Scaling Data Plane Verification via Parallelization

Sisi Wen
ByteDance

Anubhavnidhi Abhashkumar*
ByteDance

Chenyang Zhao†
Beijing University of Posts and Telecommunications

Weirong Jiang
ByteDance

ABSTRACT

The data plane verification of networks in hyperscale environments is challenging due to the complexity and size of modern networks. In this paper, we introduce *Medusa*, a novel verifier that efficiently analyzes large data plane models using parallel processing on multi-core CPUs. First, we propose a new data structure called RANGESET, which overcomes the parallelism limitations of existing popular data structures such as Binary Decision Diagrams (BDD) used in data plane verifiers. Next, we leverage multi-core processing by dividing the network into distinct groups and assigning each group to a separate thread for computation. The results are then integrated for comprehensive verification. By optimizing the use of multi-core systems, we enhance computational efficiency and accelerate the verification process. Experimental results demonstrate that *Medusa* outperforms existing tools in terms of speed and memory. For instance, in a network with O(10K) devices and O(1M) forwarding rules, *Medusa* can detect loops in approximately 5 seconds, outperforming other Data Plane Verifiers (DPVs) where some cannot model and analyze the network. Moreover, in networks that we could compare with other state-of-the-art DPVs, *Medusa* provides a substantial improvement, with speedups up to 600X, 4000X, and 800X compared to alternatives like Flash, APKeep, and Tulkun, respectively.

CCS CONCEPTS

• **Networks** → **Network reliability**; • **Theory of computation** → **Logic and verification**.

KEYWORDS

Data Plane Verification, Parallelization, Hyperscale Network

ACM Reference Format:
Sisi Wen, Anubhavnidhi Abhashkumar, Chenyang Zhao, and Weirong Jiang. 2024. Scaling Data Plane Verification via Parallelization. In *The 8th Asia-Pacific Workshop on Networking (APNet 2024), August 3–4, 2024, Sydney, Australia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3663408.3663420

*Corresponding author
†Work done while interning at ByteDance

APNet 2024, August 3–4, 2024, Sydney, Australia

ACM ISBN 979-8-4007-1758-1/24/08
https://doi.org/10.1145/3663408.3663420

1 INTRODUCTION

In recent years, the network infrastructure of multiple global service providers [4, 6] has undergone a rapid expansion resulting in hyperscale and heterogeneous networks consisting of large-scale data centers. To model and analyze the data plane of such networks, Data Plane Verifiers (DPVs) have become prevalent [11, 14].

However, as the size of the networks increases, so do the size of the network models, which pose significant challenges for existing state-of-the-art verifiers. For example, existing verifiers often face limitations when dealing with large-scale production scenarios consisting of millions of routes and thousands of devices. In our experience, these verifiers failed during attempts to analyze our network and terminated with an "out of memory" exception due to the sheer magnitude of the network. This highlights the urgent need for verifiers that can handle the ever-increasing scale and complexity of modern networks.

Recent DPV tools such as Delta-net [13] and APKeep [20] can achieve sub-millisecond per-rule verification by leveraging the equivalence class (EC) idea to partition the packet space. However, these tools still perform poorly when analyzing hyperscale networks due to their performance and/or memory limitations. For example, although Delta-net achieves linear performance due to its innovative data structure called Atom, its unmerged ECs lead to greedy memory usage. Consider an IPv6 network consisting of 10K devices with an average of 20 ports per device and 100K prefixes, the memory usage can reach 300 GB. On the other hand, APKeep aims to minimize the number of ECs, but many repeated predicate merge and split operations degrade its performance. Also, despite APKeep's efforts to limit the number of ECs, the high number (e.g. 80K in our networks) that still exist in complex networks can overwhelm a single process.

To address this limitation, researchers have explored dividing the network model and harnessing the power of parallel processing. This approach is a promising direction as it can enable distributed DPV to handle large, complex networks more effectively. Flash [11] distributes the verification workload across different subspace verifiers. It also solves the problem of redundant computation in burst update scenarios for large networks, but its continuation of the EC-based idea from previous work and the use of BDD (Binary Decision Diagram) limits its parallelization scheme to uneven subspace partitioning. Tulkun [18] transforms verification tasks into a counting problem on a Directed Acyclic Graph (called DPVNet) and decomposes global verification into lightweight on-device tasks. However, the computational performance of DPVNet, the preconditioner of its DPV model, severely limits the deployment of this algorithm in real networks.

