Mahjong: A Generic Framework for Network Data Plane Verification

Anonymous Author(s)

ABSTRACT

Existing network data plane verification approaches check network correctness with different models and algorithms. A network operator needs to try a number of dazzling verification approaches to find a proper one with sufficient functionality and suitable performance for his/her network and intents. The inconsistent input and output of existing verification tools also cause problems for operators. With respect to a specific scenario, it is hard to judge which network model is the most efficient one, because existing verification approaches are implemented with different languages and evaluated against different datasets on different hardware platforms in their papers. To solve the problems above, we propose a division scheme that can divide approaches into general modules. Based on this division scheme, we propose a generic data plane verification framework, Mahjong. We refactor three classic verification approaches by modular programming and merge them into the Mahjong framework to perform easy use and fair comparison as examples. We also propose a uniform input format and a module configuration file for easy use. With the well-defined interfaces, future new approaches can be merged into Mahjong conveniently.

CCS CONCEPTS

• **Networks** → **Error detection and error correction**; • **Computing methodologies** → *Modeling methodologies.*

KEYWORDS

Network verification and modeling; Modular programming

1 INTRODUCTION

Network verification has been intensively studied to ensure the correctness of networks. Figure 1 shows an overview of network verification. The verification methods can be classified into three categories by the network targets: (1) telemetry checks the consistency of packet behavior and high-level intents [12][5][17]; (2) data plane verification checks the consistency of firmware rules/functions, *e.g.*, forwarding, firewall, and high-level intents; (3) control plane verification checks the consistency of control protocol configurations, *e.g.*, BGP policies[4], and high-level intents[2][16].

The focus of this study is on data plane verification, which checks the reachability and invariants of a network by statically analyzing the network state snapshots. We use the network shown in Figure 2 as a basic example to show the basic concept of data plane verification. The network operator wants to add a new rule(rule 0) to the switch R1. To check whether S1 and S2 still reach each other, the operator applies a data plane verification tool. The tool then finds the rule 0 drop the packets send from S1, therefore this buggy insertion is prevented.

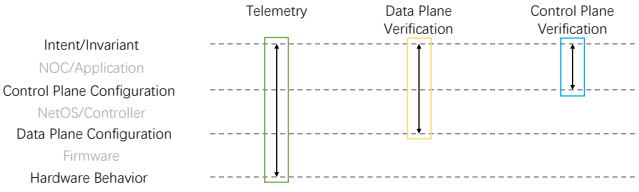


Figure 1: An Overview of Network Verification

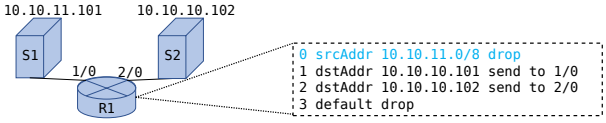


Figure 2: An simple example network

Compared to telemetry, data plane verification is done offline, and can find failures in routing synchronization which can not be found by control plane verification. Data plane verification has a relatively long history back to 2005[14] and is still rapidly progressing[20]. Various data plane verification tools have been proposed, such as Anteater[11], HSA[8], APVerifier[15] and others[10][19][9][7][6].

The question is which data plane verification approach is suitable when a network operator prepares to verify a network. Previously proposed methods are based on different network models and choose different algorithmic solutions. Moreover, these verification tools have their own implicit input constraints. It is usually infeasible to directly deploy the original verification implementation. Therefore, large Internet companies tend to re-develop the verification tool by themselves, and small and medium-sized institutions have no appropriate tools, adopting network verification slowly.

We use some examples to show the versatility of verification tools. HSA[8] cannot deal with large-scale networks due to its time costing header space set operations and the large state space on which its transfer function, *i.e.*, an algorithm to simulate the network packet processing procedure, have to search. Anteater[11] uses the SMT solver as its transfer

function, hence it can only find a single path between two ports. However, the performance of Anteater is far better than HSA in large-scale networks. BDD[1] header expression used by APVerifier[15] supports fast header space set operation, and whereas BDD makes header-rewrite representation complex. To find all possible paths in a large network without packet header rewriting, the composition of BDD Header expression and HSA transfer function is a better choice.

