ICIC Express Letters
Volume **13**, Number **10**, October **2019**

ICIC International ©2019 ISSN 1881-803X
pp. **963–970**

FAST TEST PACKET GENERATION FOR NETWORK DATA PLANE

Yaxin Li[1] and Yong Xie[2]

[1]College of Agriculture and Animal Husbandry
[2]Department of Computer Science and Technology
Qinghai University
No. 251, Ningda Road, Xining 810016, P. R. China
{ 1936978302; Mark.y.xie }@qq.com

Received February 2019; accepted May 2019

Abstract. *Modern networks are growing increasingly complex, and various network faults occur inevitably. Recently, researchers have proposed solutions to generate test packets to test network data plane. However, they either take very long time to generate test packets or suffer poor coverage for data plane. In this letter, we propose a new test packet generation framework named NetGen. It can generate test packets for entire data plane in a very short time. In addition, it supports real-time test packets update in an incremental way. We evaluate NetGen using actual networks, and the evaluation shows it accelerates test packet generation by an order of magnitude compared with state-of-the-art tools.*

Keywords: Network data plane, Formal methods, Incremental test packet update

1. **Introduction.** Network faults (e.g., configuration errors, software errors in devices and broken interfaces) can lead to network outages. These network outages can compromise availability, security, and performance of the network. Thus, network users usually need to ensure the correct behavior of their networks by systematically reasoning about the networks, which has inspired the field of network verification and network testing. Towards reliable network, researches have recently proposed a series of network verification and testing tools. Control plane verification tools are proposed to check network properties by inspecting configuration files [1, 2]. However, these tools can only check the bugs or errors in configuration files. Software errors in devices and broken interfaces are out of their scope of work. An increasing number of data plane verification tools, like Configchecker [3], and Delta-net [4], check whether forwarding information bases (FIBs) violate generic network reachability properties based on formal methods (e.g., SAT Solver, and Datalog Solver). These approaches [5, 6] can detect network errors which impact FIBs' generation (e.g., configurations errors). However, faults in actual implementation of data plane, like mislabeled cables and crashed devices, still cannot be detected by these approaches. While, according to the survey conducted by North American Network Operators' Group, network faults in actual implementation of data plane are a leading cause of network misbehave [7]. To detect these faults, network users always require to ensure that each interface in data plane executes correctly. With primordial tools (e.g., ping, and traceroute), they have to execute the command thousands times. For each run, users can only analyze whether a single given packet reaches its destination. Some data plane testing works, like RuleScope [8], are proposed to check whether forwarding rules are correctly installed in a switch. However, they are designed to generate one probe for each rule. These tools will generate too many probes, and hurt regular traffic. Tools, like ATPG [7] and Pronto [9], generate test packets to exercise every rule and link based on

DOI: 10.24507/icicel.13.10.963

FIBs. However, either of them can generate or update test packets in a short time. ATPG [7] even takes thousands of seconds to generate test packets for even a campus network.

To address above problems, we develop a new framework, NetGen, to detect the errors of actual implementation of data plane. For saving bandwidth, it generates the minimum number of test packets that travel every possible forwarding rule and port, so that any fault will be observed by at least one test packet. In actual networks, forwarding rules may undergo updates frequently, as users always need to deploy new policies (e.g., traffic engineering) [10]. It is rather time-consuming to regenerate test packets. With novel algorithms, NetGen can efficiently update test packets to handle FIBs changes in an incremental way.

The state-of-the-art data plane testing tools, like ATPG [7] and Pronto [9], can generate test packets to cover all forwarding rules. Compared with them, our novel test packet generation framework avoids redundant calculations of FIBs, which can accelerate the test packets generation. In addition, ATPG can only work for static FIBs, which limits its application in real networks. In contrast, our tool supports FIBs updates by updating test packets. To the best of our knowledge, NetGen is the first data plane testing tool that can update test packets in real time.

The remainder of this paper is organized as follows. We provide the formal definition of the research problem in Section 2. Section 3 provides an overview of our framework, and then the details of the approach are introduced. In Section 4, we present the evaluation results of our approach compared with state-of-the-art methods. Finally, we present concluding remarks in Section 5.