Finally, it is important to note that many recent DPVs [11, 18, 20] that achieve high performance are designed for incremental verification and are not suitable for modeling a network from scratch, i.e., for clean slate verification. In a real production environment, storing models of different networks as long-running processes may be impractical due to the significant amount of resources required. This could lead to a situation where it becomes necessary to give up the resources and remodel the network from scratch, similar to what clean slate verifiers [7, 17] do.

To address these limitations, we propose a new DPV called *Medusa*. *Medusa* leverages multi-core processing by dividing the network into distinct groups. It introduces a new data structure called RANGESET to model the IP address space of the network, which enables parallelization of creation and analysis/operations for different network submodels. In the first step, *Medusa* divides the network into groups and builds the Trie data structure concurrently to identify the overlap of IP address space within each group. Then, *Medusa* concurrently traverses the Trie to compute the RANGESETs for each submodel. With the obtained RANGESET formula, range-based operations are performed to determine the transmission of RANGESETs from one node to another within a RANGESET forwarding graph. Finally, the IP packet transmission is simulated by traversing the graph using a single thread.

In our experiments, we compared *Medusa* against Tulkun, Flash, and APKeep for public datasets and large-scale private datasets from production data centers. We show that *Medusa* can detect loops in a network containing O(10K) devices and O(1M) forwarding rules in just about 5 seconds, surpassing other DPVs where some cannot model and analyze the network. Additionally, in networks where we compared *Medusa* with other DPVs, it provides a remarkable improvement with up to 600X, 4000X, and 800X speedups compared to alternatives like Flash, APKeep, and Tulkun, respectively.

2 CHALLENGES

2.1 Dividing the network model

Analyzing a large DPV model in one shot may be impractical due to the size of the model. An alternative approach would be to divide the network model into submodels and analyze each of them in parallel. There are two possible methods to divide the network model [19]: dividing the IP address space and dividing the network into different groups based on devices.

Dividing the IP space has limitations. The rapid growth of equivalence classes (ECs) as the network grows in size cannot be curbed, leading to greater memory requirements. Additionally, prefixes belonging to the same EC may not be merged if they are divided into different groups. For instance, a tool such as Flash that uses this approach requires four to five times more memory than other DPVs such as APKeep (§5). It is also challenging to determine a uniform cutoff point to divide the routing tables properly, given the potential range of routing tables on a device from O(1K) to O(100K).

Dividing the network into subgroups based on devices presents several advantages. It not only reduces memory usage and the number of ECs generated, but it also enables prefixes to merge within smaller networks that they might not merge throughout the entire network. Since the network is already divided into different groups based on network features, it is easy to divide the network into subgroups, analyze them separately, and then merge the analyses in the final step.

However, merging analyses from different groups can bring its own challenges. For instance, correlating the ECs computed from and within each group can be complicated in the merge of analysis results. Additionally, policy alignment between different subgroups may be necessary to ensure consistent behavior across the entire network. Despite these challenges, dividing the network into subgroups can provide an efficient way to analyze a large DPV model.

2.2 Data structure for parallelism

Many verifiers, such as APKeep [20], Flash [11], and Batfish [10], use Binary Decision Diagrams (BDDs) to model the network due to their simplicity, memory efficiency, and ability to express complex forwarding rules concisely. However, BDDs were not originally tailored for parallelism due to maintaining a global hash table, which becomes a bottleneck when multiple threads use BDD.

Attempts to develop parallel BDDs have been made, including mutex-based structures [16], thread-pools and lock-free data structures [8], and on-demand thread creation [9]. However, these methods suffer from various issues, including mutex locking overhead, deadlocks, and thread scheduling overhead, respectively.

Existing general-purpose BDD packages aim to speed up a single BDD operation but not the overall throughput needed to handle a large number of BDD operations. Due to the large number of BDD operations involved in DPV, the aforementioned methods may not be suitable. Nanobdd [12] claims to be the first-ever high-performance thread-safe BDD library, but our experiments show that its performance is worse than popular BDD libraries like BuDDY [15], which do not support multiple threads. We randomly generate AND/OR computational tasks for predicates on BDD for Nanobdd and distribute them evenly across varying numbers of threads to execute. However, increasing the number of threads from 1 to 20 leads to a longer completion time (of 2 to 10 seconds), since the increased number of threads creates lock contention, thereby causing a degradation in performance. While it is possible to use multiple processes/threads to create multiple hash tables for parallelism, BDDs between different processes/threads cannot operate directly, and extra costs must be paid to achieve a certain mapping between BDD of different processes/threads.

