Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries

Jiarong Xing Wenqing Wu Ang Chen
Rice University

Abstract

Link-flooding attacks (LFAs) aim to cut off an edge network from the Internet by congesting core network links. Such an adversary can further change the attack strategy dynamically (e.g., target links, traffic types) to evade mitigation and launch persistent attacks.

We develop Ripple, a programmable, decentralized link-flooding defense against dynamic adversaries. Ripple can be programmed using a declarative policy language to emulate a range of state-of-the-art SDN defenses, but it enables the defenses to shapeshift on their own without a central controller. To achieve this, Ripple develops new defense primitives in programmable switches, which are configured by the policy language to implement a desired defense. The Ripple compiler generates a distributed set of switch programs to extract a *panoramic* view of attack signals and act against them in a fully decentralized manner, enabling successive waves of defenses against fast-changing attacks. We show that Ripple has low overheads, and that it can effectively recover traffic throughput where SDN-based defenses fail.

1 Introduction

Distributed denial-of-service (DDoS) attacks [5, 9, 10] have always been a significant threat, but as of late, adversaries have taken DDoS attacks to the next level. In a *link-flooding attack* (LFA), an attacker can surgically remove an edge network from the Internet without it perceiving any attack traffic. Such an adversary identifies a set of network links that serve the victim edge, and orchestrate attack traffic to congest these links [40]. Victim destinations may experience severe performance degradation or complete disconnection. To mitigate these new attacks, traditional endpoint-based DDoS protections [26, 29, 50, 61] are fundamentally handicapped—since attack traffic never has to reach endpoint destinations, defenses must be deployed inside the network core.

Link-flooding attacks are significantly more challenging than traditional volumetric DDoS in their detection, classification, and mitigation. *(a)* Unlike volumetric attacks, which are easy to detect by thresholding, LFAs can leverage low-rate flows to stay under the detection threshold [40]. *(b)* Classification algorithms are also challenging to design: although some LFA traffic has distinctive features (e.g., spoofed or UDP-based flows) [53], more advanced attacks rely entirely on legitimate flows that are indistinguishable from normal traffic [40]. *(c)* Adversaries can launch adaptive attacks [40] that dynamically change target links or traffic types while targeting the same victim network. An effective LFA defense must simultaneously match the significant *diversity* and *dynamicity* of the attacks.

A promising line of work has developed link-flooding defenses [4,28,39,43,55,61] based on software-defined networking (SDN) [46]. In this architecture, defense algorithms run as software apps in a centralized controller; this *programmability* is key to implementing a wide range of LFA defenses in an otherwise fixed-function network. Although SDN switches are hardwired for packet forwarding, the software defense apps can receive OpenFlow messages from the switches at runtime, construct a global defense view, and compute new defense decisions when needed. This *feedback loop* enables the hardwired switches to work with the controller for dynamic defenses. The switches send traffic samples or statistics to the software apps, which run detection, classification, and mitigation algorithms. The defense decisions are then populated to each switch for link-flooding defense.

However, these defenses share a common assumption and limitation—the efficacy of the feedback loop itself. After the controller samples, recomputes, and reinstalls defense decisions to the switches, the decisions must remain effective for a sufficiently long period of time. If the adversary rapidly changes her attack strategies, e.g., traffic types or target links, then the controller decisions would constantly lag behind. Such dynamic adversaries can force the defense apps to always act on stale data, which would in turn result in suboptimal defenses or even additional harm. These TOCTOU-style strategies have been proven effective and are known as *rolling attacks* [40]. Developing effective defenses against adaptive adversaries remains an open research question.

In this paper, we propose Ripple, a *programmable* and *decentralized* link-flooding defense against adaptive adversaries. Like OpenFlow-based SDN defenses, Ripple can be programmed to implement a wide range of defenses; however, the defenses can shapeshift on its own without centralized software control. This makes Ripple a powerful match against dynamic attacks. In fact, Ripple can activate new defenses as fast as attack waves propagate. Defense decisions only take RTT-timescale—the diameter of the network—to take effect. This enables successions of defense waves to replace older ones as the attacks change. The technology enabler for Ripple is the emergence of *programmable switches* [18, 19], which Ripple leverages to develop new defense primitives in switch hardware. However, raw hardware speeds are only a starting point; there is a range of challenges in the Ripple defense.

The key challenge Ripple needs to address is *decentralization*; the same factor that enables rapid attack responses also brings new design challenges. In traditional SDN, the software controller has a centralized view, so it has a global vantage point to enforce network-wide defense. Ripple, however, eliminates central control, so it needs to choreograph the switch-local decisions carefully for synchronized defense. To this end, Ripple develops a policy language, a compiler, and a distributed runtime for link-flooding defense. The Ripple language exports a key abstraction that we call a defense *panorama*, whose goal is to precisely extract network-wide threat signals from high-speed traffic. Users of Ripple program against this panoramic view, and the compiler generates switch programs that implement the panoramic defense in a fully distributed manner. Attack signals are first extracted by switch-local primitives, and then propagated by the distributed runtime protocol for view synchronization.

In summary, we make the following contributions:

- A decentralized defense architecture for mitigating adaptive link-flooding attacks.
- The Ripple system, which develops defense primitives in programmable switch hardware. It can be programmed by a policy language to emulate state-of-the-art SDN defenses. The compiler generates switch programs to implement the policy. The defense programs run in a fully distributed manner to react to changing attacks without central control. The runtime protocol synchronizes switch-local views for panoramic defense.
- Hardware and software prototypes[1], and extensive evaluation that demonstrates defense effectiveness.

