dedicated for packet forwarding. It maintains several copies of the Forwarding Information Base (FIB), the entries of which are derived from the routes, selected and pushed by the control plane. An FIB contains the IP prefixes of different length and the corresponding next hops, i.e., output ports. To guarantee fast next hop lookup for each incoming packet, FIB memory resides on highly expensive [3,17,18] line cards with Ternary Content-Addressable Memory (TCAM) chips. As the number of entries in the global FIB constantly grows, network operators try to compress the forwarding tables using different aggregation techniques, in order to prolong the lifetime of the legacy TCAM chips [6].

2.2. Longest prefix match rule

When a packet arrives at a router’s data plane, its next hop is determined according to the Longest Prefix Match of packet’s IP destination address in the FIB. An example of the Longest Prefix Match selection is shown in Table 1. The table represents a sample FIB for IPv4 addresses with 32-bit address space. Several cases may happen during the matching process:

1. An IP destination address does not have a match in the FIB. In such case, the packet will be dropped by the router. For example, an IP destination address that does not start with the prefix 128.153.0.0/16, for example, 45.56.76.120, will be discarded.

2. An IP destination address has a single match in the table. In such case, the packet will be simply forwarded to the corresponding next hop. An example of such an IP address is 128.153.0.11 with the match 128.153.0.0/16 and next hop A.

3. An IP destination address has several matches in the table. In such case, a match with the longest prefix length will be selected. An example of such an IP address is 128.153.124.35, that matches prefixes 128.153.0.0/16, 128.153.64.0/18, 128.153.96.0/19. However, only the prefix 128.153.96.0/19 will be selected by the data plane engine, since it is the Longest Prefix Match. Thus, the packet will be forwarded to the next hop E.

2.3. The equivalence of forwarding tables

The forwarding tables are equivalent if and only if when applied to a router, a router’s forwarding engine behaves similarly for a packet with any possible IP destination address (see Section 3.1 for a more formal definition).

In other words, a packet will be forwarded to the same next hop, regardless of what equivalent forwarding table was used for the FIB. We use Table 2 with the binary representation of prefixes as an example. In this example, the FIB tables 1 and 2 are equivalent, even though their default routes (with the prefix “...”) are different. For example, a packet with destination IP address starting with 000 will be forwarded to the next hop B in both tables (LPM is “...” for the first table and 000 for the second table). On the contrary, if the FIB Table 1 changes its default next hop to A, the forwarding tables will be no longer equivalent. For example, an IP address starting with 011 will be forwarded to the next hop A by the first FIB (LPM: “...”), and to the next hop B by the second FIB (LPM: 01).

The naive way for verifying the forwarding equivalence of two or more forwarding tables is to match each possible IP address against those tables. However, such approach is not feasible, since the IP address space for IPv4 and IPv6 protocols contains $2^{32}$ and $2^{128}$ addresses, respectively. We present the current state-of-the-art approaches, TaCo and Normalization in the following section.

---

1. Default route matches the IP addresses that don’t have any match among other table entries. The default route can be used if only the table entries do not fully cover the IP address space.
2.4. State of the art

2.4.1. Taco: Semantic equivalence of IP prefix tables

ToCo verification algorithm [19] bears the following features:

1. ToCo is designed for comparing two tables simultaneously.
2. It uses separate binary trees to store all entries for each forwarding/routing table.
3. ToCo leverages leaf pushing to obtain non-overlapping prefixes in both trees.
4. ToCo performs two types of comparisons: (a) direct comparisons of the next hops for prefixes, common between two tables, and (b) comparisons that involve LPM lookups of the IP addresses, extended from the remaining prefixes of each table.

More specifically, ToCo needs to use four steps to complete equivalence verification for the entire routing space. We illustrate each step using two FIBs, shown at Table 2a and b:

1. Building a binary tree for each table, as shown in Fig. 3. The binary tree is built as a traditional prefix tree, where a left branch represents the bit 0 and the right branch represents the bit 1. The root node of such a tree is a default node with the prefix length equal to zero. To add a prefix to the binary tree, a node should be generated according to the bits of the prefix, with the depth equal to the prefix length. Such nodes will contain the value of the next hop from the forwarding table. In the meantime, binary prefix tree requires generating auxiliary nodes at each level of the prefix tree.

2. Perform leaf pushing for each binary tree. Leaf pushing operation requires each node with a single child on one branch to generate a second child on another branch. The next hops of the generated nodes should be inherited from the closest ancestor with the next hop value. Fig. 4 shows the resultant binary trees. Note, that after leaf pushing, TaCo’s internal nodes do not need to carry next hop information, since all the possible Longest Prefix Matches are the leaf nodes.

3. Finding common prefixes and their next hops, then making comparisons. In Fig. 4, those are the prefixes 000, 001, 01, 100, 11.

4. Extending non-common prefixes, found in both binary trees, to IP addresses and making comparisons. Prefix extension is performed by adding enough zero bits at the end of a prefix, to obtain valid IP address (32-bit or 128-bit address for IPv4 and IPv6 protocols respectively). In the example, Table 1 needs to extend the prefix 1010 and 1011 to an IP address and perform an IP lookup traversal through the Table 2. At Table 2, the non-common prefix is 101. TaCo extends this prefix to an IP address and performs an IP lookup traversal through the Table 1. The IP lookups in the binary tree are performed in the following way: starting from the root node, the algorithm reads the IP address bit by bit. If 0 bit is encountered, the algorithm moves to the left branch; otherwise, it moves to the right branch. The IP lookup traversal ends at the leaf node and returns the next hop value of that node.

