Mew: Enabling Large-Scale and Dynamic Link-Flooding Defenses on Programmable Switches

Huancheng Zhou*, Sungmin Hong*, Yangyang Liu†, Xiapu Luo†,Weichao Li‡, Guofei Gu*
* SUCCESS Lab, Texas A&M University, † The Hong Kong Polytechnic University, ‡ Peng Cheng Laboratory

***Abstract*—Link-flooding attacks (LFAs) can cut off the Internet connection to selected server targets and are hard to mitigate because adversaries use normal-looking and low-rate flows and can dynamically adjust the attack strategy. Traditional centralized defense systems cannot locally and efficiently suppress malicious traffic. Though emerging programmable switches offer an opportunity to bring defense systems closer to targeted links, their limited resource and lack of support for runtime reconfiguration limit their usage for link-flooding defenses.**

We present Mew[1], a resource-efficient and runtime adaptable link-flooding defense system. Mew can counter various LFAs even when a massive number of flows are concentrated on a link, or when the attack strategy changes quickly. We design a distributed storage mechanism and a lossless state migration mechanism to reduce the storage bottleneck of programmable networks. We develop cooperative defense APIs to support multi-grained co-detection and co-mitigation without excessive overhead. Mew's dynamic defense mechanism can constantly analyze network conditions and activate corresponding defenses without rebooting devices or interrupting other running functions. We develop a prototype of Mew by using real-world programmable switches, which are located in five cities. Our experiments show that the real-world prototype can defend against large-scale and dynamic LFAs effectively.

I. Introduction

Distributed denial-of-service (DDoS) attacks remain a threat to network services [1]–[3]. While traditional server-oriented DDoS attacks [4] try to exhaust the resource of servers, the link-oriented DDoS attack, i.e., link-flooding attack (LFA) [5], [6], aims to disconnect servers in chosen networks (i.e., *victim areas*) by flooding selected network links (i.e., *target links*). For example, instead of attacking tens of data centers, attackers used LFA to flood the critical links connected to CloudFlare's servers [7], whose services are slow or inaccessible for some people. In 2015, NetEase's game services are unavailable for 9 hours due to LFAs, resulting in a loss of more than 15 million Chinese Yuan [8]. On April 19th, 2022, LFAs caused the Internet connection at Ithaca College to cut off intermittently, disrupting operations across campus [9].

LFAs are hard to detect or mitigate due to the following features: **(1) Normal-looking.** Attackers can create a lot of normal-looking and low-rate flows. To identify the malicious traffic, defenders have to inspect a massive number of flows, which is costly. **(2) Invisible.** The *target links* are usually far away from the victim so that victim-side defense systems cannot obtain a precise view. **(3) Dynamic.** The attacker may dynamically adjust attack strategies (e.g., bot sets, attack types, target links) to bypass a slow or unadaptable defense system.

To defend against LFAs, the first step is to collect data from the network and identify suspicious traffic [10]–[17]. Most existing solutions deploy the defense system on centralized servers, which request network statistics from the data plane (i.e., forwarding devices). Ideally, the centralized servers can detect the congestion events and mitigate them by analyzing the collected statistics. However, if the centralized server collects fine-grained statistics, a huge amount of data could overwhelm the centers. On the other hand, if the centralized server samples packets at a low ratio (e.g., 1/1000), the classification process could be inaccurate and time-consuming (e.g., from tens of minutes to tens of hours) [16]. Therefore, there is a trade-off between scalability and accuracy.

In this respect, deploying defense systems in internet service providers (ISPs) equipped with programmable switches [18] becomes a promising solution. An ISP network is an infrastructure for routing the Internet traffic, which is close to *target links* and hence does not need to request information elsewhere. Besides, the emerging programmable switches are being used more and more to achieve high-performance and up-to-date network functions [19]–[23]. By programming the parser and the pipeline, we can reconfigure the programmable switch to realize customized defense mechanisms while maintaining high performance (e.g., Tbps-level throughput), as shown in Ripple [24], Poseidon [25], and Jaqen [26]. Therefore, the programmable ISP-centric defense system has the potential to counter LFAs locally and timely.

However, there are several challenges. First, the resource of programmable switches is limited. The *on-chip* memory of the current programmable switch [27] is only tens of megabytes. Such a low space is not scalable because most defense mechanisms require monitoring flow-level statistics to identify malicious flows [24]. Second, though the programmable switches can be reconfigured to counter different attacks[2], reloading a new program requires the reboot of switches, which opens a time window (e.g., a few seconds) for the attacker to adjust attack strategies [24], [26]. Even worse, attackers may deliberately trigger the reconfiguration process to interrupt other running functions. It is possible because the ISP networks may run several functions (e.g., load balance and firewall) on the same programmable switch for saving costs.

[1]Mew is a powerful Pokemon who can constantly learn new skills to counter all kinds of adversaries.

[2]To counter different attacks, we require different defense functions, but it is impossible to deploy all possible functions at the same time.

To address these issues, we present Mew, a memory-efficient and runtime adaptable link-flooding defense system. First, we design a lightweight distributed storage protocol to reduce the storage overhead of the data plane. A state migration mechanism is proposed to further mitigate the traffic concentration. Second, we design cooperative defense APIs to support flexible *co-detection* and *co-mitigation* within any group, which greatly reduces the communication and storage overhead. Third, we design a memory access proxy to achieve fast-switching functions *on the fly*. The memory access proxy isolates, shares, and reuses the memory among different functions.

To sum up, we make the following contributions:

- A lightweight distributed storage protocol for reducing the storage bottleneck of the programmable ISP-centric defense system. A series of APIs to support multi-granularity cooperation among switches to counter complex LFAs.
- A runtime memory allocation mechanism to support dynamic defenses.
- A prototype of Mew deployed on a real-world testbed as well as evaluation results showing that Mew can handle large-scale LFAs even if attack strategy changes in second-level.

II. Background and Motivation

In this section, we describe link-flooding attacks and point out the difficulties to mitigate them. Then we discuss the opportunities and challenges for programmable switches, as well as the motivation for using approximate data structures.

A. LFA and Traditional Defense

LFA. LFAs try to isolate a *victim area* (e.g., servers of an organization, a city, a state, and even a country) from the Internet by flooding *target links*. Usually, *target links* are a part of links of the paths between the Internet and the *victim area*. These links can be far away from the *victim area*. By collecting and analyzing the routing information, the adversary can construct a "*link map*" containing a set of *target links*. Then, the adversary instructs botnets to create traffic flood *target links*. The attacker-generated traffic is often normal-looking (e.g., using real IP addresses), low-rate (e.g., 1 KBps), and distributed (e.g., from different locations). Thus, network operators cannot identify malicious traffic even if they detect congestion events. To date, there have been many types of LFAs, such as Coremelt [6], Crossfire [5], Rolling Crossfire [5], and CrossPath [28]. Table I shows their attack patterns, consequence, and metrics for defenses.

Traditional centralized defenses. Most traditional defense systems [12], [14]–[16], [29] are centralized. That is, the defense systems are deployed on the control plane (i.e., centralized servers) and collect network statistics from the data plane (i.e., forwarding devices) [30]. However, there is a trade-off between scalability and accuracy. On the one hand, requesting fine-grained statistics can improve defense effectiveness, but it is not scalable in the real-world network because the huge amount of data can overwhelm the centers [15]. On the other hand, requesting coarse-grained can reduce communication overhead, but the defense process can be time-consuming (e.g., several hours) and error-prone [12], [14], [16]. Thus, they can be easily bypassed by changing attack strategies [24].

B. Opportunities and Challenges

ISP networks are the infrastructure for providing Internet access service and are also the target of LFAs. To achieve high performance and flexible functions, it is trending to deploy programmable switches on ISP networks [21] [22], [23].

Opportunities of programmable switches. Emerging programmable switches are becoming desirable choices for large-volume stream processing due to their advantages of flexibility, performance, and cost-efficiency: (1) Programmable: based on domain-specific languages such as P4 [18], network operators can reprogram the programmable switches to realize customized functions without upgrading the devices [19], [20]. (2) High-performance: programmable switches support line-rate performance [31]. Their throughput can reach Tbps-level and even higher [27]. (3) Cost-efficiency: compared with legacy switches, programmable switches provide the same speed but with an order of magnitude less cost and power consumption [27], [32]–[34]. Given such advantages, there are a lot of industrial and academic efforts to implement programmable switches on the ISP networks for higher QoS [19], [20], [34]–[36] and safety guarantee [16], [24], [26].

Challenges of programmable ISP-centric defense system. Though programmable switches improve the flexibility and functionality of ISP networks, it is still difficult to build an ISP-centric link-flooding defense system, as explained below.

Limited memory (C1). A programmable switch must use *on-chip* memory to achieve link-rate speed. Current programmable switches only have tens of megabytes of *on-chip* memory, which is insufficient for state-intensive monitoring [37]–[39]. Considering Crossfire attacks [5], malicious flows are normal-looking and low-rate. One pattern of this attack is that many low-rate flows with the same source IP address may appear on the congested links. Ripple [24], a state-of-the-art link-flooding defense system, designs a defense program to monitor each flow and count the number of low-rate flows for each source IP. However, this defense policy is not scalable because the *on-chip* memory of programmable switches could be exhausted even though there are only hundreds of thousands of flows.

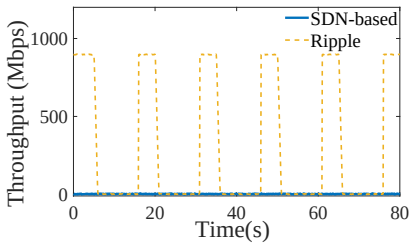


Fig. 1: Throughput of users of prior work under a dynamic attack including Coremelt, Crossfire, and Pulsing attack.