Therefore, with *a framework which unifies existing data plane verification approaches and switches between different modules of different approaches flexibly*, operators can find a suitable data plane verification solution for a specific scenario conveniently and precisely. To realize this idea, we need to tackle the following challenges:

- *Clear division.* The data plane verification structure needs to be explicitly defined to support all data plane verification tools. In this structure, verification tools are divided into modules, such as header expression and transfer function model.
- *Flexible composition.* Network operators should be able to customize verification solutions according to their requirements. All the compositions should have a uniform input format, and hence operators can navigate these compositions conveniently.

In this paper, we present Mahjong, *a generic framework for flexibly compositing network data plane verification solutions*. To address the aforementioned challenges, the main approaches of Mahjong are as follows. First, Mahjong analyzes the function and performance of data plane verification by divided modules instead of scattered tools. Then, Mahjong defines the general interfaces of divided modules and is implemented with modular programming. The following summarizes our main contributions:

- *Verification approach division scheme.* We propose a modular division scheme for data plane verification approaches, including header expression, rule set, and transfer function. We refactored three classic data plane verification approaches based on our modular scheme, proving the universality of the scheme.
- *Generic data plane verification framework.* We propose a generic framework, Mahjong, which defines interfaces between divided modules with abstract classes. Operators can use the Mahjong framework to compose verification solutions with these modules through a simple configuration file. The interfaces are well-defined and future new verification tools can be easily merged into the framework.

The rest of this paper is organized as follows. Section II summarizes and analyzes related works, and states the problem Mahjong going to solve. Section III proposes the design of Mahjong, including the module division and pipeline procedure. The implementation of Mahjong is presented in Section IV. Section V shows the evaluation results of Mahjong. Section VI concludes this paper.

2 BACKGROUND

Since static reachability analysis has been proposed by Xie et al. in 2005[14], data plane verification research has gone a long way. Existing works are mostly devoted to designing data plane verification tools with more comprehensive functions and faster computation speed. Typical data plane verification approaches and their theoretical models are shown in Table 1. We analyze representative approaches in detail as follows:

- *HSA* uses a bit array to express a packet header. To be exact, it uses two physical bits to express one logical bit in packet headers which is implemented by int arrays in code. Then the header match procedure is calculated by int AND operation. Rules in HSA have the same match field as that of the network device snapshot. They may overlap with each other but assigned different priorities. The transfer function is the same as proposed in static reachability analysis[14], which uses a graph search algorithm and pushes packets to flow from the source node to the destination node.
- *Anteater* uses two bit arrays to express one packet header. One is used for match fields and the other is used for mask fields. The original rules are converted to non-overlapped rules, lifting the priority constraint. And the transfer function is expressed by an SMT solver.
- *APVerifier* uses BDD to express packet headers. The original rules are converted to atomic equivalent classes. An equivalent class represents a set of packet headers which are processed identically, and is physically expressed by one bit. During preprocessing, headers are also cast into equivalent classes. Therefore, packet matching can be done by comparing equivalent bit arrays. The transfer function is also static reachability analysis with a graph search algorithm.

The development of data plane verification lacks continuity on network models and verification algorithms. Some subsequent data plane verification approaches are completely new methods and have no common theatrical foundations with the previous ones. As a result, network operators are easily confused by these dazzling solutions and researchers can hardly grasp the developing progress of data plane verification works. It is quite guideless and directionless to choose and deploy verification tools. Therefore, a generic data plane verification framework which makes solution selection convenient and precise is necessary and urgent for network

Table 1: The comparison between existing network data plane verification works

Tool	Header expression	Rule Set	Transfer Function	Remark
HSA[8]	Wildcard	Priority	Static reachability analysis	
Anteater[11]	Bitmask	Uncovered	SMT solver	
APVerifier[15]	BDD	Atomic	Static reachability analysis	
NoD[10]	BDD/Wildcard	naive	μZ	
Libra[19]	IP/Prefix length	Subnet slicing	Graph algorithm	Use MapReduce to speed up
APKeep[20]	BDD	PPM	Static reachability analysis	
NetPlumber[7]	Wildcard	Equivalent class	Static reachability analysis	Incremental
Veriflow[9]	Bitmask	Equivalent class	SMT solver	Incremental
Delta-net[6]	IP/Prefix	Edge labeling	Static reachability analysis	Incremental
Jinjing[13]	BitVec	Equivalent class	SMT solver	Only ACL. Gives fix advice.

operators. Mikado[18] is proposed to solve the difference between network testing tools, but there is no solution for network verification tools yet.