We then describe related work and conclude the paper.

2 Overview

Figure 1 illustrates the key mechanisms of link-flooding attacks [39, 40, 53]. The adversary controls a large botnet, and she constructs a link map using traceroute-like utilities. Next, she identifies a set of critical links that carry all or most of the victim destination's traffic. She creates congestion at some or all of the critical links using her botnet to degrade the victim's network performance.

2.1 Key challenges

Diversity. The first challenge in link-flooding attacks is the wide range of possible attack strategies [4, 28, 39, 40, 43, 53, 55, 61]. We describe three representative attacks and their adversary assumptions [39, 40, 53]. In all cases, the attack traffic always originates from the bots controlled by the adversary, but traffic types and patterns vary: *(a) Coremelt* [53] uses

[1] https://github.com/jiarong0907/Ripple

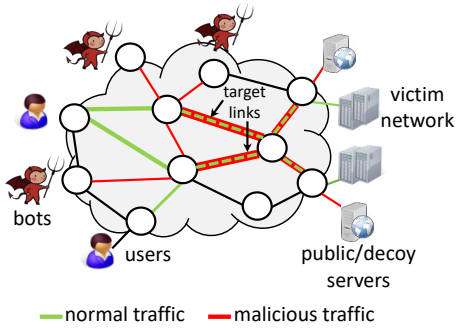


Figure 1: Link-flooding attacks congest network links to disconnect a target victim edge.

bot-to-bot traffic patterns, and the attack flows are volumetric in nature (i.e., UDP or spoofed TCP traffic). *(b) Crossfire* [40] generates traffic from bots to public servers that are not controlled by the adversary; the attack traffic consists of normal web requests, which are typically low-rate and regulated by TCP congestion control. Since these flows are protocol-conforming TCP requests, the attack traffic is "indistinguishable" [40] from flash crowds. *(c) SPIFFY* [39] falls somewhere between (a) and (b). It assumes a "cost-sensitive" adversary [39] that generates as much traffic as possible from each bot—for instance, because the attacker wants to maximize the utility of a moderate-sized botnet. The individual attack flows are TCP-conforming instead of volumetric floods.

Takeaway: Although all these attacks aim to congest network links, they require different detection, classification, and mitigation algorithms. This calls for a *programmable* defense that can be customized to mitigate a host of different attacks.

Dynamicity. Another formidable feature of link-flooding attacks is dynamicity. The feasibility of *rolling attacks* was first identified and validated by Crossfire [40], but it remains an open question to this day. *(a) Changing targets:* An adversary can dynamically shifts its attack traffic to a different set of critical links while attacking the same victim destination. The attack can therefore evade non-adaptive defenses and persist for a long time [40]. *(b) Mix-vectored attacks:* The adversary can easily change the attack traffic and patterns, or use mix-vectored attacks—related to the diversity of attack strategies discussed above. *(c) Pulsewave attacks:* The adversary can generate short-lived attack pulses, which may have already subsided by the time that defense algorithms are activated [2]. Recent results have also shown that pulses can be synchronized to arrive at the same location very precisely [49].

Takeaway: Link-flooding attacks can rapidly change. Defenses against adaptive adversaries must explicitly account for such *dynamicity*. This is a key design goal of Ripple.

2.2 State of the art

State-of-the-art defenses developed in the security community are based on OpenFlow SDN [4, 28, 39, 43, 55, 61]. At the heart of SDN-based defenses is a central controller that can host a

range of "defense apps". These apps receive traffic samples or statistics from the OpenFlow switches, perform detection and classification, and installs new mitigation decisions back to the switches. The operator can customize each step using attack-specific algorithms.

Detection can be achieved by collecting link utilization data from all switches to the SDN controller, either periodically or when certain links are overwhelmed. The SDN apps have a global view of the network by virtue of running in the central controller. This provides a useful vantage point to collect network-wide traffic statistics and detect which parts of the network are congested.

Classification algorithms implement attack-specific logic for identifying malicious traffic—e.g., volumetric flows in Coremelt [53], or cost-sensitive flows in SPIFFY [39]. Attempts to classify Crossfire-like attacks must be aware of false positives and negatives that may result, as the attack flows are indistinguishable from flash crowds [40].

Mitigation algorithms include dropping malicious packets if attack traffic has distinguishable features [53], or more conservative strategies that reroute traffic away from congested links—e.g., for "indistinguishable" attack flows where classifiers inevitably produce inaccuracy [40].

Compared to traditional networks, SDN-based solutions enable *programmable* defenses that can be customized for different types of link-flooding attacks.

2.3 Limitations of existing work

However, SDN-based defenses also have notable limitations, and they stem from the fact that a *feedback loop* is required for the defense.

Rolling attacks [40] remain a prominent open question in link-flooding defense. An adaptive adversary can rapidly change her attack strategies to prevent an effective feedback loop from forming—e.g., by dynamically changing victim links, attack traffic types, patterns, or by using short-lived pulses. SDN apps are forced to always use an outdated view of the attack. Concretely, the defense lags behind due to three latency components: a) sampling latency from OpenFlow switches to the controller, b) computing new responses in the SDN apps using traffic engineering algorithms [34, 36], and c) installing the decisions back to the network. When defense decisions are installed back to the switches, the attack strategies may have already changed. Moreover, detection algorithms that desire higher accuracy may require higher sampling rates for fine-grained analysis; but this in the case of rolling attacks may further increase defense latency.

