SDNProbe: Lightweight Fault Localization in the Error-Prone Environment

Yu-Ming Ke*, Hsu-Chun Hsiao*†, Tiffany Hyun-Jin Kim‡
*Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan
†, Research Center for IT Innovation, Academia Sinica, Taiwan
‡ HRL Laboratories, USA

***Abstract*—Probe-based fault localization identifies potential faulty nodes, which are manually inspected for confirmation. This work explores efficient and accurate fault localization, which is crucial for reducing manual effort without affecting network functionality. Prior work suffers from either high bandwidth overhead or false detection (i.e., incorrectly attributing good nodes or missing faulty nodes), especially in the presence of multiple or inconsistent faults. We propose SDNProbe, a lightweight SDN application that sends a provably minimized number of probe packets to pinpoint malfunctioning switches. We extend SDNProbe to randomize tested paths and packet headers to further improve detection accuracy. Using realistic topologies and flow rules, our evaluation results confirm that SDNProbe can rapidly localize faulty switches while reducing the number of required test packets by 30%, compared to prior approaches. Even with 50% of switches being faulty, Randomized SDNProbe can detect all faulty switches in 33 seconds, whereas prior approaches have false negative rates of 15-40%.**

I. Introduction

Active probing has been a widely adopted and effective approach to analyze end-to-end network performance. The most notable benefits of active probing (i.e., sending test packets) include simplicity of its approach while accurately depicting all available and cooperating nodes on a network at the time of the probe, and the capability to determine testing regions or time as the network administrators want.

However, since active probing has been mainly used for Quality of Service (QoS) and troubleshooting, the nature of the active probing mechanism hinders the identification of faulty nodes. For example, probing can identify a faulty node that persistently drops, modifies, or misdirects packets, since the probe issuer never receives the probe packets back. On the other hand, existing probe-based approaches are insufficient for tightly localizing advanced faulty nodes in error-prone environments, due to limited tested paths and simple localization mechanisms. For example, if two nodes are inseparable by any tested paths, it would be impossible to attribute a fault to either of them. In addition, faulty nodes can act non-persistently such that they drop, modify, or misdirect packets selectively. They can collude and deviate the normal traversing path such that additional, unauthorized nodes eavesdrop on the packets without being detected. Besides security limitations, another shortcoming with active probing is its invasiveness with the sheer volume of traffic. Straightforward approaches send one test packet per flow rule [12], [31], and despite the recent effort to minimize test packets [35] and to speed up test packet generation [38], they still incur high bandwidth overhead.

Given the sophistication spectrum of faulty nodes, we explore the extent to which simple active probing can reduce false detection rates while minimizing communication overhead. Our solution, SDNProbe, is a lightweight data-plane troubleshooting tool that significantly increases the accuracy of fault localization, more so than existing probe-based solutions, while curtailing communication overhead. Our insights to achieve these desired properties are:

- Prior work [35], [38] greedily solves the test packet minimization problem, which is modeled as a known NP-complete problem in general graphs, and thus obtains suboptimal results. However, we observe that this problem is solvable in polynomial time when there is no loop in the routing policy maintained by the controller, and loop detection can be done in polynomial time. Based on this insight, we enhance well-established graph algorithms to construct test packets in a lightweight manner that provably minimizes the number of test packets, such that the controller wastes no bandwidth.
- Prior work fails to accurately detect multiple or non-persistent faults in error-prone environments due to limited tested paths and simple localization mechanisms. To tightly localize such advanced faults, we propose applying SDN's programmability and randomization techniques such that the set of possible tested paths can sufficiently distinguish different failure patterns, eliminating the blind spots of the detection algorithm. As a proof-of-concept, we propose a randomized variant of SDNProbe.

We implement SDNProbe and its variant using realistic topologies and flow rules. Our evaluation results confirm that SDNProbe can rapidly localize faulty switches by sending the minimum number of test packets. More specifically, SDNProbe reduces the number of test packets required to localize all existing malicious switches by 30%, compared to existing approaches. Even when a network consists of 50% advanced faulty nodes, the randomized SDNProbe can detect all faulty nodes in 33 seconds, whereas prior work suffers from high false negative rates.

Contributions. We propose a probe-based fault localization mechanism that utilizes logically centralized control and programmability in SDN. Our main insight is efficiently eliminating disconnected paths (i.e., finding *legal* paths on which test packets can travel), and sending a minimum set of test packets that cover all flow entries. We prove that our approach finds a minimum set of test packets in $O(n^{2.5})$. Unlike existing work, we consider a strong, realistic switch failure model and propose a modification that randomizes the path selection to prevent faulty nodes from evading detection. We present evaluation results that show the reduced performance overhead while tightly localizing faulty switches.

II. Background

Software-Defined Networking (SDN) is a network architecture that decouples the control plane from the data plane [19]. This abstraction enables centralized control and programmability to enhance the manageability of traditional networks. A typical SDN comprises a *controller*, whose main task is to control the flow of network packets, and one or more *switches*, whose task is to forward packets based on rules built into its firmware. In other words, switches require no additional complexities such as maintaining routing tables as paths are centrally and dynamically maintained by the controller.

OpenFlow [29] is a widely-adopted communication protocol between the control plane and the data plane in SDN. In OpenFlow, a virtually-centralized controller regulates how packets are forwarded by installing and deleting *flow entries* on switches. The controller can send an OpenFlow-defined message to manage switches using a secure channel (i.e., TLS protected) and install flow entries on the switch. Note that a switch can have more than one *flow table* to store flow entries.

When a packet arrives at a switch, the switch checks the first flow table to find a matching flow entry. Each flow entry contains forwarding information, including a *match field*, *action*, and *priority*. If a packet matches multiple flow entries, the entry with the highest priority is selected. The switch then processes the packet according to the corresponding action(s): *output* to a specific port, *drop*, or *set-field* to modify the packet header. For example, when an incoming packet header matches a flow entry's match field, the packet will be forwarded to the specified output port. If a flow entry contains a set field, the incoming packet's header is overwritten with the corresponding value in the set field.

III. Problem Definition

Probe-based fault localization detects faulty switches by sending test packets, where a fault indicates a mismatch between the data plane's actual behavior and the control plane's network policies. The main technical challenges of probe-based fault localization are (1) efficiently computing a small set of test packets that exercise every possible fault pattern, and (2) accurately locating faulty switches based on the test packet results.

A. Desired Properties

1. **Tight localization boundary**: to reduce manual effort, the mechanism should tightly localize suspicious switches such that both false positives (i.e., labeling obedient switches as faulty) and false negatives (i.e., letting faulty switches evade detection) are low.
2. **Low bandwidth overhead**: the mechanism should locate faulty switches by sending as few test packets as possible to reduce bandwidth overhead.
3. **Fast detection**: the mechanism should quickly identify faulty switches to minimize the impact on normal packets.

B. Switch Failure Model

A switch is faulty if it contains one or more faulty flow entries that are executed incorrectly. Hence, a faulty switch may *misdirect* a packet to a switch that differs from the intended one (such that the packet may never reach the intended destination), *drop* a packet, and *modify* the packet header or content. Multiple faulty switches may exist simultaneously, but the number of faulty switches is unknown beforehand.

