Efficient Forwarding Anomaly Detection in Software-Defined Networks

Qi Li, *Senior Member, IEEE*, Yunpeng Liu, Zhuotao Liu, Peng Zhang, and Chunhui Pang

Abstract—Data centers, the critical infrastructure underpinning Cloud computing, often employ Software-Defined Networks (SDN) to manage cluster, wide-area and enterprise networks. As the network forwarding in SDN is dynamically programmed by controllers, it is crucial to ensure that the controller intent is correctly translated into underlying forwarding rules. Therefore, detecting and locating forwarding anomalies in SDN is a fundamental problem in production networks. Existing research proposals, roughly categorized into probing-based, packet piggybacking-based, and flow statistics analysis-based, either impose significant overhead or do not provide sufficient coverage for certain forwarding anomalies. In this article, we propose FADE , a controllable and passive measuring scheme to simultaneously deliver detection efficiency and accuracy. FADE first analyzes the entire network topology and flow rules, and then computes a minimal set of flows that can cover all forwarding rules. For each selected network flow, FADE decides the optimal number of monitoring positions on its path (much less than total number of hops), and installs dedicated rules to collect flow statistics. FADE controls the installation and expiration of these rules, along with unique flow labels, to guarantee the accuracy of collected statistics, based on which FADE algorithmically decides whether a forwarding anomaly is detected, and if so it further locates the anomaly. On top of FADE , we propose iFADE (a more scalable version of FADE) to further optimize the usage and deployment of dedicated measurement rules. iFADE achieves over 40 percent rule reduction compared with FADE . We implement a prototype of both FADE and iFADE in about 12 000 lines of code and evaluate the prototype extensively. The experiment results demonstrate (i) FADE and iFADE are accurate, e.g., they achieve over 95 percent true positive rate and 99 percent true negative rate in anomaly detection; (ii) FADE and iFADE are lightweight, e.g., they reduce the overhead of control messages compared with state-of-the-art by about 50 and 90 percent, respectively.

Index Terms—Software defined networking, cross-plane consistency check, forwarding anomaly

1 Introduction

Data centers are critical infrastructure underpinning the Cloud computing. Nowadays, production data centers often employ Software-Defined Networking (SDN) to manage both cluster networks [1], wide area networks [2], [3] and enterprise networks [4]. SDN adopts a new networking paradigm by separating the control plane from the data plane [5]. However, SDN itself does not ensure the flow rule consistency between what is intended in the control plane and what is actually programmed in the data plane, which may result in forwarding anomalies, i.e., packets are forwarded along wrong paths [6], [7], [8]. In production SDN networks, forwarding anomalies can be caused by hardware faults [8], [9], software bugs introduced by the routing agent when translating the controller intent into data-plane forwarding rules [10], and even attacks [6], [11].

To detect forwarding anomalies, our research community has proposed several categories of approaches, including network probing [12], [13], [14], [15], path piggybacking [16], [17], [18], [19], and flow statistics analysis [20], [21], [22], [23], [24]. However, these solutions either impose significant overhead or are not comprehensive enough to detect certain forwarding anomalies. For instance, path piggybacking approaches typically require non-trivial protocol changes and hardware capabilities that are not available on commodity switches; flow statistics based approaches incur significant control-plane overhead for ubiquitous statistics collection from all flows; and the probing-based approaches have a tradeoff between accurate detection and probing overhead (see more detailed analysis in Section 2). We summarize the major properties of these solutions in Table 1.

In this paper, we propose an efficient *F*orwarding *A*nomaly *D*etection archit*E*cture (FADE) to detect forwarding anomalies in SDN. The key insight of FADE is that flow rules matching the same network flow should have consistent view on the flow's statistics. By intelligently computing a small set of measurement rules, placing them optimally across the network, and accurately collecting and analyzing their statistics, FADE is able to detect and locate forwarding anomalies more efficiently than prior proposals. Towards this end, FADE is designed with a group of tightly coupled components: (i) a flow selection module that models the network into a forwarding graph and provably decides a

- *Qi Li, Yunpeng Liu, and Chunhui Pang are with the Institute for Network Sciences and Cyberspace and Beijing National Research Centre for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China. E-mail: qli01@tsinghua.edu.cn, {liuyp20, pch14}@mails.tsinghua.edu.cn.*
- *Zhuotao Liu is with the Institute for Network Sciences and Cyberspace and Beijing National Research Centre for Information Science and Technology (BNRist), Tsinghua University, Beijing 100084, China, with the Google Inc., CA 94043 USA, and also with the University of Illinois at Urbana-Champaign, IL 95051 USA. E-mail: zhuotaoliu@tsinghua.edu.cn.*
- *Peng Zhang is with the School of Computer Science, Xi'an Jiaotong University, Xi'an 710049, China. E-mail: p-zhang@xjtu.edu.cn.*

Manuscript received 23 June 2020; revised 13 Mar. 2021; accepted 14 Mar. 2021. Date of publication 26 Mar. 2021; date of current version 13 May 2021. (Corresponding author: Zhuotao Liu.) Recommended for acceptance by Y. Yang. Digital Object Identifier no. 10.1109/TPDS.2021.3068135

TABLE 1
Comparison With Existing Forwarding Anomaly Detection

Scheme	No New Packet Headers and Device Updates	Traffic-Interception Detection Approaches	Measurement Overhead
Networking probing [12], [13], [14], [15]	✓	×	Measure all rules and packet replay
Path piggybacking [16], [17], [18], [19]	×	✓	Hardware upgrade and protocol changes
Statistics analysis [20], [21], [22], [23], [24], [25], [26], [27]	✓	✓	Measure all rules and flows
FADE / iFADE	✓	✓	Measure a minimal set of probe rules

minimal set of flows that are able to cover all forwarding rules in the network; (ii) a probe selection module that can find the optimal number of measurement rules required for each selected network flow; (iii) a rule installation module that generates dedicated rules on selected probe positions and accurately collects their statistics by controlling the installation and expiration of these rules; and finally (iv) an anomaly detection module that analyzes the flow statistics to decide and locate anomalies.

On top of FADE , we further propose iFADE , a scalable version of FADE to support large scale deployment. The design goal of iFADE is to reduce the number of required flow rules while still achieving similar detection accuracy to FADE . iFADE is powered by two innovative designs. First, by analyzing the flow paths and forwarding actions on each hop, iFADE safely aggregates a set of flows that traverse the same sequence of switches and are forwarded by the same sequence of actions. As a result, iFADE only needs to install a single measurement rule on each switch (except for the ingress one) for the entire aggregate flow. Second, iFADE uses a detection scheduling scheme that can divide the entire detection task into multiple rounds so that iFADE is still deployable even if the available rule capacity on switches is much smaller than the number of required flow rules. We formulate the scheduling problem as an integer linear programming problem and propose heuristics to solve it with near-optimal effectiveness.

We implement a prototype of FADE and iFADE on our physical testbed in approximately 12,000 lines of Java code. We extensively evaluate both systems to report their detection accuracy / efficiency / robustness, control plane and data plane overhead, as well as the algorithm efficiency in a large-scale network topology. Overall, both systems achieve over 95 percent true positive detection rate and nearly 100 percent true negative rate, with small throughput overhead (less than 3 percent) in both control plane and data plane. Compared with prior work SPHINX [20] with similar detection accuracy, FADE and iFADE introduce much smaller detection overhead; for instance, to measure flow statistics, SPHINX generates 3x and 10x more OpenFlow messages than FADE and iFADE , respectively.

In summary, we make the following concrete contributions in this paper.

- We propose FADE , an effective and accurate anomaly detection architecture for SDN. By intelligently selecting a minimal set of network flows and optimally placing a set of measurement flow rules, FADE is able to achieve near optimal detection accuracy, while introducing very small overhead.
- On top of FADE , we propose iFADE , a more scalable version of FADE which not only greatly reduces measurement rule consumption, but also is deployable even if the available rule capacity on switches is much smaller than the total number of required measurement rules. iFADE improves scalability of FADE while retaining a comparable detection accuracy with FADE .
- We implement a prototype of FADE and iFADE in roughly 12,000 lines of Code (mostly in Java), and extensively evaluate their performance on our physical testbed. Overall, the experimental results demonstrate that FADE and iFADE have achieved their design goals. For instance, both FADE and iFADE achieve over 95 percent true positive rate for detecting forwarding anomalies in various topologies, and meanwhile iFADE reduces the number of required measurement rules by over 40 percent compared with FADE .

2 Related Work

Detecting and fixing forwarding anomalies has drawn significant attention from our research community. Prior proposals on addressing SDN forwarding anomalies can be divided into three major categories, i.e., sending probes, piggybacking path information in packets, and analyzing flow statistics.

Probing Based Anomaly Detection. Probing based anomaly detection approaches [12], [13], [14], [15] sample packets from network flows as probing packets, and then replay these packets to detect anomalies. They analyze flow rules in the control plane and then decide, for instance by solving mathematical problems [15], what probing packets are required to either traverse suspicious flow paths [14] or cover all network flows [12]. The probing packets need to be forwarded to the SDN controller on every hop via PacketIn messages. The primary challenge of such solutions is a trade-off between detection accuracy and controller overhead: although sending a large number of probes can improve anomaly detection coverage, those packets, in practice, could effectively result in Denial of Service to the SDN controller due to those excessive control messages, as well as reduced network throughput. Besides, some of these proposals [12], [13] check only the first and the last hop on network paths. Thus, they cannot detect traffic interception

forwarding anomalies where network flows are detoured from their desired paths but eventually return to the correct destinations.

Path Piggybacking Based Anomaly Detection. The path piggybacking approaches [16], [17], [18], [19] leverage customized packet headers to encode path information into packets, and verify the path when packets exit the network. The inserted path information includes cryptographic tags [17], compressed Message Authentication Codes [18] or encoded OpenFlow switch IDs [16]. Although theoretically sound, these approaches often require non-trivial protocol changes and hardware capabilities that are not available on commodity switches. Such solutions, essentially, share the same design principle of the various Internet security protocols that require new packet headers and switch / router firmware update, which have been proven to be impractical for real-world deployment [28].