2.4 A programmable, decentralized defense

The key contribution of Ripple is to address the limitations of existing work in mitigating changing attacks. The design of Ripple involves a number of techniques.

#1: Programmable data planes. Ripple leverages an emerging technology trend in the networking community. Programmable switches have reconfigurable data planes, which can be programmed in P4 [12] for flexible packet processing. A P4 program can customize the switch pipeline with new protocols, header types, and sophisticated processing logic. Ripple develops defense primitives that directly run in programmable switch hardware. Unlike SDN apps that can only process downsampled traffic or aggregate statistics, programmable switches can inspect *every single packet* with nanoseconds of extra latency at full linespeed.

#2: Decentralized defense. Raw hardware speed, by itself, is far from sufficient for mitigating adaptive attacks. This is because a link-flooding defense requires a network-wide view—the detection, classification, and mitigation policies need to precisely identify the locations and types of attack waves, which propagate across switches and change over time. If every switch simply takes actions based on its local view, the collective effect may be incoherent or even chaotic. Ripple addresses this by developing a policy language to precisely capture a defense *panorama*—a global, real-time view of attack waves and how they propagate through the network. Users of Ripple program against this panoramic view without having to reason about switch-local actions; rather, our compiler automatically generates the defense programs at every switch. To match the dynamicity of adaptive adversaries, Ripple constructs this panoramic view in a fully decentralized manner, using a distributed protocol to synchronize switch-local views. This enables successive defense waves to take effect against fast-changing attacks as they propagate.

3 Programming the Panoramic Defense

The first set of challenges in Ripple stems from the need for *programmable* defense. In OpenFlow-based SDN, the defense apps are simply software modules that can be plugged into the SDN controller platform—e.g., OpenDaylight [16]/Beacon [14] supports Java apps, POX [15] supports Python apps, and NOX [13] supports C++. However, the P4 programming model is highly restricted; it does not support familiar program constructs such as loops or recursions. The program also needs to fit in tight hardware resource constraints. Therefore, defenses mostly have to work with compact, array-like data structures. Moreover, a P4 program only specifies per-switch processing, but Ripple needs a panoramic view of the entire network. This leads to design challenges for a) providing a panoramic view of the network, b) emulating state-of-the-art SDN defenses in a restricted programming model, and c) shielding switch programming details from the defense.

3.1 The panoramic view

Ripple proposes a new abstraction, the defense *panorama*, which describes the types of signals that are relevant for link-

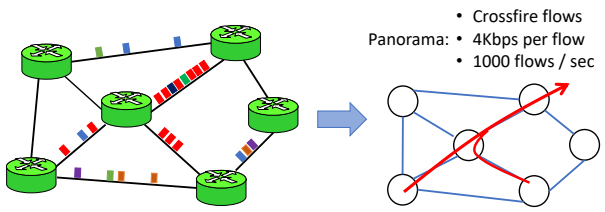


Figure 2: Ripple provides a defense panorama for network operators to program defense policies easily.

Primitive	Description
`panorama(w)`	The panorama abstraction (`w`: time window)
`map(key, vh, f)`	Apply `f` to `key`, and emit virtual header `vh`
`reduce(key, vh, agg)`	Aggregate by `key` and emit virtual header `vh`
`filter(p)`	Apply predicate `p` to the packet
`distinct(key)`	Emit unique headers as defined by `key`
`when(pred, f)`	If `pred` is true, apply `f`
`zip(key, l1, l2)`	Join two lists `l1` and `l2` by `key`

Table 1: The key language constructs in Ripple.

flooding defense. A panorama extracts network-wide threat signals from switch-local traffic using the defense policy, and it abstracts away unrelated signals so that they do not clutter the defense view. More concretely, Ripple captures a series of panoramic snapshots of the entire network, and precisely zooms in on the attack signals. These snapshots present a global view of attack waves, including their current location, traffic composition, trajectory through the network, and how they evolve over time. Under the hood, the defense policy is compiled to a distributed set of P4 programs by the Ripple compiler. Users of Ripple only need to program against the panoramic view, without having to directly reason about P4-level switch programs.

Ripple takes inspiration from recent work on network telemetry [30, 48] that customizes Spark-style functional operators [60] for traffic measurements and monitoring. Table 1 shows the key primitives in Ripple, which are customized to support detection, classification, and mitigation algorithms for link-flooding defense. The input to a Ripple policy is (logically) every single packet in the network and its trajectory over time. The Ripple operators record and transform the packet headers at every switch locally, filtering out attack-unrelated signals. Link-flooding attack signals, on the other hand, are promoted from a switch-local view to be globally visible. They are materialized as a set of panoramic variables by a distributed runtime protocol. From Ripple's perspective, a packet has physical headers such as TCP/IP, but it has additional "virtual" headers such as timestamps, link locations, or any attack signals as defined by the policy. A parameter *w* specifies the frequency at which the panoramic snapshot should be taken. Each snapshot is captured by executing the policy body, which consists of a series of functional operators.

In the following subsections, we showcase the expressiveness of Ripple by first emulating a range of SDN-based defenses in recent work and then developing new defenses.

3.2 The Crossfire defense

We start by describing how Ripple supports the Crossfire defense using the panoramic view.