We consider two types of non-persistent faults: an *intermittent* fault selectively affects packets only during certain time periods, and a *targeting* fault selectively affects only certain IPs in a flow entry. For example, given a rule matching all packets with a destination IP in the subnet 10.10.0.0/16, a targeting fault may only affect the destination IP 10.10.1.1.

We further consider advanced failures that require colluding faulty switches to *detour* a packet such that it deviates from the testing path but eventually returns to the intended path [27]. Path detouring enables unintended nodes to eavesdrop, frame, or launch denial of service on switches on the dedicated path between the colluders. Furthermore, by detouring packets, colluding switches can bypass firewalls or intrusion detection systems that are hosted by benign switches between the colluding switches.

C. Limitations of Previous Approaches

High bandwidth overhead. One type of probe-based mechanisms identifies faulty switches by sending one test packet for each flow entry. Chi et al. [12] suggest periodically sending a test packet to a randomly-selected rule. However, sending one test packet per rule is inefficient and their simple fault localization mechanism suffers from false positives and false negatives. Monocle [31], [32] focuses on checking the correctness of newly-installed flow entries by sending one packet for each newly-installed rule on the monitored switch. However, in addition to the increased overhead on the network, this fails to consider the faults caused by unmonitored switches and thus may experience high false positives.

Some researchers have focused on reducing the number of test packets. For example, ATPG [35] and Pronto [38] generate test packets by reducing the problem of finding a minimum set of test packets to the problem of finding a minimum set cover (MSC). Since MSC is a known NP-complete problem, they both use the best-known greedy algorithm to approximately solve MSC. Pronto adopts a faster implementation than ATPG's. Chao et al. [11] propose minimizing the number of test packets when localizing malicious switches, but due to the omitted details, it is difficult to assess the effectiveness.

Inaccurate detection in error-prone environments. Running on traditional networks, ATPG [35] considers a switch to be faulty if it is at the intersection of two faulty host-to-host paths. However, ATPG fails to tighten the localization boundary and is unable to identify advanced failures. In general, narrowing down the suspected region is non-trivial, especially when there is more than one faulty node. Suppose the controller spots two suspected paths intersecting each other. Without further information, it would be hard to tell whether the switch at the intersection is faulty or there are two (or more) faulty switches on the paths. Pronto [38] aims to accelerate test packet generation by leveraging the concept of Atomic Predicate [34], but does not address how to perform fault localization.

IV. Intuition and Overview

SDNProbe efficiently and accurately detects faulty switches under a realistic switch failure model because of provably minimized test packet set and path randomization techniques.

Reducing packet count. Existing, per-rule approaches generate one test packet for each flow entry given a network topology, which wastes bandwidth with redundant test packets. Per-path approaches reduce the number of required test packets by generating those that can traverse multiple flow entries. Despite this improvement, one unsolved challenge of per-path approaches is how to efficiently compute a minimal set of test packets to traverse every flow entry. Prior approaches [35], [38] reduce this minimization problem to a minimum set cover problem and use greedy algorithms to approximately solve the known NP-complete problem. Instead of using approximation, SDNProbe's core intuition to handle such intractability is to restrict the test packet minimization problem on directed acyclic graphs (DAG). This is a reasonable assumption because a well-formed policy (flow entries) should contain no loop, and efficient polynomial-time algorithms and tools exist [24], [25] to verify whether flow entries cause a loop in the network. Specifically, SDNProbe minimizes the packet count by enhancing existing graph algorithms. We define *Minimum Legal Path Cover (MLPC)* and reduce the test packet minimization problem to the MLPC problem on a directed acyclic graph, which is provably solved in $O(n^{2.5})$ time using our algorithm.

Improving detection accuracy via randomization and SDN programmability. Traversing every flow entry is insufficient to accurately localize faulty switches, especially in the presence of advanced failures. For example, if two switches appear together on tested paths all the time, it would be impossible to attribute a fault when only one of them is faulty. In other words, to tightly localize faulty switches, it is important to ensure that the set of possible tested paths for each switch is large enough such that different subsets of switches can be effectively tested. Our core idea to enlarge the set of possible tested paths (and thus improve detection accuracy) is to leverage randomization and SDN's programmability. With programmability, a tested path can flexibly start and terminate at any node. Moreover, by randomizing tested paths as well as test packet headers, SDNProbe ensures that every possible tested path can be chosen with some probability. Hence, SDNProbe can detect advanced failures with a certain probability, which increases with the number of detection runs. We propose Randomized SDNProbe based on a randomized matching algorithm [16].

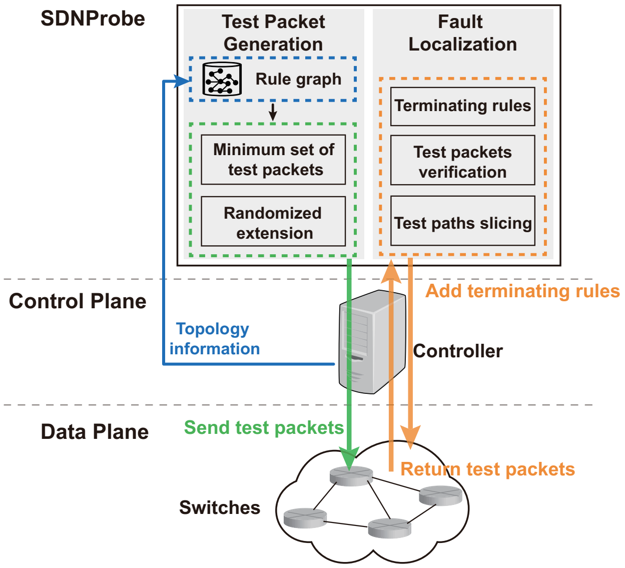


Fig. 1: Overview of SDNProbe.

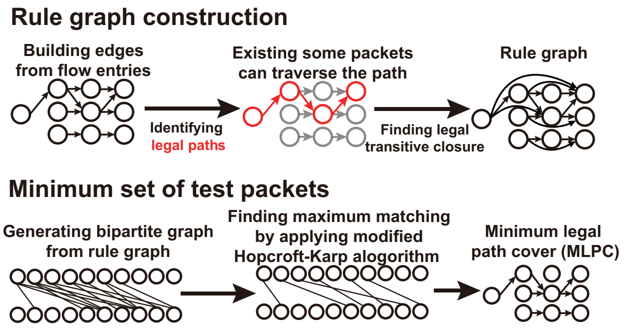

Rule graph construction
Building edges from flow entries
Identifying legal paths
Existing some packets can traverse the path
Finding legal transitive closure
Rule graph
Minimum set of test packets
Generating bipartite graph from rule graph
Finding maximum matching by applying modified Hopcroft-Karp alogorithm
Minimum legal path cover (MLPC)

Fig. 2: Flowchart of test packets generation.

A. SDNProbe Overview

As shown in Figure 1, SDNProbe detects misbehaving switches in two stages:

Test packet generation. To generate a minimum set of test packets that are guaranteed to traverse all flow entries on the network, SDNProbe defines a *rule graph*. This is a directed acyclic graph (DAG) that identifies *possible* flow directions among switches given the information of network topology and forwarding states. To further distinguish valid flows from invalid ones, SDNProbe analyzes which header can traverse a path by applying *transitive closure* to identify *legal paths* on which packets can traverse. Finally, SDNProbe generates a *bipartite graph* and applies a modified Hopcroft-Karp algorithm to construct a minimum set of test packets to traverse every rule on the network. Figure 2 illustrates a step by step construction of test packet generation.