Finally, when all comparisons end up with the equivalence results, ToCo theoretically proves that two FIB tables have forwarding equivalence. To summarize, ToCo undergoes several inefficient operations:

1. Leaf pushing for binary trees, a costly and slow operation.
2. To find common prefixes for direct comparisons, ToCo must additionally perform tree traversals.
3. IP address extension and mutual lookups for non-common prefixes are CPU-expensive.
4. Finally, to compare n tables and find the entries that cause possible non-equivalence, it may require \((n - 1) \times n\) times of tree-to-tree comparisons. For example, for three tables A, B, C there are six comparisons: A vs B, A vs C, B vs C, B vs A, C vs B, C vs A. Thus, it may require 90 tree-to-tree comparisons to compare 10 tables mutually. On the contrary, VeriTable eliminates all these expensive steps and accomplishes the verification over an entire IP routing space through a single traversal over the Patricia trie.

2.4.2. Normalization

Révai et al. in [20] show that a unique form of a binary tree for a forwarding table with the specific forwarding behavior can be obtained through Normalization, a procedure that eliminates brother leaf nodes with identical labels (e.g., next hop values) from a leaf-pushed binary tree. Indeed, if a recursive substitution is applied to the binary trees in Fig. 4, binary trees (a) and (b) will be identical (see Fig. 5). Authors in [20] prove that the set of tables with the same forwarding behaviors have identical normalized binary trees. More specifically, Normalization verification approach has three steps involved:

1. Leaf pushing; (2) tree compression and (3) side-by-side verification. Leaf pushing operation was described in details in Section 2.4.1. Tree compression involves compressing two brother leaf nodes with identical values, into their parent node. The parent node’s next hop will be then equal to the
next hop value of the compressed brother nodes. This is a recursive process until no brother leaf nodes have the same next hops. The final step for verification is to verify the sameness of the binary tree. For that goal, the algorithm needs to perform full simultaneous traversal over both trees.

Although Normalization needs to perform the expensive recursive leaf compression operation, it has several significant advantages over TaCo. First, unique binary trees contain fewer nodes, therefore the traversal over those trees is quicker. Second, verification of the identity of binary trees requires no IP address extensions and IP lookups (for satisfying the forwarding equivalence, each prefix in the first normalized tree must have a common prefix in the second normalized tree). In Section 4, we present the results of comparison between VeriTable, TaCo and Normalization.

The following section presents the design of VeriTable in details.

3. Design

In this section, first, we represent VeriTable theorem and the property derived from the theorem. Next, we introduce the data structures used in this work and the workflow of VeriTable. In addition, we used a small example to demonstrate each step of VeriTable.

3.1. Veritable theorem and property

First, we need to formalize the definition of the terms Longest Prefix Match and Forwarding Equivalence.

**Definition 1** Longest Prefix Match. Suppose $p$ is a prefix with length $I_p$ in a forwarding table $T$. We denote $p$ as $p_1 p_2 p_{i_p}$, where $p = \{0, 1\}^{i_p}$ (i.e., $p_i$ is 0 or 1 for $i = 0, 1, 2, \ldots, I_p$). Also suppose there is a string $s = \{0, 1\}^{i_s}$, where $i_s$ is the length of $s$. Then, according to the Longest Prefix Matching rule, we define that $p$ is the Longest Prefix Match for $s$ in $T$, namely, $p = \text{LPM}_T(s)$, if and only if

1. $I_p \leq i_s$
2. $p$ is a prefix for $s$, i.e., $p_1 p_2 p_{i_p} = s_1 s_2 s_{i_s}$,
3. $\exists p'$ in $T$, where $p'$ is a prefix of $s$ and $I_{p'} > I_p$. 

In the following proofs, we denote the value of the next hop for a prefix $p$ in the table $T$ as $N_T(p)$.

**Definition 2** Forwarding Equivalence. The forwarding tables $T_1, T_2, \ldots, T_m$ are forwarding equivalent, if and only if, for every single IP address $\omega = \{0, 1\}^n$, $N_{T_1}(\text{LPM}_T(T_1)) = N_{T_2}(\text{LPM}_T(T_2)) = \ldots = N_{T_m}(\text{LPM}_T(T_m))$. Where $n$ is the length of an IP address.

According to the definition above, verifying the equivalence of forwarding tables requires $2^{32}$ or $2^{128}$ IP addresses for IPv4 and IPv6, respectively. The goal of VeriTable is to reduce the number of comparisons by using prefixes, but still verify the entire routing space. To do that, we leverage a joint forwarding table, that is build by merging all comparable tables into a single table. The following theorem proves an important property of such a table; we design VeriTable based on that property.

**Theorem 1.** Let $T$ be a joint forwarding table, built by merging individual forwarding tables $T_1, T_2, \ldots, T_m$. Assume that $p = \text{LPM}_T(T)$, then we can prove that $\forall \omega = \{0, 1\}^n$, $\text{LPM}_T(T_1) = \text{LPM}_T(T_2)$, where $n$ is the length of an IP address and $i = 1, 2, \ldots, m$.

**Proof.** Let a prefix $p$ be $p_{1} p_{2} p_{i_p}$ ($1 \leq i \leq n$), and $\omega$ be $\omega_1 \omega_2 \omega_n$. Suppose, $p = \text{LPM}_T(T)$, then $\omega = p_{1} p_{2} \ldots p_{i_p} \omega_n$. We prove the theorem using contradiction. Suppose, $\text{LPM}_T(T_1) \neq \text{LPM}_T(T_2)$, namely, $\text{LPM}_T(T_1) = \text{LPM}_T(T_2)$. Then, according to the Definition 2, there exists a different prefix $p'$ in the forwarding table $T_1$, such as its length $I_{p'} > I_p$ and $p'$ is a prefix for $\omega$. But then, $p'$ exists in $T$. Thus, $p$ can not be a Longest Prefix Match for $\omega$ in $T$ (see Definition 2), which is contradictory to the initial assumption of this theorem, that $p = \text{LPM}_T(T)$. $\square$

Based on the Theorem 1, we derive VeriTable property: comparing next hops for each $\omega$ in $T_1, T_2, \ldots, T_m$ is equivalent to comparing next hops for each $p$ in $T_1, T_2, \ldots, T_m$. More formally, we rephrase the definition of Forwarding Equivalence:

**Definition 3.** The forwarding tables $T_1, T_2, \ldots, T_m$ are Forwarding Equivalent, if and only if, $\forall \omega = \{0, 1\}^n, \forall p = \text{LPM}_T(T)$, where $T$ is the union of $T_1, T_2, \ldots, T_m$. $N_{T_1}(\text{LPM}_T(T_1)) = N_{T_2}(\text{LPM}_T(T_2)) = \ldots = \text{LPM}_T(T_m)$.