3 RANGESET

We propose a new data structure called `RANGESET` to represent IP address space. The `RANGESET` (*RS*) object represents a set of ordered pairs of integers. Each pair consists of a lower-bound integer, denoted as LB_i, and an upper-bound integer, denoted as UB_i, for the i-th pair in the *RS*. The *RS* can be represented as a union of ordered pairs from $i = 1$ to n, where n is the total number of ordered pairs in the *RS*. The entries in the *RS* are sorted based on the lower-bound.

$$RANGESET = \bigcup_{i=1}^{n} (LB_i, UB_i) \tag{1}$$

We can represent any IP space using a *RS*. For example, the IP space 0.0.0.8/29 can be represented as the *RS* $\{(8, 15)\}$, which includes the IP addresses ranging from 0.0.0.8 to 0.0.0.15. Similarly,

the IP space $0.0.0.8/29 - 0.0.0.13/32$ can be represented using the *RS* $\{(8, 12), (14, 15)\}$. This *RS* includes all IP addresses in the $0.0.0.8/29$ subnet except for 0.0.0.13.

The condition for a valid *RS* is as follows:

$$LB_i \leq UB_i \qquad \text{and} \qquad UB_i + 1 < LB_{i+1} \tag{2}$$

For instance, $\{(1, 2), (4, 5)\}$ is a valid *RS*, while $\{(1, 2), (3, 4)\}$ is not. A *RS* is considered a false *RS* if there are no pairs in it. Conversely, a *RS* is considered a true *RS* if there is only one pair, and the lower bound is the minimum and the upper bound is the maximum. For example, $\{(0, 2^{32} - 1)\}$ is a true *RS* that represents the entire IPv4 address space.

Similar to other DPVs, both model creation and analysis incur *AND*, *OR*, and *NOT* operations. Note, only valid *RS* can partake in these operations, all of which must also return valid *RS*. Due to space constraint, we will provide a brief overview of these operations. The *AND* operation is computed by iterating through both input ranges, comparing the lower and upper bounds of each range pair. If there is an overlap, the function computes the intersection and adds it to the result. E.g. $\{(0, 5), (10, 15)\}$ AND $\{(3, 7), (12, 18)\}$ = $\{(3, 5), (12, 15)\}$.

The *OR* operation combines and sorts all the entries in both the *RS*. It then pops the first pair from the sorted range to initialize the current pair. The function iterates through the combined sorted range, comparing the upper and lower bounds of the current pair to the next pair in the sorted range to determine whether a union can be formed. If a union can be formed, the function updates the upper bound of the current pair and continues iterating through the sorted range. On the other hand, if a union cannot be formed, the function adds the current pair to the result and iterates forward. E.g. $\{(0, 5), (10, 15)\}$ OR $\{(3, 7), (12, 18)\}$ = $\{(0, 7), (10, 18)\}$. Both these operations can be computed in $O(n + m)$, where n and m correspond to the size of the *RS*.

The *NOT* operation computes the complement by iterating through each pair in the input range. For each gap between pairs in the input range, it pushes a new pair. The lower bound of each gap pair is equal to the upper bound of the previous pair plus one, and the upper bound of each gap pair is equal to the lower bound of the next pair minus one. E.g. NOT $\{(0, 5), (10, 15)\}$ = $\{(6, 9), (16, 2^{32} - 1)\}$. NOT operations can be computed in $O(n)$.

It is worth noting that both Delta-net's Atom and *RS* are data structures used to partition the IP space, but there is a significant difference between the two. Atom ensures global uniqueness of its data throughout the network model, which restricts its parallelization capability. On the other hand, *RS* uses equivalence classes to prevent IP space fragmentation without enforcing global uniqueness, making it a more scalable and parallelizable option for deployment.

4 DESIGN

A highly effective strategy to optimize the utilization of multi-core CPUs for DPV is to partition the network into groups assigned to specific threads. Each thread is responsible for computing the RANGESET forwarded by each device in the group to its neighbors, leveraging the parallel processing capabilities of modern multi-core systems. Once each thread has completed its computations, the results are integrated to facilitate the verification process, promote computational efficiency, and maximize parallelization. This approach harnesses the full power of parallel processing provided by these systems and enables efficient utilization of multiple CPUs, significantly speeding up the verification process.