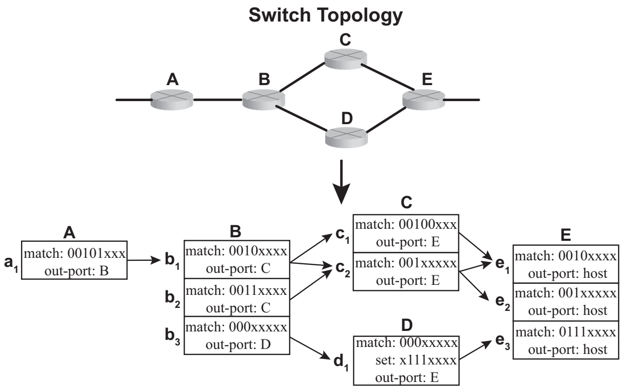


Fig. 3: In this sample switch topology, each boxed vertex is a flow entry, and flow entries in the same flow table are stacked together. The default value for the set field is `set:xxxxxxxx`, and the vertex at the top of the stack has the highest priority in each flow table.

Fault localization. Once a minimum set of test packets is determined, SDNProbe needs to insert a test flow entry at the end of each tested path such that the test packets can be returned to the controller. SDNProbe ensures that this is done without affecting normal packets.

After receiving returned test packets, if the controller notices suspicious activities on a path, including dropped, modified, or misdirected return packets, the controller tries to localize the misbehaving switch on the suspicious path by slicing it into two and repeating the localization process.

V. Test Packet Generation

We explain in detail how SDNProbe computes the minimum set of test packets using a rule graph, and how SDNProbe can be extended to randomize tested paths and headers for accuracy enhancement.

A. Rule Graph Construction

Given the switch topology and the flow entries on each switch, the controller constructs a graph, which we call a *rule graph*.[1] Edges of a rule graph represent the possible flow directions according to the routing policies in the control plane. The foundational concept of a rule graph is based on a plumbing graph [24], but since the plumbing graph is designed to verify incremental updates of routing policies, it resolves the dependency at a later time. Unlike the plumbing graph, we design a rule graph to resolve the dependency *at construction* to support efficient test packet generation.

Each vertex in the rule graph represents a flow entry in a switch and is labeled with four pieces of information, including *match field*, *set field*, *output port*, and *priority field*. An edge exists between two vertices in the rule graph if there are packets that can match the two flow entries. Figure 3 is an example of the rule graph.

Notations and terminologies. The packet header is represented as a bitstream in a $\{0, 1, x\}^L$ space [25], where L is the bit-length of the header and x represents a wildcard.

The k^{th} bit of a given header space H is denoted by $H[k]$, where $0 \leq k \leq L-1$. Given a vertex r on the rule graph, $r.m$, $r.s$, and $r.p$ represent the match field, set field, and the priority field of r, respectively. The default set field is a bitstream with all wildcards, which leaves the packet header unchanged.

Recall that among all the matched rules in a flow table, a packet is processed by the one with the highest priority. We say rule r_j is an *overlapping rule* of r_i ($r_j >_o r_i$) if r_i and r_j are in the same flow table, r_j has a higher priority than r_i ($r_j.p > r_i.p$), and there is at least one header that satisfies both of their match fields ($r_i.m \cap r_j.m \neq \emptyset$).

The *input* of a rule r ($r.in$) is the set of headers that can be processed by r. Hence, $r.in$ can be represented by its match field minus the overlapped header space: $r.in = r.m - \bigcup_{q >_o r} q.m$. Although computing a rule's input is proven to be NP-complete [9], in practice, we can obtain a header that satisfies the input using efficient SAT/SMT solvers, such as MiniSat [17] and Z3 [14].

The *output* of a vertex r ($r.out$) is the resulting header space after applying r's set field ($r.s$) to the input ($r.in$). Let $T(h, s)$ be a bitwise set-field operation such that the kth bit of $T(h, s)$ is $h[k]$ if $s[k]$ is a wildcard, and $s[k]$ otherwise; thus, $r.out = T(r.in, r.s)$. For example, in Figure 3, the input and output of rule d_1 are $000xxxxx$ and $0111xxxx$, respectively.

There are two steps to constructing a rule graph:

Step 1: Building edges. Each directed edge (r_i, r_j) on the rule graph indicates that there are some packets that can (1) trigger r_i, (2) be forwarded to the switch that maintains rule r_j, and (3) trigger r_j. Formally, there exists an edge (r_i, r_j) if and only if $r_i.port = r_j.switch$ and $r_i.out \cap r_j.in \neq \emptyset$. We check whether such an edge exists for each pair of rules on neighboring switches.

For example, edge (b_2, c_2) exists in Figure 3 because b_2's output port is C, and $0011xxxx \cap (001xxxxx - 00100xxx) \neq \emptyset$. However, there is no edge between rule c_1 and rule e_2 since $00100xxx \cap (001xxxxx - 0010xxxx) = \emptyset$. In this case, all packets with the header $00100xxx$ will match e_1, which has higher priority than e_2 in the flow table.

Step 2: Legal transitive closure. Before introducing the second step to construct a rule graph, we define a *legal path*.

Definition 1: A path $r_1, r_2, ..., r_n$ on the rule graph is a **legal path** if and only if there exists a packet that can traverse the path. That is, $O_n \neq \emptyset$ where $O_{i+1} = T(O_i \cap r_{i+1}.in, r_{i+1}.s)$ and $O_0 = \{x\}^L$.

In Figure 3, $a_1 \to b_1 \to c_2 \to e_1$ is a legal path since the packets with the header $00101xxx$ can go through this path.

To construct the rule graph, we apply *legal transitive closure* to the graph generated from Step 1, such that the resulting rule graph represents the reachability for each vertex. Specifically, given the graph $G_1 = (E_1, V_1)$ from Step 1, the rule graph $G = (E, V)$ is constructed such that $V = V_1$ and $E = E_1 \cup \{(u, v) | \exists$ legal path from u to v in $G_1\}$. We can conduct a breadth-first search on G_1 to obtain this rule graph.

Figure 4 shows the legal transitive closure of Figure 3. Because the path $b_2 \to c_2 \to e_2$ is a legal path (i.e., $0011xxxx \cap$

[1] The terms "flow entry" and "rule" are used interchangeably in this paper.

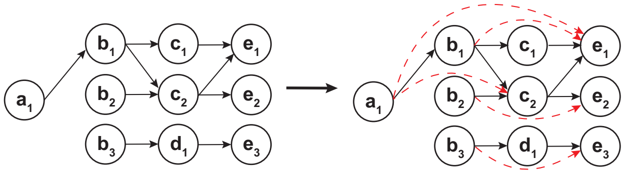


Fig. 4: The extra edges after applying legal transitive closure are shown in red on the rule graph in Figure 3.

$(001xxxxx - 00100xxx) \cap (001xxxxx - 0010xxxx) = 0011xxxx \neq \emptyset$), an edge ($b_2$, e_2) exists in the legal transitive closure (see Figure 4) to represent this legal path.

We assume that the controller's routing policy avoids routing loops, as they are generally undesirable and can be efficiently detected using static analysis [24], [25]. Hence, the constructed rule graph is a directed acyclic graph (DAG).