3 DESIGN

By analyzing existing data plane verification tools, we find that all data plane verification tools are composed of the three pillar modules to statically analyze the packet forward procedure in a network:

- *Header expression.* A way to represent packet headers. In this way, packet header intervals can present set operations such as intersection, subtraction, and equal.
- *Rule Set.* A way to record rules. A rule should have its match fields, input ports, and actions. The actions contain forward, drop or rewrite, and so on.
- *Transfer Function.* A set of algorithms that can use the rules and given input packet header to calculate how the network deals with this packet.

Figure 3 is an example showing how the above three modules are linked in a traditional data plane verification tool.

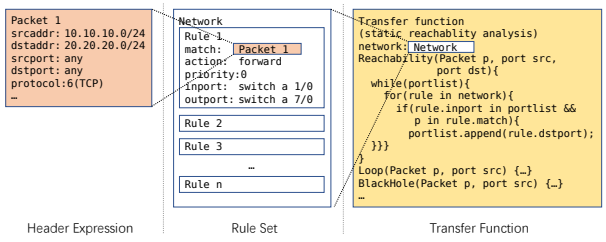


Figure 3: How header expression, rules and transfer functions are linked[14].

With the three modules, the high-level Mahjong framework is shown in Figure 4. Mahjong decomposes verification approaches into modules, each of which has a series of related operations or functionalities. The input data plane states and network operator configurations are processed by Mahjong pipeline procedure, and then the verification results and procedure performance are exported.

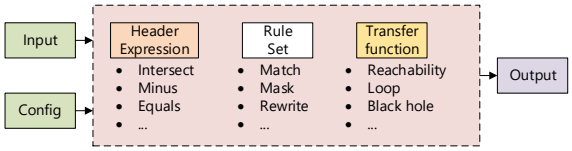


Figure 4: The high-level framework of Mahjong

The following subsections will elaborate on the pillar modules and pipeline procedure of Mahjong.

3.1 Pillar Modules

The HSA, Anteater, and APVerifier are all classic and influential data plane verification tools. The analysis in section II shows that their algorithms hugely differ. Therefore we take these three tools as examples to illustrate the feasibility of our module division. The following modules are extracted from them.

Header expression

01x wildcard uses two physical bits to represent one logical bit in packet headers. In detail, it uses physical 10 to present logical 1, 01 to present logical 0, 11 to present x which is logical all, 00 to present logical none. Packet header intersect operation is performed by bit AND operation. However, it is annoying to express non-continuous intervals, which is common in packet header subtract operation.

01 bitmask* uses two bitvectors to represent a packet header: one bitvector represents match bits and the other represents mask bits. The packet header processing, such as intersect, is more complex than 01x wildcard, but there is no need to add packet header non-empty constraint when using 01* bitmask with SMT solver.

BDD uses bool conditions to represent a packet header. It is efficient to express non-continuous packet header intervals by adding a branch in the diagram. The packet header operations are implemented by logical bool operations.

Rule Set

Priority records the whole rule match field, and applies rules to packets in priority order. If a packet header matches a high priority rule, low priority rules will do nothing to it.

Non-overlapping subtracts the high priority rule match fields from the low priority rule match fields to make rule match fields uncovered.

Atomic. Use the non-overlapping rules to calculate the greatest common divisor of all rule match fields in the whole network. Each divisor is projected to an integer index and rule match fields and packet headers are projected to integer sets. The rule match action is implemented by set operations.

Transfer Function

Static reachability analysis. Given a source port and a destination port, the transfer function use rules to calculate which packets will reach the next port start from the source port. Then use a graph search algorithm to figure out which packets reach the destination port. Recording the visited port, this transfer function can also locate loops and black holes.

Static reachability analysis inversely. Similar to static reachability analysis, the difference is the path search starts from the destination port and ends at the source port. It achieves better performance in some specific topologies.

