Detecting Congestion-Related Attacks via Fine-Grained Queue Diagnosis

Rui Dai, Dan Tang, Zheng Qin, *Member, IEEE*, Kai Chen, Keqin Li, *Fellow, IEEE*, and Jiliang Zhang, *Senior Member, IEEE*

***Abstract*—As modern networks are increasingly demanding performance, it is crucial to protect network resources from attack threats. Congestion-Related Attacks (CRAs) have become a serious threat to network infrastructures, which can cause severe degradation of network performance. The emerging programmable switch makes it possible to offload intelligence to the data plane, providing an opportunity to deploy defense strategies against CRAs in the data plane. In this paper, we present the Fine-Grained Queue Diagnosis (*FGQD*) system deployed on the programmable switch, capable of real-time monitoring of the queue and network traffic to defend against CRAs. Specifically, we propose *culprit flows for congestion* and *active flows during congestion* to track the flows culpable for congestion formation and those exhibiting abnormal activity during congestion. To effectively recognize these flows, we design the approximate data structure Time-Windows to overcome the resource and operational constraints on the programmable data plane. Furthermore, we employ an in-network machine learning model that utilizes queue and packet features to identify malicious flows of CRAs. Extensive experiments on the software-based testbed show that *FGQD* achieves 97.333% detection rate with remarkably low false positive rates of 0.018% and 0.106% when detecting Shrew attacks and Optimistic Ack attacks, outperforming existing methods including Conquest, Henna, and Hashpipe. Moreover, *FGQD* responds to Shrew attacks within 5.70 milliseconds, which is orders of magnitude faster than SDN-based control plane solutions that typically require seconds to respond. These results conclusively demonstrate *FGQD*'s exceptional effectiveness in defending against CRAs.**

***Index Terms*—Congestion-related attack, in-network machine learning, programmable data plane, queue diagnosis.**

Received 1 April 2024; revised 31 December 2024 and 11 April 2025; accepted 9 June 2025. Date of publication 17 June 2025; date of current version 19 December 2025. This work was supported in part by the National Natural Science Foundation of China under Grants 62472153 and U24A20289, and by the Hunan Provincial Natural Science Foundation of China under Grant 2025JJ50350. The associate editor coordinating the review of this article and approving it for publication was K. Zeng. *(Corresponding author: Dan Tang.)*

Rui Dai, Dan Tang, and Zheng Qin are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China (e-mail: dairui@hnu.edu.cn; Dtang@hnu.edu.cn; zqin@hnu.edu.cn).

Kai Chen is with the School of Cyberspace Security, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: kchen@hust.edu.cn).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Jiliang Zhang is with the College of Semiconductors (College of Integrated Circuits), Hunan University, Changsha 410082, China (e-mail: zhangjiliang@hnu.edu.cn).

Digital Object Identifier 10.1109/TCCN.2025.3580566

I. Introduction

WITH emerging applications such as Internet payment, online games, online education, and industrial Internet, the performance requirements for modern networks are becoming increasingly demanding [1], [2], [3]. Congestion control is a key component of the Internet, playing a vital role in ensuring efficient and fair transmission while significantly influencing network performance [4]. Due to the vulnerability of congestion control, it is susceptible to being exploited and compromised by attackers [5], who can launch attacks that disrupt network link transmission and maliciously create congestion in the network, resulting in significant damage and degradation of network performance [6], [7], [8]. In 2015, NetEase suffered an economic loss of over RMB 15 million due to a cyber attack on its backbone network that congested and paralyzed the primary ingress and egress links to its server farm [9].

In this context, several Congestion-Related Attacks (CRAs) [10] have emerged, such as Shrew attacks [11], [12], [13] and Optimistic Ack attacks [14], [15]. Invariably, these attacks cause network congestion, severely impairing network performance. Unfortunately, despite the research community's efforts [16], [17], [18], [19], detecting and mitigating CRAs still face challenges in effectiveness, flexibility, and real-time performance. The recent trend in Software-Defined Networking (SDN) has introduced programmable switches [20], [21], enabling the offload of intelligence into the network and providing an opportunity to defend against CRAs in the switch. Programmable switches support programming using domain-specific languages like P4 [22], allowing packet processing with user-defined logic. In addition, a program can operate in concert between the control and data planes, enabling flexible and advanced packet processing. The programmable switch enables line-rate packet processing with throughput orders of magnitude higher than the servers [23], [24]. Furthermore, deploying detection systems in the data plane can avoid communication delays with the control plane, enabling a timely response to CRAs. Unlike systems deployed on the host side, systems deployed on the switch can preemptively see and filter malicious traffic [25], thus preventing CRAs from harming network performance. In particular, the visibility of the network queue is critical for diagnosing network performance issues. Still, traditional network devices have minimal visibility into the queue state, making it challenging to analyze and fix performance problems. With the advent of programmable switches, fine-grained and real-time queue monitoring has become possible [10], [26].

In this paper, we propose the Fine-Grained Queue Diagnosis (*FGQD*) system deployed on the programmable switch, capable of real-time monitoring of the queue and network traffic to defend against CRAs. We propose *culprit flows for congestion* and *active flows during congestion* to track flows culpable for congestion formation and flows exhibiting abnormal activity during congestion, respectively. *FGQD* monitors the queue in real-time to identify culpable flows that cause packet backlogs in the queue to find out the primary culprits that cause network congestion. Furthermore, *FGQD* measures *the flow's activity during congestion* based on the intention and behavior of CRAs to create congestion. Those flows that remain active during network congestion may be malicious flows attempting to exacerbate congestion. However, programmable switches are highly restricted, with low memory available and limited mathematical operations support [27], [28]. It is not easy to implement the identification of *culprit flows for congestion* and *active flows during congestion* in the data plane. To overcome these limitations, we design Time-Windows, a compact data structure consisting of two windows supporting access and clean operations, to achieve accurate tracking of *culprit flows for congestion* and *active flows during congestion*. Notably, benign flows can also become *culprit flows for congestion* and *active flows during congestion* due to network bursts in normal circumstances [26], [29]. To further reduce false positives for benign flows, *FGQD* also leverages packet features to identify malicious flows. Specifically, *FGQD* first uses queue diagnosis to preliminarily identify suspicious flows, and then employs an in-network Random Forest (RF) model that combines both queue and packet features to ultimately identify malicious flows. Generally, our contributions can be briefly summarized as follows:

- We propose *culprit flows for congestion*, considering flows that cause packet backlogs in a queue to be blamed for congestion, which provides a flow-level analysis of the reasons for network congestion from a queuing perspective.
- We propose *active flows during congestion* to track flows that are abnormally active during congestion based on the intention and behavior of CRAs to congest the network.
- We design Time-Windows, a compact data structure, to overcome the limitations on the programmable switch and implement the tracking of *culprit flows for congestion* and *active flows during congestion* in the data plane.
- We design an in-network RF model that combines queue and packet features to identify malicious flows, which can effectively reduce false positives for benign flows.
- We prototype *FGQD* and use extensive experiments on the software-based testbed to validate its performance.

The rest of the paper is organized as follows. Section II introduces the background and motivation. Section III presents the high-level design of *FGQD*, with the design details in Section IV. In Section V, we experimentally evaluate the *FGQD*'s performance. Section VI discusses several practical issues and Section VII reviews related works. Finally, Section VIII concludes this paper.

II. Background and Motivation

In this section, we first present the threat model of CRAs (Section II-A), then discuss the limitations of existing defenses (Section II-B), and finally introduce the programmable data plane along with its constraints (Section II-C).

A. Threat Model

We refer to attacks that maliciously congest target links as Congestion-Related Attacks (CRAs) [10], which can be divided into two categories. The first category involves directly manipulating protocol flaws to create congestion [5], such as the Optimistic Ack attack [14], [15]. The second category consists in generating traffic to create congestion and may further trigger protocol vulnerabilities to enhance the attack's effectiveness, such as the Shrew attack [11], [12]. The attack methods and tactics of CRAs may be varied, but they share a common characteristic of congesting network links to impair network performance.

B. Limitations of Existing CRAs Defenses

Effectiveness shortcomings. CRAs typically exploit vulnerabilities in network protocols and congestion control, making it challenging to combat them. For example, the Optimistic Ack attack manipulates acknowledgment packets where the sending host is not compromised. Thus, systems [30], [31] that rely on IP address-based information struggle to accurately detect the Optimistic Ack attack. Another solution is to modify protocol specifications and implementations, which is practically difficult to implement [19], and protocols always have unknown flaws that attackers may exploit [32], [33]. Additionally, CRAs typically target network links far from the client side, preventing the client side defenses from obtaining an accurate view. Even if the systems [34], [35] on the client side effectively counter the attack, the malicious traffic still traverses the network, impairing the network's performance.

Flexibility shortcomings. There may be various CRAs, each with different strategies and ways. The system designed for a specific CRA lacks flexibility and is difficult to effectively scale to address other types of CRAs. For instance, the techniques [16], [17], [36] for detecting Shrew attacks are tailored to the pattern of Shrew attacks but cannot effectively defend against Optimistic Ack attacks.

Real-time performance shortcomings. Most defense systems deploy in the control plane or middlebox, which inevitably introduces communication latency and fails to achieve timely response to attacks, e.g., Softguard [37], P&F [38], PeakSAX [18], and GASF-IPP [39] deployed in the control plane have a response delay of seconds or even tens of seconds, failing to achieve a timely response to attacks.