Detection. The detection policy looks for significant congestion (>80% link utilization) anywhere in the network, and it populates the panoramic variable 'victimLks' every 100 ms:

```
1  victimLks = panorama(100ms)
2  .map(link, ld, f_load)
3  .filter(ld > 80)
```

Line 2 maps the virtual header field 'link', which indicates the location of the packet, into its current link load 'ld'. By default, header fields that are not referred to by the policy, such as TCP/IP headers in this policy, are left untouched. Here, the link load computation uses an intrinsic function 'f_load', which will be expanded by the compiler; internally, it computes an Exponentially Weighted Moving Average (EWMA) of the traffic rate at 'link'. Line 3 checks the newly generated virtual header 'ld' against a threshold. A packet's headers (virtual and physical) are kept strictly local unless 'ld' passes the filter, in which case they are populated as panoramic variables and can be accessed by all switches via the 'victimLks' variable. In fact, the Ripple compiler later will see that only the size of 'victimLks' is needed, so it only propagates data needed for computing the set size; all other header fields are abstracted away from the panorama.

Classification. The Crossfire flows [40] are low-rate HTTP requests. One could identify such flows for special treatment (e.g., rerouting these flows), as long as we are aware that Crossfire classifiers may result in false positives/negatives:

```
1  suspicious = panorama(100ms)
2  .filter(victimLks.sz > 3)
3  .reduce([sip,dip,sport,dport], flowsz, f_sum(sz))
4  .filter(flowsz < 1KB)
5  .distinct([sip,dip,sport,dport])
6  .map([sip,dip,sport,dport], one, f_id)
7  .reduce([sip, dip], cnt, f_sum(one))
8  .filter(cnt > 1000)
```

At a high level, line 2 specifies that classification will be triggered if there is significant congestion (more than three congested links). Further, lines 3-4 select low-rate flows; lines 5-7 counts the number of distinct flows for each source and destination IP address pair. Line 8 selects IP address pairs with more than 1000 such flows, and populates the selected headers to a panoramic variable 'suspicious'. Notice that, 'victimLks' in line 2 is defined in the detection policy, but it is panoramic thus accessible by any switch in the network. Lines 3 and 7 use 'reduce' to aggregate packet headers per-flow and per-IP pair, respectively; the aggregation function 'f_sum' aggregates packet sizes into flow sizes at line 3, and it counts the number of distinct flows at line 7 after the 'distinct' operator. The 'map' operator at line 6 invokes 'f_id', which

produces a virtual header that always evaluates to 1, a constant. The compiler will later recognize that only 'sip' and 'dip' are needed for 'suspicious'.

Mitigation. Since Crossfire flows are indistinguishable from flash crowds, simply blocking the traffic will result in collateral damage. Existing work has proposed to reroute flows to less congested regions of the network for mitigation [43, 51]:

```
1  mitigation = panorama (100ms)
2  .when([sip, dip] in suspicious, fwd=f_reroute)
```

As before, 'f_reroute' is an intrinsic function. It forwards packets to a switch's least-utilized ports by setting the virtual header 'fwd' to the outgoing port. The compiler will recognize that 'mitigation' is never accessed by another policy, so it is filtered from the panoramic view.

Summary. The mitigation policy can be easily modified to invoke 'f_drop' as a more aggressive defense, if so desired. The detection policy can also be parameterized to use different thresholds, as can the classification policy for different numbers of congested links or different types of attack flows. Users operate at a higher level of abstraction, and our compiler automatically ensures that the panoramic view will be implemented in the switch programs.

3.3 The Coremelt defense

Next, we show how one can implement a defense against volumetric Coremelt attack flows. Assuming the same detection policy as before, we can classify volumetric flows and drop them.

Classification. Line 2 remains the same as before. Line 3 aggregates the traffic volume for each source IP, and line 4 selects the ones with high traffic volume:

```
1  suspicious = panorama(100ms)
2  .filter(victimLks.sz > 3)
3  .reduce([sip], flowsz, f_sum(sz))
4  .filter(flowsz > 100MB)
```

The mitigation policy will use the panoramic variable 'suspicious', which is keyed on 'sip'.

Mitigation. The operator could specify a more aggressive defense against volumetric flows by dropping such packets:

```
1  mitigation = panorama(100ms)
2. when([sip] in suspicious, fwd=f_drop)
```

The mitigation policy highly resembles that in Crossfire, except that suspicious traffic will be dropped.

3.4 The SPIFFY defense

SPIFFY [39] proposes a more advanced classification algorithm to identify *cost-sensitive attackers*—i.e., adversaries that generate protocol-conforming traffic from their bots at their highest possible rates. The key mechanism of SPIFFY classification is a *rate change test*, which reroutes traffic to less congested regions and checks whether the aggregate throughput for a source IP address increases or not. Normal TCP flows typically will ramp up, because they were originally bottlenecked at the network link. In contrast, attack flows will have stable rates as each bot has already been utilized to the full. SPIFFY identifies IP addresses with stable rates after rerouting, and drops their traffic. We can specify this rerouting based classification in Ripple as follows. 'flowsz1' and 'flowsz2' policies compute the traffic rate of each source IP address before and after rerouting, respectively. The 'rerouteip' policy reroutes traffic once the attack is detected, and records the source IP addresses that have experienced rerouting. The 'suspicious' policy implements the rate change test for classification. The 'drop' policy implements the defense.

```
1  flowsz1 = panorama(100ms)
2  .when(rerouteip.isempty)
3  .reduce([sip], flowsz1, f_sum(sz))

4  flowsz2 = panorama(100ms)
5  .when(!rerouteip.isempty)
6  .reduce([sip], flowsz2, f_sum(sz))

7  rerouteip = panorama(100ms)
8  .when(victimLks.sz>3, fwd=f_reroute)
9  .distinct([sip])

10  suspicious = panorama(100ms)
11  .filter(!rerouteip.isempty)
12  .zip([sip], flowsz1, flowsz2)
13  .filter(flowsz2-flowsz1 < 100KB)

14  drop = panorama(100ms)
15 .when([sip] in suspicious, fwd=f_drop)
```