B. Generating a Minimum Set of Test Packets

Constructing the rule graph enables a controller to traverse every flow entry. To minimize the number of test packets, under the condition that the rule graph is a directed acyclic graph (DAG), we can reduce the problem to one of finding a *Minimum Legal Path Cover (MLPC)*, and we prove that finding a MLPC on DAG can be efficiently solved. Our MLPC problem and solution are inspired by the well-studied Minimum Path Cover (MPC) problem [8], [15], [23] with modifications to incorporate the concept of the header space. In the rest of this section, we define the Minimum Legal Path Cover (MLPC) problem, and then present an algorithm that can efficiently solve MLPC in DAG.

Minimum Path Cover (MPC). A path cover of a directed graph is a set of paths such that each vertex in the graph belongs to at least one path. Finding a minimum path cover of the rule graph may fail to generate feasible test packets because the edges in a rule graph only represent the pairwise relationships between flow entries. Hence, no packet may match any rule on a given path in the path cover. As depicted in Figure 3, the MPC contains three directed paths: (1) $a_1 \to b_1 \to c_1 \to e_1$, (2) $b_2 \to c_2 \to e_2$, and (3) $b_3 \to d_1 \to e_3$. The match fields of path (1) are $00101xxx$, $0010xxxx$, $00100xxx$, $0010xxxx$, respectively. However, no packet can go through this path, because $00101xxx \cap 0010xxxx \cap 00100xxx \cap 0010xxxx = \emptyset$.

Minimum Legal Path Cover (MLPC). To address the aforementioned limitation of MPC above, we define the *Minimum Legal Path Cover (MLPC)* problem.

Definition 2: **A legal path cover** of a directed graph is a set of legal paths such that every vertex belongs to at least one of the legal paths.

Given a directed graph, the Minimum Legal Path Cover (MLPC) problem is to find the minimum size legal path cover of the graph. With the above definition, generating the minimum set of test packets (P_{min}) is equivalent to solving the MLPC problem in the rule graph.

Algorithm 1 presents the steps to generate P_{min} by solving the MLPC problem on a DAG. Our proposed algorithm extends existing MPC solutions [8], [15], [23]. The proposed algorithm utilizes two graphs: (1) A rule graph G, as explained in Section V-A, to identify flow directions and legal paths; (2) A bipartite graph B of G to identify legal augmenting paths. We explain the construction of each graph and how B and G are used to find the minimum number of legal paths in Figure 3. When the reachability in graphs is defined over legal paths, the main technical challenges are: efficiently validating a legal path without increasing time complexity and ensuring the correctness of the algorithm.

Algorithm 1: Test Packet Generation based on MLPC

```
Input  : Rule graph $G$.
Output: Minimum set of test packets $P_{min}$.
$B \longleftarrow$ bipartite_graph($G$);
$C_{min} \longleftarrow$ Hopcroft-Karp($B$) w/ legal augmenting paths;
$P_{min} \longleftarrow$ construct headers from $C_{min}$;
return $P_{min}$;
```

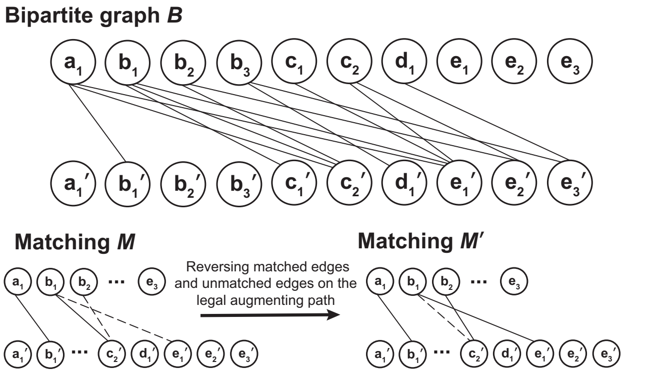


Fig. 5: Bipartite graph B of G in Fig. 4. The solid and dotted edges in M and M' represent matched and unmatched edges, respectively. (We omit other unmatched edges in M and M'.) A legal augmenting path is $b_2 \to {c_2}' \to b_1 \to {e_1}'$, where $(b_2, {c_2}')$ and $(b_1, {e_1}')$ are unmatched and $(b_1, {c_2}')$ is a matched edge in M.

Step 1: Bipartite graph. To identify legal augmenting paths, we generate a bipartite graph B using G, as illustrated in Figure 4. If G has vertices $r_1, r_2, ..., r_n$, we generate two disjoint sets of vertices, $r_1, r_2, ..., r_n$ and ${r_1}', {r_2}', ..., {r_n}'$, for the bipartite graph B, where $r_i = {r_i}'$, and convert a directed edge (r_i, r_j) in G into an undirected edge $(r_i, {r_j}')$ in B. Then, we can apply a modified Hopcroft-Karp algorithm [23] to find the maximum matching in B and transform it to MLPC to generate P_{min}.

Step 2: Modified Hopcroft-Karp algorithm. A *matching* of a bipartite graph can be obtained by finding an augmenting path, one that starts and ends with an unmatched vertex and the edges belonging alternatively to the matched and unmatched [13]. If there is an augmenting path p on a matching M, we can obtain a new matching M' by finding the symmetric difference of M and p (reversing the matched and unmatched edges on p), and $|M'| = |M| + 1$. Hopcroft-Karp algorithm is based on Berge's Theorem [8], which states that a matching is maximum if and only if no augmenting path exists on this matching. However, using this maximum

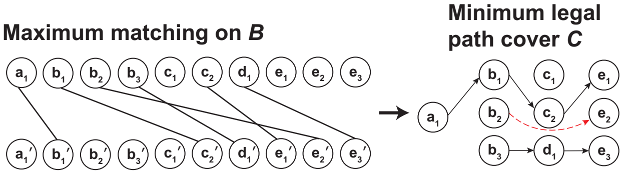


Fig. 6: Maximum matching with legal augmenting path transforms to MLPC.

matching algorithm directly may construct test packets that cannot traverse every rule on the network. Therefore, we apply the Hopcroft-Karp algorithm with a *legal augmenting path* on MLPC.

Definition 3: Given a bipartite graph B transformed from a rule graph G, an augmenting path p on B is **legal augmenting** if and only if the unmatched edges on p can be connected with current matched ones on G to form a legal path.

In Figure 5, edges (a_1, b_1') and (b_1, c_2') are matched, shown as a solid line, and (b_1, e_1') and (b_2, c_2') are not matched, shown as a dotted line in M. Then, $b_2 \rightarrow c_2' \rightarrow b_1 \rightarrow e_1'$ is a legal augmenting path, because it is an augmenting path on B and $a_1 \rightarrow b_1 \rightarrow e_1$ and $b_2 \rightarrow c_2$ are legal paths on G. We can check the validity of a legal augmenting path on B in constant time by checking whether it corresponds to an edge in G.

Theorem 4: Given the bipartite graph B, which is transformed from a rule graph G, a legal path cover C is minimum if and only if no legal augmenting path exists on the matching M of B.