C. Programmable Data Plane

Implementing new network functions on traditional switches is an expensive and complex process, so the community has proposed SDN that decouples the control plane from the data plane, transferring network intelligence to logically

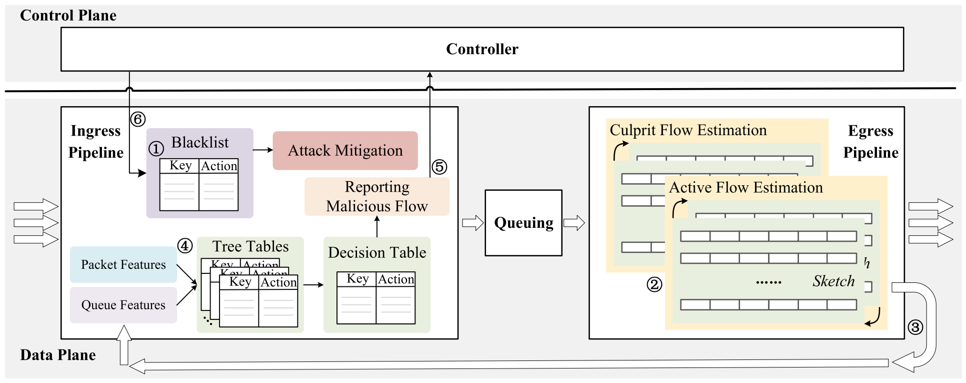


Fig. 1. The overview and workflow of *FGQD*.

centralized controllers to enable fully programmable software-driven networks [21]. Emerging programmable switches are programmed using domain-specific languages such as P4 [22] and support stateful packet processing using user-defined logic, extending network programmability from the control plane to the data plane. Despite its many advantages, the programmable data plane has the following limitations regarding computing and storage [27], [28].

Limited support for mathematical operations. P4 supports boolean, shift, addition, and subtraction operations but not floating-point, loop, division, multiplication, and complex conditional operations.

Low available memory. The programmable switch has limited SRAM and TCAM, e.g., only 120MB of SRAM and 6.2MB of TCAM per pipeline in Tofino 1 [27]. Additionally, as packets traverse the switch pipeline, each register can only be accessed once.

III. Overview of *FGQD*

In this section, we present *FGQD*, a Fine-Grained Queue Diagnosis system deployed in the data plane that leverages the emerging P4 programmable switch. *FGQD* utilizes the queue metadata provided by the switch to measure *the flow's contribution to the queue backlog* and *the flow's activity during congestion*, thus enabling the detection of CRAs. Fig. 1 shows the five modules of *FGQD* deployed in the data plane.

Culprit Flow Estimation Module implements the measurement of the *culprit flows for congestion*. The purpose of CRAs is to congest the target link, and the malicious flow involved in the attack is often a significant contributor to the queue backlog.

Active Flow Estimation Module implements the measurement of the *active flows during congestion*. Malicious flows generated by CRAs are typically the perpetrators of congestion, aiming to congest the target link, and therefore are active when the network is congested.

In-Network RF Module performs the final identification of malicious flows of CRAs, which consists of multiple tree tables and a decision table. Each tree table represents a decision tree model, where each rule in the tree table corresponds to a decision path from the root to a leaf node in the decision tree. The action triggered by the rule depends on the type of the leaf node in the decision tree. Based on the decision outcomes from the tree tables, the decision table makes the final classification.

Reporting Module performs the control plane reporting of malicious flows. After detecting malicious flows via the in-network RF module, *FGQD* reports these malicious flows to the control plane so that mitigation rules can be issued to the blacklist table.

Mitigation Module handles the mitigation of CRAs. Based on the malicious flows submitted by the reporting module, the control plane instructs the data plane to take mitigation actions (e.g., marking malicious flows, rate-limiting malicious flows, deprioritizing malicious flows, dropping the packets, etc.), thus preventing CRAs.

There are six steps in the *FGQD* workflow, as shown in Fig. 1. ①*Match blacklist*, when a packet arrives at the switch ingress pipeline, it is first matched against the blacklist table, and if *flow_ID* of the packet is found in the blacklist, mitigation actions are applied to prevent the CRAs. If the packet does not match any entries in the blacklist table, it is forwarded normally.

②*Perform fine-grained queue diagnosis*, since the queue metadata can only be obtained in the egress pipeline, both the culprit flow estimation module and active flow estimation module are deployed in the egress pipeline. Queue diagnosis monitors *the flow's contribution to the queue backlog* and *the flow's activity during congestion*, and if a flow is identified as being both the *culprit flow for congestion* and the *active flow during congestion*, *FGQD* initially recognizes the flow as a suspicious flow for CRAs.

③*Recirculate packets of suspicious flows*, when the culprit flow estimation module and active flow estimation module in the egress pipeline detect the suspicious flow, *FGQD* recirculates the packet of the suspicious flow to the ingress pipeline.

④*Recognize malicious flows*, when the packets of a suspicious flow are recirculated to the ingress pipeline, the in-network RF model uses the packet and queue features to match the tree tables and decision table, ultimately identifying the malicious flows of CRAs.

⑤*Report malicious flows*, after the in-network RF model finalizes the malicious flow, reporting module uses digest messages to report the *flow_ID* of malicious flows based on the classification result of the decision table.

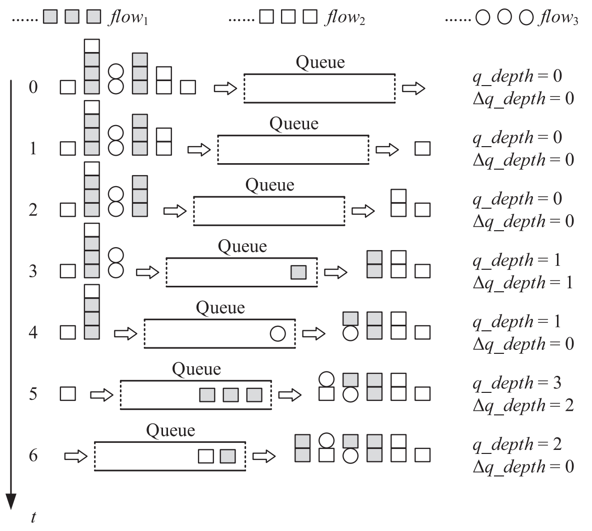


Fig. 2. Demonstration of the queue backlog formation process.

⑥*Add entries to the blacklist table*, after the control plane receives the reported *flow_ID* of the malicious flow, it issues the entry to the blacklist table through the Runtime API.

IV. Design Detail

In this section, we present the design details of *FGQD*, including the details of Fine-Grained Queue Diagnosis (Section IV-A), the design of Time-Windows (Section IV-B), and the details of in-network RF model (Section IV-C).

A. Fine-Grained Queue Diagnosis

1) Culprit Flows for Congestion: Queue backlog is an essential reflection of the network conditions. It is a feasible idea to track the flows culpable for network congestion based on the process of queue backlog formation [10], [26]. Fig. 2 shows a demonstration of the queue backlog formation process. Assuming that two packets can pass on the queue per unit of time ($MaxRate_{out} = 2\ packets/t$). During the time range $t \in (0, 6]$, there are three flows of packets that have passed through the queue. At the beginning ($t = 0$), the queue depth is 0 ($q\_depth = 0$), the queue depth change is 0 ($\Delta q\_depth = 0$), and there is no packet backlog in the queue. After $t = 3$, there are backlogs in the queue, in which within $t \in (2, 3]$ and $t \in (4, 5]$, the backlog of packets in the queue increases to 1 and 3.

To prevent CRAs, it is necessary to identify suspicious flows from the perspective of congestion formation. More precisely, we ask: *when packets block in a queue, which flows through the queue are culpable for the backlog*?

Definition 1: The flow's contribution to the queue backlog. n flows ($\{flow_1, \dots, flow_n\}, n > 0$) have passed through the queue during the time range $(t_s, t_e]$ ($t_s < t_e$), where each $flow_i$ consists of m_i ($m_i > 0$) packets, i.e., $flow_i = \{p_1, \dots, p_{m_i}\}$. For the time range $(t_j, t_{j+1}]$, x_j ($x_j > 0$) packets have passed through the queue. In these x_j packets, the proportion of packets in each $flow_i$ is r_i^j, in which $\sum_{i=1}^{n} r_i^j = 1$. If the change in queue depth over the time range $(t_j, t_{j+1}]$ is Δq_depth_j ($\Delta q\_depth_j >= 0$[1]), the $flow_i$'s contribution to the queue backlog during $(t_j, t_{j+1}]$ is $r_i^j \times \Delta q\_depth_j$. In the total time range $(t_s, t_e]$, the $flow_i$'s contribution to the queue backlog is Eq. (1). Notably, *definition 1* is independent of the packet scheduling algorithm.

$$contribution_{flow_i} = \sum_{t_j = t_s}^{t_{e-1}} \left(r_i^j \times \Delta q\_depth_j \right) \tag{1}$$

Taking the queue backlog formation process in Fig. 2 as an example, initially each flow's contribution to the queue backlog is 0. Within $t \in (0, 1]$, the packets of $flow_2$ pass through the queue, the Δq_depth of the process is 0, and the $contribution_{flow_2}$ remains 0. Similarly, within $t \in (1, 2]$, $contribution_{flow_2}$ still remains 0. During the time range $t \in (2, 3]$, the packets of $flow_1$ pass through the queue and the Δq_depth for that process is 1, so $contribution_{flow_1}$ increases to 1. Within $t \in (3, 4]$, the packets of $flow_3$ pass through the queue, and the Δq_depth of this process is 0, so $contribution_{flow_3}$ remains 0. Within $t \in (4, 5]$, the packets of $flow_1$ and $flow_2$ pass through the queue, where the packets of $flow_1$ account for $\frac{3}{4}$ and the packets of $flow_2$ charge for $\frac{1}{4}$, and the Δq_depth of this process is 2. Therefore, $contribution_{flow_1}$ is increased to 2.5 (i.e., $1 + \frac{3}{4} \times 2$) and $contribution_{flow_2}$ is increased to 0.5 (i.e., $0 + \frac{1}{4} \times 2$). Within $t \in (5, 6]$, the packets of $flow_2$ pass through the queue, the change of queue depth is negative and the Δq_depth of the process is 0, and the $contribution_{flow_2}$ remains unchanged. Thus during the time range $t \in (0, 6]$, $contribution_{flow_1}$ is 2.5, $contribution_{flow_2}$ is 0.5 and $contribution_{flow_3}$ is 0.