The functional operator 'zip' performs a join between two sets of tuples. A zip join between (k_a, a) and (k_b, b) will produce (k, a, b) if $k_a = k_b = k$; otherwise, the result is empty. Line 12 above performs a join between 'flowsz1' with 'flowsz2'. Ripple also supports self-joins that join a panoramic variable with its previous snapshot in the last time window: 'zip([sip], flowsz)' would join the flow sizes in two consecutive time windows. Similarly, this can also be extended to support joins across multiple windows, using a similar syntax: '.zip([sip], flowsz, t)' would zip join the flow sizes in t adjacent windows. Line 13 takes in the list of $(sip, flowsz1, flowsz2)$ tuples, and selects those with negligible rate differences.

3.5 New defense policies

So far, we have shown how Ripple can support several state-of-the-art defenses that are developed in the context of OpenFlow-based SDN. Next, we present a few new policies that can be supported in Ripple.

P1: Blocking pulsewaves. The following policy identifies flows that generate high-rate, short-lived pulsewaves to the victim. It relies on detecting significant rate differences across time windows, and uses 10ms for capturing the panorama. It can be further extended to monitor t consecutive windows at line 3, and by counting

the number of pulses across these windows after line 4.

```
1  pulsewaves = panorama(10ms)
2  .reduce([sip, dip], flowsz, f_sum(sz))
3  .zip([sip, dip], flowsz, flowsz)
4  .filter(flowsz1/flowsz2 < 1/16)
```

P2: Victim detection. The next policy distinguishes normal congestion from link-flooding attacks by examining whether congestion affects all IP ranges roughly evenly, or if there are victim IPs that experience significantly higher packet loss. Traffic to victim IP ranges will be rerouted to least congested links for special protection:

```
1  inflowsz = panorama(100ms)
2  .filter(link==0 || link==1)
3  .reduce([dip], inflowsz, f_sum(sz))

4  egflowsz = panorama(100ms)
5  .filter(link==2 || link==3)
6  .reduce([dip], egflowsz, f_sum(sz))

7  victim = panorama(100ms)
8  .zip([dip], inflowsz, egflowsz)
9  .filter(egflowsz/inflowsz < 0.5)

10 mitigation = panorama (100ms)
11 .when([dip] in victim, fwd=f_reroute)
```

Assuming links 0-1 are the network ingress and links 2-3 are the egress, the 'inflowsz' and 'egflowsz' policies measure the incoming and outgoing traffic volume for each IP address, respectively. The 'victim' policy performs a zip join on 'inflowsz' and 'egflowsz', and identifies IP addresses that experience 50%+ loss rate. The 'mitigation' policy reroutes such traffic.

P3: Protecting key networks. The operator could further customize the 'victim' policy above to specifically protect key customers as a value-add service:

```
7  keyflows = panorama(100ms)
8  .zip([dip], inflowsz, egflowsz)
9  .filter(egflowsz/inflowsz < 0.5)
10 .filter([dip] in 1.2.0.0/16)
```

P4: Multi-vectored attacks. Multiple defense policies can co-exist in Ripple:

```
1  coremelt_sip = panorama(100ms)
   .. //omitted for brevity
2  xfire_flow = panorama(100ms)
   .. //omitted for brevity
3  mitigation = panorama(100ms)
4  .when([sip] in coremelt_sip, fwd=f_drop)
5  .when([sip, dip] in xfire_flow, fwd=f_reroute)
```

Summary. Users of Ripple can easily customize the panoramic view needed for defense, without having to reason about how the view will be captured locally at each switch or reconstructed globally. Rather, the Ripple compiler automatically infers the required header fields for populating panoramic variables. At runtime, the protocol only synchronizes the data required by the defense across the network.

4 Decentralized Panorama Construction

Next, we describe how the Ripple compiler generates switch programs to enforce the defense policies in a fully decentralized manner. The compiler analyzes the policy to generate switch-local defense programs in P4, and it augments these programs with a runtime protocol that synchronizes switch-local views and constructs the network-wide panorama.

4.1 Programmable switch primitives

Ripple compiles switch-local defense programs leveraging the following hardware primitives. The Ripple switch programs can process *every single packet* without downsampling.

Stateful registers. A programmable switch has several megabytes of SRAM, and a P4 program can allocate register arrays from stateful memory. The registers can be indexed, read from, and written to on a per-packet basis.

ALUs and hash units. Programmable switches have Arithmetic Logic Units (ALUs) that can operate on packet headers and register data. P4 programs can perform arithmetic and bitwise operations, as well as CRC hash and checksum functions at Tbps linespeed. Ripple uses these building blocks to compute defense decisions locally at every switch.

Ripple generates the runtime synchronization protocol using the following hardware features. The synchronization protocol runs between the programmable switches to construct the panoramic view.