Theorem 5: Algorithm 1 can solve the minimum legal path problem in $O(n^{2.5})$ time.[2]

With Theorem 4, a legal path cover C is minimum if and only if no legal augmenting path on the corresponding matching of C can be extended.[2] Therefore, the modified Hopcroft-Karp algorithm with legal augmenting paths can find minimum legal path cover as illustrated in Figure 6. The minimum legal path cover C_{min} on the rule graph is $a_1 \rightarrow b_1 \rightarrow c_2 \rightarrow e_1$, $b_2 \rightarrow e_2$, $b_3 \rightarrow d_1 \rightarrow e_3$, c_1, respectively. $b_2 \rightarrow e_2$ can be further converted to $b_2 \rightarrow c_2 \rightarrow e_2$ on the rule graph.

Step 3: Construct headers. After finding the minimum legal path cover C_{min}, we can compute a header space $HS(\ell)$ for each legal path ℓ in C_{min} by intersecting the rule headers on the path. Header intersection can be efficiently computed. For example, the header space of the legal path $a_1 \rightarrow b_1 \rightarrow c_2 \rightarrow e_1$ in Figure 6 is $00101xxx \cap 0010xxxx \cap (001xxxxx - 00100xxx) \cap 0010xxxx = 00101xxx$. For each path in C_{min}, we select one packet from $HS(\ell)$ to form the minimum set of test packets, P_{min}.

C. Randomized SDNProbe

To localize basic failures defined in §III-B, SDNProbe as well as previous probe-based approaches [35], [38] generate test packets to traverse all flow entries. However, traversing all flow entries is inadequate to effectively localize advanced failures for existing probe-based approaches. Specifically, in prior probe-based approaches including SDNProbe, packets may be silently detoured when there is more than one faulty switch on the same tested path. If two faulty switches send packets to each other along a different path, such packet detouring will remain unnoticed since the test packets will still arrive at the same destination. Also, non-persistent failures will stay undetected with high probability. Suppose in Figure 3 switch E is faulty and the detection algorithm generates four tested paths $a_1 \rightarrow b_1 \rightarrow c_2 \rightarrow e_1$, $b_2 \rightarrow c_2 \rightarrow e_2$, $b_3 \rightarrow d_1 \rightarrow e_3$, c_1. The faulty switch E is detected if the fault only affects the packets with the header $00100xxx$, because the detection algorithm only selects test packets from the header space $00101xxx$ (which covers $a_1 \rightarrow b_1 \rightarrow c_2 \rightarrow e_1$). Hence, the header $00100xxx$ on E is a *blindspot* of tested paths.

Tested path randomization. The above advanced failures may occur because the test packet generation algorithm is static, and thus cannot test the two detouring switches separately if they happen to be on the same tested path at the first place.

As a proof-of-concept, we propose a randomized algorithm by substituting the modified Hopcroft-Karp Algorithm in Algorithm 1 with randomized matching [16] to find legal augmenting paths. This randomized algorithm generates different test packets and test paths every time, making it difficult for the adversary to bypass detection because it can no longer know the possible set of test packets and the location of switches is not always at the end of a test path.

Note that tested path randomization can be efficiently performed because it can reuse the same rule graph to compute different randomized instances.

Test packet header randomization. Since every packet with the header in $HS(\ell)$ can traverse ℓ, we sample one header from the header space as the test packet for this path. The sampling method can be flexibly chosen, either uniformly at random or based on the past traffic distribution (e.g., sFlow [6]). For each time period t, we collect the set of headers $h^t(\ell)$ from the switches on each path ℓ. As a result, we can randomly select one packet whose header is in $HS(\ell)$ and $h^t(\ell)$ for each path ℓ and time period t to generate a minimum set of test packets.

VI. Fault Localization

To verify the correctness of the data plane using the returned test packets from switches, SDNProbe requires installing an additional test flow entry at the terminal switch on each tested route, and the match field of the test flow entry is the same as the test packet. Note that this installation stage is performed before the test packets are sent.

To ensure that the normal packets remain unaffected, test packets and test flow entries should be unique enough such that the test flow entries can be matched only by the test packets. To achieve this requirement, for each tested path (i.e., a sequence of flow entries to be tested), SDNProbe selects a unique header u as follows: (1) u cannot match any flow entry on the on-path

[2] Due to space limitations, the proofs of Theorems 4 and 5 can be found in the full report [7].

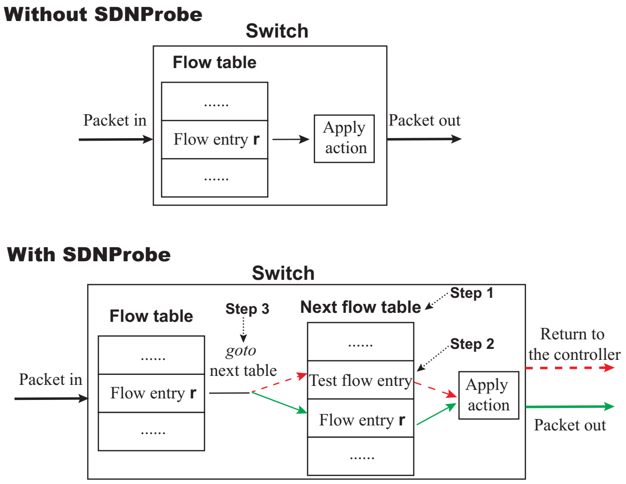


Fig. 7: Appending a test flow entry to verify the correctness of data plane.

switches except those to be tested. (2) u should differ from the values chosen by other test packets. In practice, we can use an SAT solver to find a unique header for a test packet with these constraints.

As shown in Figure 7, the modification in each terminal switch ensures that SDNProbe does not affect normal packets and normal switch operations, based on the following procedure: (1) duplicate the table and copy flow entry r; (2) insert the test flow entry with higher priority in the following flow table compared to the matched flow entry; (3) change the action of the flow entry r in the original flow table to `goto next table` action.

Duplicating the table entry (Step (1) above) is an important procedure for SDNProbe to cover all flow entries in the presence of failures. If we skip this procedure and directly insert the test flow entry on the same table, the last flow entry will remain untested because the test packet will match the test flow entry and be returned immediately.

Algorithm 2 shows the fault localization procedure. When the controller does not receive an expected test packet or receives a modified test packet, it marks the path as suspected and increases the *suspicion level* of all switches on this path. To determine which switch is likely to be faulty, the controller needs to narrow down the suspected path to one switch. Hence, the controller slices the suspected path to two sub-paths and recursively performs the same procedure. Once the suspicion level of a switch exceeds a certain detection threshold, the switch is considered faulty and requires further manual inspection. In addition, by tracking the suspicion level of each switch, a network administrator can make better decisions in choosing which switch to manually inspect first.

The use of suspicion level is needed for detecting an intermittent fault that reoccurs from time to time, each occurrence lasting for less than one detection round, such that SDNProbe may be unable to attribute the correct faulty switch in a round. Although the idea seems straightforward, we note that it can only work with fault localization schemes that have no or low false positives against persistent failures (such as SDNProbe).