Definition 2: Culprit flows for congestion. Given a queue backlog contribution threshold δ, $flow_i$ is considered to be the culprit flow for congestion in the time range $(t_s, t_e]$ if $contribution_{flow_i}$ is greater than the threshold δ.

CRAs aim at congesting the target link, so the malicious flows of the attack typically are the *culprit flows for congestion* as well. For the benign flows, the contribution to the queue backlog will be much smaller than that of the malicious flows. Per-flow measurements are challenging to implement due to the constraints on the P4 switch [27], [28]. Therefore, *FGQD* applies approximation techniques (Time-Windows in Section IV-B) to estimate the *culprit flows for congestion* within the limitations imposed by the P4 switch and makes a simplification to accommodate the constraints on the P4 switch when measuring *the flow's contribution to the queue backlog*. In the culprit flow estimation module, *FGQD* sets a queue depth record using the registers. When a packet arrives at the egress pipeline, *FGQD* reads and updates the queue depth record to calculate the $\Delta q\_depth'$ between the current packet and the previous queue depth. The simplified calculation of *the flow's contribution to the queue backlog* is Eq. (2).

$$contribution'_{flow_i} = \sum_{t_j = t_s}^{t_{e-1}} \Delta q\_depth'_j \tag{2}$$

[1]A negative change in queue depth indicates that the queue backlog is easing. For this case, we set Δ *q_depth* to 0.

Due to the simplification of calculating $contribution'_{flow_i}$, there will be some errors with the actual $contribution_{flow_i}$. The purpose of *FGQD* is to detect the malicious flows for CRAs whose contribution to the queue backlog usually be much more significant than that of most benign flows, so although there is an error in the actual measurement of $contribution_{flow_i}$, it does not affect *FGQD*'s effectiveness in detecting CRAs, as will be evaluated in Section V.

2) Active Flows During Congestion: In normal circumstances, there may also be bursts in the network traffic that cause a backlog in the queue [26], [29]. There exists a small benign traffic whose contribution to the queue backlog will be a considerable value and thus falsely reported as malicious by *FGQD*. In addition, the $contribution'_{flow_i}$ calculated by *FGQD* is in error with the real $contribution_{flow_i}$. Therefore, other aspects of malicious flows need to be measured to further improve the accuracy of detecting CRAs.

FGQD measures *the flow's activity during congestion* to detect CRAs in terms of the intent and behavior of the attack. Attackers use CRAs to congest the target link, so the malicious flows typically show significant activity when the network is congested. For benign TCP flows, they are limited by the sender's congestion control, which reduces the congestion window (e.g., Multiplicative Decrease) and thus reduces the sending rate when the sender senses that network congestion is occurring (e.g., RTO Timeout or Duplicate ACK). Therefore, the activity of benign TCP flows decreases significantly during network congestion. Although benign UDP flows aren't subject to congestion control, they lack any intent or behavior to maliciously congest the network. Therefore, benign UDP flows typically do not behave abnormally active when the network is congested.

Definition 3: The flow's activity during congestion. Given the thresholds τ and γ for determining the congested state, where the network is considered to be congested when packet p_{k_i} of $flow_i$ passes through the queue if its queuing time $q\_time_{k_i}$ is greater than the threshold τ or the queuing depth $q\_depth_{k_i}$ is greater than the threshold γ. For each $flow_i$ in the time range $(t_s, t_e]$, $activity_{flow_i}$ in the congested state is Eq. (3), where $f(\cdot)$ is Eq. (4).

$$activity_{flow_i} = \sum_{k_i=1}^{m_i} f(\cdot) \times p_{k_i} \tag{3}$$

$$f(\cdot) = \begin{cases} 1, & \text{if } \left(q\_depth_{k_i} > \gamma \ or \ q\_time_{k_i} > \tau\right); \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

$q\_time_{k_i}$ and $q\_depth_{k_i}$ are the queuing time and queue depth of the packet p_{k_i} of $flow_i$ passes through the queue. p_{k_i} can be either the packet size or the number of packets. The $flow_i$'s activity during congestion is actually the flow size of $flow_i$ in the congested state.

Definition 4: Active flows during congestion. Given a threshold α, $flow_i$ is considered to be an active flow during congestion in the time range $(t_s, t_e]$ if $activity_{flow_i}$ is greater than the threshold α.

As with estimating *culprit flows for congestion*, *FGQD* also uses Time-Windows to identify the *active flows during congestion*. The per-packet procedure in egress pipeline is shown in Algorithm 1. The inputs to the algorithm include a threshold δ for queue backlog contribution, an active flow estimation threshold α, and thresholds τ and γ for determining the congestion state. When a packet arrives at the egress pipeline, first, *FGQD* gets the queue information using the queuing metadata provided by the P4 switch to get the timestamp, queue depth, and queuing delay of the packet as it passes through the queue. Next, *FGQD* calculates the index of Time-Windows based on the timestamp of the packet. Based on the *clean_index*, *FGQD* performs the cleaning operation on the corresponding window. Then, *FGQD* reads and updates the queue depth record to compute the change in queue depth of the packet. Additionally, *FGQD* determines whether the queue is in congestion when the packet passes through, based on the values of *qdepth* and *qtimedelta*, and counts the $f(\cdot)$. Finally, *FGQD* reads and writes the corresponding window based on *access_index* to measure *the flow's contribution to the queue block* and *the flow's activity during congestion*. If a flow is found as both the *culprit flow for congestion* and the *active flow during congestion*, *FGQD* uses metadata to carry *the flow's contribution to the queue backlog* and *the flow's activity during congestion*, then recirculates the packet to the ingress pipeline. Otherwise, the packet is forwarded normally.

The operation details for cleanWindow (line 7) and accessWindow (lines 15-16) are given in Section IV-B. The conditional statements for determining whether the queue is congested (lines 11-14), and for identifying whether a flow is both *culprit flow for congestion* and *active flow during*

Algorithm 1: Procedure in Egress

```
Input: Threshold $\delta$ for culprit flow estimation, threshold
       $\alpha$ for active flow estimation, congestion
       thresholds $\tau$ and $\gamma$.
1  for packet arriving at Egress Pipeline do
2      timestamp ← queueing metadata;
3      qdepth ← queueing metadata;
4      qtimedelta ← queueing metadata;
5      access_index ← getAccessIndex(timestamp);
6      clean_index ← getCleanIndex(timestamp);
7      cleanWindow(clean_index);
8      last_qdepth ← readRegister(qdepth_record);
9      qdepth_record ← writeRegister(qdepth);
10     Δqdepth ← qdepth - last_qdepth;
11     if qdepth > $\gamma$ or qtimedelta > $\tau$ then
12         $f(\cdot) = 1$;
13     else
14         $f(\cdot) = 0$;
15     contribution ← accessWindow(access_index, Δ
       qdepth);
16     activity ← accessWindow(access_index, $f(\cdot)$);
17     if contribution > $\delta$ and activity > $\alpha$ then
18         recirculate the packet;
19     else
20         forward the packet;
```

```
action set_cnt_activity(bit<32> cnt) {
    meta.cnt_activity = cnt;
}

table tb_cnt_activity {
    key = {
        meta.qdepth:     range;
        meta.qtimedelta: range;
    }
    actions = {
        set_cnt_activity;
    }
    default_action = set_cnt_activity(0);
}
```

Listing 1. P4 pseudocode for $f(\cdot)$.

```
action recirculate_packets() {
    ...
}

table tb_queue_diagnosis {
    key = {
        meta.contribution: range;
        meta.activity:     range;
    }
    actions = {
        recirculate_packets;
        NoAction;
    }
    default_action = NoAction();
}
```

Listing 2. P4 pseudocode for queue diagnosis.

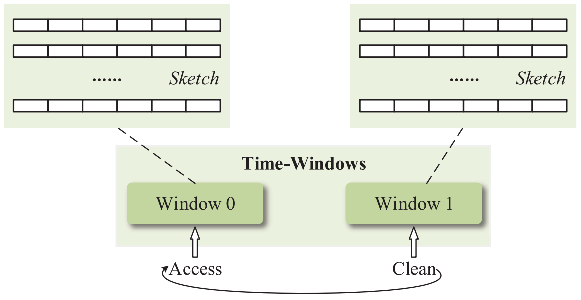


Fig. 3. The structure of Time-Windows.

congestion (lines 17-20), are implemented using two tables, whose P4 pseudocode is shown in Listing 1 and Listing 2.

B. Time-Windows

1) The Structure of Time-Windows: Fig. 3 shows the structure of Time-Windows. Time-Windows consist of two windows, where one window performs the access operation to fetch the recorded data and record the data to be measured, and another window performs the clean operation to prepare for the next round of access. Time-Windows measure and query the flow's information in the time range $(t_s, t_e]$, and when it reaches the following time range $(t_e, 2t_e - t_s]$, the index of the windows that perform the access and clean operations move backward cyclically, thus realizing the continuous measurement of the flow.