Programmable parsers. P4 programs can define new header and protocol types using programmable switch parsers and deparsers. New protocols are fully compatible with TCP/IP traffic, as a P4 switch recognizes each protocol type when parsing packet headers, and activate different processings as needed.

Traffic generator. Programmable switches have hardware-based traffic generators that can serve as an out-of-band traffic source. The generators can further be configured to send packets of customized formats at prescribed rates. Ripple uses this for generating synchronization protocol messages.

4.2 Panoramic data structures

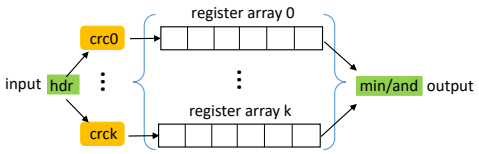
All panoramic variables are backed by a uniform representation: *key/value stores (KVS)*. Switch-local programs can access any panoramic variable *as if it is locally present* using KVS-based APIs. Concretely, a panoramic variable pv can be indexed by a key k: $pv(k)$ returns a value v associated with that key, or *nil* if k does not exist. The KVS size can be obtained by $pv.sz$, which returns the number of distinct keys in the KVS. An API call $pv.isempty$ will return a binary value indicating whether or not the KVS size is zero. In the policy language, the panoramic variables are accessed in a declarative though equivalent form—e.g., k *in* pv is equivalent to $pv(k)$. The key k to the pv is a vector of packet headers. In

fact, a policy may potentially reference *pv* using any header field; so by default, *k* contains all physical headers and virtual headers that are defined by the policy as relevant to the attack. However, under the hood, not all headers will be materialized in *pv*—the Ripple compiler chooses the best implementation for each variable, depending on its size, access methods, and the packet headers referenced.

Inferring access keys. Although policy programs can freely access any virtual/physical header in a *pv*, in practice most headers may not be relevant to the policy. Therefore, the Ripple compiler optimizes away unreferenced header fields and only preserves the access keys. For instance, most detection policies produce a *pv* called 'victimLks', which in principle contains all packet headers on congested links; however, the compiler will detect that only link IDs (a virtual header) are accessed elsewhere in the policies, so the resulting *pv* is only keyed on link ID. This minimizes the amount of data that needs to be synchronized, and also guides the compiler to infer how large the KVS may be.

Inferring sizes. The Ripple compiler infers the size of the panoramic KVS by checking which packet headers are used as access keys. The key range (e.g., link IDs vs. source IPs) will determine the upperbound of a KVS size, and Ripple uses this as optimization hints to choose the best implementation. In its simplest form, a KVS is backed by a *register array*, which is natively supported by programmable switches. Here, the KVS size grows linearly with the number of keys, but it maintains exact information for each key. If larger KVS sizes are needed, Ripple will dynamically choose between a *count-min sketches (CMS)* [23] or *bloom filters (BF)* [17], which are approximate data structures that trade off accuracy for space efficiency. These data structures support *count* and *membership* queries, respectively, but their sizes do not increase with key insertions. Rather, they use constant memory and may produce overcounting (in CMS) or false positives (in BF). Nevertheless, the accuracy/efficiency tradeoff provides strong theoretical guarantees and has been proven effective for network monitoring tasks [23]. The Ripple compiler uses them to back *pv*'s with arbitrary key counts. It further chooses an implementation based on the access methods.

Inferring access methods. The Ripple compiler checks how a panoramic variable *pv* is accessed by the policies. *(a)* If *pv* is never accessed in any policy body—e.g., 'mitigation' in most of the policies is not further accessed by other policies—no panoramic KVS will be instantiated by the compiler. *(b)* If *pv* is only accessed by *.isempty*, the compiler only maintains a binary value using a single register. *(c)* If *pv* is further accessed by *.sz*, Ripple maintains the distinct keys and key counts but it abstracts away the values using the BF implementation. *(d)* If $pv(k)$ is invoked in a policy, then depending on whether $pv(k)$ is used as a binary check or for arithmetic computation, Ripple uses a BF or CMS:

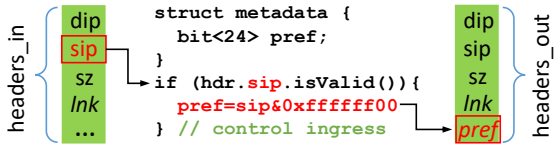


For both BF and CMS, the input key is a set of packet headers. The headers are hashed using *k* CRC functions to produce an index to each register array. CMS arrays contain *counts*, and an insertion will increment *k* elements, one in each array. BF arrays have *binary* entries, and an insertion will set *k* elements to one. The same headers are used for querying the KVS, and the same *k* indexed will be computed by the hash units. The CMS will return the minimum of all *k* counts as the estimated count, and the BF will return the logical AND of the queried values (1: key exists, 0: key does not exist).

4.3 Extracting local panorama fragments

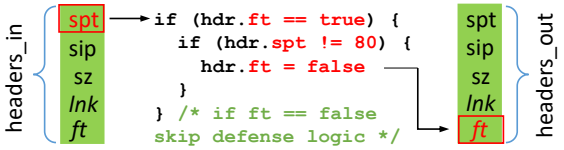
Next, we describe how the Ripple compiler generates P4 programs to extract switch-local threat signals. These fragments will later be synchronized across the network to construct the global panorama. The compiler analyzes each operator in the policy sequentially, and generates P4 programs to examine every single packet and filter out attack-unrelated signals.

map applies a function to input packet headers, and generates one or more new headers. For instance, 'map(sip,pref,f_pref24)' takes the source IP of a packet, applies 'f_pref24' to identify the /24-prefix, and generates a virtual header 'pref' for the output:



In the input and output header stacks, italic variables (*lnk*, *pref*) are virtual headers. We also show (much simplified) P4 program snippets for computing the IP prefix from source IP and generating a new header.