Lack of support for runtime reconfiguration (C2). Because programmable switches have limited resources, they can only run a few defense functions *at the same time*. An attacker can

TABLE I: Comparison among LFAs

	Attack patterns	Consequence	Metrics for defenses
Coremelt	a few bots with high aggregated speed	link congestion	aggregated speed (host-level)
Crossfire	bots generating many low-rate flows	link congestion	number of low-rate flows (flow-level)
Rolling Crossfire	multiple botnets congest links in turn	link congestion	correlation* between links and flows (flow-level)
Crosspath/Pulsing	bots sending periodic pulse traffic	periodic disturbance	flow entropy (flow-level)

*Correlation means the relationship between flows and links. High correlation rates mean the speeds of a flow and a link increase/decrease at the same time.

change attack patterns frequently (e.g., in seconds) to evade defenses, e.g., using relatively high-speed flows [6] to bypass a low-rate detection policy or launching a pulse attack to disable a coarse-grained detector [28]. Although defenders can reconfigure the programmable switch after an attack posture changes, it takes a few seconds to back up runtime contexts and reload new defense programs. Meanwhile, other running functions such as the firewall will be interrupted for a while. Fig. 1 shows the throughput of prior works, i.e., Ripple and SDN-based solutions. Ripple shows periodic and non-trivial downtime against dynamic LFAs due to the process of reconfiguration. Also, an SDN-based solution RADAR [12] that collects flow information in a centralized manner shows all-time low throughput, which is not suitable for quickly tracking the changes of dynamic attacks.

C. Approximate Data Structure

When dealing with large sets of data, we may not need to fully record the data. Instead, we focus on key questions such as whether an instance has appeared, which instance appears the most, or how many times an instance appears. A common approach is to use *deterministic* data structures, such as hashset or hashtable. However, when working with stream data, this approach might require many queries to update an instance, whereas stream data often requires updating an instance in one pass. To address this issue, we can use the key of instances as the index. However, this approach is not scalable because the size of the index space is 2 to the length of the data key. For example, a 5-tuple TCP key is 102 bits, so the size of the index space is 2^{102}.

Approximate data structures [40]–[42] are good choices for dealing with large data sets or traffic data. In general, an approximate data structure implements a hash function that maps objects randomly and compactly to certain items. Compared with deterministic storage, the length of the index of an approximate data structure is the length of the hash output. If the number of instances expected to be stored is less than a threshold, their collision rate is negligible. That is, the size of the index space is predetermined by the number of instances expected to be stored and the expected collision rate. Theoretically, given a Bloom Filter (BF) with a fixed size, the expected collision rate is $(1 - e^{-kn/m})^k$, where n is the number of instances, m is the size (index space) of BF, and k is the number of hash functions. For example, a BF with the size of 1,000,000 can record states of 1,000 instances with 0% collision rate, or record states of 2,000,000 instances with 86% collision rate. Common approximate data structures are Bloom Filter (BF) [40], Counting Bloom Filter (CBF) [41], and Count-min sketch (CMS) [42].

Takeaway: Given the same space, an approximate storage structure can record more instances than a deterministic storage structure, but its accuracy is affected by the number of recorded instances.

III. Mew Overview

A. Definitions

- **Flow**: A flow is a sequence of packets from a source to a destination. In this paper, we distinguish different flows by using flowkeys computed by 5-tuple (i.e., source IP, destination IP, source port, destination port, and protocol). We interchangeably use the terms flow and traffic.
- **Edge switch**: An edge switch is a forwarding device that directly connects to external networks or internal hosts. Thus, packets from other ISPs or hosts always arrive at the edge switch first.
- **Core switch**: A core switch is a forwarding device positioned within the backbone. The core switch only receives packets from other switches.
- **Capture**: A switch *captures* a flow if this switch is responsible for recording the flow. In the distributed storage case, a flow is only *captured* by one switch to save memory.

B. Problem scope

Deployment scenario. We focus on a programmable ISP network, which owns many programmable switches. The programmability of these emerging devices makes it possible for network operators to provide diverse services without upgrading the hardware. It is reasonable for ISP networks to deploy link-flooding defenses to protect critical links and ensure a high quality of service (QoS). For saving costs, some devices may run multiple functions, including defense functions and basic functions such as forwarding and load balance. The basic network functions should not be interrupted.

Threat model. We focus on large-scale and dynamic LFAs. We assume that the adversary controls some distributed botnets and instructs them to create a large number of flows, which pass through some critical links[3]. Due to link congestion, a victim area is unable to access the Internet. To remain stealthy, each bot creates seemingly legitimate flows (e.g., real IP and low-rate) and has a maximal number of concurrent flows (e.g., 10,000). The adversary can change bot sets, *target*

[3]In Crossfire [5], attackers have hundreds of thousands of bots and decoy servers around the world to congest target links with the bandwidth of 40Gbps.

links, or attack types. We suppose that the programmable switches cannot be compromised by adversaries. Besides, the communication between switches cannot be tampered with, which is a common assumption of LFAs. Finally, we assume that the routing path of each flow always changes *predictably*[4], which is also the basic condition of most LFAs [5].

Requirements of a programmable defense system.

- **Storage balance.** Due to the limited memory of the programmable switches, most local flow monitoring mechanisms for link-flooding defenses will exhaust the memory of a switch when there are many flows concentrating on a link. A robust defense system should be able to distribute flow monitoring tasks to different switches for storage balance. Hence, we propose a distributed storage protocol in §IV.
- **Precise cooperation.** Although sharing information among all switches can construct a global network view, it may introduce a huge overhead. To construct a precise network view for defenses, we design a series of APIs to support cooperative defense in §V.
- **On-the-fly update.** Current programmable switches cannot be reprogrammed on the fly, which is easy to be exploited if attackers deliberately trigger the rebooting process to interrupt running functions on switches. In §VI, we propose a dynamic resource allocation mechanism to change defense policies without halting switches.

C. Workflow

As Fig. 2 shows, Mew is based on the following techniques:

- *Distributed Storage*: When the defense mode is activated and a new flow appears, an edge switch records its existing state and starts a negotiation protocol. Without suspending the flow, a switch on the routing path is voted to *capture* this flow in a predefined strategy (e.g., least-utilized first). For flexibility, captured flows can be migrated between switches.
- *Cooperative Defense*: By using cooperation APIs, network operators deploy co-detection and co-mitigation modules on multiple programmable switches. For example, a switch on congested links can request related flow states from other switches or inform others of its link states.
- *Runtime management*: Due to resource limitations, we first allocate minimum resources[5] for each deployed function (including non-defensive functions) to support as many kinds of defenses as possible. Depending on real-time network conditions, network operators allocate memory for activated modules and recycle memory[6] after mitigation.

IV. Distributed Storage

As mentioned in §III-B, it is important to achieve storage balance among programmable switches in link-flooding defense. Mew designs an in-band negotiation protocol to elect switches to *capture* flows uniformly.

[4]A change is predictable if it can be anticipated by network operators, such as proactively updating flow rules.

[5]When a function is not activated, switches only maintain some states of their activated conditions (e.g., link utilization).

[6]After Mew mitigates the attacks, Mew can flip the current version to discard the stale states to release memory.

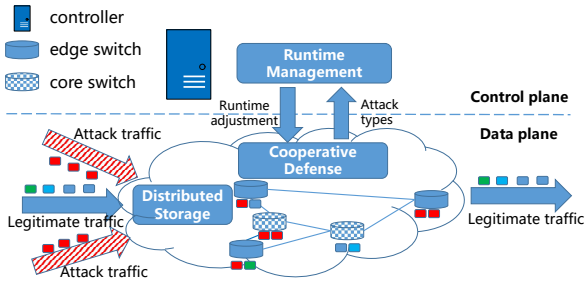
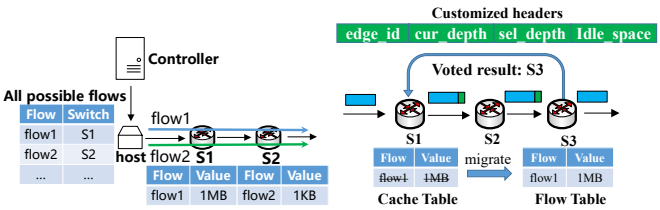


Fig. 2: Overview of Mew.

A. Performant distribution protocol

A straightforward way is to use a mapping function (e.g., hash) to elect a switch as shown in Fig. 3(a). However, it has two problems: (1) A center/controller needs to determine the routing path of all possible flows in advance and assign switches to record them, which is prone to be imbalanced due to changing network conditions. (2) If a center/controller assigns switches to capture flows in real-time, it must receive a lot of packets and introduce extra latency. Even worse, attackers may create many flows to flood the controller [43].



(a) Out-of-band assignment. **(b)** In-band negotiation.

Fig. 3: The structure of Mew's distributed storage.

Hence, we propose a lightweight distribution protocol to achieve storage balance. The key idea is to use a greedy algorithm to choose a least-utilized switch, and adjust the state distribution over time. Here, the least-utilized switch refers to the switch with the most available memory for defenses.

Inspired by the emerging in-band network telemetry (INT) [44], [45], our distribution protocol selects the least utilized switch without prior knowledge or suspending the flows. As shown in Fig. 3(b), when a new flow F enters the protected network, its first packet is identified by the edge switch through the existing table. Then, the edge switch caches the state of flow F. Meanwhile, the first packet is added with an extra header containing $edge\_id$, cur_depth, sel_depth, and $idle\_space$ and flow F is forwarded without pause. The subsequent switches will increment the cur_depth. If their available memory is larger than the $idle\_space$ in the packet header, they replace the sel_depth and $idle\_space$ with their cur_depth and available memory. Therefore, the sel_depth in the packet always indicates the depth from the edge switch to the least-utilized switch among the passed switches. The last-hop switch removes the extra packet header and sends the edge switch a *mirror* packet containing the sel_depth.