The goal of *FGQD* is to find malicious flows for CRAs that are not only the *culprit flows for congestion* in the queue, but are also *active flows during congestion*. Therefore, any approximate data structure that supports write and read

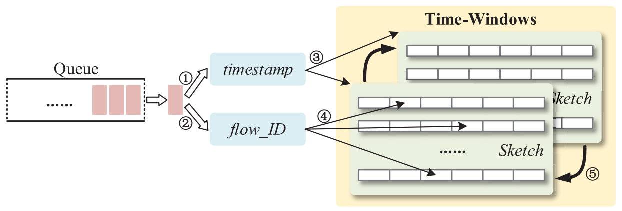


Fig. 4. Per-packet Procedure of Time-Windows.

operations and provides reasonable accuracy can be employed by Time-Windows to enable the detection of malicious flows. Sketch [40], [41], as an approximate measurement solution, provides good estimation accuracy under computational and memory constraints. To address the memory limitation, we employ Count-Min Sketch (CMS) [40] in *FGQD*, since it is easy to implement in the data plane. CMS may suffer from overestimation errors due to the hash collision, the margin of which depends on the size of the chosen structure.

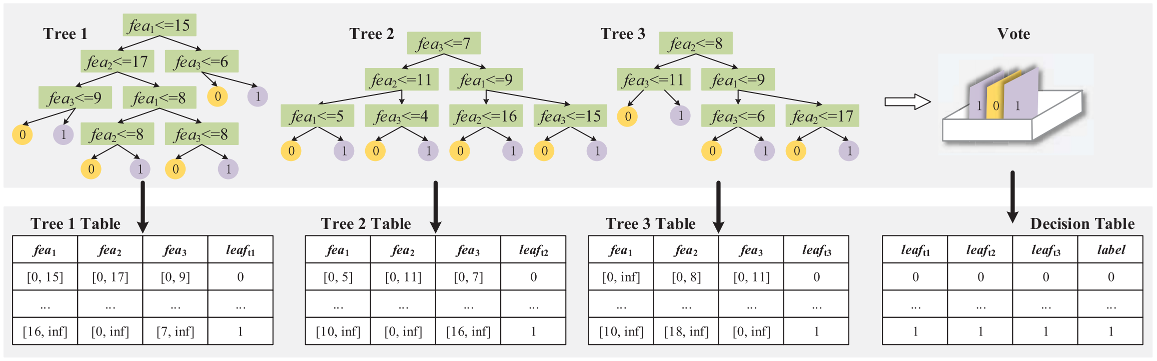
In *FGQD*, each CMS consists of R register array of length C, each register with W bit width, and then each CMS will consume $R \times C \times W$ bit and a Time-Windows consumes $2 \times R \times C \times W$ bit registers. To detect malicious flows for CRAs, *FGQD* needs to perform the estimation of *culprit flows for congestion* and *active flows during congestion*, using two Time-Windows to measure *the flow's contribution to the queue backlog* and *the flow's activity during congestion*, respectively, and thus *FGQD* consumes registers with a size of $4 \times R \times C \times W$ bit.

2) Per-Packet Procedure of Time-Windows: To determine whether a packet is part of a malicious flow for CRAs, *FGQD* uses Time-Windows to maintain information about the flow to which the packet belongs. Fig. 4 shows the per-packet procedure of Time-Windows. There are five steps from the time the packet arrives at the egress pipeline to the time the *FGQD* completes the recording and querying, which are: ①*Get timestamp from the queueing metadata*; ②*Get flow_ID from the packet header*; ③*Calculate Time-Windows index*; ④*Calculate sketch index*; ⑤*Perform access and clean operations on sketches*.

①*Get timestamp from the queueing metadata*. To estimate the *culprit flows for congestion* and the *active flows during congestion* in the time range $(t_s, t_e]$, *FGQD* uses the packet's timestamp information to determine the time range to which the packet belongs. Specifically, *FGQD* uses the queuing metadata to obtain the packet's timestamp information. In bmv2, the packet's timestamp is measured in microseconds and is set when the packet first enters the queue.

②*Get flow_ID from the packet header*. *FGQD* uses the five-tuple <*srcIP*, *dstIP*, *srcPort*, *dstPort*, *protocol*> from the packet header as the *flow_ID* for its corresponding flow, noting that other fields can also be used as the *flow_ID* if needed, such as the three-tuple <*srcIP*, *dstIP*, *protocol*>.

③*Calculate Time-Windows index*. Time-Windows consist of two windows, and *FGQD* uses the timestamp from the queueing metadata to calculate the window index, determining which windows are used for access and clean operations. The index for the access window is $(timestamp \div (t_e - t_s)) \bmod 2$, and the calculation of the index for the clean window is



fea_1	fea_2	fea_3	$leaf_{t1}$
[0, 15]	[0, 17]	[0, 9]	0
...	...	...	...
[16, inf]	[0, inf]	[7, inf]	1

fea_1	fea_2	fea_3	$leaf_{t2}$
[0, 5]	[0, 11]	[0, 7]	0
...	...	...	...
[10, inf]	[0, inf]	[16, inf]	1

fea_1	fea_2	fea_3	$leaf_{t3}$
[0, inf]	[0, 8]	[0, 11]	0
...	...	...	...
[10, inf]	[18, inf]	[0, inf]	1

$leaf_{t1}$	$leaf_{t2}$	$leaf_{t3}$	$label$
0	0	0	0
...	...	...	...
1	1	1	1

Fig. 5. Example of mapping RF model with 3 trees.

$(timestamp \div (t_e - t_s) + 1) \mod 2$. Due to the computation constraint, P4 does not support division and modulus operations directly. To overcome this limitation, *FGQD* implements division via right shift and modulus using bit slicing. Specifically, *FGQD* sets the window's time range $t_e - t_s$ to be an integer powers of 2 (e.g., 2^n), allowing $(timestamp \div (t_e - t_s))$ to be replaced with (*timestamp* $>>$ *n*). Consequently, *access_index* is calculated as (*timestamp* $>>$ *n*) & 1, and the computation for *clean_index* follows a similar approach.

④*Calculate sketch index. FGQD* determines the slot position for performing access operation on the register in the sketch by hashing *flow_ID*. Due to hash collisions, different *flow_ID*s may be mapped to the same slot, leading to overestimation errors.

⑤*Perform access and clean operations on sketches.* Each register can only be accessed once as the packet travels through the switching pipeline [27], [28]. For the access window, *FGQD* utilizes a single access register to first read the recorded data and then write updated data back to the sketch. To clean the sketch, one approach is to use the Runtime API from the control plane to clear the registers. However, this method is challenging to implement effectively in practice due to operational delays. *FGQD* addresses this problem by introducing an additional clean window. Specifically, when a packet reaches the egress pipeline, a write-zero operation is performed on each column of the sketch in the clean window. If the register array forming the sketch has a length of *C*, the sketch will be fully cleared after *C* packets pass through it within the time range $(t_e - t_s)$.

3) Error Analysis: We now analyze the estimation error of the Time-Windows due to the hash collision. In *FGQD*, we use the CMS [40] with *R* rows and *C* columns in a window, where $R = \lceil e \ / \ \varepsilon \rceil$ and $C = \lceil ln(1/\delta) \rceil$. For CMS, it can achieve ε additive error with failure probability δ, which means that with $1-\delta$ probability, a query with ground truth flow size w will return an estimate w' that satisfies $w \leq w' \leq w + \varepsilon W_{CMS}$, where W_{CMS} is the total size of all the flows inserted into the CMS.

C. In-Network RF

Network traffic is influenced by various factors, such as user behavior, device type, and application service. During network traffic surges caused by factors other than malicious attacks, benign flows may also trigger queue congestion and could potentially be identified by *FGQD* as *culprit flows for congestion* and *active flows during congestion*. Consequently, detecting malicious flows of CRAs solely based on queue diagnosis in the egress pipeline may lead to false positives for benign flows.

Algorithm 2: Procedure in Parser and Ingress

```
1  for packet arriving at Parser do
2      flow_ID, packet_fea ← packet's header;
3  for packet arriving at Ingress Pipeline do
4      instance_type ← standard metadata;
5      if instance_type is INGRESS_RECIRC then
6          queue_fea ← metadata;
7          label ← RF(packet_fea, queue_fea);
8          if label is 1 then
9              report flow_ID;
10         else
11             forward the packet;
12     else
13         if flow_ID in blacklist then
14             perform mitigation actions;
15         else
16             forward the packet;
```

To this, we employ the in-network RF model to further improve the detection accuracy and reduce the false positives of benign flows, which is deployed in the switch ingress pipeline and consists of multiple tree tables and a decision table. Fig. 5 shows an example of mapping RF models with 3 trees. Each tree table represents a decision tree model, where each rule in the tree table corresponds to a decision path from the root to a leaf node in the decision tree. The action triggered by the rule depends on the type of the leaf node in the decision tree. Based on the decision outcomes from the tree tables, the decision table makes the final classification. Overall, tree tables and decision table implement the decision-making process of the RF model directly within the data plane.

The per-packet procedure in parser and ingress pipeline is shown in Algorithm 2 and P4 pseudocode for in-network RF

```
action setClass0(bit<1> class) {
    meta.class0 = class;
}

table tb_dt0 {
    key = {
        meta.contribution:   range;
        meta.activity:       range;
        hdr.ipv4.ihl:        range;
        hdr.ipv4.diffserv:   range;
        ...
    }
    actions = {
        setClass0;
        NoAction;
    }
    default_action = NoAction();
}
...

action report_maliciousFlowID() {
    ...
}

table tb_vote {
    key = {
        meta.class0: exact;
        meta.class1: exact;
        meta.class2: exact;
    }
    actions = {
        report_maliciousFlowID;
        NoAction;
    }
    default_action = NoAction();
}
```

Listing 3. P4 pseudocode for in-network RF.

is shown in Listing 3. When a packet reaches the switch parser, *FGQD* parses the packet's header to extract the *flow_ID* and packet features. Upon arriving at the switch ingress pipeline, *FGQD* first determines whether the packet is a recirculated packet. If it is, the in-network RF model is invoked for classification. If the classification result identifies the packet as belonging to the CRAs, its *flow_ID* is reported. Otherwise, the packet is forwarded normally. For packets forwarded normally to the switch, *FGQD* checks whether the *flow_ID* is found in the blacklist. If it is, mitigation measures are applied, otherwise, the packet is forwarded normally.