2. **Problem Statement.**

2.1. **Forwarding basics.** *Rule*: Each rule r is a tuple $< rv, p >$, where rv is a bit-vector to encode matching conditions (using 1, 0 and $\star$), and p refers to an output port. rv models a range of IP address. For example, 10.0.1.0/24 corresponds to an rv 00001010 00000000 00000001 $\star\star\star\star\star\star\star\star$.

Device: Each device (e.g., switch, and router) R contains a set of forwarding rules (e.g., FIBs) $r_1, r_2, r_3, \ldots, r_n$. A packet header h, which is a bit-vector, matches rule r_j, if each of the vectors $rv_1, rv_2, rv_3, \ldots, rv_{j-1}$ contains a conflicting bit, whereas rv_j has no such conflicting bit. The matching condition for rule r_j can be defined by the Boolean function representing the set of bit-vectors $rv_j \backslash \{rv_1, rv_2, \ldots, rv_{j-1}\}$[1]. A packet with the header h is forwarded to the port defined in the rule r_j, only when h is evaluated to true by the matching Boolean function of rule r_j.

Network: A network N consists of a set of devices $R_1, R_2, R_3, \ldots, R_n$. Device R_i contains the list of ports $p_i^1, p_i^2, p_i^3, \ldots, p_i^n$, where n is the number of ports. The link from port p_j^l to its adjacent port p_i^k can be denoted as a tuple $< p_i^k, p_j^l >$.

2.2. **Forwarding faults.** The problem in this letter is to reveal all the forwarding faults of the rules and links in network N. We tackle the problem by generating test packets to systematically exercise all rules or links. A fault happens to a rule r_j in device R, for any packet with the header h satisfies matching condition for rule r_j, the device processes h without forwarding to the port defined in the rule r_j. A fault happens to a link $< p_i^k, p_j^l >$, if packet with header h is forwarded by the port p_i^k but port p_j^l does not receive any packet with the header h.

To check whether a rule r is correct, the test packet h should satisfy the rule's Boolean match condition. It is equivalent to verifying whether assignments make a Conjunction Normal Form (CNF) true. Therefore, rule-forwarding fault probe generation is in NP-complete. Link-forwarding fault probe generation is similarly handled.

[1]For example, the $1\star\star\backslash\{110, 101\}$ encodes the set $\{111, 100\}$.

3. **NetGen Overview.** Figure 1 shows the work flow of NetGen. First, it collects FIBs from the devices in the network. By computing reachability-to-rules table, it generates the minimal test packets which can exercise every rule and link. Then, via an agent, it requests terminals to send test packets. If the destination host or switch receives the packets, it confirms the monitor that these exercised rules and links work correctly (①-④ in the figure). To make sure the test packets can work for the changed forwarding rules, it updates reachability-to-rules table and test packets incrementally (⑤-⑦ in the figure).

3.1. **Test packets generation.** The test packet generation mainly consists of constructing reachability-to-rules table and selecting test packets. Table 1 shows an example table for the toy network in Figure 2. r_{11} denotes that packets with the header representing 10.0.1/24 (0000101 0 00000000 000 00001 $\star\star\star\star\star\star\star\star$) are forwarded to port p_1^1. Other rules are similarly defined. For each entry in Table 1, any packet with the header (i.e., the 6th column) can cover all rules (i.e., the 4th column) and links (i.e., the 5th column) associated with that entry. For example, entry 1 refers to that packets with the header whose destination is 10.0.1/24 can be forwarded to exercise rules $\{r_{11}, r_{22}\}$ and links $\{< p_A^0, p_1^0 >, < p_1^1, p_2^0 >, < p_2^1, p_B^0 >\}$.

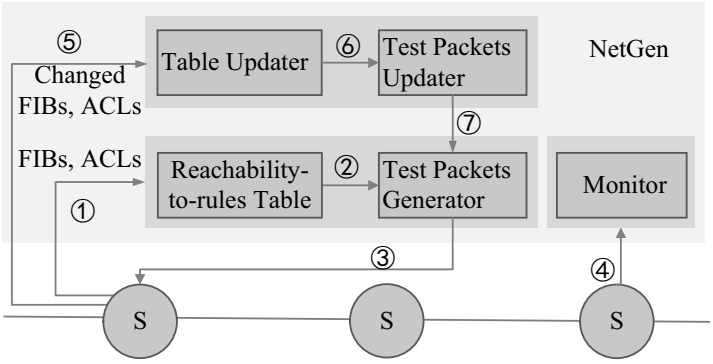

FIGURE 1. High level image of NetGen. S: Switch.