Finally, the edge switch records the depth for flow F and transfers the cached states of flow F to the voted switch. It is worth noting that only the first packet of a flow will trigger the distribution negotiation protocol, so the communication overhead is negligible.

We introduce a Counting Bloom Filter (CBF) to the edge switch to store the captured depth as well as the existing state of each flow. The captured depth means the hop from the edge switch to the voted switch, and a depth of 1 means the flow is captured by the edge switch. Usually, it is a 1-6 bits state (i.e., 1-63 hops), depending on the path length. Through the captured depth, the edge switch can know if a flow is new or not. The captured depth of any new flow is 0 (new). If the captured depth is 0, the edge switch will add extra headers for the negotiation process. Otherwise, the edge switch will add the captured depth to the packet header. To reduce communication overhead, the captured depth can use reserved headers such as 6-bit reserved headers in TCP packets. When receiving the packets, each switch will check if the captured depth is equal to 1. If so, the switch will record this packet and remove the extra packet header or clear the reserved header. Otherwise, the switch subtracts the captured depth by 1.

B. State migration

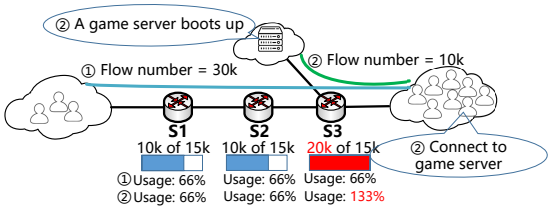
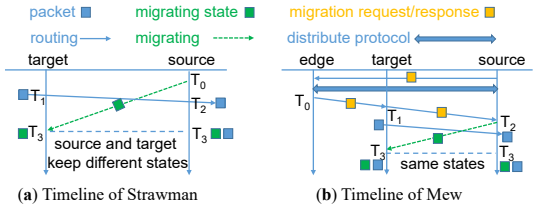


Fig. 4: Flows concentrate on a link and the switch overloads.

Although Mew's distribution protocol uniformly distributes flow states to multiple switches, Mew needs to support state migration in order to adapt to the changing network conditions. As shown in Fig. 4, at first the load of each switch is balanced (i.e., stage ①). At stage ②, a game server completes maintenance and boots up. Then a lot of hosts try to connect it so that many flows appear in the specific links. Because $S1$ and $S2$ are far away from those links, they fail to share the burden and the storage imbalance happens. Thus, we need to actively migrate some flow states from the overloaded switch (i.e., S3) to idle switches (e.g., S1 and S2).

Strawman: To move the flow state from one switch (source) to another switch (target), the source can send the target a migration packet along with migrating state. After receiving the migration packet, the target begins to record the flow state. Finally, the source deletes the migrating flow state to release space. This solution has two problems. First, the source needs to know which switches are suitable for migrating state. Second, if the source is **behind** the target, some packets may be missed recording due to the intermediate state. As shown in Fig. 5(a), the source migrates a state to the target at T_0. Before the target receives the state (at T_1), the target has handled a packet while the target does not record it. At T_2 and T_3, the source and the target keep different states. With this intermediate state, the source cannot safely delete its state.



(a) Timeline of Strawman (b) Timeline of Mew

Fig. 5: The timeline of different migration strategies.

Mew: To address these problems, we leverage edge switches and break the migration process into two steps. First, the source sends a migration request to edge switches. Then, the edge switches run the negotiation protocol to select the switch with the least load to accept migrating flows. Second, the edge switch sends a migration response containing the same key (e.g., IP addresses) of the migrating flow. Thus, the migration response follows the same routing path of the migrating flow and reaches the target as well as the source. As shown in Fig. 5(b), the target receives the migration response at T_1 and begins to record the migrating flow. At T_2, the source receives the migration response and sends the stored state to the target. At T_3, the target keeps the same state as the source. In both cases, the source does not stop recording the migrated flow until it receives an acknowledgment from the target. If any packets get lost, the source will time out and resend a request to restart the protocol.

V. Cooperative Defense

Some LFAs such as Crossfire occur over multiple links, so a single switch cannot detect and mitigate them. In this section, we design a cooperative defense mechanism to defeat such complex LFAs.

Strawman: At first glance, we can directly use the global synchronization approach of Ripple [24] to realize the cooperation among switches. However, Ripple chooses a full storage strategy (i.e., switches store states of all passing flows), so they only synchronize a little information (e.g., blocklist). Under a distributed storage scenario, switches need to share a large number of flow-level states. Thus, directly adopting global synchronization will introduce high synchronization and extra storage overhead.

If we assume that there are n switches in the network, each switch has to send 1 packet (size = m) and receive $n-1$ packets (size = m) from others in the p2p mode, whose communication overhead is $O(mn)$ for each node. On the other hand, if we choose a central entity (leader-mode) to aggregate packets, the central entity has to receive $n-1$ packets and send 1 **big** packet to $n-1$ switches. The size of the **big** packet is about $n \times m$ in the worst case (i.e., each switch sends a unique packet). Thus, the communication overhead of the central entity is $O(mn^2)$ in the worst case. In addition, global synchronization requires

each switch to store full information even though they are not attacked. This introduces unnecessary storage overhead to each switch. Finally, the leader-mode is vulnerable because attackers can launch DDoS on the central entity to disable the global synchronization process.

Mew: Instead of periodically synchronizing information to the whole network, Mew support that switches synchronize information within customized groups only when the predefined conditions are met. To support multi-granularity cooperation, we design a set of APIs:

- **Monitor(state, mode)** allows the defender to assign the storage mode of a state. For example, **Monitor(load[port], full)** means that **each** switch monitors the load of each port locally (i.e., full storage). **Monitor(byte_count[flow], dist)** means that only the **chosen** switch monitors the byte count of each flow (i.e., distributed storage). Defenders can change the storage mode of a specific state in real time.
- **Sync(state, range)** supports a switch to send a specific state to other switches within a certain range. The state is a tuple data structure and the range can be set to global or a group. Synchronizing objects are shared among all switches (range=global) or limited to a given group (range=group).
- **Request(state, mode, period)** defines the request mode and the period of a state. For example, **Request(load[port], global, 100ms)** is to get the load of each port from all switches every $100ms$. To analyze the traffic correlation on different links [12], we can narrow the querying range by using **Request(entropy[port], group, 1000ms)**. Switches in the given group then return the entropy (i.e., the change of states) of each port every $1000ms$.
- **Trigger(condition, actions)** is a conditional statement to perform **actions** once the **condition** is true. For example, **Trigger(byte_count[IP]>100Mb, block(IP))** is used to block an IP address if its byte count is larger than 100Mb within a window.

In the following cases, we show that defenders can easily design the detection and mitigation mechanisms by using proposed APIs. It is worth noting that these mechanisms may already exist, but the proposed APIs make them more flexible and more efficient. We show more cases in Appendix A.

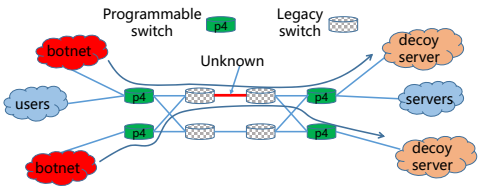


Fig. 6: A distributed Crossfire attack example

Case Study: switch-native Crossfire defense. In Crossfire attack, the attacker has a lot of available bots. To avoid being identified easily, each bot only generates a certain number of low-rate flows (e.g., 1000 4Kbps flows). A potential classification approach from Ripple [24] is to monitor flow speeds and count the number of low-rate flows of each host pair (e.g., a source IP and destination IP pair): if a pair establishes too many low-rate flows, this pair could be responsible for the congestion event. Unfortunately, the memory limitation of the programmable switch makes it hard to maintain too many flow states. In the real world [46], there could be millions of flows regardless of whether an attack is taking place. Even if we use approximated data structures such as Count-Min Sketch (CMS), a small CMS (e.g., size is 65,536 in Ripple) is not enough to store them and causes an unacceptable collision rate. To address this issue, we use proposed APIs to design a cooperative defense mechanism as shown in the below code:

```
0 # Crossfire Detection
1 Monitor(byte_count[port],full)
2 Monitor(byte_count[flow],dist)
3 Monitor(number_of_flows[port],full)
4 Sync(byte_count[port],100ms,global)
5 Trigger(byte_count[port]>1000Mb &&
6     number_of_flows[port]>100K,Activate(Crossfire))
```

By calling Monitor API, we let each programmable switch records the byte count of ports (line1). To reduce storage overhead, the byte count of a flow is recorded only by a chosen switch (line 2). To detect the attack types, each switch records the number of passing flows of each port (line 3). It is worth noting that switches on congested links are not always programmable switches, as shown in Fig. 6. To recover port states between non-programmable switches, the programmable switches synchronize the byte count of ports every 100ms using global mode (line 4). Whenever the byte count of a port exceeds a threshold (line 5) and the number of passing flows of that port exceeds a threshold (line 6), the Crossfire mitigation is activated (line 6).

```
0 # Crossfire Mitigation
1 Monitor(slow_flow_count[host_pair], full)
2 Request([byte_count[flow]<=1Kb],group)
3 Trigger(slow_flow_count[host_pair]>1000,
4     Sync([suspicious,host_pair,Crossfire], global))
```

When the Crossfire mitigation is activated, defenders can allocate memory to monitor finer-grained states for classification and mitigation. For example, in line 1, switches activating Crossfire mitigation begin to monitor the count/number of slow flows (i.e., with a byte count < 1Kb) of each host pair. Due to the distributed storage design, the activated switches need to request states from other switches (line 2). To reduce communication overhead[7], switches only return a boolean state indicating if the flow speed is less than a threshold (e.g., 5Kbps). If the number of slow flows of a host pair exceeds a threshold (line 3), the switch synchronizes the host pair with suspicious Crossfire tags globally (line 4).