V. Experimental Evaluation

In this section, we present the experimental evaluation of the *FGQD* prototype, including the experimental setup (Section V-A), detection performance evaluation (Section V-B), response time evaluation (Section V-C), and comparison of different detection systems (Section V-D).

A. Experimental Setup

We prototype *FGQD* using P4 and Python, and conduct experiments on the software-based testbed with mininet[2] and bmv2.[3]

Topology. We set up a dumbbell topology consisting of four hosts (h1, h2, h3, and h4) and two switches (s1 and s2). h1 and h2 are connected to s1, h3 and h4 are connected to s2, and the link between s1 and s2 is the bottleneck link. Due to the performance limitations of bmv2,[4] we set the bandwidth of the bottleneck link to 50*Mbps* and the bandwidth of other links to 100*Mbps*. h1 sends background traffic to h3 via s1 and s2, and h2 sends attack traffic through s1 and s2 to h4. The *FGQD* prototype is deployed on s1, and the queue depths on s1 and s2 are set to the default value of 64.

Traffic Generation. We utilize the WIDE dataset on January 22, 2023 as background traffic to emulate normal network traffic. Since the WIDE dataset does not contain CRAs, we assume its traffic is all benign. The average rate of the original traffic is 357.79*Mbps* [42]. To match the link rate, we replay the original traffic at a rate of 0.1x using tcpreplay, so that the background traffic would be approximately 70% of the bottleneck link's bandwidth, simulating a busy network scenario. For malicious traffic of CRAs, we implement a script using the Python socket to generate attack traffic for simulating Shrew attacks and use the tool in [43] to simulate Optimistic Ack attacks.[5] To get the training data for the RF model, we use the first 0–50 seconds of WIDE traffic combined with simulated malicious traffic of CRAs to emulate the network and record the traffic data. To evaluate *FGQD*'s performance, we use the 50–100 seconds of WIDE traffic along with simulated malicious traffic of CRAs to emulate the network traffic.

Threshold Selection. *FGQD* uses the threshold δ for culprit flow estimation, threshold α for active flow estimation, and thresholds τ and γ to indicate queue congestion. These thresholds are set based on the 95th percentile of the values under normal network conditions.

Evaluation Metrics. We choose the detection rate (DR), false negative rate (FNR), and false positive rate (FPR) to evaluate *FGQD*'s detection performance comprehensively. The DR is the proportion of actual malicious flows that the *FGQD* correctly identifies. The FNR is the proportion of actual malicious flows incorrectly identified as benign. The sum of the DR and FNR is 1. The FPR is the proportion of actual benign flows incorrectly identified as malicious flows. DR and FNR are used to evaluate *FGQD*'s ability to recognize malicious flows, where the higher the DR and the lower the FNR, the better the ability of *FGQD* to recognize malicious flows for CRAs. FPR is used to evaluate the probability of *FGQD* misidentifying benign flows as malicious flows, where the lower the FPR, the lower the probability of *FGQD* misidentifying benign flows as malicious flows.

B. Detection Performance

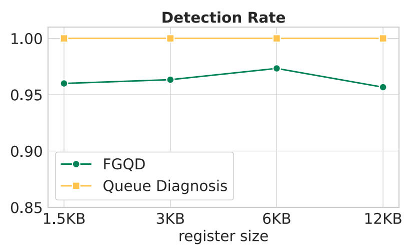
1) Evaluation for Shrew Attacks: Fig. 6 shows the detection performance of *FGQD* for Shrew attacks when Time-Windows consume different register sizes (1.5KB, 3KB, 6KB, and 12KB). The green curve represents the detection performance of *FGQD*, while the yellow curve indicates the detection performance of queue diagnosis. Queue diagnosis

[2]https://github.com/mininet/mininet

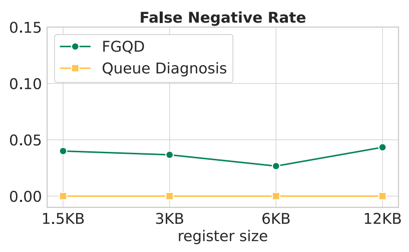
[3]https://github.com/p4lang/behavioral-model

[4]https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md

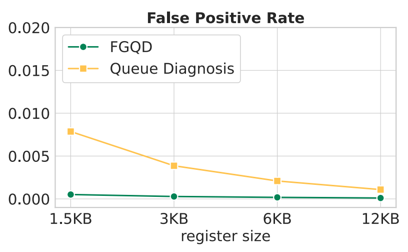
[5]We collect traces of Optimistic Ack attacks in advance using tcpdump and replay the collected traffic when evaluating *FGQD*.



(a)



(b)



(c)

Fig. 6. The detection performance of *FGQD* for Shrew attacks.

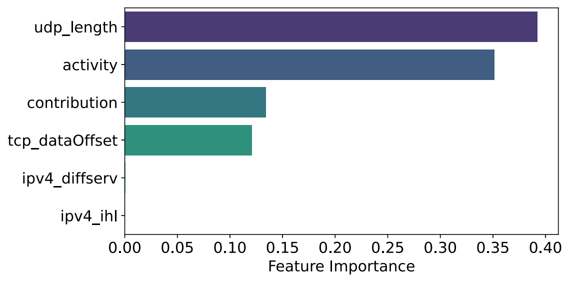


Fig. 7. The feature importance of RF in evaluation for Shrew attacks.

here refers to utilizing only the culprit flow estimation module and active flow estimation module in the egress pipeline, without employing the in-network RF module. When a flow is identified as both a *culprit flow for congestion* and an *active flow during congestion*, queue diagnosis reports it as a malicious flow. In contrast, *FGQD* integrates the in-network RF module to further validate the suspicious flows identified by queue diagnosis. The DR and FNR of queue diagnosis are slightly better than those of *FGQD*, as the in-network RF performs additional screening of suspicious flows identified by queue diagnosis. This verification process may occasionally result in the unintended omission of malicious flows. However, compared to queue diagnosis, *FGQD* significantly reduces the false positive rate. Beyond identifying malicious flows, an equally important objective is to minimize the impact on benign flows. Consequently, the design of *FGQD* prioritizes minimizing the false positives of benign flows.

Time-Windows employ sketches to measure *the flow's contribution to the queue backlog* and *the flow's activity during congestion*, which may be prone to overestimation errors. Such overestimations may inflate these two metrics, potentially leading to false positives for benign flows. With more registers consumed by Time-Windows, the lower the FPR of the queue diagnosis. As seen in Fig. 6(c), FPR is lowest when Time-Windows utilize 12KB of registers. This is because an increased register capacity reduces the likelihood of hash collisions, mitigates overestimation errors, and consequently decreases the occurrence of false positives. Note that, even with a low register consumption (e.g., 1.5KB), Time-Windows manage to maintain a reasonably low FPR, highlighting the effectiveness and robustness of its design.

Fig. 7 shows the feature importance of the RF model in evaluating Shrew attacks. The queue features include the contribution and activity, while the packet features comprise header fields such as udp_length, tcp_dataOffset, ipv4_diffserv, and ipv4_ihl. The contribution feature denotes *the flow's contribution to the queue backlog*, and the activity feature refers to *the flow's activity during congestion*. Among these, the packet feature udp_length has the highest importance score, with the two queue features also ranking high in importance. In Shrew attacks, attackers aim to maximize the destructiveness of the attack while maintaining a low average traffic rate to enhance stealthiness. To achieve this, they often use lightly loaded packets to seize network bandwidth, rather than heavily loaded packets. In contrast, normal traffic, primarily intended to carry information for network communication, typically has a packet load that is not excessively small. This distinction explains why the packet feature udp_length holds a high importance score in the RF model for detecting Shrew attacks. Furthermore, causing queue congestion is a critical characteristic of Shrew attacks, which justifies the high importance scores of the two queue features.

2) Evaluation for Optimistic Ack Attacks: Fig. 8 shows the detection performance of *FGQD* for Optimistic Ack attacks when Time-Windows consume different register sizes (1.5KB, 3KB, 6KB, and 12KB). It can be seen that the detection performance of *FGQD* remains well regardless of the amount of register consumption, maintaining a DR for Optimistic ACK attacks above 95% and a FPR below 1%, which demonstrates the effectiveness of *FGQD*. Additionally, the register size allocated to Time-Windows has no significant impact on the DR and FNR for detecting Optimistic Ack attacks. With an increase in the register size assigned to Time-Windows, the probability of hash collisions decreases, leading to a reduction in the FPR.

Fig. 9 shows the feature importance of the RF model in evaluating Optimistic Ack attacks. Among these features, the queue feature contribution has the highest importance score. In Optimistic Ack attacks, the receiver forges ACK packets to deceive the sender to accelerate the data transmission, thus causing congestion in the network. Thus, the contribution feature which reflects the congestion condition of the queue plays a critical role in identifying Optimistic Ack attacks, which also underscores the effectiveness of queue diagnosis in detecting Optimistic Ack attacks.