Statistics Analysis Based Anomaly Detection. The statistics-based approaches [20], [21], [22], [23], [24], [25], [26], [27] discover forwarding anomalies by performing heavyweight flow statistics collection and analysis. Towards this end, they propose to collect various statistics, including OpenFlow flow statistics from all flow rules [20], port-based statistics [21], network counter rules [22], [23], [27], and packet trajectory data [24]. However, these solutions often impose significant communication overhead as they require to collect statistics from a vast majority of, if not all, flow rules. For instance, SPHINX [20] generates up to ten times of control messages than our scheme. Additionally, their collection and detection are *static* since they cannot evolve their methodology (such as dynamically adjusting statistics collection frequency and refining the scope of suspicious or benign flow rules) to agilely react to network dynamics.

Flow Measurement. Flow measurement is the foundation of networking engineering tasks, including anomaly detection, traffic engineering, capacity planning, fabric migration and so on. There are two approaches in traditional flow measurement: sampling and streaming. The sampling-based approaches [29], [30], [31] sample network packets and send them to remote servers for analysis or being used to maintain local flow statistics that would be consumed later by remote servers. Such methods provide only coarse-grained measurements, and therefore their accuracy is not satisfactory [32]. The streaming approaches [32], [33], [34] leverage specialized algorithms and data structures to store measurement results, e.g., flow counters. They achieve higher accuracy on specific metrics with few resources. In this paper, both of our schemes require collecting flow statistics accurately from a set of dedicated measurement rules. The key to realize that is to ensure that during their lifetime, these rules process the same set of network packets. Based on prior work on leveraging timers or triggers [35] to notify specific events to the control plane, it is possible to build a synchronous measurement system for our schemes.

3 Problem Statement

3.1 Background

SDN, conceptually, is a centralized networking architecture where the network forwarding (i.e., the data plane) is computed by a controller (i.e., the control plane). As a result, routing policies in SDN go beyond the typical shortest path routing. This benefit is truly realized in the era of Cloud computing when Cloud providers build numerous data centers across the globe. To hyperconnect these data centers (both connecting the computing clusters within a data center [1] and interconnecting different data centers [2]), data center networks are required to deliver extremely high bisection bandwidth. Towards this end, Cloud providers often build rich topologies in data center networks, such as Fat-tree [36], and then apply SDN to intelligently and dynamically program routing paths (e.g., perform traffic engineering) to fully utilize the available network paths.

However, SDN itself does not ensure that the intended paths by the controller are faithfully translated or programmed on switches, i.e., SDN networks may experience *forwarding anomalies* where packets are wrongly dropped or detoured from the paths intended by the controller. In production SDN networks such as B4 [2] and Jupiter [1], a gigantic software, conceptually a Routing Agent [10] or a control plane [37], is responsible for translating the desired forwarding intent given by the controllers to the actual forwarding rules on switches, and the translation is often performed on different levels (i.e., hierarchical), including the global level that controls the entire network and the domain level that controls a sector of the network. Because of its complexity, the Routing Agent is bug-prone. As a result, checking the consistency between the control-plane intent and data-plane realization to detect and rectify forwarding anomalies is critical in production SDN networks. In addition, prior works (such as [11], [38]) show that deliberate attacks may also result in forwarding anomalies in SDN networks.

3.1.1 An Introductory Example

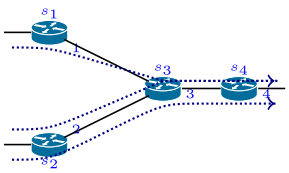
In this paper, a network *flow* (or flow) represents a stream of packets that are processed by the same sequence of switch flow rules while they traverse the network. The *correct rule path* of a flow means the sequence of rules intended by the controller, while the *actual rule path* of the flow is the actual sequence of rules matched by the flow in the data plane. In the rest of this paper, we use *rule path* to denote the correct rule path unless otherwise specified. The design goal of FADE is to detect forwarding anomalies where the actual rule paths are inconsistent with the correct rule paths.

Consider an introductory example in Fig. 1. Given the forwarding intent in Fig. 1a, we build *forwarding graph* (Fig. 1c) to represent the intent. A forwarding graph is a unidirectional graph that consists of sequences of flow rules matched by the same network flow, i.e., the forwarding graph is composed of all intended rule paths. For instance, in the forwarding graph, the correct rule path of the flow starting from s_1 and destined for $10.0.1.0/24$ should be $\{r_{11}, r_{31}, r_{41}\}$. If we assume that switch s_3 has an abnormal rule r_{33} installed (either due to software bugs or attacks) to drop the flow, the actual rule path of the flow will be $\{r_{11}, r_{33}\}$. Thus, network flow destined for $10.0.1.0/24$ experiences a forwarding anomaly because the intended rule path for the flow and its actual flow path are inconsistent.

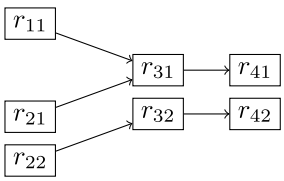
In this paper, we categorized forwarding anomalies into two major groups: *traffic hijacking* and *traffic interception*. In

rule	switch	match	action
r_{11}	s_1	ipv4_dst=10.0.1.0/24	output:1
r_{21}	s_2	ipv4_dst=10.0.1.0/24	output:2
r_{22}	s_2	ipv4_dst=10.0.2.0/24	output:2
r_{31}	s_3	ipv4_dst=10.0.1.0/24	output:3
r_{32}	s_3	ipv4_dst=10.0.2.0/24	output:3
r_{41}	s_4	ipv4_dst=10.0.1.0/24	output:4
r_{42}	s_4	ipv4_dst=10.0.2.0/24	output:4

(a) flow rules.



(b) network topology.



(c) Forwarding graph.

Fig. 1. The forwarding flow rules (a), the network topology (b), and the forwarding graph built for the network (c).

traffic hijacking, flows are dropped, or redirected to wrong rule paths, and they never return to the correct rule paths. In traffic interception, flows are first detoured to wrong rule paths, but finally return to the correct rule paths before exiting the network. FADE is designed to detect both types of anomalies. In reality, it is very rare for flows to experience the so-called "compound forwarding anomalies"; for instance, a flow is first detoured, then returned back to the correct the path, and finally dropped. Thus we ignore such cases in this paper and assume a flow experiences at most a single type of forwarding anomaly (as the definitions of two types of anomalies are mutually exclusive).

4 FADE Design

In this section, we present the detailed design of FADE , starting with its overview.

4.1 Design Overview

FADE is a *F*orwarding *A*nomaly *D*etection archit*E*cture designed particularly for SDN. As shown in Fig. 2, FADE is designed with four tightly coupled components. The flow selection module first organizes the rule paths of all flows into a forwarding graph based on the network topology and flow rules. Then, it selects a minimal set of flows that is provably able to cover all rule paths in the forwarding graph. For each selected flow, FADE algorithmically decides the optimal number of required measurement probes on the flow's rule path. Afterwards, FADE generates and installs dedicated measurement rules for those probes and starts to collect flow statistics. FADE ensures that all dedicated flow rules matched by the same flow process the same set of packets by synchronizing the installation and expiration of these rules. Finally, FADE collects flow statistics of these rules upon rule expiration, and verifies whether the collected statistics are consistent. If not, a forwarding anomaly is detected and FADE continues to localize the anomalous rules.

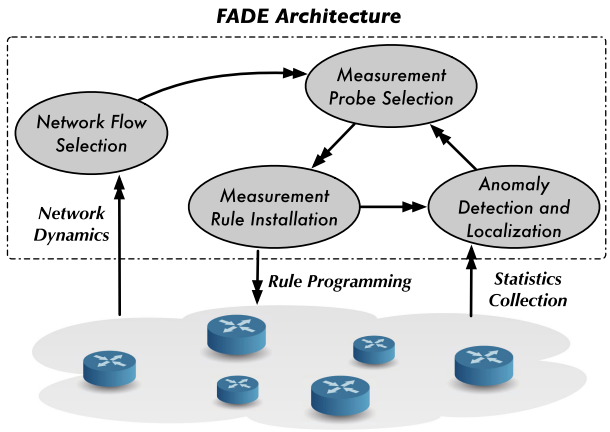


Fig. 2. The architecture of FADE . FADE contains four tightly coupled components: (i) a flow selection module that selects a minimal set of flows provably covering all forwarding rules in the network; (ii) a probe selection module that can find the optimal number of measurement rules required for each selected flow; (iii) a rule installation module that generates and programs these measurement rules, and finally (iv) an anomaly detection module that analyzes the flow statistics collected from these measurement rules to decide and locate anomalies.

4.2 Flow Selection

The goal of flow selection is to identify a minimal set of flows whose rule paths can cover those of all other flows such that FADE can uses this subset of flows instead of all network flows to validate packet forwarding. Flow selection is performed atop the forwarding graph. In order to construct the forwarding graph, FADE intercepts the flow rules intended by the controller and virtually computes the graph. To this end, we leverage HSA [39] to analyze the dependencies among different flow rules, and incrementally update forwarding graph whenever any rules are updated due to network dynamics.