Algorithm 2: Localizing faulty switches using test packets.

```
Input  : Set of test packets $S$.
Output: Faulty switch(es).
Test packet need to be sent $P \longleftarrow S$;
while detection scheme starts do
    send test packets $P$;
    wait and receive the set of test packets $P'$;
    for each test packet $p \in P$ and correspond $p' \in P'$ do
        $path \longleftarrow$ the path traversed by $p$;
        if $p \notin P'$ or $p \neq p'$ then
            increase suspicion level of rules on $p$;
            if $length\ of\ path > 1$ then
                slice_path($path$, $p$);
            else if $suspicion\ level\ of\ rule > threshold$ then
                identify this switch as faulty;
    $P \longleftarrow \emptyset$ ;
    $P$ add all $p$ where $p \notin P'$ or $p \neq p'$;
    if $P = \emptyset$ then
        $P \longleftarrow S$;
```

TABLE I: Comparisons of detection accuracy between fault localization techniques.

	SDNProbe	Randomized SDNProbe	Per-rule -based	Intersection -based
1 faulty node	✓	✓	✓	✓
> 1 faulty nodes	✓	✓	FP	FP
Intermittent fault	✓	✓	FN, FP	FN, FP
Targeting fault	FN	✓	FN, FP	FN, FP
Detour (colluding)	FN	✓	FN, FP	FN, FP

✓: Detectable, FN: False Negative, FP: False Positive

If a fault localization scheme is prone to high false positives even against persistent failures (such as per-rule and per-intersection techniques compared in Sections VII and VIII), using suspicion level will be ineffective.

VII. Detection Accuracy Analysis

We compare SDNProbe and Randomized SDNProbe with two common classes of fault localization (per-rule-based and intersection-based) used in prior work, with respect to the failure models defined in Section III-B. The analytical comparisons are aligned with the experiment results presented in Section VIII.

SDNProbe. SDNProbe can always detect basic failures on single or multiple faulty switches (i.e., no false negative) because misdirection, dropping and modification will alter the content or path of test packets, which is observable by SDNProbe. That is, SDNProbe's controller application can identify the occurrence of a basic fault along a tested path if the corresponding test packet returns unexpectedly or does not return. SDNProbe will never attribute a basic fault to a good node (i.e., a false positive) because SDNProbe effectively reduces the set of suspected switches by slicing the tested path, ensuring exact detection of a faulty switch. However, SDNProbe may fail to detect colluding nodes that happen to be on the same tested path and detour test packets between each other. Because SDNProbe keeps track of the suspicion level of each switch in fault localization, it can detect intermittent faults after a sufficient number of detection rounds. However, SDNProbe may fail to detect a targeting fault that does not overlap with any test packets.

Randomized SDNProbe. Similar to SDNProbe, Randomized SDNProbe ensures exact detection of basic failures on single or multiple switches. In addition, because Randomized SDNProbe randomly constructs tested paths for each detection round, the probability that two colluding nodes are on the same tested path in every round decreases exponentially. Randomized SDNProbe also randomly selects a test packet within the valid header space according to real-traffic distribution, and thus can increase the probability of detecting targeting faults over time.

Per-rule-based techniques. Per-rule-based fault localization [12], [31] checks a switch's flow rule by sending a test packet from the switch's previous hop to its next hop. However, when the test packet terminates unexpectedly, it is difficult for per-rule-based approaches to distinguish which of these three switches should be held accountable when there are multiple faulty switches in the network. Thus, false positives may occur for all but the single faulty switch case. Although per-rule approaches might also set a suspicion level for detecting intermittent faults, the high number of false positives will likely make every switch seem suspicious. Since per-rule approaches have a shorter tested path length (i.e., three hops), detouring is less likely but still possible. Targeting faults are likely undetected without using randomized test packet headers.

Intersection-based techniques. Intersection-based fault localization [35] considers a switch to be faulty if it is at the intersection of two host-to-host faulty paths. This technique can be applied to traditional networks where test packets can only be sent and received at the edges of the network. However, since there might be no alternative path to intersect with a suspicious switch, intersection-based approaches may fail to narrow down the set of faulty switches (i.e., false positives). In addition, if there is more than one faulty switch in the network, a benign switch at the intersection of two faulty paths might be wrongly blamed. Similar to per-rule approaches, intersection-based ones do not have proper mechanisms to detect intermittent and targeting faults.

VIII. Implementation and Evaluation

Implementation. We implemented SDNProbe and Randomized SDNProbe using approximately 3K lines of code [5] in Python and C++. Our implementation is based on the Ryu framework [1] in accordance with OpenFlow 1.3 specifications, and can be easily ported to other OpenFlow controller frameworks such as OpenDaylight [30].

In our implementation, the Ryu controller provides the topology information and flow entries on switches to the test packet generation component, which is implemented in C++ for optimized performance. Once obtaining the generated test packets and the associated tested paths, the controller sends them and perform fault localization.

To facilitate evaluation, our implementation can support the emulation of the data plane using Mininet [2], which can create multiple instances of Open vSwitch [3] as a virtual network. The data-plane emulation is useful for testing the functionalities of SDNProbe. SDNProbe comes with an easy-to-use interface and is available on GitHub.

With this implementation, we show that SDNProbe and Randomized SDNProbe achieve the desired properties defined in Section III.

Evaluation methodology. To evaluate probe-based fault localization schemes under a realistic network setting, we used a randomly-generated topology and flow entries that were synthesized based on real datasets. Specifically, we sampled the router-level topology from the Rocketfuel dataset [4] and incorporated flow entries obtained in a campus network. To generate additional flow entries for comprehensive evaluation, we also inserted flow entries to forward packets along paths computed by an all-pairs K-th shortest path algorithm [18].

The controller sends test packets at a rate of 250 KBytes per second, and the default threshold is set to be 3. In our experiments, attacks are simulated by modifying the flow entries. It is worth noting that SDNProbe is designed to detect inconsistencies between expected and actual data-plane behaviors, regardless of the cause of the inconsistencies.

Evaluation metrics. To examine the performance and feasibility of SDNProbe, we consider three metrics for evaluation:

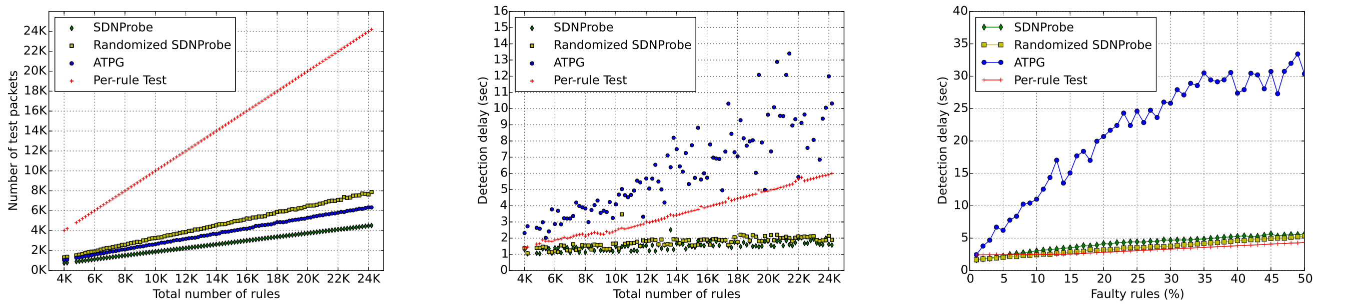
- **Number of test packets** reflects the communication overhead caused by the detection scheme.
- **Detection delay** consists of the time to generate and send test packets and to localize *all* faulty switches.
- **False positive rate (FPR) and false negative rate (FNR)** quantify the fraction of incorrectly detected good switches and the fraction of undetected faulty switches, respectively.