SMT solver encodes rules and constraints into an SMT model and uses an SMT solver like Z3 to find a possible solution. For example, in Figure 1, the policy *dstAddr 10.10.10.101 send to 1/0* can be encoded as $(R1_{1/0in}\ OR\ R1_{2/0in})\ AND\ p == 10.10.10.101\ implies\ R1_{1/0out}$. The reachability to S2 is encoded as $assert\ R1_{2/0out}$. The encoding way is not monopolized. The way used by Anteater can find loops, black holes, and one possible path but cannot find all possible paths between 2 ports.

3.2 Pipeline Procedure

The detailed design of Mahjong is shown in Figure 5. The verification pipeline flows from the input data plane policies and module configurations to rule set processing, and then to the network transfer function for global simulation, and finally to the output results. Each of the three modules has its corresponding selection mechanism. Preprocessing optimizations, such as slicing and symmetry, are applied to the network rule set, therefore the preprocessing stage is contained within the rule set module.

Note that some verification approaches perform unique preprocessing optimizations such as APKeep and Libra, and the rule set calculation procedure in other approaches may overlap. The raw input rules from network devices are priority rules. Non-overlapping rules are calculated from priority

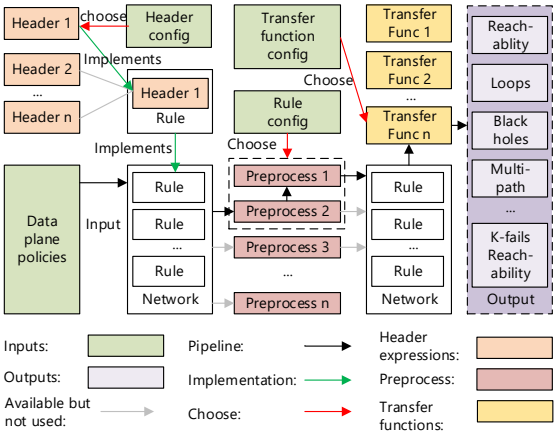


Figure 5: The detailed framework of Mahjong

rules, and atomic rules are calculated from non-overlapping rules. It is more efficient to process the rules with a serialized function pipeline and choose a position to break rather than build mutually exclusive rule modules.

To solve the partly-sequential rule granularity and total-mutual header expression and transfer function, we design the module selection mechanism in different ways as a solution.

- For header expression and transfer function, different modules have different basic data structures and basic operations. Therefore, headers and transfer functions are implemented with mutually exclusive modules.
- Rule match and rewrite operation depend on intersect and equal operations implemented in header expression, therefore the data structure of a single rule is no need to refactor. Instead, we differ the rule granularity by the choice of preprocessing functions applied to the whole network rule set.

As mentioned above, we need to implement various header expressions and transfer functions but only one rule expression. The header expressions and transfer functions are mutually exclusive, while the rule granularity which is determined by the preprocessing procedure has both mutually exclusive alternatives and serialized processing.

With module division, we can analyze functionality in the granularity of modules, and the function of the whole verification tool can be calculated by intersecting the function of modules. For example, in the calculation of reachability, 01x wildcard and atomic rules support finding all possible paths and finding one path, but SMT solver (with Anteater encoding) only supports finding one path. Therefore, if we build a verification tool with 01x wildcard, atomic rules, and SMT solver, this tool only supports finding one path. In conclusion, Analysis in module granularity helps researchers to find the bottleneck of verification tools, and make improvements.

4 IMPLEMENTATION

As Mahjong is built to facilitate operators to easily switch among modules, we choose to implement the framework with the idea of modular programming. Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains all necessary to execute only one aspect of the desired functionality [3]. The interfaces expressing the elements that are provided and required by the module need to be defined first.

- *Header expression:* Set operations include intersect, complement, add, minus, rewrite, contains, and isempty.
- *Rules:* Only a general rule class is needed. It contains elements including in_ports, out_ports, match, rewrite mask, rewrite value, action.
- *Network:* The network class contains port set and rule set.
- *Network preprocessing:* This stage is implemented as static functions, whose input is a network and returns a preprocessed network.
- *Transfer function:* A transfer function has an element, network. The interfaces are verification functions, *e.g.*, find reachability, find loops, and verify all. The outputs of the interfaces are verification results.

There are several programming languages which provides interface classes, for example, C++, C#, Java, etc. Considering the portability and performance, we choose Java to implement the Mahjong. We then describe problems and corresponding solutions in Mahjong implementation.