TABLE 1. Reachability-to-rules table for the network in Figure 2

Entry#	Ingress	Egress	Rules	Links	Header
1	p_{A0}	p_{B0}	r_{11}, r_{22}	$< p_A^0, p_1^0 > < p_1^1, p_2^0 > < p_2^1, p_B^0 >$	$dst = 10.0.1/24$
2	p_{B0}	p_{A0}	r_{21}, r_{12}	$< p_B^0, p_2^1 > < p_2^0, p_1^1 > < p_1^0, p_A^0 >$	$dst = 10.0.2/24$
...	...	...	...	...	...

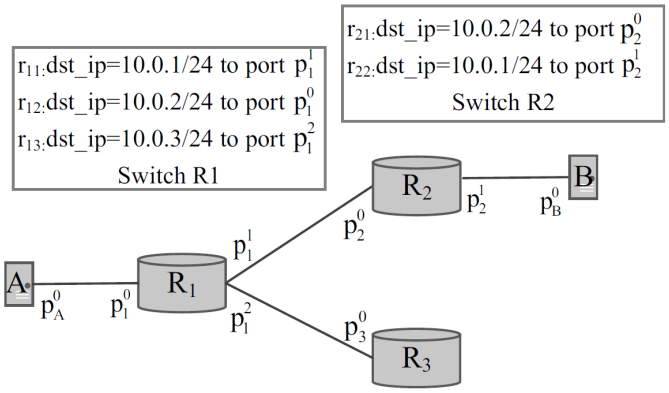

FIGURE 2. An example network

3.1.1. *Constructing maps for rules in each device.* To construct Table 1, we need to reason about how packets are processed in the network. As a rule in each device is a tuple $< rv, p >$ which maps bit-vector to output ports, it is rather time-consuming (like ATPG [7]) to compute packet forwarding with these bit-vectors. To speed up the computation, we convert rules in each device to a map from labels of disjoint difference normal form (ddNF) to output ports [11]. A ddNF is constructed as a directed acyclic graph, a four-tuple $< N, E, l, root >$ where N are nodes, $E \in N \times N$ are edges, l is a labeling function which maps each node to a bit-vector, and root r is a root node $l(root) = \star\star\star^k$. For two nodes $n, m \in N$, if $E(n, m)$, then $l(m) \subset l(n)$. Figure 3 shows an example ddNF. The top most node denotes the $\star\star\star \backslash \{\star 1\star, 1\star\star\}$, the left-most node $\star\star\star \backslash 11\star$, and the bottom node denotes $11\star$. Now, with ddNF, the same process can be used to compute reachability, but this time we use lists of labels instead of wildcard matching and intersection.

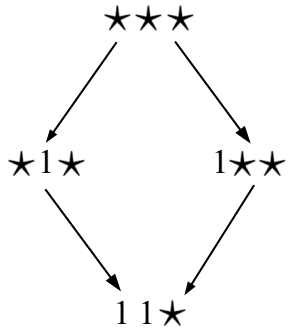

FIGURE 3. An example ddNF

The computation of the label-to-port map in Algorithm 1 follows algorithms in [11]. Whereas, for the test packets generation, we also provide how to construct the port-to-label map and the rule-index-to-label map for the rules in each device. The inputs are *Devices* that refer to the rules of the devices in the network. First, we compute a map $rv2node$ from bit-vectors to labels, and a map $node2rv$ from labels to bit-vectors by constructing a ddNF (line 2). Then, to extract labels of ddNF for each rule, we subtract previously labels by computing $DC(rv2node[rv]) \backslash seen$, where *seen* are the nodes that have been used (lines 3-8). $S[p]$ refers to the map from ports to labels and $H[index(rv, p)]$ is a map from the rule index to labels (line 9).