4.1 Design Goal

There is an intuitive idea to compute the RANGESETs forwarded from a device to its neighbors. We first sort each device's routing rules by priority and iterate through them. For each rule, we subtract the current rule's *RS* from the *RS* of all other ports and merge the current rule's *RS* into the specified port's *RS*. This helps calculate the *RS* forwarded from each port of each device. However, this approach has several redundant computations. Firstly, there could be duplicated *RS* operations on different devices, such as $P0 - P1$ happening on both device B and C simultaneously in Figure 1. Secondly, there could be redundancies in determining overlaps between different *RS*. To reduce the redundant computations as mentioned earlier, we suggest a new and efficient approach using tries.

We explain the method combined with an end to end example in Figure 1. Note that the first three steps will be paralleleized and only the last step is single threaded. The network has four prefixes - $P0$, $P1$, $P2$, and $P3$ - where $P0$ is the default route, represented by $0.0.0.0/0$ or :: /0. These prefixes follow a hierarchical structure, with $P3 \subset P2 \subset P1 \subset P0$. Additionally, these prefixes have a priority ranking, $P3.priority > P2.priority > P1.priority > P0.priority$. The routing table for each device in the network is shown in Figure 1. To perform network verification efficiently, we can initially divide the network nodes into groups. For example, node A can be divided into group1, and nodes B and C can be divided into group2. Each group will have its own thread for computation.

4.2 Build Trie

We adopt a trie-based approach to identify all the overlap and priority in the network. This trie is constructed based on the prefixes in the network. Each trie node comprises two parts: the prefix and the forwarding behavior of all devices for that prefix. The forwarding behavior specifies how traffic is forwarded for the prefix from each device to its neighbors, which is obtained from the routing table. For instance, let's consider the trie node $\{P1, [(B : C), (C : A)]\}$ in group 2 which is made up of separate rules on devices B and C. This indicates that $P1$ is forwarded on device B to neighbor C, and on device C to neighbor A. The prefix space of a child node is always a subset of its parent node. We can create a distinct trie for each network group. Notably, every device has a default rule to drop all packets, which is represented by $P0$ in this example. In this way, we can avoid any redundant overlap judgments when traversing the trie using a Depth-First Search (DFS) method.

4.3 Compute RANGESET

To obtain all RANGESETs, we traverse the trie using DFS. We can illustrate this with an example that revolves around the traversal of the group2's trie.

To simplify the explanation, in Figure 1B, we use forwarding action tables for each node to demonstrate how the *RS* splits based on changes in forwarding as it passes through that trie node. Initially,