Once the forwarding graph is constructed, we apply Algorithm 1 to select the minimal set of flows. At the highest level, the selection algorithm proceeds as follows. It starts from egress rules with out-degree of 0, representing the last-hop rules installed on the network borders (line 2-3 of Algorithm 1). For each egress rule, it performs *reversed* depth-first traversal (DFT) of the graph until reaching a rule that has an in-degree of 0, i.e., an ingress rule (line 5-17). We explain later why the algorithm performs DFT on a reversed graph. A rule may have multiple precedent rules in the forwarding graph, meaning multiple network flows may share this rule. Such rule paths are later handled by backtracking of the DFT (line 8). Once the DFT exits without exception, it also ensures that no loops are found in the forwarding graph. Otherwise, loop-based forwarding anomalies are detected and FADE prunes the looping rule paths and continues with a directed acyclic graph. Note that some network flows are unnecessary to be investigated if their rule paths are fully covered by those of other flows. For example, as shown in Fig. 1, a flow entering from switch s_1 and destinated to $10.0.1.0/24$ has rule path $\{r_{11}, r_{31}, r_{41}\}$, which is a superset of the rule path of a flow entering from switch s_3 and destinated to $10.0.1.0/24$. Thus, the latter flow is not necessary for further analysis. By constructing rule paths from egress rules, i.e., a reversed DFT, our algorithm guarantees to only select the rule paths that are not part of any other paths.

After graph traversal finishes, one sequence of rules from an egress rule to an ingress rule represents a *reversed* and 'longest' rule path matched by a network flow, which will be selected for subsequent forwarding verification. We formalize the flow selection logic in Algorithm 1.

Algorithm 1. Flow Selection Algorithm

Input: G $= (V, E)$: the forwarding graph
Output: result: saves selected flows and their rule paths
1 **begin**
2 result= $\emptyset$, R_e= G.GetEgressRules(), wq= $\emptyset$ /* wq stands for the *waiting queue* of the DFT. An item of wq consists of selected flows and its rule path./*
3 **forall** r_e **in** R_e **do**
/* start DFT from all egress rules/*
4 wq.add ({***new*** rulePath(r_e), ***new*** flow(r_e)}
5 **while** !wq.empty() **do**
/* search until all rules are covered/*
6 rp, f= wq.popFront()
7 tmp= rp.front() /* the first rule of a rule path/*
8 **while** (R_{pre}= G.getPreviousRules(tmp)) $\neq \emptyset$ **do**
/* search all previous rules of a rule path/*
9 tmp= R_{pre}.get(0)
10 rp.pushFront(tmp) // extend the rule path
11 f= f $\cap$ **new** flow(tmp)
/* if there are multiple search choices, search the first one and temporarily store other choices into wq/*
12 **for** i= $1 \rightarrow R_{pre}$.size() **do**
13 newRp= copy(rp)
14 newRp.insert(0, R_{pre}.get(i))
15 newF= f$\cap$ **new** flow(R_{pre}.get(i))
16 wq.pushBack({newRp, newF})
17 result.insert({newRp, newF})
18 **return** result

Theorem 1. *The set of selected network flows by Algorithm 1 provably covers all rule paths in the forwarding graph.*

Proof. From the definition of the forwarding graph, it is clear that the graph may contain multiple connected components. Since the algorithm prunes any possible loops during traversal, each connected component eventually becomes a directed acyclic and fully connected graph (i.e., a DAG). Thus, to prove the above algorithm, it is sufficient to prove that the same condition holds within a DAG.

By performing depth-first traversal in a DAG starting from an egress rule and terminating at an ingress rule, the algorithm ensures it can find all *longest* reversed rule paths in the DAG. For any network flow $\mathcal{F}$ in the DAG, its rule path must start with an ingress rule and terminate at an egress rule. Thus, its rule path must be a subset of one of the longest reversed rule paths returned by the algorithm. Therefore, if we denote the network flow matched by the longest rule path as $\mathcal{F}^*$, then $\mathcal{F}^*$ guarantees to cover the rule path of $\mathcal{F}$. In other words, only selecting the network flows matched by the longest rule paths is sufficient to cover all possible rule paths in the DAG. □

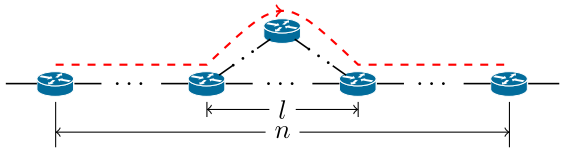


Fig. 3. Optimizing the number of probes.

4.3 Probe Selection

For all selected network flows by Algorithm 1, FADE invents *a controllable and passive probing scheme* to collect their flow statistics. In particular, given a selected network flow, FADE chooses a set of rules on its rule path as the probing spots. The selected rules are referred to as *probes*. For each probe, FADE generates a dedicated rule to *overwrite* it. By overwriting, we mean the dedicated rule has the same action with the probe, but with higher priority. Thus, these dedicated rules do not change the network forwarding behaviors, i.e., they are passive rules. However, FADE can configure and update the installation and timeout of these dedicated rules for the sake of statistics collection. We elaborate on how to select probes in this subsection and discuss the generation of dedicated flow rules in the following subsection.

Ideally, two probes, installed on the first and last hop of the rule path, are required to measure a flow. Unfortunately, this approach is not sufficient for traffic interception anomaly, which detours the network flows intermediately without changing their sources and destinations. Thus, extra probes are required in FADE . To figure out the best number of probes needed to maximize the detection probability, we tackle the probe selection problem via a probabilistic model. Recall that we assume that a flow rule incurs only one type of forwarding anomaly. Consider the case where a rule path has n rules (hops) and we select k probes from the rule path, as shown in Fig. 3. Note that the first rule and the last rule must be selected as probes. Therefore, the remaining $k-2$ probes should be chosen from the $n-2$ intermediate rules. A traffic hijacking anomaly can be detected if the anomalous rule hijacking the traffic is not selected as a probe. Thus, the detection probability is $p_1(k) = \frac{\binom{n-3}{k-2}}{\binom{n-2}{k-2}} = \frac{n-k}{n-2}$. An interception anomaly can be detected if the anomalous rule detouring the traffic is not selected as a probe and meanwhile at least one probe exists on the intercepted path. Under the constraint that the anomalous rule is not selected, there are $\binom{n-3}{k-2}$ combinations, including $\binom{n-l-2}{k-2}$ combinations in which no probes on the intercepted path are selected. Therefore, the detection probability is $p_2(k,l) = \frac{\binom{n-3}{k-2}-\binom{n-l-2}{k-2}}{\binom{n-2}{k-2}} = \frac{n-k}{n-2} - \frac{(n-k)\ldots(n-k-l+1)}{(n-2)\ldots(n-l-1)}$, where l is the length of the intercepted path and $2 \leq l < n$.

For simplicity, we assume the interception and hijacking anomalies happen with an equal probability (although it is straightforward to consider a weighted linear combination of the two probabilities). Thus, the overall probability of finding a forwarding anomaly for a flow rule is the sum of the aforementioned two probabilities. Further, the probability of detecting traffic interception attack is also factored by

TABLE 2
The Optimal Number of Probes

Rule path length	3	[4, 8]	[9, 13]	[14, 21]	[22, 32]
Optimal k	2	3	4	5	6

the length of the hijacked path, which is unknown a priori. Thus, we enumerate all possible length of the hijacked paths, and compute overall probability as follows:

$$p(k) = p_1(k) + \sum_{l=2}^{n-1} \frac{p_2(k,l)}{(n-1-l)}, \quad (2 \le k \le n). \tag{1}$$

In practice, the length of a rule path is often less than 32 [40]. We list the optimal k for varying rule path lengths in Table 2.

4.4 Rule Generation

The rule generation stage controls the installation and expiration of the dedicated measurement rules for selected probes. We divide the dedicated rules into two categories, denoted by $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively. For each flow, the rules in $\mathcal{R}_1$ are responsible for matching packets and stamping a unique label onto the headers of the matched packets. Thus, $\mathcal{R}_1$ should be installed to overwrite the first probe, i.e., in an ingress point. Since the flow label assignment process is centralized, FADE guarantees its uniqueness across network flows. In practice, based on what fields are still available, FADE may use MPLS label, VLAN label, ToS field, or their combinations to carry the label (for instance, our prototype uses a combination of VLAN label and ToS field). Besides attaching a flow label, the action of $\mathcal{R}_1$ should copy its probe's action to not change the forwarding behaviors. In general, FADE only generates *one* $\mathcal{R}_1$ for each network flow, but it may generate multiple $\mathcal{R}_1$ rules if the network flow is originated from multiple sources. Rules in $\mathcal{R}_2$, installed to overwrite all subsequent probes, are responsible for counting packets with the label attached by $\mathcal{R}_1$. The actions of rules in $\mathcal{R}_2$ are also copied directly from the corresponding probes, except that the last rule in $\mathcal{R}_2$ should strip the label to ensure that packets are intact for subsequent forwarding. At the end of this subsection, we have a concrete example of the dedicated flow rules generated for the network shown in Fig. 1.