Algorithm 1 Constructing maps for rules in devices

```
1: procedure ConstructMaps(Devices)
2:     rv2node, node2rv ← constructDdnf(Devices)
3:     for R ∈ Device do
4:         for < rv, p >∈ R do
5:             labels ← DC(rv2node[rv])\seen
6:             for l ∈ labels do
7:                 R'[l] ← p
8:             end for
9:             S[p], H[index(rv, p)] ← labels
10:            seen ← seen ∪ DC(rv2node[rv])
11:        end for
12:    end for
13:    return S, H
14: end procedure
```

3.1.2. *Constructing reachability-to-rules table.* The inputs of Algorithm 2 are the output maps from Algorithm 1 and topology. For each port, we compute the reachable packets from it to all other ports in the network by the *travelNet* procedure (lines 3-5). Label la corresponds to the set of all packets at p, and ru refers to the exercised rules (line 4). In *travelNet*, when packets arrive at a port, the device *dev* that contains the input port is applied to these packets by *devForw*, producing a list of new packets (lines 11-12). If packets reach the edge, a corresponding entry is recorded in reachability-to-rules table (lines 14-15). Otherwise, we invoke the device containing the new port (line 16). We will repeat the previous process until the packet is no longer able to reach any other ports. In *devForw* procedure, for each port that is not equal to inport *inp* on the device, we intersect labels la with the labels of that port $S[p]$ (line 23). For each rule in the rules which guard that port, if its corresponding labels are intersected with la, the rule is exercised by the packets. Then we can get the history rules exercised along the forwarding path (lines 24-28). When the *travelNet* procedure completes, we can get the reachability-to-rules *table* (line 6).

Algorithm 2 Generating test packets

```
1: procedure GeneratePROBES(S, H, topo)
2:     table ← φ
3:     for protp ∈ ports do
4:         travelNet(initVar (p, la, ru), topo)
5:     end for
6:     tps ← minSetCover(table)
7:     return table, tps
8: end procedure
9: procedure TravelNET(in, topo)
10:    for inf[i] ∈ inf do
11:        dev = findDev (in[i].p, topo)
12:        t ← devForw(dev, in[i].p, in[i].la, in[i].ru)
13:        for t[i] ∈ t do
14:            if t[i].p ∈ edgeports then
15:                table ← addEntry(in[i].p, t[i].p, t[i])
16:            else travelNet(t, topo)
17:            end if
18:        end for
19:    end for
20: end procedure
21: procedure devForw(dev, inp, la, ru)
22:    for port p ∈ getPorts(dev) do
23:        la ← la ∩ S[p]
24:        for r ∈ getRules (p) do
25:            if H[index(r)] ∩ la ≠ φ then
26:                ru ← update(ru, r)
27:            end if
28:        end for
29:        tmp[p] ← record(p, la, ru)
30:    end for
31:    return tmp
32: end procedure
```

3.1.3. *Selecting test packets.* For each entry, a packet with the header (i.e., the 6th column) can exercise all links and rules in the entry. A rule may appear in more than one entry. We select a minimum of test packets *tps* to exercise all rules (line 6). Test packets to cover all forwarding links can be similarly computed. The algorithm follows the greedy Min-Set-Cover algorithm. The greedy algorithm guarantees to find a cover which is at most a logarithmic factor $1 + \ln(n)$ (n is the number of entries) larger than the optimal solution [12].

3.2. **Test packets update.** NetGen provides an incremental test packets update approach to make sure test packets can still cover all links and rules including added, removed and modified ones.

3.2.1. *Updating reachability-to-rules table.* In Algorithm 3, the inputs are the original reachability-to-rules *table*, and the original *ddNF*, the rules in *Devices*′, and the changed topology *topo*′. First, we update the labels of ddNF to reflect the changed rules (line 2). Then, we recompute the label-to-port map $S'[p]$ with *ddNF*′ (line 3). To update the table, we compute the ports p^t which are tagged different labels in label-to-port maps (line 4). If the changes in new network cover link changes, the reachability-to-rules table should be recomputed with the new *ddNF*′ (line 11). Otherwise, we identify the changed entries in the reachability-to-rules table. For each entry, if any port along the path (i.e., the 5th column) is in ports p^t, we recompute the entry to reflect the changes (lines 5-10).