4.1 Mechanism gap in transfer functions

```
# forward rules
# action$inport$match$outport
fwd$[10001]$11x$[10002]
fwd$[10001]$xxx$[10003]

# rewrite rules
# action$inport$mask$rewrite
rw$[10001]$01x$11x

# links
# action$inport$outport
link$[10002]$[20001]...
```

(a) Uniform input format

```
# config file
public class TypeConfig{
    public static String HEADER_TYPE =
        "APHeader";//BDD
    public static String TF_TYPE =
        "Z3";
    ...
}
```

(b) Module choose config file

Figure 6: Code pieces

Transfer functions mainly have two types. One is generating constraints according to rules first and then getting the verification result in one breath, including model checking ways like SMT solver, μZ, and equivalent class graph algorithm used by Libra. The other is using rules to get a temporary result at each step and get the final result by iteration. A typical example is static reachability analysis used by HSA and APVerifier. It is difficult to meet these two types of methods, while the base data structure used by header expression should not expose to transfer functions. To bridge the gap, we defined constraint generation function and rule apply function in the rule class separately.

4.2 Input format

Existing data plane verification approaches mainly use data plane snapshots directly dumped from network devices as inputs. These devices are made by various providers and the formats of snapshots also vary. Writing parsing modules for each of them is an artificial job and hardly contributes to the framework. To keep the framework clean, we choose to define a uniform input format recording the general rule data structure shown in Figure 6(a).

The priority is recorded by the line order. We write an example script which can parse data plane snapshots of Cisco devices and generate an input file of the uniform input format. In this way, other network devices can easily be merged into the framework by adding parsing scripts.

4.3 Module selection

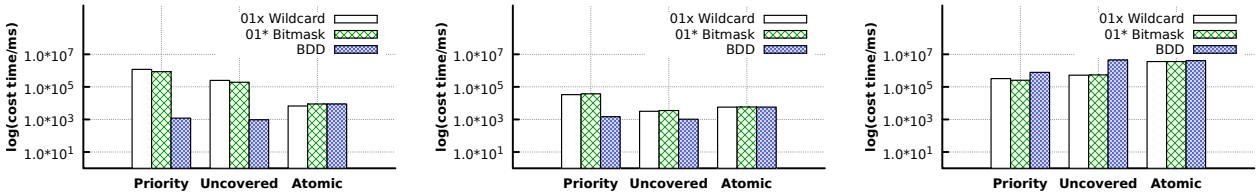
To select modules, operators only need to modify a config file like Figure 6(b). We use the factory design concept in Java to implement instantiation based on the config.

To add new modules to the Mahjong, developers only need to add a class to the project and register the module in the factory class. The project is open-source and a module merge instruction is provided to potential developers.

We need to mention that we can't assert the combination of 01x wildcard, priority rules, and static reachability analysis has the same performance as HSA because minor details in implementation can cause a huge performance difference. Here is an example. In the 01x wildcard header expression we implemented, there is a reorganization function which can simplify the result of add or minus operation, such as transform $xx-10-11$ to $0x$, and then speed up the following add or minus operation. However, sometimes it will brings huge time cost, such as transform $xxx-111-000$ to $110+101+011+100+010+001$. When the rule list is tedious, the calculation time will be unacceptable and worse than HSA. If an intelligent optimization algorithm is applied to this function, the performance may be better than HSA.

5 EVALUATION

Our evaluation aims to answer whether Mahjong makes the development and customization of data plane verification tools more convenient, and how a network operator adjusts a verification tool precisely suitable for her/his network. All experiments are conducted on a commodity server with 16GB memory and Intel i5 8400 CPU. Currently, Mahjong contains three header expressions, three rule granularity, and three transfer functions mentioned in Section III.



(a) Static reachability analysis (b) Inverse static reachability analysis (c) SMT solver

Figure 7: The verification time of various compositions

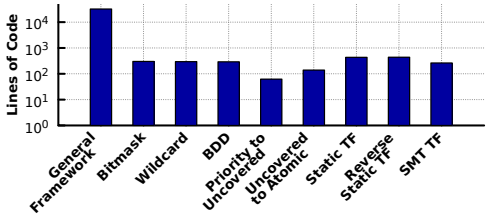


Figure 8: LoC cost of each module.