C. Evaluation for Response Time

An essential objective of defense systems is to respond promptly to attacks to mitigate their impact. The faster *FGQD* responds, the less damage CRAs can inflict on the network. The response time here refers to the interval between *FGQD*

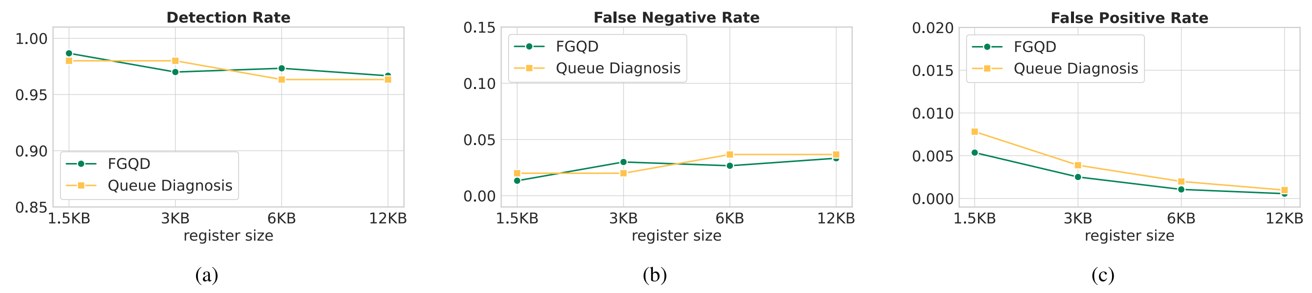


Fig. 8. The detection performance of *FGQD* for Optimistic Ack attacks.

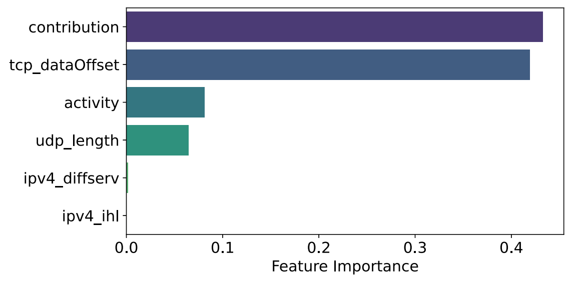
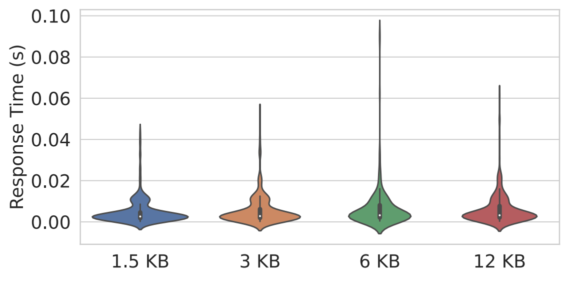


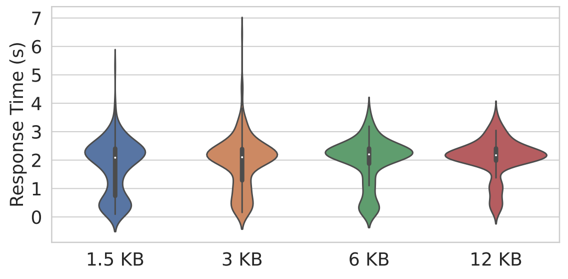
Fig. 9. The feature importance of RF in evaluation for Optimistic Ack attacks.

TABLE I
THE COMPARISON OF DIFFERENT METHOD'S RESPONSE LATENCY FOR SHREW ATTACKS

Methods	Deployment	Response Latency
Spectral feature [16]	\	5s
Small signal model [17]	\	30s
Softguard [37]	Control plane	12-24s
P&F [38]	Control plane	9.39s
PeakSAX [18]	Control plane	1-21s
GASF-IPP [39]	Control plane	6.77s
FGQD	Data plane	5.70ms



(a) Response latency for Shrew attacks.



(b) Response latency for Optimistic Ack attacks.

Fig. 10. Response latency of *FGQD*.

receiving the attack traffic and successfully blocking it. To measure it, we capture traffic as a pcap file and the time span during which the attack flow appears in the file is considered *FGQD*'s response time to CRAs. Considering the false negatives of malicious flows, we measure only the duration of malicious flows detected by *FGQD* in the pcap file.

The response latency to attacks is a critical performance metric for evaluating the effectiveness of a defense system, and we assess the response performance of *FGQD* concerning both Shrew attacks and Optimistic Ack attacks. Fig. 10 shows the response latency of *FGQD*, where Fig. 10(a) is the response latency for Shrew attacks and Fig. 10(b) refers to the response latency for Optimistic Ack attacks. For Shrew attacks, *FGQD* responds within a few milliseconds, swiftly blocking the malicious flow's packets with an average response time of approximately 5.70 milliseconds. For Optimistic Ack attacks, the average response time of *FGQD* is about 1.93 seconds. The reason *FGQD*'s response time to Optimistic Ack attacks is significantly longer than its response to Shrew attacks lies in the nature of the Optimistic Ack attacks. In Optimistic Ack attacks, the traffic initially appears normal and does not cause network congestion, effectively behaving as benign traffic during this phase. It is only after the malicious receiver forges Ack packets that the sender increases its transmission speed, leading to network congestion. As a result, some time may pass before *FGQD* identifies and responds to the malicious traffic. Note that this does not imply *FGQD* is slow to respond to Optimistic Ack attacks, but rather that the attack traffic behaves as normal traffic in its early stages. Overall, *FGQD* demonstrates its ability to respond promptly to malicious flows of CRAs, effectively mitigating the attacks.

Table I shows the comparison of response latency for different detection methods for Shrew attacks. Spectral feature [16] and small signal model [17] perform sophisticated time-frequency analysis of network traffic using sampling techniques to detect Shrew attacks, which inevitably results in considerable response latency. Softguard [37], P&F [38], PeakSAX [18], and GASF-IPP [39] are systems deployed in the SDN control plane. In these systems, the controller is responsible for collecting network traffic, making decisions, and installing defense policies on the switches, which inevitably introduces significant latency, and the response latency of these systems is at the level of seconds or even tens of seconds. Compared to the above methods, *FGQD* has a low response latency of approximately 5.70 milliseconds, as

TABLE II
FGQD VERSUS OTHER SYSTEMS FOR SHREW ATTACKS

Methods	**DR** (%)	**FNR** (%)	**FPR** (%)	**Register** (KB)
Conquest [10]	98.000	2.000	0.048	6
Henna [44]	71.667	28.333	1.070	/
Hashpipe [45]	83.000	17.000	0.474	51.375
FGQD	97.333	2.667	0.018	6

TABLE III
FGQD VERSUS OTHER SYSTEMS FOR OPTIMISTIC ACK ATTACKS

Methods	**DR** (%)	**FNR** (%)	**FPR** (%)	**Register** (KB)
Conquest [10]	76.667	23.333	0.033	6
Henna [44]	100.000	0.000	0.029	/
Hashpipe [45]	88.000	12.000	0.471	51.375
FGQD	97.333	2.667	0.106	6

it is deployed directly on the data plane, enabling it to quickly respond to Shrew attacks.

D. FGQD Versus Other Systems

We compare *FGQD* with Conquest [10], Henna [44], and Hashpipe [45].

- Conquest [10] employs a snapshot-based data structure to track the information of queues in the data plane, thereby identifying flows that make significant contributions to the queue. We have implemented the Conquest prototype on the software-testbed, which consumes about 6KB of register size.
- Henna [44] is an in-network classifier that relies exclusively on packet features. It consists of a hierarchical two-stage model, with a RF deployed in the ingress pipeline and a decision tree deployed in the egress pipeline. The authors have provided the code[6] for the Tofino version, based on which we implement the bmv2 version, where the tree model is implemented similarly to the tree table of *FGQD*.
- Hashpipe [45] is a flow size measurement algorithm deployed on the programmable data plane. Based on the code[7], we have implemented the Hashpipe prototype on the software-based testbed, which consumes about 51.375KB of register size.

Table II shows the comparison of different systems for detecting Shrew attacks. In *FGQD*, Time-Windows utilize 6KB of registers, while Conquest [10] also uses 6KB of registers, and Hashpipe [45] consumes 51.375KB of registers. As shown in the table, *FGQD* achieves a DR, FNR, and FPR of 97.333%, 2.667%, and 0.018%. Conquest [10] has a DR, FNR, and FPR of 98.000%, 2.000%, and 0.048%, and Henna [44] has a DR, FNR, and FPR of 71.667%, 28.333%, and 1.070%, and Hashpipe [45] has a DR, FNR, and FPR of 83.000%, 17.000%, and 0.474%, respectively. Conquest [10] detects attack flows by identifying significant contributors to the queue backlog, whereas Shrew attacks create congestion by sending bursty traffic, hence Conquest [10] can effectively detect these attacks. The primary pattern of Shrew attacks lies in their traffic behavior, Henna does not perform well in detecting these attacks since it uses only packet features. Hashpipe [45] is mainly used for detecting heavy hitter flows. Although Shrew attacks send bursty traffic, they exhibit relatively long silent periods, which also results in suboptimal detection performance for Hashpipe. In contrast, *FGQD* utilizes queue diagnosis, which is similar to Conquest, and thus can achieve about the same as Conquest's DR and FNR. Moreover, *FGQD* further reduces false positives by incorporating packet features and in-network RF, thereby attaining a lower FPR than Conquest. Beyond identifying malicious flows, an equally important objective is to minimize the impact on benign flows. Therefore, the detection system should strive to accurately identify malicious flows while reducing false positives for benign flows. Among the four detection systems, *FGQD* achieves the lowest FPR.