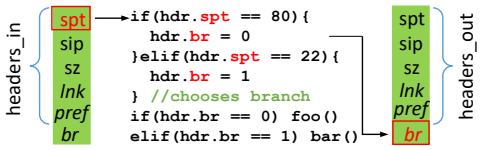
filter checks header values against a predicate, and generates a binary header ft that indicates whether or not the current packet is relevant for the defense. The Ripple policy acts on every single packet, so all packets have $ft = T$ when they enter the switch. Once a filter decides that the packet does not require further consideration, it sets the virtual header to F:



The switch program always checks the ft header before any defense processing. Packets that are filtered out will only receive forwarding related processing.

when is a control flow operator used for branching behaviors. All following statements after a 'when' (and before the next

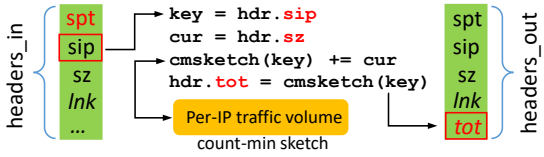
'when') are only executed if the condition evaluates to true. Consider 'when(sport==80, foo)' and 'when(sport==22, bar)':



It sets a virtual header that indicates which branch is taken. Later statements check against the branch header, and only activate defense processing for that branch.

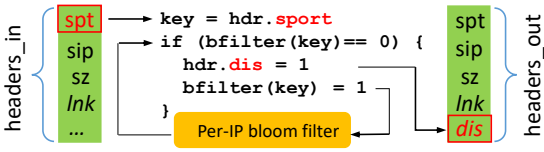
Virtual headers are carried on a special metadata bus in switch hardware, and they have the same lifetime as a physical packet. In other words, virtual headers will disappear after the packet leaves the switch, unless the policy uses one of the following operators to track cross-packet state:

reduce takes in a set of headers as the key, and aggregate all packets with the same key using the reduce function. In addition to producing a virtual header as output, it also stores the current aggregation result into a count-min sketch to persist the state. For instance, 'reduce(sip,tot,f_sum(sz))' groups packets by their source IP addresses, aggregates packet sizes, and computes the total volume per source IP:

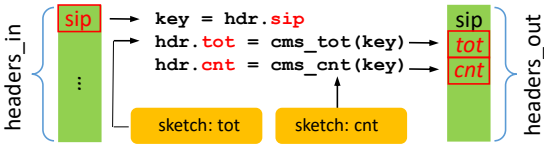


The aggregation runs throughout the current time window, and resets when a new panoramic snapshot begins. The same time window applies to 'distinct' and 'zip' below.

distinct avoids double-counting of the same key by first checking if the program has already recorded it in a bloom filter. It sets a virtual header to indicate whether the current packet carries the first distinct key in the same time window. Every unique key triggers an insertion to the bloom filter. Consider 'distinct(sport)':



zip performs a join between two sets of headers using a common key. Internally, the compiler generates two sketches (or bloom filters), and a packet triggers two queries, one to each sketch. Consider 'zip(sip,tot,cnt)', which produces a total traffic volume and a packet count for each source IP:



A self-join on a header field can be performed across two or more adjacent time windows, and the compiler generates one sketch for each window. A join that acts on t windows will generate t sketches overall.

Summary. Applying the sequence of operators to each packet will result in a set of relevant packet headers that are needed for the defense. Logically, all selected packets' headers are accessible in the policy return value—i.e., the panoramic variable; however, as discussed, the compiler performs optimizations to abstract away most physical and virtual headers.

4.4 Constructing the panorama

So far, we have described how the Ripple compiler identifies relevant attack signals and extracts them from switch-local traffic. The Ripple compiler also augments each switch program, so that they run a distributed protocol for view synchronization. Local fragments will be carried by this synchronization protocol to all switches, and switches will construct a global view based on the panorama definition. The runtime protocol executed once per time window.

Goal: At the beginning of each time window, each switch has extracted a fragment of the panorama pv from the traffic in the past period based on the policy program. Therefore, pv is initially distributed across all switches and sharded by switch IDs: $pv[s_1], pv[s_2], \cdots, pv[s_k]$, where s_i is a switch identifier. Implementing the panoramic view, therefore, requires Ripple to merge all switch-local fragments $pv[s_i], i \in [1..k]$. The goal of the Ripple protocol is to merge these fragments in a fully distributed manner.

Querying pv. In OpenFlow-based SDN, the controller can naturally serve as a vantage point to query switch data, perform the merge, and install the aggregates back; however, this would create a centralized component. Instead, we borrow recent proposals that query and synchronize switch state entirely in the data plane [45, 56]. At a high level, Ripple uses the *packet generator* to generate a stream of packets, whose destination IP addresses are the intended receiver switches. The switch program attaches the register values to the packets as customized header fields, and sweeps through all registers that need to be synchronized. Figure 3(a) illustrates the packet format for synchronizing pv.

Disseminating pv. The programmable switches disseminate pv fragments by routing the packets through the network to all switches. Ripple has 1) a spanning tree mode, and 2) a multicast mode, as shown in Figure 3.