In other words, since all IP addresses $\omega$ are covered by all Longest Prefix Matches $p$ in the joint forwarding table, VeriTable merely needs to go through all $p$s, match it against each comparable forwarding table and verify the equivalence of the next hops. The outcome of this property can be illustrated through an example with two forwarding tables with prefixes for 3-bit address space (see Table 3a and b). According to the original definition of Forwarding Equivalence, to verify it one needs to match any possible IP address against each of the tables (8 IP addresses in total). According to VeriTable property, it is necessary to do the following steps: (1) Join two tables into a single merged table; (2) Find all Longest Prefix Matches $p$ in that table; (3) For each $p$, find next hops in each table. The resulting joint table is shown on Table 3c. Note, that since the address spaces 0 and 1 are fully covered in the join table $T$ by other prefixes, the default prefix “.” is not a Longest Prefix Match (LPM) and is skipped during the verification process.

The following section shows the implementation and workflow of each of VeriTable steps in details.
3.2. VeriTable implementation and workflow

According to VeriTable property, the implementation of VeriTable should complete the following tasks: (1) Build a data structure of the joint forwarding table; (2) Identify all Longest Prefix Matches \( p \) of the joint table using that data structure; (3) For each \( p \), find the values of next hops in the comparable tables. To satisfy the forwarding equivalence requirement, the next hop values from each table shall be equal.

3.2.1. Setup the joint Patricia trie

Patricia Trie

Instead of using a binary tree to store the joint forwarding table, VeriTable uses the Joint Patricia trie data structure, derived from the PATRICIA (Practical Algorithm to Retrieve Information Coded in Alphanumeric) tree [15], a data structure based on a radix tree using a radix of two. PATRICIA Tree (PT) is a compressed binary tree and can be quickly built and perform fast IP address prefix matching. For instance, Fig. 6 demonstrates the corresponding PTs for FIB Table 2a and FIB Table 2b. The most distinguished part of 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 binary tree, 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 differences for small tables are not significant, however, they are significant for large forwarding tables with hundreds of thousands of entries. An exemplary 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. The detailed comparison in terms of running time, node accesses and memory consumption is presented in Section 4.

The above-mentioned features enable the PT to use less space and perform faster lookups. However, it 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 Fig. 6a? In fact, PT has to use an additional glue node to accomplish this task.

Building the joint Patricia Trie The first step of VeriTable algorithm is to build the joint Patricia trie (PT), the main data structure used in this work. Rather than building multiple binary trees or PTs for each individual table and comparing them in a one-to-one peering manner, as TaCo and Normalization do, VeriTable builds an accumulated joint PT using all tables one upon another. In the beginning, VeriTable takes the first table as an input and initiates all necessary fields to construct a PT accordingly. Afterward, while reading other tables, the nodes with the same prefixes will be merged. In case of new prefixes, VeriTable will add corresponding nodes to the joint PT.

For the next hops, VeriTable uses an integer array, located at each node of the joint PT. The size of the array is the same as the number of tables for comparison. The array contains next hops from each individual forwarding table. More specifically, these next hops 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. 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 at the node with prefix 001. If there is no next hop for a prefix in a particular table, the value in the array will be initialized as "." by default, or called an "empty" next hop (in the implementation, VeriTable uses "-1"). The nodes derived from at least one of the forwarding tables are REAL and contain at least one non-empty next hop in the array. The rest of the nodes are called GLUE nodes. Algorithm 1

### Algorithm 1 Building a joint PT \( T \).

1: procedure **BuildJointPT**\( (T_1, T_2, \ldots, T_n) \)
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 \( T_i \in T_1, T_2, \ldots, T_n \) do
6:   for each entry \( e \in T_i \) do
7:     Find a node \( n \) in \( T \) such as \( n.\text{prefix} \) is a longest match for \( e.\text{prefix} \) in \( T \).
8:       if \( n.\text{prefix} = e.\text{prefix} \) then
9:         \( n.\text{nexthop} ← e.\text{nexthop} \)
10:        \( n.\text{type} ← \text{REAL} \)
11:      else
12:        Generate new node \( n' \)
13:        \( n'.\text{prefix} ← e.\text{prefix} \)
14:        \( n'.\text{nexthop} ← e.\text{nexthop} \)
15:        \( n'.\text{type} ← \text{REAL} \)
16:        Assume \( n \) has a child \( n_c \)
17:        if the overlapping portion of \( n_c \) and \( n' \) is longer than \( n.\text{length} \) but shorter than \( n'.\text{length} \) bits then
18:           Generate a glue node \( g \)
19:           \( n'.\text{parent} ← g \)
20:           \( n_c.\text{parent} ← g \)
21:           \( g.\text{type} ← \text{GLUE} \)
22:           \( \text{Set } g \text{ as a child of } n \)
23:           \( \text{Set } n' \text{ and } n_c \text{ as children of } g \)
24:      else
25:         \( n'.\text{parent} ← n \)
26:         \( n_c.\text{parent} ← n' \)
27:         \( \text{Set } n_c \text{ as a child of } n' \)
28:         \( \text{Set } n' \text{ as a child of } n \)
29:       end if
30:   end for
31: end if
32: end for
33: end procedure