[6]https://github.com/nds-group/Henna
[7]https://github.com/vibhaa/hashpipe

Table III shows the comparison of different systems for detecting Optimistic Ack attacks. *FGQD* has a DR, FNR, and FPR of 97.333%, 2.667%, and 0.106%, while Conquest [10] achieves a DR, FNR, and FPR of 76.667%, 23.333%, and 0.033%. Henna [44] has a DR, FNR, and FPR of 100.000%, 0.000%, and 0.029%. Hashpipe's [45] DR, FNR, and FPR are 88.000%, 12.000%, and 0.471%, respectively. Among these four systems, Henna [44] has the highest DR and the lowest FNR and FPR. In the experiments, we employ a tool to generate Optimistic Ack attack traffic. The traffic produced by this tool exhibits relatively uniform packet header fields, allowing Henna, which relies on packet features, to easily learn all the header field patterns of Optimistic Ack attack traffic. This may lead to an overestimation of Henna's detection effectiveness. In future work, it will be necessary to identify real-world datasets for Optimistic Ack attacks or use attack tools with more diverse patterns for evaluation. Optimistic Ack attacks occur when a malicious receiver forges ACK packets, prompting the sender to accelerate its traffic. However, this increase does not elevate the total traffic volume to heavy hitter levels, so Hashpipe's detection performance for it is not very good. *FGQD*'s queue diagnosis monitors both *Culprit flows for congestion* and *active flows during congestion*, and therefore also achieves effective detection of Optimistic Ack attacks. Overall, *FGQD* demonstrates strong performance in detecting both Shrew attacks and Optimistic Ack attacks, proving it to be an effective and flexible solution.

VI. DISCUSSION

In this section, we discuss the *FGQD* from three perspectives, which are detection for other CRAs (Section VI-A), potential integration with AI-based intrusion detection systems (Section VI-B), and potential application in 5G networks, Internet of Things (IoT), and cloud security (Section VI-C).

A. Detection for Other CRAs

Beyond Shrew attacks and Optimistic Ack attacks, other CRAs include Pulsing DoS attacks [46] and Link-Flooding Attacks [8], etc. The Pulsing DoS attack methodology closely resembles that of Shrew attacks, as both leverage traffic bursts to consume link bandwidth and trigger congestion. Thus, *FGQD* is well-suited for detecting Pulsing DoS attacks through similar mechanisms used for identifying Shrew attacks. Link-Flooding Attacks represent a broader category encompassing various attack types such as Crossfire, CoreMelt, and CrossPath attacks. These attacks often employ sophisticated techniques. For instance, Rolling Crossfire can dynamically alter target links, resulting in traffic patterns that appear both legitimate and highly adaptive [8]. Due to this complexity, effective defense against Link-Flooding Attacks typically requires more comprehensive protection systems. Our proposed *FGQD* scheme shows promise as a valuable component that could be integrated into such advanced defense architectures.

Beyond evaluating the detection performance of *FGQD* against other CRAs, it is also important to evaluate its effectiveness in real-world scenarios. However, the scarcity of public network datasets containing CRAs trace data, and our lack of access to hardware devices such as Tofino switches, pose significant challenges. We will further investigate and validate *FGQD*'s ability to detect CRAs under real-world scenarios in future work.

B. Potential Integration With AI-Based Intrusion Detection System

AI-based Intrusion Detection System (AI-IDS) shifts traditional rule and signature-based detection into intelligent models, not only improving its ability to recognize security threats but also delivering significant gains in real-time responsiveness, scalability, and interpretability. As a result, it has become a cornerstone of modern network security defenses. Nevertheless, current in-network AI-IDS still faces challenges such as limited data plane resources, constrained feature and semantic richness, and the complexity of deployment and maintenance [27], [47], [48]. *FGQD* leverages the most widely adopted in-network machine-learning model (in-network RF) to reduce false positives on benign flows effectively. It introduces two novel queue-based features (*the flow's contribution to the queue backlog* and *the flow's activity during congestion*) that extend the analytical feature set available to in-network AI-IDS. Moreover, *FGQD*'s Time-Windows enable efficient recording of flow features within the resource-constrained data plane. In future work, we will further explore the prospects and potential of integrating *FGQD* into AI-IDS.

C. Potential Application in 5G Networks, IoT, and Cloud Security

In addition to traditional Internet infrastructure, 5G networks, IoT, and cloud platforms are all vulnerable to CRAs. IoT devices are particularly susceptible due to their lack of robust security measures, rendering them ideal entry points for CRAs. Consequently, defending against CRAs across 5G networks, IoT, and cloud platforms is an urgent priority. In future work, we will further investigate the potential and prospects of applying *FGQD* within 5G networks, IoT, and cloud platforms.

VII. Related Work

The detection methods for Shrew attacks can be categorized into time-frequency analysis methods [16], machine learning methods [38], and queue analysis methods [49]. Time-frequency analysis methods use time-frequency analysis techniques to detect attacks. Chen and Hwang [16] proposed a collaborative detection and filtering method using Discrete Fourier Transform and Hypothesis Testing for spectral analysis of network traffic to detect Shrew attacks. Tang et al. [38] proposed P&F, deployed in the SDN control plane, which uses machine learning models to detect Shrew attacks and uses time-frequency analysis to locate the source and victim of the attack. Queue analysis methods detect Shrew attacks by analyzing network queues. Yue et al. [49] proposed a feedback control model to describe the congestion control process and designed a queue distribution model to extract queue features, combining a simple distance-based method with an adaptive threshold algorithm to detect bursts of Shrew attacks.

The detection methods for Optimistic Ack attacks can be categorized into methods that acknowledge packet authentication [15] and protocol state monitoring [19]. Sherwood et al. [15] proposed to randomly discard data segments at the sender and recognize the receiver as a misbehaving receiver when the sender receives an optimistic ACK against one of the intentionally discarded segments. Based on the emerging programmable data plane, Laraba et al. [19] proposed the Extended Finite State Machine (EFSM) abstraction to monitor stateful protocols in the data plane to mitigate TCP protocol abuses including Optimistic Ack attacks.

VIII. Conclusion

In this paper, we develop *FGQD*, a defense system against CRAs deployed on the programmable switch, capable of real-time monitoring of the queue and network traffic. Particularly, *culprit flows for congestion* and *active flows during congestion* enable tracking flows culpable for congestion formation and flows exhibiting abnormal activity during congestion. The approximate data structure Time-Windows implement the monitoring of *culprit flows for congestion* and *active flows during congestion* in the data plane, overcoming resource and computational constraints on the programmable data plane. Additionally, the in-network RF can effectively reduce false positives for benign flows. Experimental evaluation shows that *FGQD* has achieved effective detection and rapid response to CRAs, demonstrating excellence in effectiveness, flexibility, and real-time performance.

References

[1] Z. Li, Y. Hu, L. Tian, and Z. Lv, "Packet rank-aware active queue management for programmable flow scheduling," *Comput. Netw.*, vol. 225, Apr. 2023, Art. no. 109632.

[2] S. Arslan, Y. Li, G. Kumar, and N. Dukkipati, "Bolt: Sub-RTT congestion control for ultra-low latency," in *Proc. 20th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Boston, MA, USA, Apr. 2023, pp. 219–236.

[3] D. Tang, Y. Yan, C. Gao, W. Liang, and W. Jin, "LtRFT: Mitigate the low-rate data plane DDoS attack with learning-to-rank enabled flow tables," *IEEE Trans. Inf. Forensics Security*, vol. 18, pp. 3143–3157, 2023.

[4] L. Yu, J. Sonchack, and V. Liu, “Cebinae: Scalable in-network fairness augmentation,” in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 219–232.

[5] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru, “Automated attack discovery in TCP congestion control using a model-guided approach,” in *Proc. 25th Ann. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, San Diego, CA, Feb. 2018, p. 95.

[6] D. Tang, R. Dai, Y. Yan, K. Li, W. Liang, and Z. Qin, “When SDN meets low-rate threats: A survey of attacks and countermeasures in programmable networks,” *ACM Comput. Surv.*, vol. 57, no. 4, pp. 1–32, Dec. 2024.

[7] R. Xie et al., “Disrupting the SDN control channel via shared links: Attacks and countermeasures,” *IEEE/ACM Trans. Netw.*, vol. 30, no. 5, pp. 2158–2172, Oct. 2022.

[8] H. Zhou, S. Hong, Y. Liu, X. Luo, W. Li, and G. Gu, “Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches,” in *Proc. IEEE Symp. Security Privacy (SP)*, 2023, pp. 3178–3192.

[9] “Netease’s servers crashed for 9 hours.” [Online]. Available: https://www.docin.com/p-2167939755.html

[10] X. Chen et al., “Fine-grained queue measurement in the data plane,” in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, 2019, pp. 15–29.

[11] A. Kuzmanovic and E. W. Knightly, “Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants,” in *Proc. Conf. Appl., Technol., Archit., Protoc. Comput. Commun.*, 2003, pp. 75–86.

[12] M. Yue, J. Li, Z. Wu, and M. Wang, “High-potency models of LDoS attack against CUBIC + RED,” *IEEE Trans. Inf. Forensics Security*, vol. 16, pp. 4950–4965, 2021.

[13] M. Yue, M. Wang, and Z. Wu, “Low-high burst: A double potency varying-RTT based full-buffer shrew attack model,” *IEEE Trans. Dependable Secure Comput.*, vol. 18, no. 5, pp. 2285–2300, Sep./Oct. 2021.

[14] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson, “TCP congestion control with a misbehaving receiver,” *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 5, pp. 71–78, Oct. 1999.

[15] R. Sherwood, B. Bhattacharjee, and R. Braud, “Misbehaving TCP receivers can cause internet-wide congestion collapse,” in *Proc. 12th ACM Conf. Comput. Commun. Secur.*, 2005, pp. 383–392.

[16] Y. Chen and K. Hwang, “Collaborative detection and filtering of shrew DDoS attacks using spectral analysis,” *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1137–1151, 2006.

[17] W. Zhi-Jun, Z. Hai-Tao, W. Ming-Hua, and P. Bao-Song, “MSABMS-based approach of detecting LDoS attack,” *Comput. Secur.*, vol. 31, no. 4, pp. 402–417, 2012.