Algorithm 3 Updating test packets

```
1: procedure UpdatePRobeS(table, ddNF, Devices′, topo′)
2:     ddNF′ ← updateDdnf(ddNF, Devices′)
3:     S′[p] ← constructMaps (ddNF′)
4:     p^t ← diffPorts (S[p], S′[p])
5:     if changeType(topo′) = ruletype then
6:         for e ∈ getEntry(table) do
7:             if e.path ∩ p^t ≠ φ then
8:                 table′ ← update(table, S′[p])
9:             end if
10:        end for
11:    else table′ ← consTable(ddNF′)
12:    end if
13:    e^r, e^a ← diff (table, table′)
14:    for e ∈ e^r, e^a do
15:        tps ← update (tps, e)
16:        r^m, l^m ← addUncover (tps, e)
17:    end for
18:    table′ ← prune(table′)
19:    return tps ∪ minSetCover(table′)
20: end procedure
```

3.2.2. *Updating test packets.* Not all packets in pervious test packets *tps* are affected by the updated entries. Therefore, we identify the rules and links which cannot be exercised by test packets *tps*. First, we update *tps* by removing test packets which correspond to the removed entries[2] e^r (line 14). The removed test packets may result in that some rules r^m and links l^m cannot be exercised now. The added entries e^a are similarly handled (lines 15-17). The next step is to select test packets to cover the rules and links which cannot be covered by *tps*. Because not every entry in *table*′ will contribute to the updated

[2]A modified entry can be considered as the removal of old entry and addition of a new entry.

test packets, we prune $table'$ by removing entries whose rules (the 5th column) and links (the 4th column) are not in r^m and l^m (line 18). And the final test packets are the union of pruned test packets tps and a minimum of updated test packets tps^t (lines 19). The problem is converted to generate test packets for uncovered rules and links. It is solved by the greedy Min-Set-Cover algorithm. Therefore, the test packets update algorithm provides near-optimality solution with complete coverage.

4. **Implementation and Evaluation.** We construct the test packets generation engine and test packets update engine in C language. Then, we implement the FIB parser, monitor and reporter. Pronto [9] also uses these components. All experiments are performed on a machine with 8GB of RAM and Intel Core CPU running at 3.40 GHz.

We apply NetGen to actual benchmark networks including Stanford [13] and Internet2 [14]. These networks contain more than 700,000 and 120,000 forwarding rules, respectively. We compare our tool with ATPG [7] and Pronto [9], as they also generate test packets to cover forwarding rules. In practice, not all ports in networks can be used for testing or debugging due to a variety of reasons like security concerns. First, we evaluate the rule coverage rates of our tool with available ports in the networks. Figure 4 provides the error-bar values of the rule coverage for different ports. The rule coverage rates of our tool increase with the value of ports. For most cases, the coverage values of NetGen are higher than those of ATPG [7] and Pronto [9].

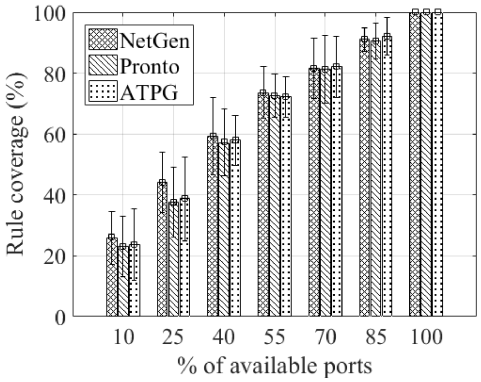

FIGURE 4. Rule coverage

We then evaluate the efficiency of NetGen. Table 2 shows the average test packets generation time of NetGen, ATPG [7] and Pronto [9] with 100% available ports. After 100 production runs, NetGen takes 1.9 seconds and 1.2 seconds in average to generate test packets for Stanford and Internet2 respectively. The generation time of NetGen is 6326 times and 25 times less than that of ATPG [7] and that of Pronto [9]. Finally, we evaluate how efficiently NetGen can deal with forwarding rules update. We do not compare our tool with ATPG [7], as it does not support test packets update. Figure 5 shows the test packets update time for forwarding rules addition and rules removal. For each of the rule addition and removal operations, we conduct 100 production runs on NetGen and Pronto [9]. For rule addition, NetGen takes less than 200 ms for 73.8% runs, and less than 230 ms for 93.6% runs. For Pronto [9], 77.4% of runs can be in less than 800 ms. For rules removal, NetGen takes less than 100 ms for 55.2% runs, and less than 200 ms for 89.7%