\[ \]

in Appendix A elaborates the detailed workflow to build a joint PT for multiple tables. Table 4 describes a joint PT’s node’s attributes. Fig. 7a shows the resultant joint PT for FIB Table 2a and b.
Table 4
Joint Patricia trie node’s attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>Node Pointer</td>
<td>Points to a node’s parent node</td>
</tr>
<tr>
<td>l</td>
<td>Node Pointer</td>
<td>Points to a node’s left child node if exists</td>
</tr>
<tr>
<td>r</td>
<td>Node Pointer</td>
<td>Points to a node’s right child node if exists</td>
</tr>
<tr>
<td>prefix</td>
<td>String</td>
<td>Binary string</td>
</tr>
<tr>
<td>length</td>
<td>Integer</td>
<td>The length of the prefix, 0–32 for IPv4 or 0–128 for IPv6</td>
</tr>
<tr>
<td>nexthop</td>
<td>Integer Array</td>
<td>Next hops of this prefix in T1...Tn, size n</td>
</tr>
<tr>
<td>type</td>
<td>Integer</td>
<td>Indicates if a node is a GLUE or REAL</td>
</tr>
</tbody>
</table>

![Diagram of VeriTable algorithm](image)

(a) Initial Joint PT  
(b) Joint PT after the 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 the True flags over to its parent recursively until finding a REAL node.

Fig. 7. VeriTable algorithm. 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 the True flags over to its parent recursively until finding a REAL node.

There are several advantages for the design of a joint PT:

1. Mostly, common prefixes among comparable tables will share the same node and prefix, which can considerably reduce memory consumption and computational time for new node creations.
2. Common prefixes and uncommon prefixes will be automatically gathered and identified in one single PT after the first step of building the joint PT.
3. Such design will greatly speed up subsequent comparisons of next hops between multiple tables without traversing multiple individual trees.

In the rest of this section, we describe the second, verification step of VeriTable, during which (1) VeriTable identifies all longest prefix matches (LPMs) in the joint PT; (2) VeriTable finds next hops for each of those LPMs in the comparable tables; (3) Based on the comparison of those next hops, VeriTable either verifies the equivalence of the comparable forwarding tables or shows the discrepancies between those tables.

3.2.2. Verification steps

The verification step consists of a single post-order traversal over the joint PT. We divide this traversal into two phases: (1) Top-down phase, during which VeriTable inherits next hops from the REAL nodes towards the first REAL descendants; and (2) bottom-up phase, during which VeriTable verifies the equivalence of the next hops for each Longest Prefix Match (LPM) in the joint PT. Note, that both phases are rotating and replacing each other during the post-order traversal.

Top-down phase VeriTable follows the intuitive property of the LPM rule, according to which 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 Fig. 7a, 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 the next hop from its closest REAL ancestor if the prefix contains an "empty" next hop. More specifically, when moving down, VeriTables is searching for "empty" next hops in the Next Hop array in each node. If "empty" next hops are found, VeriTable initializes them with the next hop of their closest REAL parent. In the meantime, "non-empty" next hops are always preserved in the Next Hop array. Note, that all GLUE nodes (hollow nodes in Fig. 7a) are skipped during this process because they are merely ancillary nodes helping to build up the tree structure and do not carry any next hop information.

After top-down phase, every REAL node will have a Next Hop array without any "empty" next hops. Fig. 7b shows the results of the top-down phase. If there is not a default route 0/0 in the original forwarding tables, for calculation convenience, VeriTable creates one, with the next hop value 0 and node type REAL.
Bottom-up phase As it was already mentioned above, this process is interwoven with the top-down phase in the recursive post-order verification step. While VeriTable 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 may be encountered, where the Next Hops array will be checked linearly, i.e., element by element. If there are any discrepancies, then VeriTable can immediately conclude that the forwarding tables are non-equivalent. If all next hops share the same value, VeriTable moves upward to its directly connected parent node.

To identify the Longest Prefix Matches in the joint PT at the internal nodes, VeriTable needs to check the prefix length difference at each internal node. 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, i.e., \( d = 1 \) 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 \( d > 1 \) for at least one child node, indicates that the parent node covers more routing space than that of all of its children nodes. If \( d > 1 \) happens at any time, VeriTable sets a LEAK flag variable at the parent node, to indicate that all of its children nodes are not able to cover the same routing space as the parent. In case if this parent is a REAL node, VeriTable identifies it as a Longest Prefix Match and verifies it, according to the VeriTable property, described in Section 3.1. If the parent with at least one LEAK flag is a GLUE node, the flag will be carried over up to the nearest REAL node, which can be an intermediate parent node of the GLUE node or a further ancestor. In this case, the verification of the “Next Hop” array will be executed at that REAL node, after which the flag will be cleared.

Intuitively, VeriTable checks the forwarding equivalence over the routing space covered by leaf nodes first, then over the remaining “leaking” routing space covered by internal REAL nodes. Fig. 7c demonstrates the bottom-up LEAK flag setting and carried-over process. For example, \( d = 2 \) between the 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 (AA), where the leaking routing space will be checked. Next, the LEAK flag will be cleared to False to avoid future duplicate checks of the same routing space.

In Algorithm 2 (see in Appendix A), we show the pseudocode of the verification step of VeriTable. Note, that VeriTable can exactly identify and print out the prefixes that cause non-equivalence of the comparable forwarding table by simply referring to the Longest Prefix Matches at the joint PT, on which the discrepancy was detected.

3.2.3. Complexity

As we show above in this work, at every stage of VeriTable algorithm we use the Patricia Tree data structure, which significantly reduces the number of memory accesses compared to the binary tree. In the worst case, when one of the comparable FIB tables consists of all possible IP addresses from the range (i.e., \( 2^{32} \) or \( 2^{128} \) prefixes for IPv4 and IPv6 respectively), the Patricia tree will be equivalent to the full binary tree. Thus, the running time of VeriTable is equal to \( O(k m n) \), where:

- \( k \) is the maximum prefix length in the comparable FIBs;
- \( m \) is the number of those FIBs;
- \( n \) is the number of nodes in a joint table.