[18] D. Tang, Z. Zheng, X. Wang, S. Xiao, and Q. Yang, “PeakSAX: Real-time monitoring and mitigation system for LDoS attack in SDN,” *IEEE Trans. Netw. Service Manag.*, vol. 20, no. 3, pp. 3686–3698, Sep. 2023.

[19] A. Laraba, J. François, S. R. Chowdhury, I. Chrisment, and R. Boutaba, “Mitigating TCP protocol misuse with programmable data planes,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 1, pp. 760–774, Mar. 2021.

[20] A. AlSabeh, J. Khoury, E. Kfoury, J. Crichigno, and E. Bou-Harb, “A survey on security applications of P4 programmable switches and a STRIDE-based vulnerability assessment,” *Comput. Netw.*, vol. 207, Apr. 2022, Art. no. 108800.

[21] A. Liatifis, P. Sarigiannidis, V. Argyriou, and T. Lagkas, “Advancing SDN from OpenFlow to P4: A survey,” *ACM Comput. Surv.*, vol. 55, no. 9, pp. 1–37, Jan. 2023.

[22] P. Bosshart et al., “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[23] G. Li et al., “IMap: Fast and scalable in-network scanning with programmable switches,” in *Proc. 19th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Renton, WA, USA, Apr. 2022, pp. 667–681.

[24] D. Tang, B. Liu, K. Li, S. Xiao, W. Liang, and J. Zhang, “PLUTO: A robust LDoS attack defense system executing at line speed,” *IEEE Trans. Dependable Secure Comput.*, vol. 22, no. 3, pp. 2855–2872, May/Jun. 2025.

[25] M. Zhang et al., “NetHCF: Filtering spoofed IP traffic with programmable switches,” *IEEE Trans. Dependable Secure Comput.*, vol. 20, no. 2, pp. 1641–1655, Mar./Apr. 2023.

[26] Y. Lei, L. Yu, V. Liu, and M. Xu, “PrintQueue: Performance diagnosis via queue measurement in the data plane,” in *Proc. ACM SIGCOMM Conf.*, New York, NY, USA, 2022, pp. 516–529.

[27] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, “An efficient design of intelligent network data plane,” in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, Anaheim, CA, USA, Aug. 2023, pp. 6203–6220.

[28] Y. Dong et al., “HorusEye: A realtime IoT malicious traffic detection framework using programmable switches,” in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, Anaheim, CA, USA, Aug. 2023, pp. 571–588.

[29] Q. Zhang, V. Liu, H. Zeng, and A. Krishnamurthy, “High-resolution measurement of data center microbursts,” in *Proc. Internet Meas. Conf.*, 2017, pp. 78–85.

[30] Y. Xiang, K. Li, and W. Zhou, “Low-rate DDoS attacks detection and traceback by using new information metrics,” *IEEE Trans. Inf. Forensics Security*, vol. 6, pp. 426–437, 2011.

[31] A. d. S. Ilha, A. C. Lapolli, J. A. Marques, and L. P. Gaspary, “Euclid: A fully in-network, P4-based approach for real-time DDoS attack detection and mitigation,” *IEEE Trans. Netw. Service Manag.*, vol. 18, no. 3, pp. 3121–3139, Sep. 2021.

[32] M. L. Pacheco, M. V. Hippel, B. Weintraub, D. Goldwasser, and C. Nita-Rotaru, “Automated attack synthesis by extracting finite state machines from protocol specification documents,” in *Proc. IEEE Symp. Security Privacy (SP)*, 2022, pp. 51–68.

[33] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, “Automata-based automated detection of state machine bugs in protocol implementations,” in *Proc. 30th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.

[34] C. Fu, Q. Li, M. Shen, and K. Xu, “Frequency domain feature based robust malicious traffic detection,” *IEEE/ACM Trans. Netw.*, vol. 31, no. 1, pp. 452–467, Feb. 2023.

[35] C. Fu, Q. Li, and K. Xu, “Detecting unknown encrypted malicious traffic in real time via flow interaction graph analysis,” in *Proc. 30th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–18.

[36] D. Tang, R. Dai, C. Zuo, J. Chen, K. Li, and Z. Qin, “A low-rate DoS attack mitigation scheme based on port and traffic state in SDN,” *IEEE Trans. Comput.*, vol. 74, no. 5, pp. 1758–1770, May 2025.

[37] R. Xie, M. Xu, J. Cao, and Q. Li, “SoftGuard: Defend against the low-rate TCP attack in SDN,” in *Proc. IEEE Int. Conf. Commun. (ICC)*, 2019, pp. 1–6.

[38] D. Tang, Y. Yan, S. Zhang, J. Chen, and Z. Qin, “Performance and features: Mitigating the low-rate TCP-targeted DoS attack via SDN,” *IEEE J. Sel. Areas Commun.*, vol. 40, no. 1, pp. 428–444, Jan. 2022.

[39] D. Tang, S. Wang, B. Liu, W. Jin, and J. Zhang, “GASF-IPP: Detection and mitigation of LDoS attack in SDN,” *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3373–3384, Sep./Oct. 2023.

[40] G. Cormode and S. Muthukrishnan, “An improved data stream summary: The count-min sketch and its applications,” *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[41] S. Kim, C. Jung, R. Jang, D. Mohaisen, and D. Nyang, “A robust counting sketch for data plane intrusion detection,” in *Proc. 30th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2023, pp. 1–17.

[42] “MAWI working group traffic archive.” [Online]. Available: https://mawi.wide.ad.jp/mawi/samplepoint-F/2023/202301221359.html

[43] “Attacks on congestion control by a misbehaving TCP receiver.” [Online]. Available: https://github.com/rameshvarun/misbehaving-receiver

[44] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, “Henna: Hierarchical machine learning inference in programmable switches,” in *Proc. 1st Int. Workshop Native Netw. Intell.*, 2022, pp. 1–7.

[45] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, “Heavy-hitter detection entirely in the data plane,” in *Proc. Symp. SDN Res.*, 2017, pp. 164–176.

[46] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, “Aggregate-based congestion control for pulse-wave DDoS defense,” in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 693–706.

[47] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, “Jewel: Resource-efficient joint packet and flow level inference in programmable switches,” in *Proc. IEEE Conf. Comput. Commun.*, 2024, pp. 1631–1640.

[48] R. Li, Q. Li, Y. Zhang, D. Zhao, X. Xiao, and Y. Jiang, “Genos: General in-network unsupervised intrusion detection by rule extraction,” in *Proc. IEEE Conf. Comput. Commun.*, 2024, pp. 561–570.

[49] M. Yue, Z. Wu, and J. Wang, “Detecting LDoS attack bursts based on queue distribution,” *IET Inf. Secur.*, vol. 13, no. 3, pp. 285–292, 2019.

Rui Dai received the B.S. degree in computer science and technology from Hunan Normal University in 2018 and the M.S. degree in computer science and technology from Hunan University, China, in 2021, where he is currently pursuing the Ph.D. degree with the College of Computer Science and Electronic Engineering. His current research interests include intrusion detection, AI for networks, and programmable networking.

Dan Tang received the Ph.D. degree from the Huazhong University of Science and Technology in 2014. He is an Associate Professor with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. His research interests include the areas of computer network security, computer information security, and architecture of future Internet.

Zheng Qin (Member, IEEE) received the Ph.D. degree in computer software and theory from Chongqin University, China, in 2001. He is a Professor of computer science and technology, Hunan University, China. He has accumulated rich experience in products development and application services, such as in the area of financial, medical, military, and education sectors. His main interests are computer network and information security, cloud computing, big data processing, and software engineering. He is a member of China Computer Federation and ACM.

Kai Chen is a Doctor, a Professor, and a Vice President with the School of Cyberspace Security, Huazhong University of Science and Technology. Devoting to talent cultivation, he has won the Hubei Provincial Teaching Achievement Award, the Provincial First-Class Course, the First Prize of the Excellent Case Selection of Network Security Industry-University Collaborative Education of the Ministry of Education, independently developed a system of six platforms for practical teaching, applied for software copyrights and patents, published five textbooks, one monograph, and published many high-level educations reform papers. He has devoted himself to scientific research, engaged in research on computer network applications and security, Internet of Things security, and industrial Internet security, published many high-level articles. He is a member of the Council of the Hubei Cyberspace Security Society and the Education Working Committee, and a Permanent Member of the CCF CSP Organizing Committee of the Computer Society.

Keqin Li (Fellow, IEEE) is a SUNY Distinguished Professor of Computer Science with the State University of New York. He is also a National Distinguished Professor with Hunan University, China. He has authored or coauthored over 850 journal articles, book chapters, and refereed conference papers. He holds over 70 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the worlds top 5 most influential scientists in parallel and distributed computing in terms of both single-year impact and career-long impact based on a composite indicator of Scopus citation database. His current research interests include cloud computing, fog computing and mobile edge computing, energy-efficient computing and communication, heterogeneous computing systems, computer networking, and machine learning. He has received several best paper awards. He has chaired many international conferences. He is currently an Associate Editor of the ACM Computing Surveys and the CCF Transactions on High Performance Computing. He is also a Member of Academia Europaea (Academician of the Academy of Europe). He is an AAIA Fellow.

Jiliang Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science and technology from Hunan University, Changsha, China in 2015, where he is currently a Full Professor with the College of Integrated Circuits. He is a Vice Dean with the College of Integrated Circuits, Hunan University, where he is a Director of Chip Security Institute, and the Secretary-General of CCF Fault-Tolerant Computing Professional Committee. His current research interests include hardware security, integrated circuit design, and intelligent system. He has been the Program Committee Member for a number of well-known conferences such as DAC, ASP-DAC, GLSVLSI, and FPT. He is a Senior Member of CCF.