To achieve high detection accuracy, FADE needs to ensure that $\mathcal{R}_1$ and $\mathcal{R}_2$ process the same set of network packets. Towards this end, FADE controls the installation and expiration of these rules. Specifically, rules in $\mathcal{R}_2$ should be ready before $\mathcal{R}_1$ becomes effective; and after $\mathcal{R}_1$ expires, $\mathcal{R}_2$ should be alive long enough to handle the packets tagged by $\mathcal{R}_1$. Formally, let i_1 and i_2 be the installation times of $\mathcal{R}_1$ and $\mathcal{R}_2$, and t_1 and t_2 be their effective times. Let d_{max} denote the larger value of the following two latencies: (i) the maximum rule programming latency and (2) the maximum forwarding latency across the network. Then, within $[i_1, i_1 + d_{max}]$, $\mathcal{R}_1$ becomes effective, and within $[i_1 + t_1, i_1 + t_1 + d_{max}]$, $\mathcal{R}_1$ expires. We can compute the similar time windows for $\mathcal{R}_2$. To ensure the correctness of the aforementioned invariant, FADE should ensure (i) $\mathcal{R}_2$ is effective before $\mathcal{R}_1$ is programmed, i.e., $i_2 + d_{max} < i_1$; and (ii) $\mathcal{R}_2$ expires after the last packet tagged by $\mathcal{R}_1$ exits the network, i.e., $i_2 + t_2 > (i_1 + d_{max} + t_1) + d_{max}$, as illustrated in Fig. 4. These constraints can be simplified to Equations (2) and (3)

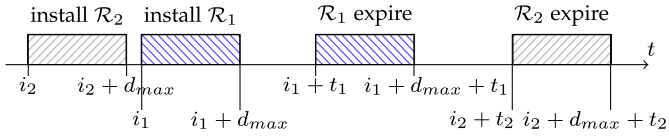


Fig. 4. Installation and hard timeout of dedicated measurement rules.

$$i_1 - i_2 \ge d_{max} \tag{2}$$

$$t_2 - t_1 \ge 3d_{max}. \tag{3}$$

Example. We wrap up this subsection with the generated flows for the network shown in Fig. 1. For flows entering from s_1 and destinated to $10.0.1.0/24$, its rule path is $\{r_{11}, r_{31}, r_{41}\}$. Then, FADE selects r_{11} and r_{41} as probes, and generates two dedicated rules. In the example, we use the ToS field to carry the label and assign label 4 to this flow. Also, we assume the duration of collecting flow statistics is 2s and d_{max} is 500ms. We list the dedicated rules in Table 3. Note that the installation time is relative. By controlling the installation time, we make sure that $\mathcal{R}_1$ rules take effect after $\mathcal{R}_2$ rules. From the table, it is clear that the first dedicated rule $\mathcal{R}_1$ matches packets that the corresponding probe could match and attaches a flow label to the packets. $\mathcal{R}_2$ only matches packets tagged by $\mathcal{R}_1$ and further strips the label (by resetting the ToS field).

4.5 Anomaly Identification and Location

FADE detects anomalies by analyzing flow statistics collected from these dedicated flow rules. Overall, the detection logic proceeds as follows. Consider a case where k probes are selected for a flow; the ith probe is on the n_ith switch (denote as s_{n_i}) on the forwarding path. For this flow, FADE generates $j+1$ $\mathcal{R}_1$ rules ($j \ge 0$) and $k-1$ $\mathcal{R}_2$ rules. We denote these rules as $m_{1,1}, m_{1,2}, \ldots, m_{1,j+1}, m_2, \ldots, m_k$. The sum of packet counters reported by $\mathcal{R}_1$ rules is p_1, and $m_i (i \ge 2)$ reports packet counter p_i. Thus, if $p_1 \ne p_k$, FADE detects a traffic hijacking anomaly. Further, the anomaly should be located between $s_{n_{u-1}}$ and s_{n_u} where u is the smallest number satisfying $p_u \ne p_1$ (line 4-5 of Algorithm 2). However, even if p_1 is equal to p_k, traffic interception may still occur. To verify that,

TABLE 3
The Generated Dedicated Rules for FADE

symbol	probe	switch	installation time	priority	hard timeout	match	action
$\mathcal{R}_1$	r_{11}	s_1	500ms	65535	1s	ip_tos=0,ip_dst=10.0.1.0/24	set_tos=4, output:1
$\mathcal{R}_2$	r_{41}	s_4	0ms	65535	3s	ip_tos=4,ip_dst=10.0.1.0/24	set_tos=0, output:4

Algorithm 2. Anomaly Identification Algorithm

Input: $m_{1,1}, \ldots, m_{1,j+1}, m_2, \ldots, m_k$: dedicated flow rules of the network flow
s_{n_i}, p_i: switch and packet count of rule m_i
rp: rule path of the flow
Output: the anomalous rule or the resubmit rule path

```
1  begin
2    SuspisiousPath= ∅, u = −1
3    u = min_i{i : p_i ≠ p_1, 1 ≤ i ≤ k}
4    if p_1 ≠ p_k then                          /* traffic hijacking/*
5      SuspisiousPath= [s_{n_{u−1}}, s_{n_u}]
6    else if u ≠ −1 then                        /* traffic interception,
     intercepted path: [s_{n_u}, s_{n_v}]/*
7      v = max_i{i : p_i = p_u, u ≤ i ≤ k}
8      SuspisiousPath= [s_{n_{u−1}}, s_{n_v}]
9    else if isResubmitted(rp) then
       /* find no anomaly in resubmitted rule path and reselect probes in the next detection round/*
10     resubmitRulePath(rp)
11     return
12   if length(SuspisiousPath) == 3 then
13     r_err = rp[n_u − 1]                      /* suspicious rule path is short enough, and the malicious rule is on the switch before s_{n_u}/*
14     return r_err
15   else
16     resubmitRulePath(SuspisiousPath)         /* suspicious rule path is not short enough, so resubmit to locate the anomaly/*
17     return
```

FADE checks whether there exists u and v ($2 \leq u \leq v \leq k-1$) such that p_i ($i \in [u, v]$) is different from p_j ($j \notin [u, v], j \in [1, k]$). If so, an interception anomaly is detected and the anomaly is located between $s_{n_{u-1}}$ and s_{n_v} (line 7). To further locate the precise anomalous rule, FADE iteratively processes the suspicious rule path as a new rule path (line 16) until the length of the suspicious path is reduced to be 3 (line 12-14). At this termination stage, FADE can conclude that the malicious rule is the middle hop of the rule path. We formalize the above logic in Algorithm 2.

5 iFADE Design

In this section, we further introduce a more scalable version of FADE , denoted as iFADE, which significantly reduces the number of required dedicated flow rules compared with FADE, while achieving almost the same detection accuracy. At a high level, FADE invents two mechanisms to reduce rule consumption and enhance the scalability. First, it aggregates a set of selected network flows if their rule paths traverse the same sequence of switches, and meanwhile the corresponding rules on the rule paths have the same actions. For instance, in Fig. 1, the two network flows entering switch s_2 and matched by two rule paths, {r_{21}, r_{31}, r_{41}} and {r_{22}, r_{32}, r_{42}}, satisfy the aforementioned conditions and therefore can be aggregated for anomaly detection. Such aggregation greatly saves the number of required dedicated rules since only one set of $\mathcal{R}_2$ rules is necessary for each aggregate flow. Second, since the $\mathcal{R}_1$ rules, installed on the ingress switches, cannot be safely aggregated, iFADE incorporates a round-based mechanism, i.e., splitting the detection for aggregate flows into multiple rounds and investigating only a part of aggregate flows in each round. By solving an Integer Linear Programming (ILP) problem, iFADE is able to minimize the maximum number of dedicated rules required on any switches.

Note that, the goal of iFADE is to solve the possible scalability issues in large-scale networks. By aggregating flow paths, iFADE trades flow table space usage with moderate time overhead introduced by more detection rounds. Therefore, iFADE should not be viewed as a substitute of FADE . It depends on the actual network scale and the requirements on detection latency to decide which method to use, and they can certainly be deployed simultaneously.

5.1 Rule Computation Under Flow Aggregation

Similar to FADE , iFADE also has four phases, i.e., flow selection, probe selection, rule generation, and anomaly identification. Except for the probe selection phase, iFADE has different designs in other phases due to flow aggregation. In the following subsections, we present the design details of iFADE.

5.1.1 Flow Selection

To guarantee coverage, iFADE also performs DFT on the forwarding graph for aggregate flow selection. However, the algorithm needs to be aware of physical locations of flow rules, i.e., whether some flow rules are installed on the same switch. In particular, the algorithm proceeds as follows. First, the algorithm groups all egress rules on the same switch with the same actions as one *rule category* (line 5-6 of Algorithm 3). Each rule category is perceived as a single virtual traversal runner which starts and drives an iteration of the DFT. During traversal, a rule category is split into multiple categories if the category's precedent rules have different actions, i.e., they cannot be grouped into the same category anymore (line 13). For any rule category, its DFT terminates when all its precedent rules contain only ingress rules (line 16-17). Once the DFT for the entire graph terminates, rules in the same category form a set of rule paths, each of which is associated with one network flow. These network flows are considered as a single aggregate flow. We formalize the traversal logic in Algorithm 3.

5.1.2 Dedicated Rule Generation

For each aggregate flow used for anomaly detection, iFADE needs to generate dedicated $\mathcal{R}_1$ rules, for matching each member flow, and a set of $\mathcal{R}_2$ rules, for matching all member flows. To this end, different rules in $\mathcal{R}_1$ need to attach the same packet label, i.e., all individual flows in the same aggregate flow should carry the same label. Again, since the label assignment is centralized controlled, iFADE can ensure the label uniqueness across different aggregate flows. Besides label tagging and striping, all dedicated rules have same actions as their corresponding probes to preserve the original networking forwarding.

Since flow aggregation only partially changes the contents of dedicated rules, it does not affect the installation and expiration of these rules. Thus, iFADE adopts the same mechanism for rule timing as FADE . We list the examples

Algorithm 3. Aggregate Flow Selection

```
Input: G: forwarding graph; S: switches in the network
Output: result: the rule paths of aggregate flows
1  begin
2    result = ∅  /* the final result set of aggregate rule paths/*
3    wq = ∅      /* temporary set of aggregate rule paths/*
4    forall s in S do
       /* get all egress rules on s and classify them/*
5      R_e = G.getEgressRules(s)
6      C_e = ClassifyByActionList(R_e)
7      forall c in C_e do
         /* initiate aggregated rule paths/*
8        wq.pushBack (new AggregatedPath(c))
9    while !wq.empty() do
10     arp = wq.popFront()      /* currently searched aggregate rule path/*
11     repeat
         /* repeat until there are no previous rules/*
12       pre = G.getPrevious(arp.front())
         /* classify previous rules of the aggregated rule path according the switches and actions/*
13       C = ClassifyBySwitchAndActionList(pre)
14       if C.size() == 0 then
15         result.add(arp)     /* find a new result/*
16       else if C.size() == 1 then
17         arp.pushFront(C.get(0))     /* extend the rule path of aggregated flow/*
18       else
           /* if there are multiple subsequent search choices, search the first one and temporarily store other choices into wq/*
19         for i = 1 → G.size() do
20           newArp = copy(arp)
21           newArp.pushFront(C.get(i))
22           wq.pushBack(newArp)
23         arp.pushFront(C.get(0))
24     until C.size() == 0
```

of generated rules in iFADE in Table 4. The first two rules are the $\mathcal{R}_1$ rules for the aggregate flow with two member flows. Both flows are assigned the same label (represented by the same ToS value), which is further matched by the single dedicated rule $\mathcal{R}_2$ on subsequent switches.