In reality, prefixes in a FIB rarely exceed the length of 24 bits. As we show in Section 4, compared to TaCo, VeriTable reduces memory accesses by at least 35 times, consuming much smaller memory space.

3.3. Applications-network problem diagnosis

The VeriTable algorithm can be extended and applied to solve very serious network problems. One application is to detect network-wide loop issues mainly caused by incorrect network configuration. Another application is to discover if there are routing blackholes, which happen when there is not any matching route in the routing table and correspondingly the network traffic will be dropped. The cause of blackholes may be due to sudden link failures or misconfiguration. We designed two algorithms to detect loops and black holes in a realistic routing topology based on the VeriTable algorithm. We verified the results detailed in Section 4.
3.3.1. Loop and blackhole detection with static FIBs

The first algorithm is shown in Algorithm 3. It uses Depth First Search (DFS) to trace every forward chain. A forward chain is to simulate the forwarding steps of a packet in a hop-by-hop manner. This algorithm allocates a temporary array visit for the current nexthop array. visit is used to record the visiting status, “-1” means already scanned, “0” means on the chain that is being scanned and “1” means not scanned. visit is initialized with all “1”s. The algorithm scans nexthop from left to right, checking nexthop and visit; If nexthop is equal to or bigger than the total FIB number, it means the package is forwarded outside the network and there is no loop or blackhole for this chain. If nexthop is “-1” and it is not the starting point, it means there is a blackhole detected and the package forwarded to this position will be dropped. Otherwise, the visit value is checked: If visit is “-1”, there is no need to check again. If visit is “1”, it is changed to “0”. The iterator then moves to index nexthop, and repeats the previous check step. If visit is 0, it means the position is in the current chain and visited again. In another word, this position is in a loop. If this is the case, the loop is reported and finally, all the visit, values are changed to “-1”.

The second algorithm takes advantage of the property that every vertex in a single loop has the Indegree of 1. Blackhole detection is also included during the trace process. Detail is shown in Algorithm 4. This algorithm allocates a temporary array Indegree for the current nexthop array. indegree is used to count the in-degrees for every position. After all the elements in the array are initialized to 0, the algorithm scans nexthop from left to right: The count of the Indegree value at nexthop, will be added by 1 unless newthop is “-1”. After this step, the algorithm scans indegree and keeps looking for indegree, with “0” value, it changes indegree to “-1”, which means visited, and subtract 1 from it. The iterator continues to index nexthop, repeats the previous step and adds the blackhole check: if nexthop is “-1”, there is a blackhole at i. The iterator repeats the previous step as long as the indegree is 0. Finally, the “1” values in indegree reveal the vertices into loops.

3.3.2. Loop and blackhole detection upon updates

The algorithm of the detections after adding an entry is shown in Algorithm 5. It has a precondition that there is no original record at that position. If the position i is valid, nexthop is set to the given next hop. At the same time, status, is set to “1”, which means there is an original record now. Then, the algorithm calls the detection function to check if there are new loops or blackholes formed. Afterwards, it keeps searching the left and right nodes of the current one, changes the nexthop values, until a node with status = 1 is found.

The detection algorithm after modifying an entry is shown in Algorithm 6. It has a precondition that there was an original record at that position. If the position i is valid, nexthop is changed to the given next hop. Then, the algorithm calls the detection function to check if there are new loops or blackholes formed. Afterwards, it keeps searching the left and right nodes of the current
Algorithm 6 Loop and Blackhole Detection after Modifying an Entry.

1: procedure Mod(ancestor, i, prefix, nxthop)
2: target = search(ancestor, prefix)
3: data = target.data
4: data.nxthop = nxthop
5: Detect Loops and Blackholes (target)
6: Change target’s left and right child nodes to data.nxthop = nxthop
7: Detect Loops and Blackholes at the same time until a node
   with node.data.statusi = 1 is found
8: end procedure

one, changes the nxthop values, until a node with statusi = 1 is found.
The algorithm of the detections after deleting an entry is shown
in Algorithm 7. It has a precondition that there was an original
record at that position. If the position i is valid, nxthop, is changed to
“-1” and statusi is changed to “0”. Then the current status array
is checked. If all the values in the array are “0”, the PT runs the
erase operation and the nearest REAL ancestor node n is returned.
Afterwards, the algorithm keeps searching the left and right nodes
of the current one, changes the nxthop values, until a node with
statusi = 1 is found.

3.3.3. Complexity
As indicated by Algorithms 3 and 4, when we need to detect
loops or blackholes, each of the algorithms needs to go through a
top-down process from the joint Patricia trie root to the leaves one
by one, and each internal node maintains an array with a constant
number of next hops. If we use n to represent the number of the
trie nodes and a to indicate the number of next hops in the array,
then the complexity of the loop and blackhole detection is O(an),
where a is a constant once the network topology is known. There-
fore, the overall complexity is O(n).

4. Results

All experiments in this work were run on a machine with In-
tel Xeon Processor E5-2603 v3 1.60GHz and 64GB memory. This
section presents three different sets of experiments: first, the com-
parsion of VeriTable against TaCo and Normalization in a two-tables
scenario. Next, it shows how scalable is VeriTable for the 10-tables
scenario. Finally, it shows the performance of the “relaxed” version
of VeriTable for “black-holes” detection in a network.