Convenience. First, We count the lines of code(LoC) of the specific modules and the reusable general framework. As shown in Figure 8, the whole program is built with over 30,000 LoC, while implementing a new module only needs several hundred LoC. It proves Mahjong hugely shortens the iteration cycle of data plane verification technology.

Then, we modify the config file to prove the convenience of module composition. We evaluate the $3*3*3 = 27$ compositions of all implemented modules with a fat-tree network. The verification objective we choose is all-pair reachability. Evaluation results are shown in Figure 7.

All of the compositions work correctly. Their performances differ and are consistent with theoretical analysis. For example, the BDD header performs well with static reachability analysis but performs worst with SMT solver, because it uses pointers to perform intersect with static reachability but is transformed into bool expression in SMT constraints.

Table 2: Composition to verify Stanford backbone

Header expression	Rule granu.	Transfer func.
01x wildcard	Priority	Static reachability analysis
BDD	Priority	Static reachability analysis
01x wildcard	Priority	SMT Solver

Precision. We choose an example calculating reachability between a pair of ports on the Stanford backbone network to simulates a module choose tweaking procedure in practice.

Table 3: Experiment on Stanford backbone dataset

	Combination	Cost time
0	01x wildcard + priority rules + static reachability analysis	More than one day
1	01x wildcard + priority rules + SMT solver	1315.6 ms
2	BDD + priority rules + static reachability analysis	371.6 ms

The dataset is generated from Cisco data plane snapshot parsing. Compositions are shown in Table 2. Evaluation results are shown in Table 3.

In the beginning, the composition 0 cannot give a result in a reasonable time. One possible solution is switching the transfer function to SMT constraints. The SMT solver only finds one path, and hence it is faster, proved by the result. This is a way of sacrificing functionality for performance. Another solution is switching the header expression from 01x wildcard to BDD. In this way, the time-consuming reorganization during minus operations is eliminated. Then the verification time becomes reasonable.

6 CONCLUSION

This paper proposes Mahjong, a generic network data plane verification framework, which allows operators to compose verification solutions with functional modules. Mahjong is designed with a modular division scheme. The interfaces of the modules are well defined, and different modules with the same functionality used by different verification approaches can be re-composed and work together. Leveraging Mahjong, researchers can evaluate the functionality and performance of verification approaches, and improve the approaches in module granularity. We implemented 9 modules divided from 3 representative works HSA, Anteater and APVerifier as examples and benchmarks. We also defined a uniform network snapshot input format and the modules can be easily chosen and combined by a configuration file.

REFERENCES

[1] Sheldon B. Akers. 1978. Binary decision diagrams. *IEEE Transactions on computers* 27, 06 (1978), 509–516.

[2] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 155–168.

[3] Kenneth Leroy Busbee. 2013. Programming Fundamentals: A Modular Structured Approach Using C++. (2013).

[4] 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)*. 469–483.

[5] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*. 357–371.

[6] 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.

[7] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 99–111.

[8] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*. 113–126.

[9] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 15–27.

[10] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking beliefs in dynamic networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 499–512.

[11] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 290–301.

[12] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 129–143.

[13] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*. 214–226.

[14] Geoffrey G Xie, Jibin Zhan, David A Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. 2005. On static reachability analysis of IP networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, Vol. 3. IEEE, 2170–2183.

[15] Hongkun Yang and Simon S Lam. 2015. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking* 24, 2 (2015), 887–900.

[16] Fangdan Ye, Da Yu, Ennan Zhai, Hongqiang Harry Liu, Bingchuan Tian, Qiaobo Ye, Chunsheng Wang, Xin Wu, Tianchen Guo, Cheng Jin, et al. 2020. Accuracy, Scalability, Coverage: A Practical Configuration Verifier on a Global WAN. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 599–614.

[17] Minlan Yu. 2019. Network telemetry: towards a top-down approach. *ACM SIGCOMM Computer Communication Review* 49, 1 (2019), 11–17.

[18] Yifei Yuan, Sanjay Chandrasekaran, Limin Jia, and Vyas Sekar. 2018. Efficient and correct test scheduling for ensembles of network policies. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 437–452.

[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.