Case Study: switch-native RADAR defense. As shown in Fig. 7, a rolling Crossfire attack [5] is hard to detect and mitigate. First, the malicious flows use normal-looking packets, making the classification time-consuming. Second, before being identified by defenders, adversaries have changed bot sets and target links. RADAR [12] proposes a potential mechanism

[7]In our programs, returned states are attached to passing packets by using reserved bits if possible, which introduces minor communication overhead.

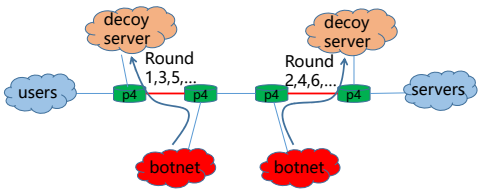


Fig. 7: A rolling link-flooding attack example

to capture the dynamic pattern of this attack. The high-level idea is to analyze the relationship between links and flows. The first relationship is that several links of a path are congested alternatively. The second relationship is that the change of the speed of malicious flows is always related to the change of the load of links. If the speed of a flow increases/decreases as the link utilization rate increases/decreases, RADAR tags this flow as a suspicious flow. The limitation of RADAR is the long identification time due to a central design. To avoid overwhelming the central entity, the center can only query a part of the states each time, which is time-consuming. To address this issue, defenders can use Trigger APIs to ensure that switches only return necessary states as shown below.

```
0 # Rolling Crossfire Detection
1 Trigger(byte_count[port]>1Gb && passing_time>100ms,
2     Sync([port,congestion], group))
3 Trigger(byte_count[port]<1Gb && passing_time>100ms,
4     Sync([port,recovery], group))
5 Trigger(link_change[port]>3, Activate(RADAR))
```

The above code defines three triggers. First, once any links begin to congest (e.g., byte count exceeds 1000Mb), an event including the port and congestion is synchronized to a given group (line 1~2). To avoid repeatedly sending events, defenders can specify a period (e.g., 100ms). Similarly, once a link recovers from congestion, an event containing the port and recovery is synchronized to others (line 3~4). When detecting the load of links dramatically changing several times, the mitigation function (RADAR) is activated (line 5).

```
0 # RADAR Mitigation
1 Monitor(speed_change[flow], dist)
2 Trigger(speed_change[flow] == cur_link_load_change),
3     count[flow]=count[flow]+1)
4 Trigger(count[flow]>3, output=honeypot)
```

When RADAR mitigation is activated, the switch records the speed changes of each flow (line 1). If the speed of a flow changes (e.g., high → low) as the load of a link changes (e.g., high → low), the count of this flow is incremented (line 2~3). When a flow shows a high correlation with a victim link (e.g., $count[flow] > 3$), it is rerouted to a honeypot for further analysis (line 4). Unlike RADAR, which sets a large request window (e.g., 10 seconds) to avoid huge communication overhead, Mew distributes identification tasks to several switches. Switches only need to synchronize the change of links, which introduces little communication overhead.

VI. Dynamic Memory Allocation

In this section, our goal is to *defeat LFAs without interrupting traffic processing even if attackers change their attack strategies rapidly* (e.g., in seconds). As we mentioned in §II-B C1, programmable switches have limited resources (e.g., memory), so we cannot directly deploy all possible defense functions on a switch at the same time.

We observe that each function can be abstracted with a few sequential match-action units (tables) and a few register accesses (memory). Current programmable switches provide hundreds of tables, which is enough for most link-flooding defenses[8]. Thus, the challenge is how to orchestrate the limited memory for different functions.

Ideally, if we can share register memory among different functions and recycle idle memory, programmable switches can load much more functions [47]. Unfortunately, there are two challenges. First, the ownership of a register is fixed after compiling. A register cannot be accessed by multiple functions/defenses. Second, the size of a register is fixed after compiling, but the memory required by a function may change depending on network conditions. For example, a load balance function requires 100kb memory when there are a few hundred flows, and requires 1Mb memory when there are a few thousand flows.

To address these issues, we propose a memory resizing mechanism to support memory sharing and reallocation.

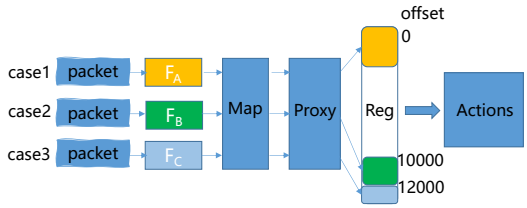


Fig. 8: Memory access proxy.

Memory resizing. To address the first challenge, we design a memory access proxy. As shown in Fig. 8, each function accesses the proxy rather than the registers. The proxy translates access requests to real register access operations. For the second challenge, it is infeasible to create many small registers and change their ownership [47], [48] to realize *fine-grained* adjustment, because the memory space can only be divided into a few tens of pieces. Thus, we design the following APIs:

- **Initial(func_id, offset)** is used for allocating memory for a function when it is activated. By using software commands such as p4runtime [49], the control plane updates the states of the function (e.g., sets a flag to 1). Then the register proxy can be accessed by the function.
- **Release(func_id)** supports us to free the memory if they are not accessed anymore. The control plane sets the flag of a function to 0 to disable its accessibility to the proxy.
- **Enlarge(func_id, size)** is a fine-grained method to adjust the memory allocation. Unlike Mantis [47], which only changes the accessibility of functions to a proxy, we can update the offset state of a function to increase its accessible memory.

[8]Although there may be tens of defenses, each defense usually occupies a few tables (from several to a few tens). Besides, some tables can be shared.

- **Shrink(func_id, size)** is more complicated. Simply releasing the register will result in data loss, so we need to migrate the valid state. To migrate exact storage data, we run the state migration protocol to move states to other switches or other available registers of the current switch. To shrink the memory storing approximated data such as CBF or CMS, the switch compresses the registers into a smaller space under the guidance of the control plane.

VII. Implementation & Optimization

Implementation. We have implemented our Mew prototype in P4 on Barefoot Tofino [50] switch, using ~9,000 lines of code. The controller for runtime configuration is implemented in Python using ~6,000 lines of code. We open-source the prototype of Mew in [51].

Optimization of monitoring state. Ripple [52] uses three windows (i.e., W_0, W_1, W_2) and corresponding registers (i.e., R_0, R_1, R_2). In every window, a switch updates the current register, clears the next register, and reads the states of the last registers as monitoring states. For example, during window W_1, a switch updates R_1 (current), clears R_2 (next), and reads R_0 (last). During the next window W_2, R_2 (current) has returned to zero. R_1 (last) is read and R_0 (next) is cleared. To reduce memory costs, we design a two-window monitor mechanism. That is, switches only use two windows (i.e., W_0, W_1), two registers (i.e., R_0, R_1), and a timestamp register (i.e., TS). During window W_1, a switch updates R_1 and reads R_0. In every update, the switch updates TS with the current timestamp. When entering the next window W_0, the switch checks the timestamp of the last update. If the timestamp is in the window W_1, the switch first clears and updates R_0. By doing so, we save a window for clearing registers. Since the register storing the timestamp is shareable and quite small (e.g., 1 bit is enough to distinguish two windows), we reduce the memory usage and ALUs by 33%.

Adaptive memory allocation. Mew supports sharing memory among different functions. To adjust the available size of each function, switches maintain an offset of each function. The real address of a function is the sum of a virtual index and its offset. Before a function runs out of space (crosses the border of other functions), the control plane updates the offset to enlarge space. Besides, we support memory slicing. For example, detectors of Crossfire [5] and pulsing attacks [28] need to monitor the number of passing flows and the times of fluctuations in a link. A straightforward way is to define two registers with a width of 32 bits. By using adaptive memory slicing, we can define a register with a width of 32 bits and "split"[9] it to two registers with widths of x and $32 - x$ respectively. When the number of passing flows increases, the control plane increases x. When the times of link fluctuations increase, the control plane decreases x. In this case, the memory usage and ALUs are reduced by 50%. In another case for classifying malicious flows of three attacks (Coremelt [6], Crossfire [5], and Pulsing [28]), the memory and ALUs are reduced by 2/3.

[9]We compose two states to a single 32-bit state before updating the register.

VIII. Evaluation

In this section, we evaluate Mew extensively and show that:

- Mew can record flow-level states with high accuracy on large-scale attack scenarios.
- Mew supports multi-granularity co-defense with low cooperation overhead.
- Mew can mitigate many kinds of LFAs efficiently.
- Mew adapts to dynamic LFAs as fast as second-levels without halting the switches.
- Mew only introduces insignificant extra latency and moderate resource overhead.

A. Prototype and setup

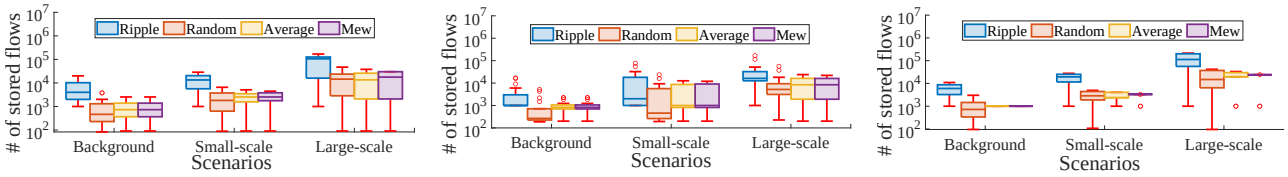
Testbed. Our testbed includes five h3c 3030g servers with Intel Xeon Silver 4216 2.10GHz CPU and 44Gbps Network Interface Card, and five 3.2Tbps H3C S9830 programmable switches with tofino chip [50]. These switches are located on five Chinese cities, including Dongguan, Shaoguan, Shenzhen, Zhongshan, and Jiangmen. They form a peer-to-peer network. The available bandwidth of each link is 1000Mbps.