A. Real Dataset

To show that SDNProbe can be applied to a realistic setting, we first examined a real dataset which is a part of the backbone network topology in a campus network. SDNProbe correctly generated 600 test packets to cover 550 and 579 forwarding entries in two routing tables. The real dataset contained overlapping rules, and the maximum number of overlapping rules was 65. For each overlapping rule, we applied an efficient SAT solver (MiniSAT [17]) to find a matching header. The time it took to generate one header was between $0.5ms$–$2.4ms$, and was consistent throughout our experiments.

B. Comparison with Other Schemes

We compared the performance and effectiveness of SDNProbe and Randomized SDNProbe with ATPG [35] and Per-rule Tests [12], [31]. The experiments were performed on 100

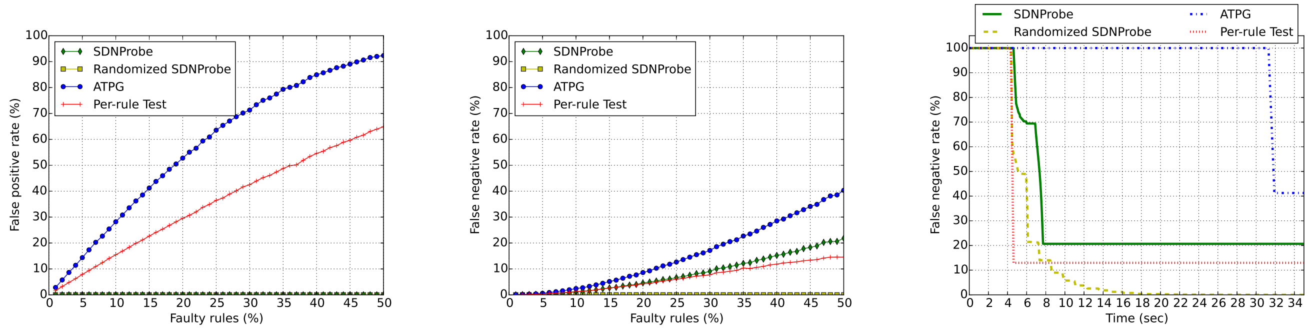


(a) Number of generated test packets.

(b) Detection delay for 1 faulty rule.

(c) Detection delay for varying faulty rule counts

Fig. 8: Performance evaluation of number of generated test packets and detection delay for SDNProbe, Randomized SDNProbe, ATPG, and Per-rule Test.



(a) Basic attack detection FPR.

(b) Path detouring attack detection FNR.

(c) Path detouring attack detection time vs. FNR

Fig. 9: False rate comparison for detecting basic and path detouring attacks. Each data point represents the average of 10 experiment runs.

topologies with varying number of flow entries. The results confirm that our schemes outperform other probe-based fault localization schemes as follows.

Number of test packets. As shown in Figure 8(a), SDNProbe generates the fewest test packets among the four schemes. The number of test packets generated by ATPG is relatively higher than SDNProbe because ATPG applies an approximation algorithm. Since the Per-rule Test sends one test packet for each flow entry, the number of test packets equals the number of flow entries. The improvement in the number of test packets depends on the size and structure of the topology. On average, SDNProbe reduces the number of test packets by 30% compared with ATPG. Randomized SDNProbe increases the number of test packets by 76% at most and 72% on average compared to SDNProbe.

Delay to localize one faulty switch. To investigate the detection delay in these four probe-based schemes, we randomly selected one flow entry to be faulty in each topology and measured the time it takes to identify the switch containing this flow entry. As Figure 8(b) shows, SDNProbe's detection delay is significantly lower than ATPG's and Per-rule Test's in every topology: SDNProbe's detection delay is 1–2.5 seconds, and Randomized SDNProbe's detection delay is 1–3.5 seconds. Since ATPG needs to compute additional test packets for fault localization, ATPG's detection delay is higher than SDNProbe's, taking at most 13.4 seconds (i.e., 5 times higher than the longest detection delay of SDNProbe). The detection delay of Per-rule Test is significantly higher because it needs to send a large number of test packets.

Our experimental results confirm that SDNProbe and Randomized SDNProbe can rapidly localize faulty switches regardless of the network scales.

Delay to localize multiple faulty switches. To examine the effectiveness of localizing multiple faulty switches, we randomly selected a fraction of flow entries to be faulty in a large-scale topology. As Figure 8(c) shows, Per-rule Test has the shortest detection delay when the rate is beyond 5%, since it does not require additional fault localization. However, as we show next, it suffers from high false positives. SDNProbe and Randomized SDNProbe are the fastest to localize faulty rules when the rate is 5% or lower, and remain competitively low when the rate is beyond 5%. ATPG performs the worst regardless of the percentage of faulty rules, since it must perform additional computation to generate each tested path during fault localization.

FPR and FNR. Figure 9(a) shows the FPR for detecting basic failures (i.e., packet misdirection, drop, and modification). As analyzed in Section VII, ATPG has a high FPR because it may incorrectly catch a benign switch that is located at the intersection of two faulty paths. Per-rule Test[3] also suffers

[3] To evaluate a target switch, Per-rule Test sends a packet to its previous-hop switch and expects to receive a return packet from its next-hop switch. However, it cannot distinguish which of the three switches is responsible for the malicious action.

TABLE II: Results of test packet generation.

Topo.	Setting (count)			Results				
	Rules	Switches	Links	MLPS	ALPS	NLPS	**TPC**	PCT (sec)
1	**4,764**	10	15	6	4.99	14,844	**954**	2.9
2	**33,637**	30	54	9	8.00	155,646	**4,203**	87.7
3	**82,740**	30	54	6	5.48	273,128	**15,098**	178.5
4	**205,713**	79	147	9	8.41	983,245	**24,456**	970.2
5	**358,675**	79	147	9	8.42	1,713,258	**42,590**	2,549.2

- MLPS: Maximum legal path length
- ALPS: Average legal path length
- NLPS: Total number of legal paths
- TPC: Test packet count
- PCT: Pre-computation time

from high a FPR because a switch may be falsely blamed for the misbehavior of its neighboring switches. Regarding FNR, all four approaches can detect all faulty switches (i.e., FNR = 0). Figure 9(b) shows the FNR when faulty switches attempt to detour packets. Randomized SDNProbe has a zero FNR because the probability that two faulty switches show up on the same tested path (and thus can remain undetected) in every test round drops to zero as time goes on. Per-rule Test's FNR is lower than SDNProbe's and ATPG's due to shorter tested paths, indicating a low chance for stealthy detours. Figure 9(c) shows the changes in FNR (y-axis) vs. detection delay (x-axis) against path detours. In this experiment, we assumed 50% of rules are faulty. In this scenario, only Randomized SDNProbe can detect *all* faulty switches (i.e., FNR = 0) in 33 seconds.

C. SDNProbe Sensitivity Test

To demonstrate its scalability, we applied SDNProbe in topologies of different scales. Table II summarizes the number of rules, switches, and links of each topology, and the results of test packet generation.

The pre-computation time includes the time to construct the rule graph, execute the MLPC algorithm, and construct test packets. Although rule graph construction is the most time-consuming step, it only needs to be computed once. In addition, SDNProbe can update the rule graph incrementally to reduce overhead. Due to space limitations, we provide details in our full report [7].