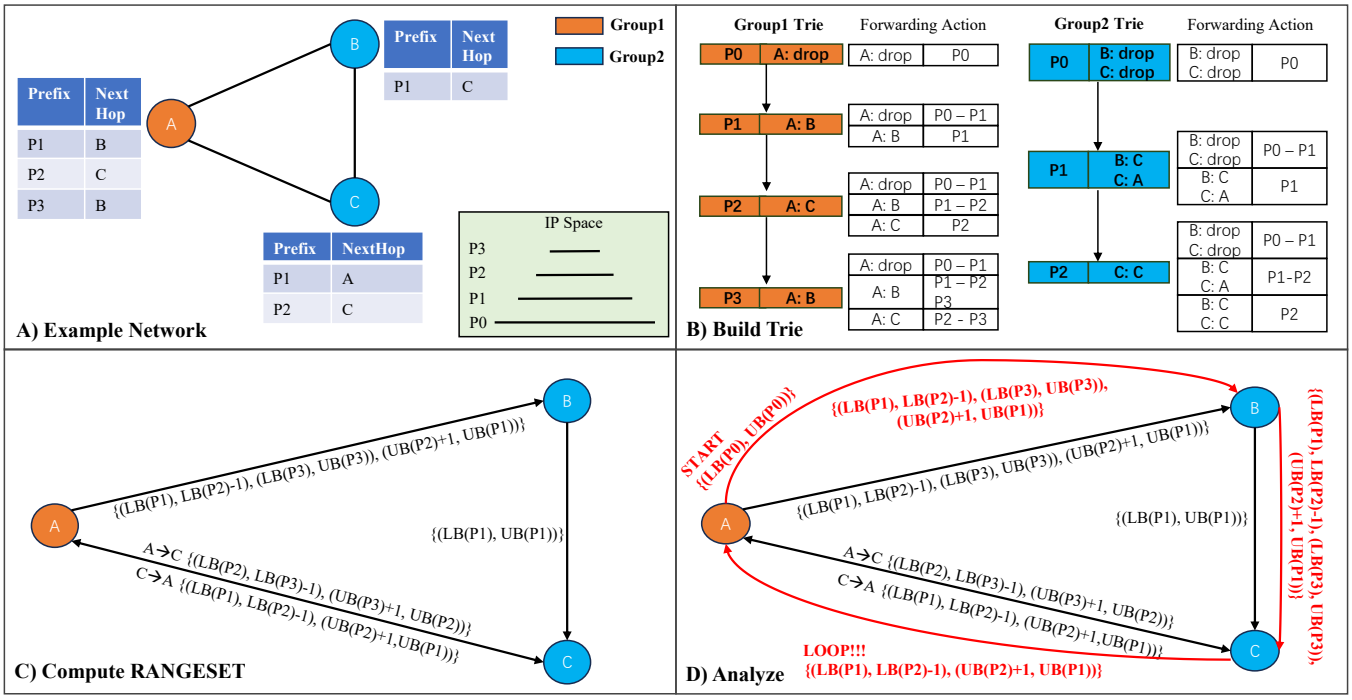

Figure 1: End to end example on a simple three device two group network.

traversing the trie node P0 shows that every device adheres to a default rule that necessitates dropping all packets. As the traversal proceeds to the trie node P1, we detect a change in the behavior relative to the default rules. Since the prefix P1 is longer than P0, it takes on a higher priority. Thus, the *RS* splits into two P0 - P1 and P1. Finally, we reach the leaf trie node P2 where, again, a change in behavior for a more specific prefix P2 is detected. Thus, P1 is split into P1 - P2 and P2. Note that if P2 had behaved the same as P1, there would have been no partition change. This is similar to APKeep's points about merging and splitting ECs. In this way, we can reduce some of the same *RS* operations. For example, there is only one $P0 - P1$ operation.

After obtaining the formula of the final *RS*, we can perform range-based operations to determine which packets are transmitted from one node to another. For example, let's assume that node A forwards packets in the range P1-P2+P3 to node B, where P1, P2, and P3 are also ranges. This operation can be symbolically represented as: $OR(AND(P1, NOT(P2)), P3)$. By evaluating this formula, we can obtain the specific *RS* or IP ranges that are transmitted from node A to B. For example, the output of the operation mentioned above could be the *RS {(LB(P1), LB(P2)-1), (LB(P3), UB(P3)), (UB(P2)+1, UB(P1))}*. These are depicted as the edge weights of the *RS* graph.

4.4 Analyze the model

Note that all the aforementioned steps can be performed in parallel for each group. For edges that cross different groups, the group that contains the source node is responsible for that edge. The final step involves traversing the *RS* graph to simulate packet forwarding. The pseudo-code for traversing the *RS* graph is provided in Algorithm 1. Although we mainly focus on identifying loops in this pseudo-code, we have extended it to identify all-all pair reachability (i.e., IP spaces reachable between all nodes), blackholes, etc.

Initially, we traverse the graph using DFS from each node using an *RS* that represents the entire IP space and an empty visited node set (lines 2-3). During DFS, if the node has been visited previously, the algorithm detects a loop (lines 6-7) and returns. Otherwise, it moves to all neighbors of the current node (lines 9-13). Before moving forward, the algorithm computes the corresponding *RS* that needs to be forwarded on that edge using the *get_RS* function (line 10). Then, it computes the intersection of the current *RS* that was forwarded to this node with this transferred *RS* using the AND operator (line 11). Finally, the intersection is forwarded to the neighbor (lines 12-13).