5.1.3 Anomaly Identification

The anomaly identification in iFADE needs to consider the flow aggregation. We use the following notations for describing the process. Consider an aggregate flow with l member flows, denoted as $f_1, f_2, \ldots, f_l$. $m_{1,i,j}$ denotes the jth dedicated rule in $\mathcal{R}_1$ generated for member flow f_i (recall that similar to FADE , iFADE may generate multiple $\mathcal{R}_1$ rules for one individual flow); p_m denotes the number of packets reported by the kth switch on path. Thus, in normal scenarios without forwarding anomalies, the following equation holds:

$$p_2 = p_3 = \cdots = p_k = \sum_i \sum_j p_{1,i,j}. \tag{4}$$

Denote $p_1 = \sum_i \sum_j p_{1,i,j}$, we have a simplified version of the equation as

$$p_2 = p_3 = \cdots = p_k = p_1. \tag{5}$$

The anomaly detection algorithm of iFADE works as follows. Starting from p_1, it checks whether the current counter is consistent with its subsequent one. If not, a suspicious sub-path is found. Since the sub-path carries an aggregate flow, to reuse the anomaly detection algorithm of FADE designed for single-flow use cases, we split the aggregate flow into smaller ones (i.e., sub-aggregate flows) and reevaluate flow statistics for each of them. It is feasible to split the aggregate flow since each member flow has its own $\mathcal{R}_1$ rules. If the statistics of a sub-aggregate flow is consistent, then it includes no anomalous member flows. Otherwise, we further split the suspicious sub-aggregate flow. The recursion terminates when a sub-aggregate flow only contains a single flow. We formalize the above logic in Algorithm 4.

Algorithm 4. Anomaly Identification Algorithm of iFADE

```
Input: arp: the aggregate rule path to be detected
Output: the anomalous rule
iFadeAnomalyIdent(arp)
begin
1    genAndInstallProbes(arp)
     /* check anomaly in the same way as Algorithm 2/*
2    if hasAnomaly(arp) then
3      if isSplitable(arp) then
4        subArp_1, subArp_2= split(arp)
5        genAndInstallProbes(subArp_1)
6        genAndInstallProbes (subArp_2)
7        result= iFadeAnomalyIdent(subArp_1)
8        result= result∪ iFadeAnomalyIdent(subArp_2)
9        return result
10     else
         /* if the aggregate flow contains only one flow, leverage the anomaly identification algorithm of FADE to locate the anomaly/*
11       result= FadeAnomalyIdent(arp)
12       return result
```

TABLE 4
Dedicated Rule Generation in iFADE

symbol	probe	switch	install time	priority	hard timeout	match	action
$\mathcal{R}_1$	r_{21}	s_2	500ms	65535	1s	ip_dst=10.0.1.0/24,ip_tos=0	set_tos=4,output:2
$\mathcal{R}_1$	r_{22}	s_2	500ms	65535	1s	ip_dst=10.0.2.0/24,ip_tos=0	set_tos=4,output:2
$\mathcal{R}_2$	r_{41}, r_{42}	s_4	0ms	65535	3s	ip_tos=4	set_tos=0,output:4

TABLE 5
Notations Used in the ILP Model

Symbol	Description
$\mathcal{A}$	The set of aggregate flows in the network.
α	One specific aggregate flow.
$\|\alpha\|$	The number of member flows in the aggregate flow α.
R	The number of rounds. R is dynamic.
r	One specific round.
$X^j_{\alpha r}$	$X^j_{\alpha r} = 1$ if a member flow j in α is being investigated in round r; otherwise $X^j_{\alpha r} = 0$.
$i_{s\alpha}$	$i_{s\alpha} = 1$ if switch s is the ingress point of aggregate flow α; otherwise $i_{s\alpha} = 0$.
$m_{s\alpha r}$	$m_{s\alpha r} = 1$ if any probe of the aggregate flow α is installed on switch s in round r; otherwise $m_{s\alpha r} = 0$
t_s	The available rule capacity on switch s.
u_{sr}	The number of dedicated rules installed on switch s in round r.
t_m	The maximum number of dedicated rules needed on every switch and in each round.

5.2 Scheduling Detection into Rounds

Since flow aggregation only saves $\mathcal{R}_2$ rule consumption, we may still encounter scalability issues for $\mathcal{R}_1$ rules in large-scale deployment. iFADE *divides and conquers* this issue by scheduling anomaly detection into different rounds such that only a subset of $\mathcal{R}_1$ rules are required in each round. The scheduling is not just random division. Instead, iFADE formalizes it as an Integer Linear Programming (ILP) problem to optimize the scheduling.

The ILP Model. The notations used in our ILP model are listed in Table 5. The optimization goal is to minimize the maximum number of rules required on every switch in each detection round. Formally, we formulate the problem as follows.

Equation (6) states the optimization goal. Equation (7) formulates the total dedicated rules, including $\mathcal{R}_1$ and $\mathcal{R}_2$ rules, required on a specific switch s. Constraints in Equations (8) and (9) indicate that the rule consumption on one switch cannot exceed the maximum reserved rule capacity (our optimization metric) and the available rule capacity on this switch. Equation (10) states each individual flow in α should be investigated in some round; together with Equation (10), Equation (11) indicates that all member flows in α are investigated eventually before the algorithm exits

$$\text{Minimize} \qquad t_m \tag{6}$$

$$\text{Subject to} \sum_{\alpha \in \mathcal{A}} i_{s\alpha} \sum_{j \in \alpha} X^j_{\alpha r} + \sum_{\alpha \in \mathcal{A}} m_{s\alpha r} = u_{sr}, \quad \forall r, \forall s \tag{7}$$

$$u_{sr} \leq t_s, \quad \forall r, \forall s \tag{8}$$

$$u_{sr} \leq t_m, \quad \forall r, \forall s \tag{9}$$

$$\sum_{r \in R} X^j_{\alpha r} = 1, \quad \forall j \in \alpha, \forall \alpha \tag{10}$$

$$\sum_{r \in R} \sum_{j \in \alpha} X^j_{\alpha r} = |\alpha|, \quad \forall \alpha. \tag{11}$$

Heuristics. Since the ILP problem is NP-hard, we develop heuristics to solve it. As $\mathcal{R}_1$ rule shortage tends to be the bottleneck, we sort all aggregate flows by their membership sizes. In each round, we schedule one or more non-investigated aggregate flows. Denote an aggregate flow under investigation as $\mathcal{F}_\mathcal{A}$. The scheduling for $\mathcal{F}_\mathcal{A}$ is feasible if the network has sufficient capacity to program all its required $\mathcal{R}_1$ and $\mathcal{R}_2$ rules. If the scheduling is infeasible due to $\mathcal{R}_1$ rule limitation, we further split $\mathcal{F}_\mathcal{A}$ into two sub-aggregate flows. We greedily include as many network flows as possible into the first sub-aggregate flow so that it is detectable in this round. Note that the greedy allocation has reservation: on each switch, while computing how many $\mathcal{R}_1$ rules can be installed, we reserve $\mathcal{K}$ rules for installing $\mathcal{R}_2$ such that we may schedule some other aggregate flows in this round. Our heuristic chooses $\mathcal{K} = 2$, considering that we have the chance of allocating two aggregate flows in one round. This value is adjustable. The second sub-aggregate flow, containing the rest flows, is deferred to some subsequent round, depending on its membership size.

Algorithm 5. Heuristic for Solving the ILP Problem

Input: the set of aggregate flows A; the number of dedicated rules t_s that could be installed on switch s.
Output: how many single flows in each aggregate flow should be detected in each run.

```
1  begin
2    result = {}
3    reserve = 2 * |S|
4    forall F_A in SortBySize (A) do
5      round= 1
6      while notEmpty(F_A) do
             /* iterate until all flows in F_A are scheduled/*
7        availableOnIngress = getAvailTCAM(r, getIngress
         (F_A))
8        if R2Installable(r, F_A) and
         availableOnIngress > reserve then
9          installR2Rules(r, F_A)
           /* split the aggregate flow if the space on ingress
              switches is not enough for R1 rules/*
10         subflow, remainedFlow =
           F_A.splitByR1Limit(availableOnIngress)
11         F_A = remainedFlow
12         installR1Rules(r, subflow)
13         result.addDetection(r, subflow)
14       round+= 1 /* schedule the remaining flows in F_A
         into the next round/*
15   return result;
```

However, if the scheduling for $\mathcal{F}_\mathcal{A}$ fails due to $\mathcal{R}_2$ limitation, this indicates that the rule capacity has been exhausted. Thus, we need to first finish the detection for some already allocated flows, free some rule space, and continue investigation for the remaining aggregate flows. We formalize our heuristics in Algorithm 5. It is straightforward that the computational complexity of the algorithm is linear.