4.1. VeriTable vs TaCo vs Normalization
Datasets were provided by the RouteViews project of the Uni-
versity of Oregon (Eugene, Oregon USA) [21]. For the evaluation,
12 IPv4 Routing Information Bases (RIBs) and 12 IPv6 RIBs from
the first day of each month in 2016 were collected. For the sim-
ulation purpose, AS numbers were used as next hops. By the end
of 2016, there were about 633K IPv4 routes and 35K IPv6 routes in
the global forwarding tables. To obtain different but equivalent
forwarding tables, an optimal FIB aggregation algorithm was ap-
piled to RIB tables. Fig. 8 shows the aggregation results for both IPv4
and IPv6 tables. IPv4 achieves a better compression ratio (about
25% of the original size) than IPv6 (about 60% percent of the origi-
nal size) because IPv4 has the larger number of prefixes. The origi-
nal and compressed tables were used to evaluate the performance
of VeriTable vs the state-of-the-art TaCo and Normalization (see de-
scription of these two algorithms in Section 2.4) verification al-
rithms in a two-table scenario. The following metrics were used
for the evaluations: data structure building time, verification time,
the number of node accesses and memory consumption.

4.1.1. Data structure building time
TaCo and Normalization need to build two separate binary trees,
while VeriTable only needs to build a single joint Patricia Trie (PT).
Fig. 9 shows the building time for both IPv4 and IPv6. VeriTa-
ble outperforms TaCo and Normalization in both cases. In Fig. 9a
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 Veri-
Table 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 Fig. 9b,
TaCo uses minimum 137.94ms and maximum 186.73ms with an
average 168.10ms to build two BTs; for Normalization these num-
bers are 162.40ms, 225.75ms and 197.25ms. VeriTable uses mini-
um 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 ta-
tables. Although IPv6 has much larger address space than IPv4, Ver-
Table achieves much less building time under IPv4 than that of
IPv4, which is attributed to the small FIB size and the usage of a
compact data structure – a joint PT. Note the slower Normalization
building time due to the operation of tree compression performed
by that algorithm.

4.1.2. Verification time
A valid verification algorithm needs to cover the whole routing
space (232 IP addresses for IPv4 and 2128 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 effi-
ciently or not. Fig. 10 shows the running time of TaCo, Normal-
ization and VeriTable for both IPv4 and IPv6, respectively. VeriTable
significantly outperforms TaCo in both cases thanks to the VeriTable
property, described in Section 3.1. TaCo takes minimum 355.06ms
and maximum 390.60ms with an average 370.73ms to finish the

---

4 We used FIFA-S algorithm, described in [9]
Fig. 8. IPv4 and IPv6 FIB size before and after aggregation

Fig. 9. IPv4 and IPv6 data structure building time

Fig. 10. IPv4 and IPv6 verification time
whole verification process. VeriTable takes minimum 51.91 ms and maximum 57.48 ms with an average 54.63 ms 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 (1356 ms VS 696 ms). 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 binary trees 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% (696.90 ms VS 1154.08 ms) for IPv4 verification.

Fig. 10 b shows the IPv6 scenario (note the Y-axis is a log scale). TaCo takes minimum 75.17 ms and maximum 103.18 ms with an average 92.79 ms to finish the whole verification process. For Normalization it is 11.47 ms, 15.58 ms, 13.87 ms respectively. VeriTable takes minimum 1.44 ms and maximum 1.97 ms with an average 1.75 ms 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 (261 ms VS 47 ms) and 4.5 times faster than Normalization (211 ms VS 47 ms) 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 selection of Longest Prefix Matches for comparisons, without tree normalization (see Section 3) for details. Note, that the leaf pushing operation over IPv6 forwarding table causes a significant inflation of the binary trees. That explains much slower speed of TaCo and Normalization verification for IPv6 tables than for IPv4 tables.

4.1.3. Number of node accesses
Node accesses, similarly to memory accesses, refer to how many nodes were visited during the verification process. The total number of node accesses is the primary factor to determine the verification time of an algorithm. Fig. 11 shows the number of total node accesses for both IPv4 and IPv6 scenarios. Due to the novel design of VeriTable, it is 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. VeriTable achieves 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, TaCo and Normalization: (a) VeriTable uses a joint PT, but TaCo and Normalization uses separate binary trees; and (b) VeriTable conducts only single post-order PT traversal. TaCo conducts several repeated node accesses over a binary tree, including Longest Prefix Match lookups. Due to the unique form of a normalized binary tree, Normalization requires no mutual IP address lookups and thus conducts significantly fewer node accesses than TaCo.

4.1.4. Memory consumptions
Memory consumption is another important metric to evaluate the performance of algorithms. Fig. 12 shows the comparisons between TaCo, Normalization and VeriTable for both IPv4 and IPv6 prefixes, in terms of their total memory consumptions. In both scenarios, VeriTable outperforms TaCo and 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, consuming only 9.3% (4.9 MB) of total memory space than that of TaCo (53 MB) and of Normalization on average. 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.

Overall, thanks to the design of VeriTable, it outperforms TaCo and Normalization in all aspects, including total running time, number of node accesses and memory consumption.

4.1.5. Scalability of VeriTable
This experiment shows the performance of VeriTable when checking the forwarding equivalence and differences over multiple forwarding tables simultaneously. For the experimental purpose, 2000 distinct errors were intentionally added to each comparable forwarding table. It was verified that the same number of errors were detected by VeriTable algorithm. The evaluation results are showed in Table 5. There are two primary observations. First, VeriTable is able to check the whole address space quickly over 10 large forwarding tables (336.41 ms) with relatively small memory consumptions (165 MB). Second, the building time, verification time, node accesses, and memory consumptions grow significantly slower than the total number of forwarding entries. This indicates
that VeriTable is scalable for equivalence checking of a large number of tables. On the contrary, TaCo and Normalization naturally were not designed to compare multiple forwarding tables. In theory, TaCo may need \( n \times (n - 1) \) 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.