Benign traffic generation. We use distributed Internet traffic generator (D-ITG) [53] to generate benign traffic (background and victim traffic) with given settings and analyze the latency, traffic rate, and packet loss rate. Typically, each background traffic has a speed range from 1Kbps and 100Kbps, and it can be UDP or TCP traffic. We mimic 500 benign hosts to generate 500 background traffic with an aggregated speed of 10Mbps (i.e., 1% bandwidth). To better observe the attack impact and defense effectiveness, we use D-ITG to generate *victim* traffic with a speed of 100Mbps (i.e., 10% bandwidth). We respectively generate a TCP traffic and a UDP traffic for experiment diversity. So far, all flows (including malicious flows) are IPv4-based. But it is easy to extend to IPv6.

Attack traffic generation. For Coremelt attacks, it requires the bots to be located in the victim network, so the number of available bots is usually fewer. We use two servers as botnets and set the number of bots of each botnet to 50 (i.e., 50 IP addresses). Each bot generates 100 traffic (e.g., with different source ports or destination ports) with a speed of 200kbps. The total throughput between two botnets is 1000Mbps (i.e., 100% bandwidth). The remaining three servers are used as a victim and two legitimate users.

For the massive number of flows scenario (i.e., Crossfire), we use two servers to simulate botnets with a total of 250 virtual IP addresses and one server as the decoy server. Each bot generates 1000 low-rate flows with a speed of 4kbps. Thus, the total throughput is 1000Mbps (100% bandwidth). The remaining servers are a victim and a legitimate user.

For the rolling Crossfire attack scenario, we assume that attackers try to congest different links alternatively to avoid triggering an alarm because some detection mechanisms will not be triggered if link congestion lasts for a short time. Attackers divide the botnets into multiple sets and instruct



(a) The storage overhead (Atmnet) **(b)** The storage overhead (Goodnet) **(c)** The storage overhead (Reuna)

Fig. 9: The boxplot shows the number of stored flows on each switch. A more even distribution is better.

them to launch attacks in different periods. We divide 250 bots into 2 sets and use them to congest different links. In more detail, each bot generates 1000 flows with a speed of 8kbps, so each round 125 bots will consume 1000Mbps bandwidth of a target link. Finally, we set two rolling periods, 5 seconds and 15 seconds. We assume that attackers will not choose a very low period (e.g., <1s), which shows an obvious pulsing pattern and turns to the pulsing attack below.

For the pulsing attack scenario, we assume that attackers send high-speed traffic every 1 second, using the same setting for launching Coremelt attacks. Then attackers send low-rate traffic every 5 seconds, i.e., sending a 4Kb packet per second.

Defense system and effectiveness metric. We evaluate the defense effectiveness of different defense systems. The baseline is an SDN-based defense, RADAR [12], [54], which contains some DDoS defense applications and a rolling Crossfire defense application. For undeveloped defenses (e.g., Coremelt), we follow the same strategy to progressively locate the suspicious traffic. For metrics of defense effectiveness, we mainly focus on benign traffic's performance, including throughput, delay, and packet loss rate. In addition, we estimate false positive rates (FPRs), meaning benign users are tagged as malicious, and false negative rates (FNRs), meaning malicious users are tagged as benign.

B. Storage Overhead and Accuracy

To show the scalability of Mew, we use three real-world topology setups, including Atmnet (tree type), Goodnet (star type), and Reuna (linear type) from Topology Zoo [55], as shown in Appendix B. We also set different traffic distributions for better comparison. Specifically, we analyze the storage overhead under background, small-scale, and large-scale attack cases. Background means that there are only benign background flows (each node generates 1000 flows). Small-scale attack assumes that attackers control a few bots and generate a few high-speed flows (e.g., 10Kbps per flow), and large-scale attack assumes that attackers control a lot of bots and generate a lot of low-speed flows (e.g., 1Kbps per flow). It is worth noting that we only have five physical programmable switches (it is the most in related work), so we use software programmable switches (bmv2 [56]) to simulate the monitoring process. The performance (e.g., throughput) of software programmable switches is far lower than physical programmable switches. However, their behavior is similar and we only use bmv2 for measuring storage overhead.

TABLE II: The defense effectiveness (Reuna)

Defense	Accuracy	FPR	FNR
Ripple (Coremelt, 6M)	99.66%	0.2%	0%
Mew (Coremelt, 3M)	100%	0%	0%
Ripple (Crossfire, 12M)	18.62%	0	100%
Mew (Crossfire, 6M)	69.57%	0%	0%

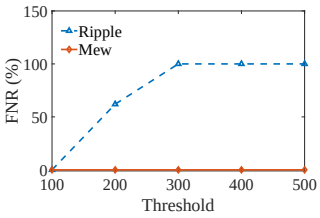


Fig. 10: The FNR of Mew and Ripple on Crossfire (Reuna)

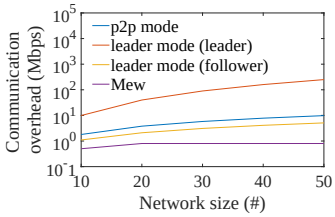


Fig. 11: Communication overhead in different networks

In Fig. 9, Ripple means full storage that switches store each passing flow. Random means random storage that a switch is randomly picked up for storing a flow. Average means distributed storage without a state migration mechanism, which allows switches to migrate flow states to others. Mew means distributed storage and state migration, as described in §IV.

As shown in Fig. 9(a), when there are more flows, the mean and maximum storage overhead of switches in Ripple increase greatly. On the contrary, random storage and distributed storage (Average and Mew) can reduce overall storage overhead greatly. Further, we observe that Random strategy achieves poor performance in highly cross topology such as Goodnet. As shown in Fig. 9(b), some switches still have to store a lot of flows (i.e., the outlier in the boxplot). It is because Random strategy randomly picks up a switch on the routing path to store a flow. For the switches with higher connectivity (i.e., connect to more switches), they have higher chances to be picked up. On the contrary, Average and Mew elect a least-utilized switch on the routing path to store a flow. When a core node stores more flows than others, it is less likely to be chosen. Thus, the maximum value is close to the average value. We further show that Mew can achieve better average storage than Average in some cases. As shown in Fig. 9(c), Average's maximum value is higher than Mew's. It is because Reuna is a linear topology and a traffic burst on a short path can easily cause storage imbalance. Mew actively migrates a part of flows to the idle switches when a traffic burst happens.

Then, we compare full storage and distributed storage on

accuracy. Specifically, each switch of Ripple uses several CMS tables with 6Mb SRAM or 12Mb SRAM while each switch of Mew uses 3Mb SRAM or 6Mb SRAM. We choose Reuna topology (bandwidth = 1Gbps) and launch Coremelt and Crossfire attacks. For Coremelt, there are 50 bots whose flow speed is 20Mbps. For Crossfire, there are 200 bots and each bot generates 1,000 low-rate flows with a speed of 5Kbps. There are 10,000 background flows with a speed of 10Kbps.

Table II shows the accuracy, FPR and FNR of different defense mechanisms for two attacks. For Coremelt attack, there are a few high-rate flows so that the accuracy of Ripple and Mew is high. However, we notice that Ripple has a small chance (0.2% FPR) to misclassify benign flows as malicious. According to the mitigation mechanism in Ripple, these benign flows will be directly added to blocklist and droped. For Crossfire attack, there are a lot of low-rate flows. To store excess flows, the accuracy of Ripple is quite low (18.62%), causing it to miss most low-rate flows and fail to identify all of malicious bots (100% FNR). On the contrary, Mew distributes flows into multiple switches and reduce storage overhead. Although the accuracy is still low (69.57%), meaning some flow collide with each other and a part of low-rate flows are ignored, it is enough for detecting most low-rate flows (>500) and identify all of malicious bots (0% FNR).

To further show the impact of accuracy on FNR, we set a different threshold for Crossfire defense. If the number of low-rate flows between two hosts is larger than a predefined threshold, they will be tagged as suspicious. As shown in Fig. 10, Mew classifies most of the malicious flows under different threshold settings. Ripple can only identify a part of malicious flows when we set a low threshold such as 100. However, a low threshold can increase the false positive rate (FPR) because legitimate users can also generate some low-rate flows.

C. Cooperative Defense Overhead

We compared the communication overhead of centralized global synchronization (leader mode [12]), decentralized global synchronization (p2p mode [24]) and group synchronization (Mew). As shown in Fig. 11, Mew introduces the least communication overhead. When the size of a network increases, the communication overhead remains constant because adversaries can only congest a few target links and group synchronization only happens in the related switches. On the contrary, global synchronization introduces much more communication overhead. In p2p mode, each node exchanges messages with other switches, so they have the same communication overhead. In leader mode, the leader (i.e., the central entity) is responsible for collecting and synchronizing information. Thus, the communication overhead of the leader is highest. Other switches (followers) still have to receive messages even if they do not need them, so their communication overhead is higher than Mew's.

D. Mitigating a single LFA

In this section, we evaluate the effectiveness of Mew against different kinds of LFAs. For each experiment, the attacker chooses a single attack strategy to congest target links. For each type of LFA, we deploy a defense program to the switch.

Coremelt attack. As shown in Fig. 12(a-c), both Mew and Ripple can identify most malicious flows and mitigates the congestion quickly. RADAR takes 22s to mitigate attacks due to a long query period. We find 1 of 500 background flows are dropped under Ripple's defense mechanism. The reason is Ripple has a higher collision rate as we discussed in §VIII-B.