6 Implementation

We implement a prototype of FADE and iFADE as applications running on the Floodlight [41] controller, in

TABLE 6
The Settings of Testbed

Tool	Version	Usage	Tool	Version	Usage
Debian	Jessie	Operating System	Open vSwitch	2.3.2	The virtual switch
Floodlight	1.2	The OpenFlow controller	Cbench	1.1	The PacketIn throughput test tool
Mininet	2.2.1	The network simulator	iperf	2.0.5	The network throughput test tool

approximately 12,000 lines of Java code.[1] For each system, we implement three modules: rule storage module, rule graph module and anomaly detection module. Specifically, the rule storage module is extended from HSA [39]. It reads and stores all flow rules by monitoring FlowMod messages and analyzes the dependencies among these rules. Additionally, it provides RESTful interfaces to accept and record manually generated flow rules. The rule graph module is responsible for constructing and updating the forwarding graph. It intercepts the flow table items sent from the control plane to the data plane, analyzes dependencies among these items and constructs the forwarding graph. It also monitors the network topology change messages and updates the forwarding graph accordingly. The anomaly detection module interacts with the other two modules and detects anomalies according to information retrieved from them. It includes three working threads, i.e., the detecting thread, checking thread, and locating thread. The detecting thread performs flow (or aggregate flow) selection and probe generation, and the actual dedicated rule installation. Afterwards, the checking thread collects flow statistics from expired dedicated rules, and executes the anomaly detection logic. Once anomalies are detected, the locating thread is responsible for finding the actual anomalous rules.

7 Evaluation

Our evaluation centers around the following questions.

(i) *What is the detection accuracy and robustness of* FADE *and* iFADE*?* On our physical testbed, we performed a series of experiments with various network topologies and parameter settings to evaluate the detection accuracy. Overall, FADE and iFADE achieve over 95 percent true positive rate and 99 percent true negative rate, and their efficiency is consistent across different settings (Section 7.2).

(ii) *What is the control-plane and data-plane overhead introduced by* FADE *and* iFADE*?* In Section 7.3, we show that both FADE and iFADE introduce only several milliseconds of latency to react to network dynamics, and achieve nearly identical control-plane throughput as if they were not enabled. In the data plane (Section 7.4), we show iFADE consumes over 40 percent less rules than FADE and both schemes impose negligible forwarding overhead.

Besides, we further evaluate the algorithm effectiveness of iFADE using a large-scale network topology, demonstrating that it achieves comparable effectiveness as the production-ready optimization solver Gurobi [43]. Finally, we compare FADE and iFADE with SPHINX [20], which is the state-of-the-art flow statistics based approach with comparable anomaly coverage and detection accuracy.

1. A copy of source code for our prototype is available at https://github.com/chunhui-pang/fade

7.1 Experiment Setup

We deploy our systems on a Linux Debian workstation with four Inter Core i5 3470 CPUs (3.2 GHZ). We use Floodlight as the OpenFlow controller, Mininet as the network emulator and Open vSwitch as the virtual switch. Cbench [44] and iperf [45] are used to benchmark data plane and control plane overhead. We list their information in Table 6. For experiment purpose, we generate aggregate flows as follows. We configure each host on our testbed with multiple *delegated IP addresses*, which are bound to the host's MAC address via static ARP entries. Thus, each host can receive traffic destined to both its real IP addresses and any of its delegated IPs. Then, we further modify the Floodlight controller so that it forwards traffic according to the packet destination addresses. Thus, an aggregate flow is generated when one host sends traffic to another host via different viable addresses.

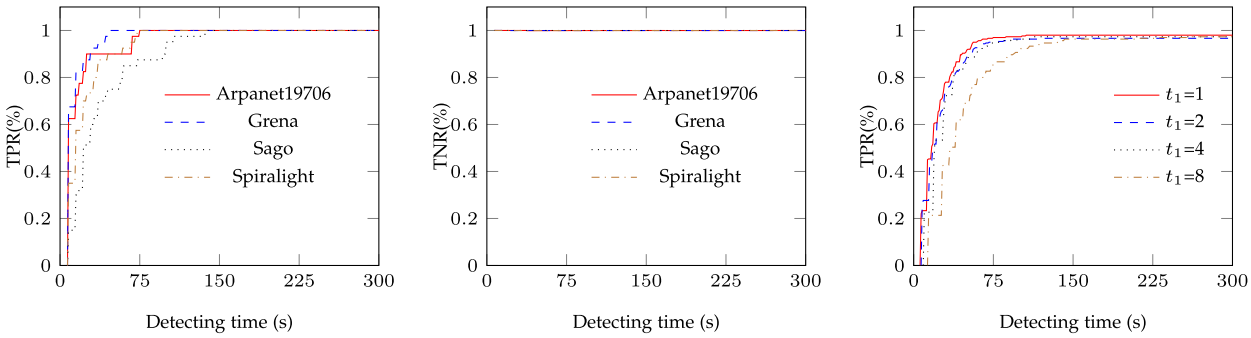
On our testbed, we build two sets of network topology. The first set contains realistic topologies selected from Topology Zoo [42], as listed in Table 7. The second set contains only linear topologies, which connect two hosts via different numbers of hops. We use ovs-ofctl [46] to inject anomalous flow rules into selected switches. To collect nonbiased results, we independently run each experiment 50 times and compute the average results. Our experimental results also depend on various parameters. For d_{max}, the maximum networking latency, we set it to 500 milliseconds based on measurements in [47]. Other parameters, such as the duration of statistics collection, the number of anomalies, and the number of flows, are evaluated explicitly.

7.2 Detection Accuracy and Efficiency

In this subsection, we perform a series of experiments to evaluate the detection accuracy and efficiency with varying parameter settings. The detection accuracy is measured by both true positive rate (TPR, i.e., sensitivity) and true negative rate (TNR, i.e., specificity); the detection efficiency is measured by the time it takes for FADE or iFADE to locate all anomalies, i.e., how long it takes for both TPR and TNR to converge.

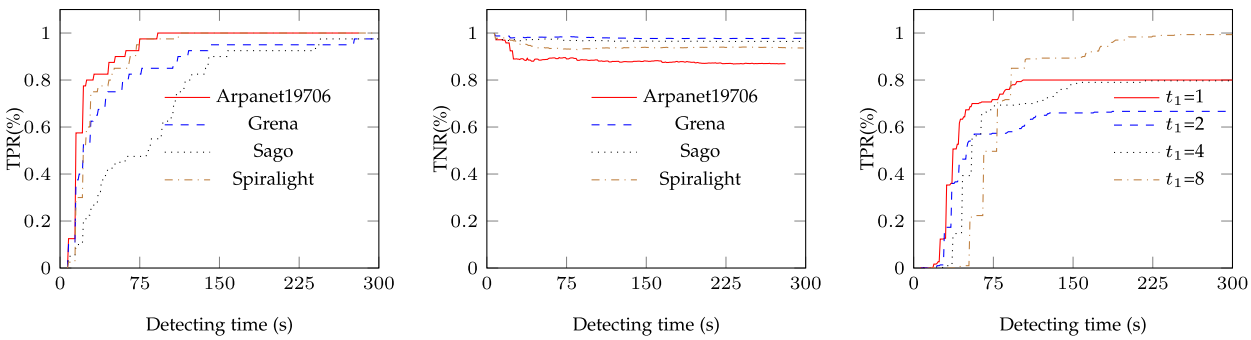
TABLE 7
Network Topologies Collected From ITZ [42]

Topology	Arpanet19706	Spiralight	Grena	Sago
Switches	9	15	16	18
Links	10	16	15	17



(a) TPR of FADE in diff. topologies. (b) TNR of FADE with diff. topologies. (c) TPR of FADE with different t_1.

Fig. 5. Detection accuracy of FADE.



(a) TPR of iFADE with diff. topologies. (b) TNR of iFADE with diff. topologies. (c) TPR of iFADE with different t_1.

Fig. 6. Detection accuracy of iFADE.

Basic Setting. In the first setting, we use the four realistic topologies. For each topology, we generate 200 network flows among all hosts and inject four anomalous flow rules. For each selected 'longest' network flow, we collect their statistics for two seconds (i.e., $\mathcal{R}_1$ rules expires after two seconds), and each experiment lasts for five minutes. We report the measured TPR and TNR for FADE in Figs. 5a and 5b. We draw two conclusions from the results: (i) both TPR and TNR are eventually converged to 1; (ii) on average, it takes FADE a few seconds to locate the first anomaly and tens of seconds to locate all anomalies. We repeat the same experiments for iFADE, and report their results in Figs. 6a and 6b. On average, both TPR and TNR of iFADE are about 5 percent less than those of FADE. We manually check encountered detection errors in the experiments and figure out they are caused by the inaccurate flow statistics reported by the Open vSwitch. Due to flow aggregation in iFADE , such inaccuracy accumulates and eventually impacts detection accuracy. Additionally, it also takes longer for iFADE to locate all anomalies. This is as expected because a single aggregate flow needs to be recursively split into individual flows in the anomaly localization stage.

Robustness. We further evaluate the algorithm robustness of FADE and iFADE under different experimental settings. We first measure the detection accuracy with larger numbers of network flows and anomalous rules. To accurately create this experimental setting, we adopt a linear topology with five hops. By changing the number of delegated IPs on both hosts, we generate different numbers of network flows, ranging from 80 to 320; for each number of flows, we inject a certain number of anomalous rules, ranging from 20 to 80, as listed in Table 8. We report the measured TPR and TNR for both FADE and iFADE in Figs. 7a and 7b, respectively. The results indicate: (i) the detection of FADE is robust, regardless of how many flows and anomalous rules are in the network; (ii) although iFADE's TNR is consistently high, its TPR fluctuates as the number of flows increases. The reason again is because the inaccuracy of flow statistics report by Open vSwitch accumulates in iFADE .

We further measure the impact of flow statistics collection duration t_1, i.e., how long the $\mathcal{R}_1$ rules last. We test four duration values: t_1 to 1, 2, 4 and 8 seconds. Figs. 5c and 6c plot the TPR for FADE and iFADE , respectively. It is clear that (i) FADE is robust for all duration values, however the entire detection period increases as the duration increases; (ii) due to inaccurate statistic report, iFADE performs much better for longer detection periods. Thus, we notice a trade-off between detection accuracy and efficiency in iFADE. Figs. 8a and 8b plot the TNR for FADE and iFADE, respectively, which demonstrates consistently good performance.