To illustrate the traversal process, let's consider a traversal starting from node A in which it initially carries the generic *RS* representing the entire IP address space of the network, denoted by *{(LB(P0), UB(P0))}*. Node A applies the AND operation with the $edge\_RS_{A->B}$ to obtain a resulting *RS* that it then transfers to node B. Node B, in turn, receives the *RS {(LB(P1), LB(P2)-1), (LB(P3), UB(P3)), (UB(P2)+1, UB(P1))}* from node A and transfers it to node C. After applying the AND operations with the $edge\_RS_{B->C}$, node C receives the same *RS* (i.e. AND with *{(LB(P1), UB(P1))}* does not change the *RS*). Finally, after applying the AND operation with the $edge\_RS_{C->A}$ (i.e. *{(LB(P1), LB(P2)-1), (UB(P2)+1, UB(P1))}*), node A receives the *RS {(LB(P1), LB(P2)-1), (UB(P2)+1, UB(P1))}*. As node A is visited again, this algorithm detects a loop in the network.

Algorithm 1: *RS* graph traversal

```
1  procedure RSTraverse()
2      for v ∈ V do
3          DFS(v, all_RS, empty_set);
4  Input: v: current node, RS: current RANGESET, visited:
       visited nodes;
5  procedure DFS(v, RS, visited)
6      if v ∈ visited then
           // Loop detected
7          return;
8      visited.add(v);
9      for n ∈ v.neighbors do
10         edge_RS ← get_RS(v, n);
11         transferred_RS ← RS AND edge_RS;
12         if transferred_RS ≠ false then
13             DFS(n, transferred_RS, visited);
14     visited.remove(v);
```

4.5 Optimization

To improve *Medusa*'s performance, we implemented several optimizations. One of these optimizations, which had a significant impact, was related to computing the RANGESET formulas. Once we traversed the trie, we obtained a formula in the form of

$$\bigcup_{i=1}^{n}(generic\_prefix_i - \bigcup_{j=1}^{m} specific\_prefix_{ij})$$

wherein the $specific\_prefix_{ij}$ is a subset of $generic\_prefix_i$. To speed up the computation, we use batch union instead of single union. Essentially, there are two options to compute the union of multiple RANGESETs from R_1 to R_N. The naive method, which involves multiple single unions, i.e., performing an iterative OR operation over the RANGESETs with time complexity of $O(nN)$, like $\texttt{OR}(R_1, \texttt{OR}(R_2, \ldots \texttt{OR}(R_{N-1}, R_N)))$, where N refers to the number of *RS* and n pertains to the total number of ordered pairs in the *RS* (as mentioned in §3). The alternative is to do Batch Union, which involves computing the union at once, like $\texttt{OR}(R_1, R_2, \ldots, R_N)$, using a technique similar to merging sorted lists. The time complexity is $O(nlogn)$, where n is again the total number of pairs in these *RS*. Generally, batch union is more efficient than multiple single unions because $logn < N$ (§5).

5 EVALUATION

We implemented *Medusa* in C++ (2.2K lines of code). To evaluate its performance, we utilized seven datasets (Table 1): three popular public datasets (Airtel [3], I2 [1], and Stanford [2]), along with four datasets sourced from our private data centers (PDC).

We evaluated *Medusa* against three state-of-the-art data plane verifiers: APKeep [20], Flash [11], and Tulkun [18]. For Flash and APKeep, we utilized the open-source implementations provided by Flash [5]. As for Tulkun, we used its open-source implementation directly. The open-source version of Flash does not support distributed deployment, so we tested it on a single server. This open-source version also only analyzes one subspace and does not fully implement the analysis of all subspaces. In order to achieve parallelism, we divided the subspace based on the first 6 bits of the IP addresses in Flash's experiments, which results in up to 64 theoretical threads. As a result, we executed all experiments for both Flash and *Medusa* using 64 threads.

Dataset	Nodes	Links	Forwarding Rules
Airtel	O(10)	O(10)	O(100K)
I2	O(1)	O(10)	O(10K)
Stanford	O(10)	O(10)	O(1K)
PDC1	O(10)	O(100)	O(100K)
PDC2	O(100)	O(1K)	O(100K)
PDC3	O(1K)	O(10K)	O(1M)
PDC4	O(10K)	O(100K)	O(1M)

Table 1: Dataset statistics. K and M stands for thousands and millions.

To create the data plane models for all verifiers, we included all IPv4 rules in each dataset. After creating the models, we instructed the verifiers to check for loop-free invariants in the networks. However, since Tulkun's open-source implementation cannot perform loop-free verification, we replaced it by verifying all-pair reachability instead. Note that Tulkun's structure is more suitable for all-pair reachability. Also, since loop-free regular expressions are more complex than all-pair reachability, we can deduce that Tulkun's loop-free computation time will likely be slower.