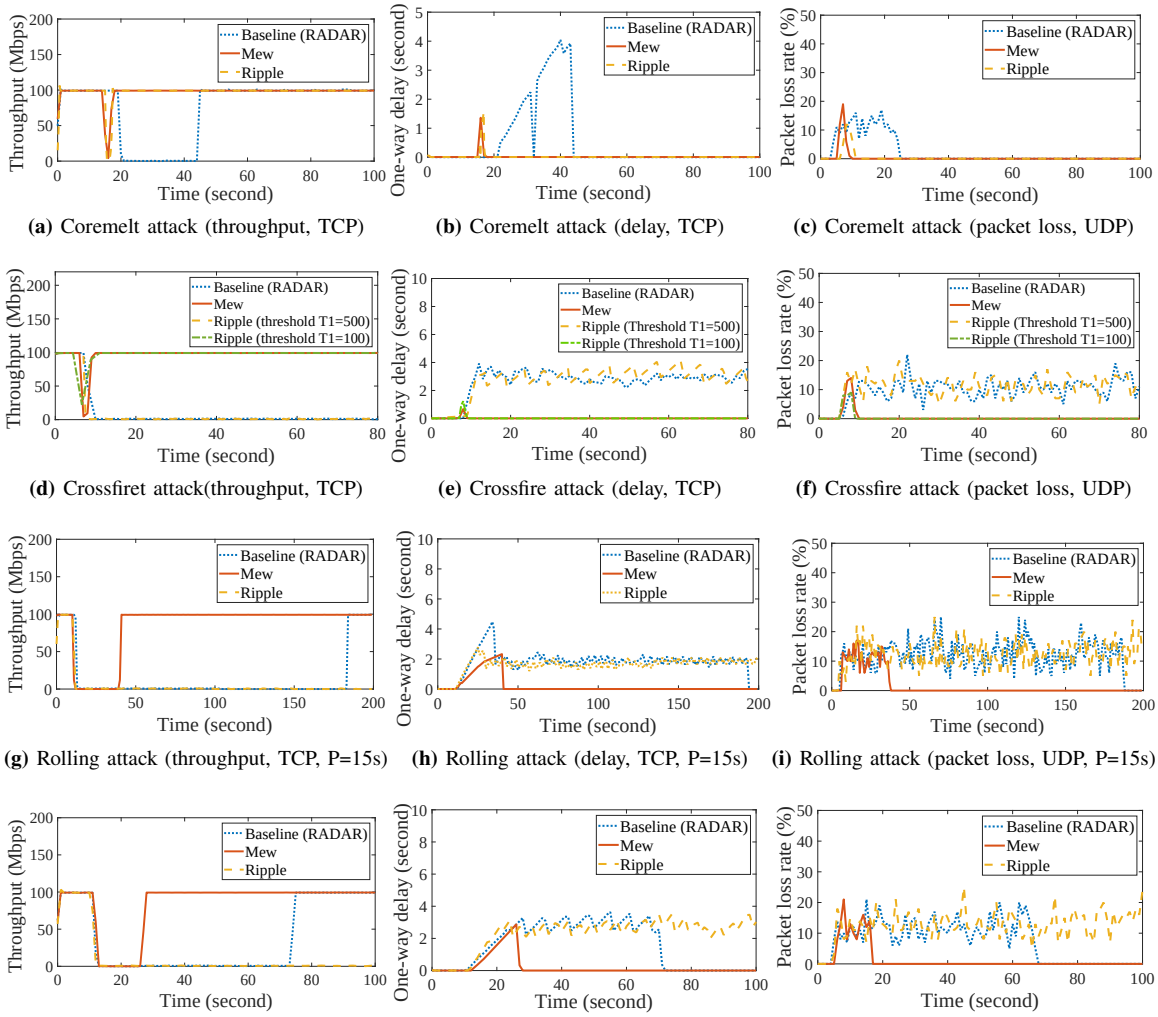
Crossfire attack. To defend against Crossfire attacks, we use a similar mitigation policy to Ripple's [24]. We record the flow speed of each flow and count the number of low-rate flows. If a source IP establishes too many ($>T_1$) low-rate flows with the same destination, it is tagged as suspicious and is rerouted to a honeypot. Here we set T_1 = 100 and 500. As shown in Fig. 12(d-f), when T_1 = 100, both Mew and Ripple can classify most malicious flows and mitigate the congestion. When T_1 = 500, Mew also classifies most malicious flows and the congestion is mitigated. However, Ripple cannot identify enough low-rate flows so that no malicious IP address is detected. The reason is that many low-rate flows are conflicted with each other. Therefore, they show a higher speed and evade the low-rate traffic detector.

Rolling LFA. For the rolling Crossfire attack, we set different rolling periods (e.g., P=5s or P=15s). Once a switch detects a link congestion event, it sends the event to other switches. When the number of congested links exceeds a threshold α and the number of link utilization changes exceeds a threshold β, the defense modules are activated. Then, a locator analyzes the correlation between the link utilization and each flow. As shown in Fig. 12(g-i), Ripple cannot identify malicious flows because their patterns are not low-rate. Instead, we need to record a speed and a speed change of each flow to capture a rolling pattern. The baseline (RADAR) needs a longer time to mitigate attacks (about 180s and 60s). Mew can locate malicious flows within 10 or 30 seconds[10] because programmable switches do not need to query flow states elsewhere. An interesting thing is that RADAR and Mew both detect rolling crossfire with a period of 5s faster. It is because a faster rolling attack shows a more obvious rolling pattern. On the contrary, a slow rolling attack has a weak rolling pattern but defenders have enough time to analyze it.

E. Mitigating dynamic LFAs

In this section, we evaluate the effectiveness of Mew to mitigate dynamic LFAs. Every T seconds, the attacker randomly picks up and launches an attack from 3 LFAs, i.e., Coremelt, Crossfire, and Pulsing attacks. We compare the mitigation effectiveness, and interruption time among Mew, Ripple, and an SDN-based defense system (RADAR). Ripple proposes a multi-vectored defense including Coremelt and Crossfire defense functions, which split the memory equally. However, this program cannot defeat Crossfire attack due to insufficient memory space. Thus, we choose the single-vectored defense programs of Ripple for comparison. Further, Ripple does not

[10] At least two periods are required to capture a rolling pattern.



(a) Coremelt attack (throughput, TCP) **(b)** Coremelt attack (delay, TCP) **(c)** Coremelt attack (packet loss, UDP)

(d) Crossfiret attack(throughput, TCP) **(e)** Crossfire attack (delay, TCP) **(f)** Crossfire attack (packet loss, UDP)

(g) Rolling attack (throughput, TCP, P=15s) **(h)** Rolling attack (delay, TCP, P=15s) **(i)** Rolling attack (packet loss, UDP, P=15s)

(j) Rolling crossfire (throughput, TCP, P=5s) **(k)** Rolling crossfire (delay, TCP, P=5s) **(l)** Rolling crossfire (packet loss, UDP, P=5s)

Fig. 12: Performance of benign traffic in different cases (attacks and defenses).

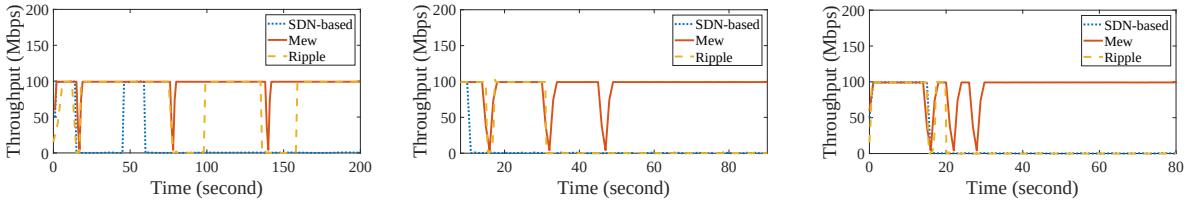
propose detectors to distinguish types of occurring attacks, so we assume that Ripple can get types of occurring attacks within 1 window (1s) by using a "magic" out-of-band detector.

Fig. 13 presents the throughput of benign traffic between users and servers. Mew always detects the attacks and mitigates them to recover the throughput between users and servers in a short time (a few seconds). After several rounds, all bots are added to suspicious lists so that attackers cannot congest links anymore. On the contrary, Ripple fails to follow the fast-changing attacks (e.g., period=15s), as shown in Fig. 13(b), (c). It is because Ripple needs to reboot and reconfigure the switch to run the new defense programs when the attack changes. In our testbed, we pre-compile all 3 defense programs but we still need more than 10 seconds to load a new program. RADAR can only detect Coremelt attack, which has an obvious pattern (i.e., malicious high-rate flows). For Crossfire and Pulsing attacks, it needs a longer time (e.g., tens of minutes) to get enough information and fails to follow the changing attacks.

We further evaluate the interruption time due to changing defense policies. RADAR is not based on programmable switches, so we compare Mew with Ripple. As shown in Fig. 14, Mew does not interrupt any flows because it can change functions on the fly. In contrast, Ripple stops handling packets whenever reloading new programs. Fig. 14 shows that three essential network functions are suspended during the updating process. We can see that: (a) a firewall is inactivated so that packets in blocklists are forwarded; (b) an NAT is stopped and thus legitimate packets cannot be forwarded correctly; (c) a load balancer is suspended and packets are concentrated on a single server, causing a load imbalance.