7.3 The Control Plane Overhead

In this experiment, we evaluate the control plane overhead of FADE and iFADE. We measure two metrics in this regard: latency for logical forwarding computation and PacketIn throughput of the SDN controller. The logical forwarding computation includes all control-plane actions taken by FADE (or iFADE) to react to network dynamics (e.g., topology changes, new flow rules), including reconstructing the forwarding graph, re-selecting probes and re-generate

TABLE 8
The Number of Network Flows and the Number of Corresponding Anomalies

Network Flows	80	100	120	140	160	180	200	240	280	320	360	400
Anomaly Rules	20	20	20	20	30	30	40	40	50	60	70	80

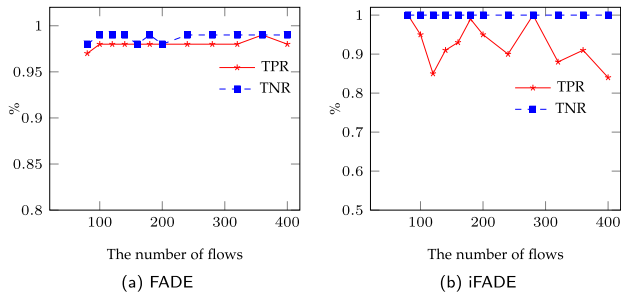


Fig. 7. The converged TPR and TNR of (7a) FADE and (7b) iFADE with different number of flows and anomalies.

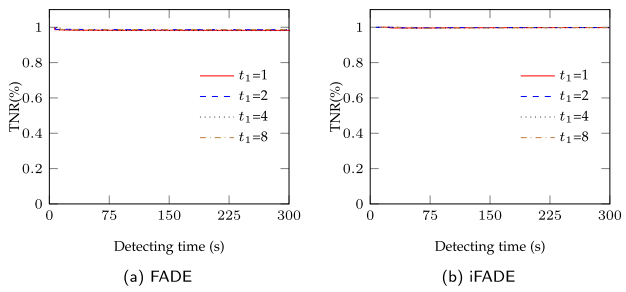


Fig. 8. The TNR of (8a) FADE and (8b) iFADE with different t_1.

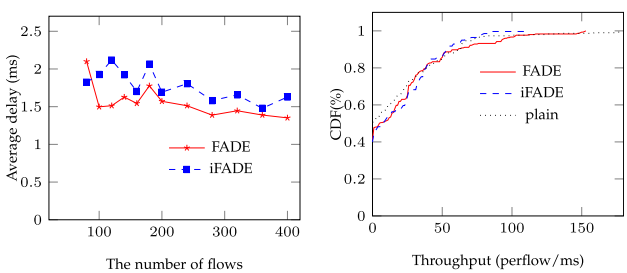


Fig. 9. Control plane overhead of FADE and iFADE.

dedicated flow rules. This metric reflects the agility of FADE (or iFADE) for handling network dynamics. We report the latency of logical forwarding computation in Fig. 9a. Overall, the latency is only about 2 milliseconds even in large scale evaluations. iFADE has slightly higher (about 5 percent more) latency due to its relatively sophisticated flow selection algorithm.

We further use the standard Cbench [44] tool to benchmark the control plane throughput. The tool sends PacketIn messages to the controller and waits for the corresponding flow-mod messages to be returned by the controller. Thus, the throughput of PacketIn messages reflects the overall capacity of the controller. We compare the PacketIn messages throughput for FADE , iFADE and a 'plain' Floodlight controller, and report the results in Fig. 9b. It is clear that the above three controllers have very close throughput, indicating negligible overhead imposed by FADE and iFADE.

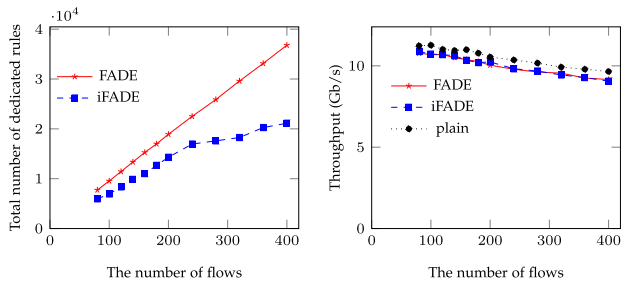


Fig. 10. Data plane overhead of FADE and iFADE.

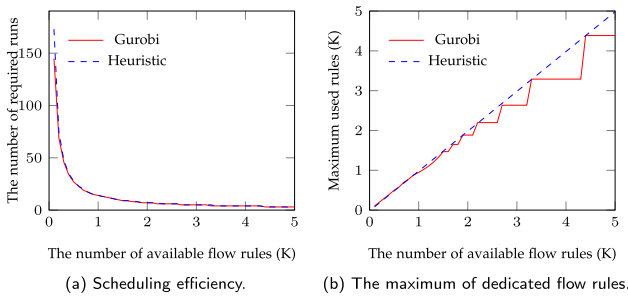


Fig. 11. Comparison experiments between Gurobi and the heuristic used in iFADE.

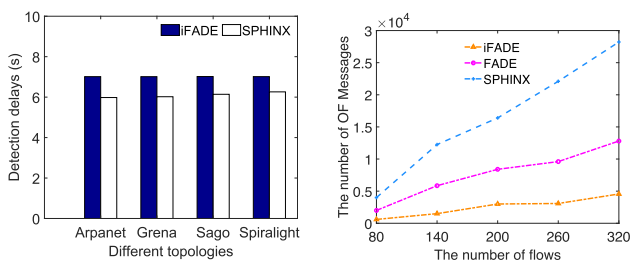


Fig. 12. Comparison experiments between FADE, iFADE and SPHINX.

7.4 The Data Plane Overhead

In this experiment, we evaluate the data plane overhead imposed by FADE and iFADE. We report two evaluation results in this regard: the number of dedicated flow rules required for detection (Fig. 10a) and the data plane network throughput (Fig. 10b). From Fig. 10a, it is clear that the number of required rules in iFADE is significantly less than FADE (about 40 percent reduction). More crucially, the number of required rules in iFADE increases *sublinearly* with the number of flows in the network. Thus, iFADE is more scalable than FADE. In addition, Fig. 10b shows that both FADE and iFADE slightly degrade the network throughput by less than 3 percent, which is negligible considering that overall

TABLE 9
The Switches of Internet2 and the Number of FIBs on Them

Switch Name	# of FIBs	Switch Name	# of FIBs	Switch Name	# of FIBs	Switch Name	# of FIBs
ATLA	17429	CHIC	17386	NEWY32AOA	17419	PAIX	17477
CLEV	17433	HOUS	17434	SALT	17450	SEAT	17695
KANS	17443	LOSA	17424	WASH	17401	WILC	17479

link utilization (even for those bottleneck links) in production SDN networks is about 90 percent [48].

7.5 Effectiveness of Heuristics in Large-Scale Topology

We further evaluate the effectiveness of the heuristic used in iFADE to perform detection scheduling (Algorithm 5). We construct a large-scale network topology based on the forwarding information base (FIB) of the Internet2 [49].[2] Eventually, we extract 12 switches, and every switch has over 17,000 FIB entries, as shown in Table 9. We use HSA to analyze these FIBs offline and construct the forwarding graph, based on which we extract 61553 flows and 115 aggregate flows (recall that a flow in our paper is defined as a sequence of switch rules, as described in Section 3.1.1).

We run Algorithm 5 to schedule detection for all aggregate flows. As a benchmark, we adopt Gurobi [43], a production-ready optimization solver, to solve the same ILP problem. We compare the number of rounds needed and the maximum consumed rules of our heuristics and Gurobi. In Fig. 11a, we report the number of rounds it takes for both solvers to solve the problem under different numbers of available rules on switches. It is clear that when number of available rules is more than 200, both solvers have almost the same effectiveness. When the maximum number of available flow rules is smaller, Algorithm 5 needs about only 10 percent more rounds than Gurobi. In addition, we also report the maximum number of rules used by both solvers in Fig. 11b. We find that Gurobi requires relatively less rules than Algorithm 5, especially when the number of available rules is large. However, the gap between the two algorithms is not large. Thus, by comparisons in multiple dimensions, we show that our heuristics have comparable performance with Gurobi.

7.6 Experimental Comparison With SPHINX [20]

Finally, we report some experimental comparison with SPHINX [20], the state-of-the-art flow statistics based work that can achieve similar detection accuracy with FADE and iFADE. We focus on two metrics in this regard. First, we report the average detection latency for finding the first anomaly in different network topologies in Fig. 12a (FADE and iFADE have very similar results so we only plot iFADE in the figure). On average, iFADE takes about one more second than SPHINX. This is because iFADE has to install two kinds of dedicated measurement rules in a strict order, which introduces additional delays. Further, we evaluate the detection overhead introduced by three systems, measured by the number of OpenFlow control messages generated during flow statistics collection. As shown in Fig. 12b, on average, the number of OpenFlow messages generated by SPHINX is about three times and ten times of that generated by FADE and iFADE , respectively. Thus, compared with ubiquitous flow statistic collection, the intelligent collection mechanism invented by FADE and iFADE significantly reduces communication overhead of anomaly detection. Besides, the number of control messages generated by iFADE is only about half of that of FADE . This is because that iFADE installs fewer rules than FADE . In conclusion, compared with SPHINX, FADE and iFADE greatly reduce the communication overhead at a small cost of detection latency.