TABLE 2. Average test packets generation time for networks

Time (s)	ATPG [7]	Pronto [9]	NetGen
Internet2 [14]	11240.1	51.8	1.9
Stanford [13]	8086.4	29.3	1.2

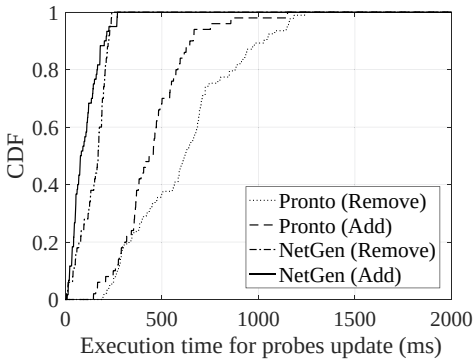

FIGURE 5. Update time

runs. By comparison, Pronto [9] takes less than 600 ms for 84.1% runs. NetGen is an order of magnitude faster than state-of-the-art tools.

5. **Conclusion.** This letter proposes a new test packet generation framework. It can generate the minimal test packets to cover all rules and links. In addition, it supports test packet update to deal with changed rules. Experiments on actual networks show its efficiency. In future, we will extend the work to support networks containing stateful functions, like firewall.

REFERENCES

[1] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan and T. Millstein, A general approach to network configuration analysis, *Proc. of the 12th USENIX Conference on Networked Systems Design and Implementation*, Oakland, CA, pp.469-483, 2015.

[2] A. Gember-Jacobson, A. Akella, R. Mahajan and H. H. Liu, Automatically repairing network control planes using an abstract representation, *Proc. of the 26th Symposium on Operating Systems Principles*, Shanghai, China, pp.359-373, 2017.

[3] E. Al-Shaer, W. Marrero, A. El-Atawy and K. Elbadawi, Network configuration in a box: Towards end-to-end verification of network reachability and security, *The 17th IEEE International Conference on Network Protocols*, pp.123-132, 2009.

[4] A. Horn, A. Kheradmand and M. R. Prasad, Delta-net: Real-time network verification using atoms, *The 14th USENIX Symposium on Networked Systems Design and Implementation*, Boston, MA, pp.735-749, 2017.

[5] P. Kazemian, G. Varghese and N. Mckeown, Header space analysis: Static checking for networks, *The 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

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

[7] H. Zeng, P. Kazemian, G. Varghese and N. Mckeown, Automatic test packet generation, *IEEE/ACM Trans. Networking*, vol.22, no.2, pp.554-566, 2014.

[8] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li and X. Chen, Is every flow on the right track?: Inspect SDN forwarding with RuleScope, *IEEE INFOCOM 2016 – The 35th Annual IEEE International Conference on Computer Communications*, San Francisco, CA, USA, 2016.

[9] Y. Zhao, H. Wang, X. Lin, T. Yu and C. Qian, Pronto: Efficient test packet generation for dynamic network data planes, *IEEE the 37th International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, GA, USA, 2017.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. Mckeown and S. Whyte, Real time network policy checking using header space analysis, *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, pp.99-112, 2013.

[11] N. Bjørner, G. Juniwal, R. Mahajan, S. A. Seshia and G. Varghese, ddNF: An efficient data structure for header spaces, *Haifa Verification Conference*, pp.49-64, 2016.

[12] S. Chakravarty and A. Shekhawat, Parallel and serial heuristics for the minimum set cover problem, *Journal of Supercomputing*, vol.5, no.4, pp.331-345, 1992.

[13] *Header Space Library*, http://bitbucket.org/peymank/hassel-public/.

[14] *Internet2*, https://www.internet2.edu/news/detail/1934/.