### 4.1.6. Routing space comparisons

A relaxed version with minor changes of VeriTable algorithm is able to quickly identify 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 node, it checks if there is a case when the corresponding Next Hop 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 certain routing space while another FIB covers it, which may potentially lead to routing “black-holes”. In our experiments, we used data from RouteViews [21] project, where 10 forwarding tables that contain the largest number of entries were collected and then merged into a super forwarding table with 691,998 entries. Subsequently, we compared the routing spaces of the 10 individual forwarding tables with the super forwarding table. The results of these comparisons (see Table 6 in detail) show that none of these 10 forwarding tables fully cover the routing space of the merged one. The leaking routes in Table 6 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 “black-holes” may take place between routers in the same domain or between different domains. To this end, VeriTable verification algorithm can find out routing space differences quickly.

### 4.2. Network-wide loop and blackhole detection

In order to test the efficiency and scalability of the VeriTable’s loop and blackhole detection algorithms. In the evaluation, we conducted 6 runs of the experiments. The number of the entries in each forwarding table in the network ranges from 100K, 200K, all the way to 600K. IP prefixes in each table are chosen from the routing tables on RouteViews [21] project. The network topology is shown in Fig. 13. We select the next hops for each prefix of each FIB in the following way: We can see from Fig. 13 that each node has some directly connected nodes, for example, the node numbered 8 is connected to nodes numbered 3, 5, 7 and 9. As a result, for each prefix entry in forwarding table of node 8, therefore, the next hops is randomly selected from values of \( \{3,5,7,9\} \).
4.2.1. Time and memory consumption

Fig. 14 shows the results of time used to detect loops and blackholes in the given topology versus the number of entries in each node. According to the figure, in general, the average time spent for loop and blackhole detection is 2838 ms for 350K entries, which is relatively small. It reveals the algorithm is very efficient as it has the time complexity better than linear. The advantage is outstanding especially when there is a huge amount of entries. The possible reason for this saving is, although the number of entries is increasing linearly, there might be more branch nodes hidden into the Patricia Trie and are fully covered by their child nodes. As a result, the number of nodes under detection does not increase that fast when compared with the situation with a small number of entries.

Fig. 15 shows the results of memory consumed to detect loops and blackholes. In general, the average memory spent for loop and blackhole detection is 362 MB for 350K entries, which is also relatively small. As a result, the algorithm used is very scalable, especially when there is a huge amount of entries. The possible reason for this saving is similar to that analyzed in the time consumption part: with the number of entries increases, the ratio of inner branch node increases. This results in a decrease of the ratio for the nodes that need to be checked.

4.2.2. Loop and blackhole size distribution

Fig. 16 shows the loop size distribution for the test results with 100K, 300K and 500K entries individually. It is concluded from the figure that, normally most of the loops have sizes no bigger than 10 and for a given size, the loop number increases with the entry number increases. the number of loops has a logarithmic decrease with the size increases. This reveals the detection algorithm works very well to detect a large scale of potential loops in a network.

Fig. 17 shows the blackhole size distribution for the test results with 100K, 300K and 500K entries individually. It is concluded from the figure that, normally most of the blackholes have sizes no bigger than 10. It is worth to note that, for a given blackhole size, the peak values of the blackhole numbers occur when there is a half number of prefixes chosen from the routing table. The possible reason for this result is, the chosen routing table has no blackhole when all the prefixes are chosen in the experiment. Before the routing space is half-filled, with the increase of the number of entries, there are more forwarding chains and the possibility for a forwarding chain to end up with a "-1" increases. After half of the
entries are filled into the FIB array, with the number of entries increasing, there are fewer new forward chains and more and more blackholes will get values instead of ending up with “-1”. In addition, the number of blackholes has a logarithmic decrease with the size increases. This reveals the detection algorithm can scale very well to a large number of FIB entries and block holes.

5. Related work

The natural application for verifying the forwarding equivalence or routing tables is to verify the correctness of FIB aggregation of FIB caching algorithms. To this end, multiple solutions for compressing an FIB were proposed. FIB aggregation algorithm implementations [9,22–24] can incur misbehavior of the router after the entry merging process or when applying BGP updates to the compressed forwarding table. FIB caching approach implies storing the popular routes in a fast and expensive memory, while rest of the routes are stored on a slower memory [17,25–27]. Although such approach can be extremely efficient in compressing the routing table (2% of the cache can reach 99.5% cache hit ratio [28]), it poses a risk of a problem called “cache-hiding”, when instead of a route with the longer prefix, which stays in the slower memory, the shorter prefix in the cache is selected for obtaining the next hop. Thus, “cache-hiding” may cause a forwarding behavior of a router different from its original forwarding behavior.

The above-mentioned risks of forwarding misconfiguration indicate the importance of developing efficient tools to verify the forwarding equivalence between original and compressed forwarding tables. TaCo algorithm, proposed by Tariq et al. [19], 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 2.4 elaborates the algorithm in detail. VeriTable is different from TaCo, since it builds a single joint Patricia tree for multiple tables and leverages novel ways to avoid duplicate tree traversals and minimize node accesses. Thus, as shown in Section 4, VeriTable outperforms TaCo in all aspects.

Inconsistency of forwarding tables within one network may lead to different types of problems, such as “black-holes”, 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. In [29] authors present a patent for a system that validates the equivalence between an RIB and an FIB in a network. Mai et al. introduced Anteater in [30], 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. Anteater was evaluated with 178 FIBs with the mean size of 1627 entries. Zeng et al. introduced Libra in [31]. Libra used MapReduce [32] 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 [33], proposed by Khurshid et al., leverages software-defined networking to collect forwarding rules and then slice the network into equivalence classes (ECs). Each EC consists of prefixes that exhibit the same forwarding behavior on the network. Upon routing update, VeriFlow finds all ECs affected by the update and generates a forwarding graph for each of them. Thus, VeriFlow confines the verification by the entries of affected ECs. Kazemian et al. introduced NetPlumber in [34], a real-time network analyzer based on Header Space Analysis protocol-agnostic framework, described in [35]. NetPlumber is compatible with both Software-Defined and conventional networks. It incrementally verifies the network configuration upon every policy change in a quick manner. However, NetPlumber is not designed for networks with a high rate of routing updates (e.g. networks, operating in Default-Free Zone), because of high processing time for update verification. The problem of FIB equivalence in a programmable data plane is studied in [36]. Sanger et al. in [37] use a Multi-Terminal Binary Decision Diagram in order to verify consistency of the data plane in a Software-Defined Network.