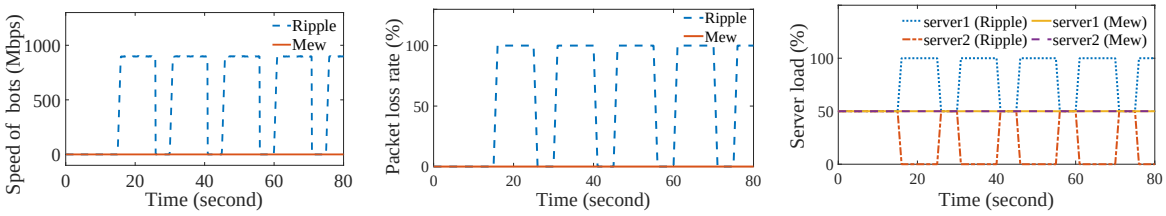
F. System overhead

We first evaluate the resource utilization of Mew. As shown in Table III, Mew mainly consumes the ALU and SRAM. Due to our design, edge switches use more computational resources (e.g., ALU) while core switches use more storage resources (e.g., SRAM). Besides, the memory (SRAM) usage of Mew is highly efficient due to the resource reallocation mechanism in



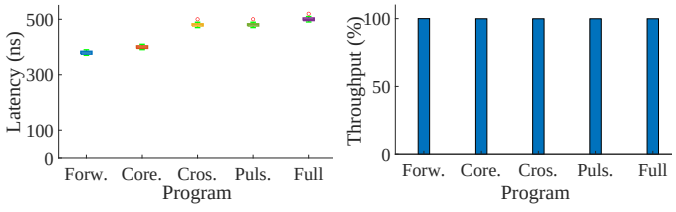
(a) Dynamic attack (T=60s) **(b)** Dynamic attack (T=15s) **(c)** Dynamic attack (T=5s)

Fig. 13: The throughput between the legitimate user and victim servers in different defense systems.



(a) Firewall interruption **(b)** NAT interruption **(c)** Load balance interruption

Fig. 14: The interruption impacts due to program reloading (every 15s, attackers change attack types)



(a) Processing latency **(b)** Throughput

Fig. 15: The impact on latency and throughput of different programs (Forward, Coremelt, Crossfire, Pulsing and Full defenses).

TABLE III: The hardware resource usage of different programs

Resource per pipeline	**Coremelt** edge/core	**Crossfire** edge/core	**Pulsing** edge/core	**Full** edge/core
Stage	10/7	11/10	12/11	12/11
SRAM%	15.0/11.9	24.4/30.1	24.4/22.3	24.4/32.0
ALU%	22.9/16.7	29.2/27.1	31.3/29.2	31.3/29.2
Hash unit%	22.2/15.3	23.6/25	27.8/30.6	27.8/30.6
VLIW%	8.1/6.8	9.1/10.2	9.4/12.2	10.2/12.2

§VI. Full defense, including Coremelt, Crossfire, and pulsing defenses, incurs the similar consumption of memory, ALU, and hash unit as the single defense for Crossfire.

We further measure the extra latency and impact on throughput introduced by Mew. We compare Mew's defense programs with a baseline program that only forwards packets (Forward). As shown in Fig. 15, Mew incurs a nanosecond-level additional latency and negligible impacts on throughput.

IX. Related Work

Traditional centralized defenses. Most traditional defense systems [11], [12], [14]–[16], [29] are centralized. They are deployed on the control plane (i.e., centralized servers) and collect network statistics from the data plane (i.e., forwarding devices). For example, RADAR [12] aims at the rolling LFAs that the adversary congests multiple links alternatively to avoid detection. RADAR tries to find out the correlation between malicious traffic and link fluctuation. By requesting flow states (e.g., with different prefixes) many times, RADAR can iteratively narrow the scope of the suspicious flow sets. Unfortunately, the request period is relatively long, making detection time-consuming and error-prone. SPIFFY [14] considers a cost-sensitive attacker who adopts an optimal and fixed strategy to send traffic. Thus, the botnets are unable to follow the changing network condition. By enlarging bandwidth shortly, defenders can identify the malicious flows, which maintain the same rate due to the fixed strategy. However, SPIFFY is limited to a single LFA (i.e., attackers never change flow speed). In the real world, attackers can change their attack strategy.

DDoS and link-flooding defense via programmable switches. To date, programmable switches have emerged for the use of security functions such as DDoS or link-flooding defenses [24]–[26]. Ripple [24] develops a decentralized link-flooding defense system whose defense policies require switches recording flow-level states locally. Unfortunately, the memory of current programmable switches is insufficient to correctly record too many flows while most LFAs will generate tens of millions of flows. In addition, Ripple misses discussing how to distinguish types of occurring attacks and reconfigure programmable switches seamlessly, so it fails to counter dynamic LFAs. Poseidon [25] designs a traffic scrubbing center to counter DDoS attacks. By rerouting all traffic to the

center, the malicious traffic can be identified by programmable switches, which run a series of defense mechanisms. However, when adversaries change attack strategies, Poseidon needs to reroute all traffic to servers, which have relatively lower throughput. Attackers can leverage that and force Poseidon to stay low-throughput. Jaqen [26] deploys a programmable defense system on ISP networks so there are more available switches for countering different kinds of DDoS attacks. Edge switches usually run detection modules and a control plane distributes traffic to switches with mitigation modules. However, the dynamic defense mechanism of Jaqen requires shutting down all filters (i.e., for defense) during the updating process. Thus, malicious traffic can pass through protected regions if attackers trigger reconfiguration. In addition, both Poseidon and Jaqen are designed for countering volumetric DDoS attacks instead of LFAs.

X. Discussion

Possible attacks. An adversary may try to disturb Mew by using the following attacks:

- **Flow flood.** Like many flow tracking systems, Mew's edge switches can run out of memory if attackers create a lot of flows. As a result, the performance of distributed storage is reduced. One solution is to set a filter to skip the tracking of flows with small packets (i.e., tracking flows with at least one big packet) because the bots should connect to the edge switch as the first hop, which restricts the uplink bandwidth[11]. However, some flows related to LFAs may also be skipped if LFAs generate small packets, e.g., sending ICMP requests (64B) instead of HTTP requests (512B), but this behavior can decrease the stealthiness of original attacks due to higher packet per second. Besides, attackers can fragment packets. By deliberately creating a small "first" fragment, the "first" fragment with 5-tuples will be skipped by Mew. Meanwhile, Mew cannot extract the flow pattern from the subsequent fragments without 5-tuples. However, ISP networks usually adopt load balancing. Without layer 4 headers (e.g., TCP headers), the fragments can be forwarded via different routing paths, so LFAs lose their potency.
- **One-time flow flood.** Mew's edge switches only track flow states after detecting LFAs and activating mitigation modules. If attackers can accurately capture the tracking window, they can launch a one-time flow flood attack instead of a long-term flow flood attack. In this case, we can reset Mew's defense type to "discard" flow states of old flows to release consumed memory after a one-time flow flood attack. Although this solution can move the defense period back, the length of the defense period is the same.
- **Disrupting defense.** An adversary with prior knowledge about Mew may tamper or spoof negotiated packets to disrupt Mew's distributed storage protocol (e.g., missing recording or sharing wrong information). However, we have assumed that the communication between switches cannot be tampered with. The spoofing packets from external adversaries will be filtered at the first hop if they contain the special packet headers related to Mew's protocol.

Limitations. We also analyze the limitations of Mew.

- Distributed storage may lose some fine-grained information such as packet loss position. However, our goals are detecting attacks and classifying malicious traffic instead of achieving complete network monitoring. Besides, we can also adjust the storage strategy by setting the mode to **full**.
- Once a switch fails, Mew cannot recover the information fully. To mitigate that, we can back up key states of each switch periodically so we only lose one-period information, which is acceptable in link-flooding defense scenarios.
- Attackers may change their strategies to bypass defense systems. For example, each bot establishes connections with more decoy servers. In this case, we can count the number of low-rate flows of a host instead of a host pair. Second, attackers may change bot sets rapidly if they have several times bots required to congest the target links[12]. As a result, Mew needs a longer learning phase and there could accumulate many benign flows, causing the memory to run out. A straightforward but costly solution is to deploy more switches because Mew provides scalability. Another solution is to limit the number of hosts (e.g., based on network prefix) on a link so that the number of available bots is restricted. Meanwhile, some benign users may use sub-optimal routing paths with higher latency.

XI. Conclusion

In this paper, we design a memory-efficient and adaptive link-flooding defense system based on programmable switches. We develop a lightweight distribution protocol and show how to schedule the massive number of flow states on different switches to reduce the memory bottleneck. A series of APIs are designed to support multi-grained and dynamic co-defense against complicated LFAs. Our evaluation demonstrates that Mew can defend against various LFAs with high accuracy and zero interruption even if attack strategies change in seconds.

Acknowledgements: We want to thank the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 1700544, 2148374, and 2226339, DHS Grant No. 518700-00001, ONR Grant No. N00014-20-1-2734, Hong Kong ITF Project (GHP/052/19SZ), the Research and Development Program of Shenzhen under Grants SGDX20190918101201696, and The Major Key Project of PCL (PCL2021A15). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, DHS, ONR, and any agency mentioned above.

[11] An attacker with 10 Gbps uplink bandwidth can create 19M unique ICMP requests (64B) per second while she can only create 2.5M unique requests per second if the packet size is increased to 500B to pass the filter.

[12] Some botnets have millions of bots (e.g., Mirai), and attackers can use a part of the bots which have paths passing the target links. In [5], 10,000 bots are required to congest a 40 Gbps link. Our testbed has 1 Gbps links, so we only use 250 bots, 30% of the memory of a single pipeline (i.e., 7.5% of the memory of a switch with 4 pipelines), and 5 switches. With Mew, 10 switches with the tofino3 chip with 4x bigger memory than ours can store 25 million flows generated by 25,000 bots (100x more than our settings).

References

[1] "Q1 2018 DDoS Trends Report". https://bit.ly/2JDR1D9.

[2] S. Moss. Major DDoS attack on DYN disrupts AWS, Twitter, Spotify and more. [Online]. Available: http://www.datacenterdynamics.com/content-tracks/securityrisk/major-ddos-attack-on-dyn-disrupts-aws-twitter-spotify-and-more/97176.fullarticle

[3] D. Pauli. Chinese gambling site served near record-breaking complex DDoS. [Online]. Available: https://www.dailydot.com/debug/lizard-squad-hackers/

[4] R. Rasti, M. Murthy, N. Weaver, and V. Paxson, "Temporal lensing and its application in pulsing denial-of-service attacks," in *S&P '15*.

[5] M. S. Kang, S. B. Lee, and V. D. Gligor, "The Crossfire attack," in *S&P '13*.