All experiments were conducted on a server running on an Intel Xeon Platinum 8336C processor with 128 cores and 512GB memory, clocked at 3.5GHz We used two primary metrics for performance evaluation: runtime, which is the time required to create and analyze the network model, and memory, which denotes the size of the network model. We took the average of five runs for both metrics and timed out all experiments after an hour. We also use *speedup* and *memory reduction* to refer to the performance factor by which *Medusa* surpasses other tools and the factor of memory that it saves compared to other tools, respectively.

Tool	Runtime in seconds (speedup)			
	Flash	APKeep	Tulkun	*Medusa*
Airtel	2.76 (1.8)	32.99 (22.2)	1206 (814.8)	1.48
I2	0.55 (2.7)	5.54 (27.7)	1.46 (7.3)	0.20
Stanford	0.25 (25)	1.00 (100)	1.63 (163)	0.01
PDC1	1.13 (11.3)	14.73 (147.3)	4.01 (40.1)	0.10
PDC2	6.26 (20.2)	243.24 (784.6)	TO	0.31
PDC3	18.89 (48.4)	1629.83 (4179)	TO	0.39
PDC4	3002 (613.9)	TO	TO	4.89

Table 2: Runtime comparison. TO means the tool timed out after running for an hour.

As indicated in Table 2, *Medusa* outperforms Flash, APKeep, and Tulkun by a significant margin. *Medusa*'s verification time is the lowest among all network topologies, offering up to 600X, 4000X, and 800X speedup compared to Flash, APKeep, and Tulkun, respectively. Furthermore, *Medusa* successfully completed the verification process for all network topologies without encountering any timeouts, which is a prevalent issue with some existing state-of-the-art

Tool	Memory in GB (memory reduction)			
	Flash	APKeep	Tulkun	*Medusa*
Airtel	4.23 (4.45)	1.26 (1.32)	3.91 (4.11)	0.95
I2	0.99 (5.82)	0.30 (1.76)	0.68 (4)	0.17
Stanford	1.01 (101)	0.11 (11)	0.62 (62)	0.01
PDC1	2.04 (10.2)	0.48 (2.4)	1.54 (7.7)	0.20
PDC2	5.56 (10.9)	1.16 (2.27)	TO	0.51
PDC3	7.10 (8.98)	2.93 (3.7)	TO	0.79
PDC4	93.51 (4.56)	TO	TO	20.49

Table 3: Memory comparison.

tools. In summary, when compared to other tools, *Medusa* is faster and more efficient for network verification.

As mentioned in §1, Tulkun uses DPVNet as a DAG for verification. However, as the network topology grows in size, the computational time required for Tulkun also increases significantly. This is because of the nature of DPVNet, which involves enumerating all the paths based on the topology and verified policies. Also, to the best of our knowledge, DPVNet creation is not parallelized, which further exacerbates the computation time. Thus, it is crucial to consider DPVNet computation time when calculating the modeling time of the verifier. On average, DPVNet computation accounted for 65% of the total runtime on the four networks that we were able to run Tulkun on. For example, DPVNet computation took 2.3 seconds out of 4.01 seconds runtime for PDC1, which represents a significant overhead for real-time verification.

As indicated in Table 3, the memory used by *Medusa* model is also significantly lower than other tools. It is important to note that BDD libraries use cache to optimize computation speed. As a result, DPV tools that rely on BDD initialization for verification will have an unavoidable memory overhead.

We also evaluated the effect of batch union on the same dataset. Our results show that, on average, $logn$ is two orders of magnitude less than N. Specifically, we found that the value of N is 12 times (small network) to 277 times (large network) greater than the value of $logn$.

6 FUTURE WORK

In our current approach, we divide the network into different groups based on device regions, such as grouping devices by city, state, etc. However, this approach may not be ideal as devices may not be evenly distributed across different regions. Therefore, we aim to identify the optimal method of network division by exploring different approaches.

We observed that increasing the number of threads did not result in a significant improvement beyond a certain point, although the exact point at which performance plateaus may differ depending on the input network. This is because the final step of traversing the graph remains single-threaded, requiring the combination of subgraphs before traversal. This remains a significant bottleneck for *Medusa*. As future work, we aim to explore further optimizations for parallelization of the final stage and the whole tool in general to enhance the performance of *Medusa*, allowing for better scaling with an increase in the number of threads.