Different from the network-wide verification methods above, VeriTable aims to investigate whether multiple static forwarding tables achieve the same forwarding behaviors, given an IP packet with an arbitrary IP destination address, or whether those tables cover the same routing space. VeriTable will be able to quickly identify if the forwarding tables return the same next hop after the Longest Prefix Matching lookups, and which prefixes result in discrepancy if the answer is no.

6. Conclusion

This paper presents the design and the implementation of VeriTable, which can quickly determine if multiple routing or forwarding tables achieve the same or different forwarding behaviors. The evaluation results of VeriTable show that VeriTable significantly outperforms its counterparts. The novel algorithm and compact data structures can offer the benefit not only in quick and efficient verification of the correctness of an FIB compression, but also in several other scenarios, when the Longest Prefix Matching rule is used for performing IP lookups. For example, VeriTable is able to check if routing updates in the control plane are consistent with the updates in the data plane. Moreover, the principles used in this work can be applied to network-wide abnormality diagnosis of network
problems. To this end, we extended VeriTable to conduct a scalable and efficient forwarding loop detection and avoidance in the data plane of a network. The newly extended algorithms can handle incremental updates, applied to the forwarding tables in a network. Our evaluation results show that the designed algorithms are efficient enough to handle large scale of forwarding tables for identifying their loop and blacklist problems.

**Declaration of Competing Interest**

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

**Acknowledgement**

The research work has been partially supported by Comcast Innovation Fund. We thank the anonymous reviewers for their insightful comments and feedback.

**Appendix A. Algorithms of VeriTable**

Algorithm 1 shows the pseudo code of the first step of VeriTable, when the joint Patricia Tree is built. At this step, VeriTable reads the entries from each table, starting from the first table, and updates or generates the updates corresponding to the prefixes from those tables. Note, that GLUE nodes need to be generated in case if two REAL nodes have the same parent and located at the same branch of that parent.

Algorithm 2 shows the pseudo code of the second step of VeriTable, when it performs single traversal over the joint Patricia tree, in order to:

1. Initialize empty next hops at the nodes of the joint Patricia tree.
2. Identify Longest Prefix Matches (LPM) in the joint Patricia tree.
3. Compare the next hops in the Next Hop array at those LPM nodes, in order to verify, if they have the same value.

For the description of each node field in the algorithms see Table 4.

**References**


Dr. Yaoqing Liu received his Master’s and PhD degrees from the University of Memphis in Computer Science. He is the program coordinator of M.S. in Cybersecurity and Information Assurance program (MSCIA), and an Assistant Professor in Gildart Haase School of Computer Sciences & Engineering at Fairleigh Dickinson University (FDU). His research interests are networked systems (security, routing, algorithm and measurement), currently focusing on Named Data Networks (NDN) and Blockchain applications. His publications appear in highly reputed conference proceedings and journals, such as IEEE INFOCOM, ACM SIGCOMM CCR and ACM/IEEE ANCS. He has been a TPC member and technical paper reviewer for IEEE conferences, journal magazines and transactions. He is the inventor of four patents. His research work has been selected as one of the 2018 TechConnect Defense Innovation Awards recognizing the potential positive impact for the warfighter and national security.
Garegin Grigoryan was born in 1991 in Moscow, Russia. He received a Diploma degree in “Automated Data Processing and Management Systems” from the Department of Cybernetics of Moscow Engineering Physics Institute, in 2012. The same year, he joined Diasoft, a Moscow-based company that provides a wide range of financial software solutions. In 2016, he entered the graduate program of Clarkson University (Potsdam, NY, USA). After receiving his MS degree in Computer Science, he joined Rochester Institute of Technology as a Ph.D. candidate. His current research interests include routing scalability, aggregation and caching of forwarding tables, Software-Defined Networking, software switching, programmable data planes, Blockchain technology, and Information-centric networking.

Dr. Jun Li is a Professor in the Department of Computer and Information Science and founding director of the Center for Cyber Security and Privacy at the University of Oregon. He received his Ph.D. from UCLA in 2002 (with Outstanding Doctor of Philosophy honor), M.E. from Chinese Academy of Sciences in 1995 (with Presidential Scholarship), and B.S. from Peking University in 1992, all in computer science. His research is focused on networking, distributed systems, and network security, with about 90 peer-reviewed publications. He has served on US National Science Foundation research panels and more than 70 international technical program committees, including chairing six of them. He currently serves on the editorial board of IEEE Transactions on Dependable and Secure Computing and a few conference or workshop steering committees. He is a senior member of ACM and IEEE and an NSF CAREER awardee in 2007.

Guchuan Sun was born in the city of Zhenjiang, Jiangsu Province, China in 1968. He received a Master’s degree in Computer science from Clarkson University in 2019, under the supervision of Dr. Yaqing Liu. His major research interests include network optimization algorithms, network problem diagnosis and blockchain applications.

Tony Tauber is a distinguished engineer at Comcast since Jan 2014. His specialties include routing, instrumentation, measurement, scaling, and network planning to accommodate new services and achieve business objectives. He works on the design and engineering of Comcast’s national backbone network including initial implementation and ongoing evolution. He served as the chair of NANOG from 2011 to 2016, and led the technical committee which solicit, review, and develop material for three-yearly conference on the state of the art in the Internet Service Provider and computer networking arena. He has been an active member of several standards groups.