2. The real-time data has been removed from the website. The data snapshot we use in our experiments is available at https://github.com/chunhui-pang/fade/tree/master/real-data/internet2

8 Conclusion

In this paper, we propose FADE, an efficient and accurate forwarding anomaly detection system in SDN. FADE intelligently computes a small set of measurement rules, places them optimally across the network, and accurately collects and analyzes their statistics to comprehensively detect and locate all forwarding anomalies in the network. On top of FADE, we further propose iFADE, a more scalable version of FADE that not only saves large numbers of measurement rules, but also is deployable even if the available switch rule capacity is much less than the total required rules. We implement both FADE and iFADE on our physical testbed in roughly 12,000 lines of code and extensively evaluate their performance. Overall, both systems achieve over 95 percent true positive detection rate and 99 percent true negative rate, while imposing small overhead on both the control plane and data plane.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant 2018YFB1800304, in part by NSFC under Grant 61572278 and Grant 61772412, and in part by BNRist under Grant BNR2020RC01013. The work of Peng Zhang was supported in part by the K. C. Wong Education Foundation.

References

[1] A. Singh *et al.*, "Jupiter rising: A decade of Clos topologies and centralized control in Google's datacenter network," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 183–197, 2015.

[2] C.-Y. Hong *et al.*, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in Google's software-defined WAN," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 74–87.

[3] C.-Y. Hong *et al.*, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 15–26.
[4] Z. Liu, Y. Cao, X. Zhang, C. Zhu, and F. Zhang, "Managing recurrent virtual network updates in multi-tenant datacenters: A system perspective," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1816–1825, Aug. 2019.
[5] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, 2008.
[6] M. Antikainen, T. Aura, and M. Särelä, "Spook in your network: Attacking an SDN with a compromised OpenFlow switch," in *Proc. Nordic Conf. Secure IT Syst.*, 2014, pp. 229–244.
[7] K. Benton, L. J. Camp, and C. Small, "OpenFlow vulnerability assessment," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, 2013, pp. 151–152.
[8] M. Kuźniar, P. Perešíni, and D. Kostić, "What you need to know about SDN flow tables," in *Proc. Int. Conf. Passive Active Meas.*, 2015, pp. 347–359.
[9] OpenFlow switch specification, 2012. Accessed: 2019. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf
[10] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 58–72.
[11] G. Pickett, "Staying persistent in software defined networks," in *Black HatUSA*, 2015. [Online]. Available: https://paper.bobylive.com/Meeting_Papers/BlackHat/USA-2015/us-15-Pickett-Staying-Persistent-In-Software-Defined-Networks-wp.pdf
[12] 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 *Proc. 35th Annu. IEEE Int. Conf. Comput. Commun.*, 2016, pp. 1–9.
[13] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," in *Proc. 8th Int. Conf. Emerg. Netw. Experiments Technol.*, 2012, pp. 241–252.
[14] K. Agarwal, E. Rozner, C. Dixon, and J. Carter, "SDN traceroute: Tracing SDN forwarding without changing network behavior," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, 2014, pp. 145–150.
[15] P. Zhang, C. Zhang, and C. Hu, "Fast data plane testing for software-defined networks with RuleChecker," *IEEE/ACM Trans. Netw.*, vol. 27, no. 1, pp. 173–186, Feb. 2019.
[16] P. Zhang *et al.*, "Mind the Gap: Monitoring the control-data plane consistency in software defined networks," in *Proc. 12th Int. Conf. Emerg. Netw. Experiments Technol.*, 2016, pp. 19–33.
[17] T. Sasaki, C. Pappas, T. Lee, T. Hoefler, and A. Perrig, "SDNsec: Forwarding accountability for the SDN data plane," in *Proc. 25th Int. Conf. Comput. Commun. Netw.*, 2016, pp. 1–10.
[18] P. Zhang, "Towards rule enforcement verification for software defined networks," in *Proc. IEEE Conf. Comput. Commun.*, 2017, pp. 1–9.
[19] Q. Li, X. Zou, Q. Huang, J. Zheng, and P. P. C. Lee, "Dynamic packet forwarding verification in SDN," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 6, pp. 915–929, Nov./Dec. 2019.
[20] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "SPHINX: Detecting security attacks in software-defined networks," in *Proc. Annu. Netw. Distrib. Syst.Secur. Symp.*, 2015. [Online]. Available: http://dx.doi.org/10.14722/ndss.2015.23064
[21] A. Kamisiński and C. Fung, "FlowMon: Detecting malicious switches in software-defined networks," in *Proc. Workshop Automated Decision Making Active Cyber Defense*, 2015, pp. 39–45.
[22] T.-W. Chao *et al.*, "Securing data planes in software-defined networks," in *Proc. IEEE NetSoft Conf. Workshops*, 2016, pp. 465–470.
[23] P. Zhang *et al.*, "FOCES: Detecting forwarding anomalies in software defined networks," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 830–840.
[24] A. Shaghaghi, M. A. Kaafar, and S. Jha, "WedgeTail: An intrusion prevention system for the data plane of software defined networks," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, 2017, pp. 849–861.
[25] Q. Li, X. Zou, Q. Huang, J. Zheng, and P. P. C. Lee, "Dynamic packet forwarding verification in SDN," *IEEE Trans. Dependable Secure Comput.*, vol. 16, no. 6, pp. 915–929, Nov./Dec. 2019.
[26] P. Zhang, H. Wu, D. Zhang, and Q. Li, "Verifying rule enforcement in software defined networks with REV," *IEEE/ACM Trans. Netw.*, vol. 28, no. 2, pp. 917–929, Apr. 2020.
[27] P. Zhang *et al.* "Network-wide forwarding anomaly detection and localization in software defined networks," *IEEE/ACM Trans. Netw.*, vol. 29, no. 1, pp. 332–345, Feb. 2021.
[28] Z. Liu, H. Jin, Y.-C. Hu, and M. Bailey, "Practical proactive DDoS-attack mitigation via endpoint-driven in-network traffic control," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1948–1961, Aug. 2018.
[29] sFlow.org. sFlow version 5, 2004. Accessed: 2021. [Online]. Available: https://sflow.org/sflow_version_5.txt
[30] B. Claise, "Cisco systems NetFlow services export version 9," RFC 3954, Oct. 2004, doi: 10.17487/RFC3954.
[31] R. Cohen and E. Moroshko, "Sampling-on-demand in SDN," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2612–2622, Dec. 2018.
[32] A. Ramachandran, S. Seetharaman, N. Feamster, and V. Vazirani, "Fast monitoring of traffic subpopulations," in *Proc. 8th ACM SIGCOMM Conf. Internet Meas.*, 2008, pp. 257–270.
[33] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *Proc. Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2002, pp. 323–336.
[34] A. Lall, V. Sekar, M. Ogihara, J. Xu, and H. Zhang, "Data streaming algorithms for estimating entropy of network traffic," in *Proc. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2006, pp. 145–156.
[35] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 413–424.
[36] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 63–74.
[37] A. Ferguson *et al.*, "Orion: Google's software-defined networking control plane," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/ferguson
[38] J. Cao *et al.*, "The crosspath attack: Disrupting the SDN control channel via shared links," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 19–36.
[39] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. 10th USENIX Conf. Netw. Syst. Des. Implementation*, 2013, pp. 99–111.
[40] Y. Vanaubel, J.-J. Pansiot, P. Mérindol, and B. Donnet, "Network fingerprinting: TTL-based router signatures," in *Proc. Conf. Internet Meas. Conf.*, 2013, pp. 369–376.
[41] The floodlight controller. [Online]. Available: http://www.projectfloodlight.org
[42] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, pp. 1765–1775, Oct. 2011.
[43] I. Gurobi Optimization, "Gurobi optimizer reference manual." Accessed: 2019. [Online]. Available: http://www.gurobi.com
[44] Cbench. Accessed: 2019. [Online]. Available: http://ctuning.org/wiki/index.php?title=CTools:CBench
[45] Iperf. Accessed: 2019. [Online]. Available: https://iperf.fr
[46] ovs-ofctl. Accessed: 2019. [Online]. Available: http://www.openvswitch.org//support/dist-docs/ovs-ofctl.8.txt
[47] C. Yu, C. Lumezanu, A. B. Sharma, Q. Xu, G. Jiang, and H. V. Madhyastha, "Software-defined latency monitoring in data center networks," in *Proc. Int. Conf. Passive Active Netw. Meas.*, 2015, pp. 360–372.
[48] A. Kumar *et al.*, "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 1–14.
[49] Internet2 flow rules. [Online]. Available: http://web.archive.org/web/20160423132602/http://vn.grnoc.iu.edu/Internet2/fib/index.cgi

Qi Li (Senior Member, IEEE) received the PhD degree from Tsinghua University, Beijing, China. Currently, he is an associate professor with Institute for Network Sciences and Cyberspace, Tsinghua University. He has worked with ETH Zurich and the University of Texas at San Antonio. His research interests include network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an editorial board member of the *IEEE Transactions on Dependable and Secure Computing* and the *ACM Digital Threats: Research and Practice*.

Yunpeng Liu received the BSc degree in computer science from Tsinghua University, Beijing, China. Currently, he is working toward the docter's degree with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. His research interests include network security and data-driven security.

Zhuotao Liu received the BS degree from Shanghai Jiao Tong University, Shanghai, China, and the PhD degree from the University of Illinois at Urbana-Champaign, Champaign, Illinois. He is currently an assistant professor with the Institute for Network Sciences and Cyberspace, Tsinghua University. Before joining Tsinghua, he was a technical lead at Google, managing massive-scale software-defined datacenter networks. His research interests include network security & privacy, Blockchain infrastructure, datacenter networking, and systems security.

Peng Zhang received the PhD degree in computer science from Tsinghua University, Beijing, China, in 2013. He was a visiting researcher with the Chinese University of Hong Kong and Yale University. He is currently an associate professor with the School of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. He is also with the MOE Key Lab for Intelligent Networks and Network Security. His research interests include verification, measurement, and security in computer networks.

Chunhui Pang received the master's degree from Tsinghua University, Beijing, China. His research interests include network security and network measurement.