Although group-based parallelization was our main focus in this paper, we are also exploring other parallelization techniques, such as IP address space parallelization. To be precise, we are examining whether or not we can effectively integrate multiple parallelization techniques.

Our verifier is a clean slate verifier, which means we need to remodel the network from scratch every time we do an analysis. To update the verifier incrementally, we can explore incremental verification techniques that update only the relevant group. Currently, we only focus on the destination IP field when forwarding packets. However, in reality, the data plane also includes ACLs dependent on other fields such as source and destination ports, etc. Additionally, features like tunneling include actions such as encapsulation, decapsulation, and header rewriting. We are exploring ways to improve our approach to incorporate these features.

Furthermore, while we are currently focusing on parallelizing specific components across different threads within the same machine, we may need to investigate how to conduct this parallel analysis across multiple machines as the network grows larger. However, this poses challenges such as how to reduce the communication burden across multiple servers during final result analysis.

An alternative direction is to enhance the parallelization capabilities of BDDs even further. But, the major challenge would be to correlate BDDs representing different ECs across multiple groups.

Acknowledgements. We would also like to thank our colleagues Jiawei Chen, Xu Liu, and the anonymous reviewers for their valuable insights and feedback.

REFERENCES

[1] [n. d.]. *The Internet2 Observatory.*
[2] 2012. *Hassel,the header space library.*
[3] 2017. *Airtel dataset.*
[4] 2021. Hyperscale Data Center Count Grows to 659 – ByteDance Joins the Leading Group. https://www.srgresearch.com/articles/hyperscale-data-center-count-grows-to-659-bytedance-joins-the-leading-group. Last accessed on Sep 14, 2023.
[5] 2022. *Flash code.*
[6] 2023. Cloudflare Radar. https://radar.cloudflare.com. Last accessed on Sep 14, 2023.
[7] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 201–219.
[8] Dirk Beyer, Karlheinz Friedberger, and Stephan Holzner. 2021. PJBDD: a BDD library for Java and multi-threading. In *Automated Technology for Verification and Analysis: 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18–22, 2021, Proceedings 19*. Springer, 144–149.
[9] Luigi Capogrosso, Luca Geretti, Marco Cristani, Franco Fummi, and Tiziano Villa. 2023. HermesBDD: A Multi-Core and Multi-Platform Binary Decision Diagram Package. In *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. IEEE, 87–90.
[10] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 469–483. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/fogel
[11] Dong Guo, Shenshen Chen, Kai Gao, Qiao Xiang, Ying Zhang, and Y Richard Yang. 2022. Flash: fast, consistent data plane verification for large-scale network settings. In *Proceedings of the 2022 Conference of the ACM Special Interest Group on Data Communication*. 314–335.
[12] Dong Guo, Jian Luo, Kai Gao, and Y Richard Yang. 2023. Poster: Scaling Data Plane Verification with Throughput-Optimized Atomic Predicates. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 1141–1143.
[13] Alex Horn, Ali Kheradmand, and Mukul Prasad. 2017. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 735–749.
[14] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, et al. 2019. Validating datacenters at scale. In *Proceedings of the ACM Special Interest Group on Data Communication*. 200–213.
[15] Jørn Lind-Nielsen. 1999. BuDDy: A binary decision diagram package. (1999).

[16] Alberto Lovato, Damiano Macedonio, and Fausto Spoto. 2014. A thread-safe library for binary decision diagrams. In *International Conference on Software Engineering and Formal Methods*. Springer, 35–49.

[17] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 953–967.

[18] Qiao Xiang, Chenyang Huang, Ridi Wen, Yuxin Wang, Xiwen Fan, Zaoxing Liu, Linghe Kong, Dennis Duan, Franck Le, and Wei Sun. 2023. Beyond a Centralized Verifier: Scaling Data Plane Checking via Distributed, On-Device Verification. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 152–166.

[19] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. 2014. Libra: Divide and conquer to verify forwarding tables in huge networks. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 87–99.

[20] Peng Zhang, Xu Liu, Hongkun Yang, Ning Kang, Zhengchang Gu, and Hao Li. 2020. APKeep: Realtime Verification for Real Networks. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 241–255.