[6] A. Studer and A. Perrig, "The Coremelt attack," in *ES0RICS '09*.

[7] Can a ddos break the internet? sure… just not all of it. [Online]. Available: https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/2/

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

[9] Cyberattack causes ic internet connection to cut out. [Online]. Available: https://theithacan.org/news/cyberattack-caused-ic-internet-connection-to-cut-out/

[10] D. Gkounis, V. Kotronis, and X. Dimitropoulos, "Towards defeating the crossfire attack using sdn," in *arXiv '14*.

[11] L. Xue, X. Ma, X. Luo, E. W. Chan, T. T. Miu, and G. Gu, "Linkscope: Toward detecting target link flooding attacks," *TIFS 18*.

[12] J. Zheng, Q. Li, G. Gu, J. Cao, D. K. Yau, and J. Wu, "Realtime DDoS defense using COTS SDN switches via adaptive correlation analysis," in *TIFS '18*.

[13] J. Wang, R. Wen, J. Li, F. Yan, B. Zhao, and F. Yu, "Detecting and mitigating target link-flooding attacks using sdn," in *TDSC '18*.

[14] M. S. Kang, V. D. Gligor, V. Sekar *et al.*, "Spiffy: Inducing cost-detectability tradeoffs for persistent link-flooding attacks." in *NDSS '16*.

[15] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI '14)*.

[16] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi *et al.*, "Flow event telemetry on programmable data plane," in *SIGCOMM '20*.

[17] S. B. Lee, M. S. Kang, and V. D. Gligor, "Codef: Collaborative defense against large-scale link-flooding attacks," in *CoNEXT '13*.

[18] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors."

[19] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *NSDI '18*.

[20] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Contra: A programmable system for performance-aware routing," in *NSDI '20*.

[21] (2018) "Multi-function Platform for Cloud Networking.". https://bit.ly/2JhJQB6.

[22] (2018) "EX9200-Flexibility and scalability for business agility and growth.". https://juni.pr/2JnC1tY.

[23] (2018) "Google Cloud using P4Runtime to build smart networks.". https://bit.ly/2Q7zG6B.

[24] J. Xing, W. Wu, and A. Chen, "Ripple: A programmable, decentralized link-flooding defense against adaptive adversaries," in *USENIX Security '21*.

[25] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating volumetric DDoS attacks with programmable switches," in *NDSS '20*.

[26] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric DDoS attacks with programmable switches," in *USENIX Security '21*.

[27] "Intel® Tofino™ 2". https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html.

[28] J. Cao, Q. Li, R. Xie, K. Sun, G. Gu, M. Xu, and Y. Yang, "The CrossPath attack: Disrupting the SDN control channel via shared links," in *USENIX Security '19*.

[29] L. Wang, Q. Li, Y. Jiang, X. Jia, and J. Wu, "Woodpecker: Detecting and mitigating link-flooding attacks via sdn," *Computer Networks 18*.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review '08*.

[31] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Computer Communication Review '13*.

[32] "Edge-Core Networks - WEDGE100BF-65X-O-AC-F-US QSFP 100g". https://bit.ly/2HiZFW0.

[33] (2018) "Compare Kemp LoadMaster, F5 Big-IP & Citrix Netscaler.". https://kemptechnologies.com/compare-kemp-f5-big-ip-citrix-netscaler-hardware-load-balancers/.

[34] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *SIGCOMM '17*.

[35] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *SOSP '17*.

[36] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-RTT coordination," in *NSDI '18*, 2018, pp. 35–49.

[37] N. K. Sharma, A. Kaufmann, T. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation," in *NSDI '17*.

[38] L. Zeno, D. R. Ports, J. Nelson, and M. Silberstein, "Swishmem: Distributed shared state abstractions for programmable switches," in *HotNet '20*.

[39] D. Kim, Z. Liu, Y. Zhu, C. Kim, J. Lee, V. Sekar, and S. Seshan, "Tea: Enabling state-intensive network functions on programmable switches," in *SIGCOMM '20*.

[40] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[41] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[42] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[43] H. Wang, L. Xu, and G. Gu, "Floodguard: A dos attack prevention extension in software-defined networks," in *DSN '15*.

[44] T. P. A. W. Group. In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[45] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *SIGCOMM '17*.

[46] W. Project. Mawi working group traffic archive. [Online]. Available: http://mawi.wide.ad.jp/mawi/

[47] L. Yu, J. Sonchack, and V. Liu, "Mantis: Reactive programmable switches," in *SIGCOMM '20*.

[48] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, "P4NFV: An NFV architecture with flexible data plane reconfiguration," in *CNSM '18*.

[49] P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html.

[50] "Barefoot® Tofino™ ". https://www.barefootnetworks.com/technology/#tofino.

[51] (2022) Mew prototype repo. [Online]. Available: https://github.com/hczhou574/Mew-prototype

[52] Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries. https://github.com/jiarong0907/Ripple.

[53] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre, "D-itg distributed internet traffic generator," in *QEST '04*.

[54] Radar: A sdn-based realtime denial-of-service defense system. [Online]. Available: https://github.com/zhengjingts/floodlight_radar

[55] The internet topology zoo. [Online]. Available: http://www.topology-z

[56] P. L. Consortium *et al.*, "Behavioral model (bmv2)," *URL: https://github.com/p4lang/behavioral-model [cited 2020-01-21]*, 2018.

Appendix

In this appendix, we include more cooperative defense cases for Coremelt attack and pulsing attack, and some details of our simulation topology setups.

A. Cooperative defense cases

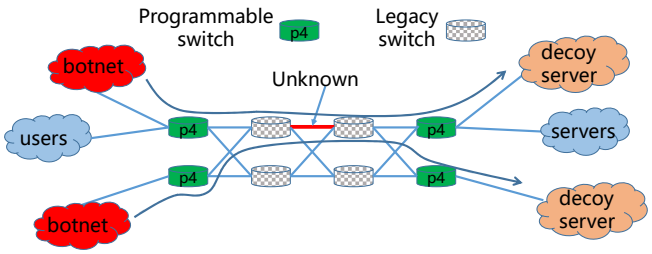


Fig. 16: The programmable switches cannot detect the link congestion directly

Case Study: switch-native Coremelt defense. In Coremelt attack, the attacker needs to instruct the bots within the victim area to congest the links connecting to the Internet. Normally, the bot number is limited so that each bot needs to generate high-speed flows. Therefore, we can monitor the aggregated flow speed for per source IP address and block the suspicious IP address as Ripple does [24]. However, Ripple fails to detect the congestion happened between the non-programmable switches (i.e., shadow area) and the mitigation will not be activated, as shown in Fig. 16. In contrast, Mew supports a group-grained synchronization to recover the link states of the shadow area. The following code shows an example:

```
#Coremelt Detection
Monitor(byte_count[port],full)
Sync(byte_count[port],100ms,group)
Trigger(byte_count[port]>1Gb, Activate(Coremelt))
```

Each programmable switch records the byte count of ports (i.e., links) using full mode (line1). To recover the link states of the shadow area, the programmable switches synchronize byte count of ports per 100ms using group mode (line2). Whenever the aggregated byte count of a port (a link) is larger than a threshold, the Coremelt mitigation is activated:

```
#Coremelt Mitigation
Monitor(byte_count[IP], full)
Trigger(byte_count[IP]>100Mb, block(IP))
```

The activated switch first records the byte count of each host (i.e., each source IP address) (line1). If the byte count of a host is larger than a threshold, the switch adds its IP address to a suspicious list and blocks the IP address.

Case Study: switch-native pulsing defense. In pulsing attack, adversaries send high-speed traffic for a short term to congest links shortly. Before triggering defenses, adversaries slow down the speed of traffic. After a period, adversaries repeat this process. To counter this attack, defenders can count the times of changes of links. If the load of a link changes dramatically and frequently in a short time, flows that dramatically change speed should be responsible to the attack. The following code shows an example:

```
#Pulsing Detection
Monitor(byte_count[port], full)
Trigger(byte_count[port]>1Gb && passing_time>10ms,
    Sync([port,congestion], group))
Trigger(byte_count[port]<500Mb && passing_time>10ms,
    Sync([port,recovery], group))
Trigger(link_change[port]>3, Activate(Pulsing))
```

Once a port/link begins to congest (line 1), an event including the port id and congestion flag is synchronized to a given group (line 2~3). The event is sent only once every 10ms to avoid repeatedly sending. Similarly, once the load of the link is less than a threshold, an event containing the port id and recovery flag is synchronized to a given group (line 3~4). When the times of dramatic changes of a link exceeds a threshold (e.g., 3), the pulsing mitigation is activated.

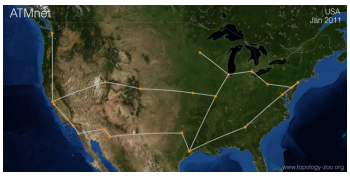
```
#Pulsing Mitigation
Monitor(speed_change[flow], dist)
Trigger(speed_change[flow] == cur_link_change),
    count[flow]=count[flow]+1)
Trigger(count[flow]>3, output=honeypot)
```

The activated switch first records the byte count of each flow. If the speed of a flow changes as the load of the victim link changes, the count of this flow is incremented. When the count of a flow exceeds a threshold, we consider this flow is suspicious and reroute it to a honeypot for further analysis.

B. Real-world topology setups

We use three topology setups from Topology Zoo [55]. There show different network types, including linear, star and tree. As shown in Fig. 17. Table IV shows the number of nodes and links of each topology setup.

(a) Linear type topology



(b) Tree type topology

(c) Star type topology

Fig. 17: Real-world topology setups (Topology Zoo)

TABLE IV: The hardware resource usage of different programs

Topology	Renua (linear)	Atmnet (tree)	Goodnet (star)
Node (#)	11	22	17
Link (#)	10	22	31