IX. Related Work

Most closely related work is reviewed in Section III-C. This section discusses additional related work in automated networking troubleshooting and path compliance enforcement.

Automated network troubleshooting. Systematically detecting network failures and misconfigurations is an active research field, given growing network scale and complexity [10]–[12], [20], [21], [24]–[26], [28], [31], [32], [35]. Heller et al. [22] propose a workflow for troubleshooting in the SDN. They show how to combine existing SDN diagnostic tools to pinpoint the origin of faults to one SDN layer. Our work can be classified as tools that check the consistency between the high-level policy and the actual behavior of forwarded packets.

NetSight [21] and its application ndb [20] modify every flow entry such that a switch can report to the controller the history of each received packet. However, they significantly increase overhead on both switches and controllers because a switch needs to return an extra packet to the controller for each forwarded one.

While we focus on checking the *actual behavior* against network policies, a complementary research area is to check *configurations* against network policies. HSA [25] checks the correctness of configurations in the data plane, and detects failures such as blackholes, loops, etc. VeriFlow [26] is a debugging tool which implements efficient algorithms and checks the network-wide invariants with very low latency by tracking rule updates. NetPlumber [24] is a network policy verification tool based on HSA and is similar to VeriFlow. NetPlumber checks policies and invariants when updating rules. Anteater [28] also diagnoses the network configurations and checks invariants on the data plane. It transforms the information of the data plane to Boolean expressions and models network invariants as Boolean satisfiability problems. Anteater uses a SAT solver to check if network policies are violated. NICE [10] uses model checking and symbolic execution to check the OpenFlow controller program.

Path compliance enforcement. A number of path verification protocols [27], [33], [36], [37] aim to enforce path compliance via cryptographic operations. For example, ShortMAC [37] secures the data plane by adding a small authenticator on each packet for each router on the forwarding path; DynaFL [36] records a sketch of all packets on a router and detects faulty routers by comparing the sketches within a neighborhood. In OPT [27], nodes on a path establish symmetric keys with the source and destination before sending packets. The source then inserts a validation field for each node on the path such that all nodes can authenticate the source and validate the path. SDNsec [33] uses symmetric-key cryptography to protect path integrity. It embeds forwarding information in the packet header, which can be verified by every node on the path. We leave it to future work to study whether and to what extend we can localize strong adversaries using probe-based techniques without cryptographic operations.

X. Conclusion

Probe-based fault localization is a practical and widely-deployed technique for identifying faulty nodes in networks. This paper explores the extent of fault complexity that can be identified using probe-based fault localization. SDNProbe and its randomized variant show that it is possible to achieve tight and even exact localization of faulty nodes under a wide range of advanced fault behaviors. Thus, SDNProbe can drastically reduce manual effort for network troubleshooting, with minimal bandwidth overhead.

Acknowledgment

This work was supported in part by the Ministry of Science and Technology of Taiwan under grants MOST 105-2221-E-002-146-MY2 and 107-2636-E-002-005- and by National Taiwan University under grant NTU-CCP-106R891204.

REFERENCES

[1] https://osrg.github.io/ryu/.

[2] http://mininet.org/.

[3] http://openvswitch.org/.

[4] "Rocketfuel," http://research.cs.washington.edu/networking/rocketfuel/.

[5] "SDNProbe: A Lightweight Tool for Securing SDN Data Plane with Active Probing," https://github.com/ymktw/SDNProbe.

[6] "sFlow," http://www.sflow.org.

[7] "SDNProbe: Lightweight Fault Localization in SDN with Active Probing (Full Report)," https://www.dropbox.com/sh/5jh956m6r35ojcn/AABpoihDIiSTMbxG7T8HAZTda?dl=0, Aug 2017.

[8] C. Berge, "Two theorems in graph theory," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 43, no. 9, p. 842, 1957.

[9] 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," in *IEEE INFOCOM*, 2016.

[10] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A NICE way to test OpenFlow applications," in *USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[11] T.-W. Chao, Y.-M. Ke, B.-H. Chen, J.-L. Chen, C. J. Hsieh, S.-C. Lee, and H.-C. Hsiao, "Securing data planes in software-defined networks," in *IEEE NetSoft Conference and Workshops (NetSoft)*, 2016.

[12] P.-W. Chi, C.-T. Kuo, J.-W. Guo, and C.-L. Lei, "How to detect a compromised sdn switch," in *IEEE NetSoft*, 2015.

[13] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[15] R. P. Dilworth, "A decomposition theorem for partially ordered sets," *Annals of Mathematics*, pp. 161–166, 1950.

[16] M. Dyer and A. Frieze, "Randomized greedy matching," *Random Structures & Algorithms*, vol. 2, no. 1, pp. 29–45, 1991.

[17] N. Eén and N. Sörensson, "An extensible sat-solver," in *International conference on theory and applications of satisfiability testing*, 2003.

[18] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[19] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, D. Meyer, and O. Koufopavlou, "Software-Defined Networking (SDN): Layers and Architecture Terminology," RFC 7426, Jan 2015.

[20] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *ACM HotSDN*, 2012.

[21] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[22] B. Heller, C. Scott, N. McKeown, S. Shenker, A. Wundsam, H. Zeng, S. Whitlock, V. Jeyakumar, N. Handigol, J. McCauley *et al.*, "Leveraging sdn layering to systematically troubleshoot networks," in *ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[23] J. E. Hopcroft and R. M. Karp, "An nˆ5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973.

[24] P. Kazemian, M. Change, and H. Zheng, "Real Time Network Policy Checking Using Header Space Analysis," in *USENIX NSDI*, 2013.

[25] P. Kazemian, G. Varghese, and N. McKeown, "Header Space Analysis: Static Checking for Networks," in *NSDI*, 2012.

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

[27] T. Kim, C. Basescu, L. Jia, S. B. Lee, Y.-C. Hu, and A. Perrig, "Lightweight Source Authentication and Path Validation," in *ACM SIGCOMM*, 2014.

[28] 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, p. 290, 2011.

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[30] J. Medved, R. Varga, A. Tkacik, and K. Gray, "Opendaylight: Towards a model-driven sdn controller architecture," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2014.

[31] P. Peresini, M. Kuzniar, and D. Kostic, "Monocle: Dynamic, fine-grained data plane monitoring," Tech. Rep., 2015.

[32] P. Perešíni, M. Kuzniar, and D. Kostić, "Rule-level data plane monitoring with monocle," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 595–596, 2015.

[33] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "SDNsec: Forwarding accountability for the SDN data plane," in *IEEE International Conference on Computer Communication and Networks (ICCCN)*, 2016.

[34] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 887–900, 2016.

[35] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic Test Packet Generation," *IEEE/ACM Transactions on Networking*, vol. 22, pp. 554–566, 2014.

[36] X. Zhang, C. Lan, and A. Perrig, "Secure and Scalable Fault Localization under Dynamic Traffic Patterns," in *IEEE S&P*, 2012.

[37] X. Zhang, Z. Zhou, H.-C. Hsiao, T. H.-J. Kim, A. Perrig, and P. Tague, "ShortMAC: Efficient Data-Plane Fault Localization," in *NDSS*, 2012.

[38] Y. Zhao, H. Wang, X. Lin, T. Yu, and C. Qian, "Pronto: Efficient test packet generation for dynamic network data planes," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.