In mode 1), the switches run a spanning tree protocol to identify a *root* switch, and all other switches use this root as a rendezvous point. Across different rounds, different switches can act as the root. Each switch sends its pv fragment along edges of this tree to the root. The root merges all fragments and distributes the panoramic view back to the switches. Compared to the multicast mode, this saves traffic overhead, since each pv fragment is only propagated in the network once. It takes roughly one round-trip time (RTT) for each synchronization round. In mode 2), the switches multicast the pv packets

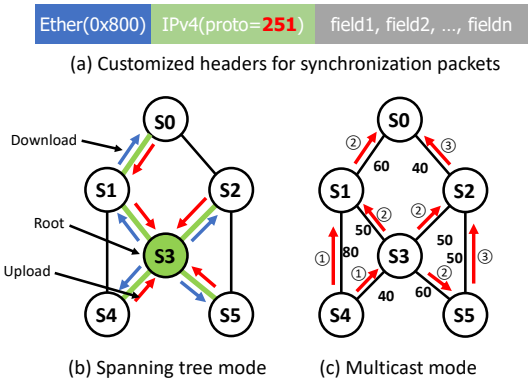


Figure 3: The packet format for synchronizing panoramic views and the two modes of the Ripple protocol.

to all neighbors, and every switch will receive all fragments from all other switches. This incurs higher traffic overhead, since each fragment is multicast to multiple neighbors. However, synchronization finishes within 0.5×RTT time.

By default, Ripple uses mode 1) to synchronize most types of *pv* in favor of traffic savings. The only exception is for implementing the rerouting-based defense, in which case it favors response time and uses mode 2) to compute least-utilized paths (i.e., in the f_reroute function). This essentially implements a distance-vector protocol that discovers best paths using probes [35, 42]. Probes are generated from each egress switch, and they are tagged with switch identifiers. The probes identify best paths in the current network to each destination switch. Data packets are forwarded from the ingress switches to the egress switches along the current best paths. A new round of probes may update the routing decisions across the network. Figure 3(b) shows an example where links are associated with costs (utilization), and probes propagate link costs across the network to identify least-utilized paths.

Merging pv. When a switch receives a *pv* fragment, it merges it with its local copy by simply adding up all the register values (for CMS) or preforming an OR (for BF) at the same indexes. Because of the linearity of these data structures, they can be easily combined by this merge [22]. When a switch receives all fragments, *pv* becomes panoramic.

5 Security Considerations

Next, we discuss potential ways that an attacker might disrupt the Ripple defense, and outline self-defense techniques. As discussed in Section 2, link-flooding adversaries are typically at the Internet edge, so in addition to launching link-flooding attacks, these adversaries can also inject crafted packets to manipulate Ripple. Attackers that can actively compromised network switches, eavesdrop, or modify existing traffic are outside the threat model.

Disrupting the synchronization protocol. The synchronization protocol in Ripple propagates panoramic variables across the network. If an attacker can disrupt this protocol—e.g., by creating congestion in the network to drop synchronization packets in a targeted manner, then this would prevent the Ripple switches from constructing a panoramic view. However, precisely disrupting the Ripple protocol is not easy, since the synchronization schedule is unknown to the attacker. Therefore, such an adversary can at best resort to congesting edges of the spanning tree to delay or prevent view synchronization. As a possible defense, Ripple could use the multicast mode for synchronization when the panoramic view cannot be constructed for multiple time windows. Since the multicast traffic does not follow spanning tree edges, the attacker can only disconnect the network by taking down a much larger portion of network links, which is inherently difficult. As another alternative, Ripple could dynamically rebuild spanning trees by changing the root switches, so that the attacker cannot predict which links are part of the spanning tree.

Spoofing synchronization packets. A strong adversary that has knowledge of synchronization packets could potentially inject spoofed packets into the network to "poison" the switch views. A classic defense is to use cryptography, where a message carried a MAC (Message Authentication Code) for source authentication; the MAC could also include timestamps or sequence numbers to prevent replay attacks. However, as a practical challenge, today's P4 programming model does not support cryptographic operations natively. To solve this, there are two possible approaches. (a) Crypto modules can be integrated to programmable data planes as "extern" hardware modules and invoked by P4 programs [1]. (b) Recent work has designed different cryptographic primitives using P4 [31, 32], including AES [21]; another related project explicitly considers authenticating inter-switch communication in the data plane [56]. These techniques are all useful building blocks for packet authentication.

6 Evaluation

We have evaluated Ripple in order to answer three research questions: a) How well does the Ripple compiler work? b) How much overhead do the Ripple defense programs incur? c) How effective can Ripple defend against link-flooding attacks, especially in the presence of adaptive adversaries?

6.1 Prototype and setup

Software and hardware prototypes. We have developed our Ripple compiler in ∼ 6000 lines of code in C++. It currently supports the `bmv2` [11] switch backend, which is a widely used software P4-16 switch model [25, 27, 35, 42, 45]. Our compiler takes in a Ripple policy, a network topology, and emits a P4 program for each switch. We have also developed a hardware prototype by converting one of the generated P4-16 programs to P4-Tofino—a special P4 dialect for Intel/Barefoot Tofino hardware switches—in 1600 line of P4 code.

Baseline defenses. To understand the benefits of Ripple, we have evaluated it against three SDN-based defenses as baseline systems. SDN-S and SDN-R are representative of classic SDN setups: SDN-S samples traffic from OpenFlow switches at a prespecified sampling rate to the controller; the controller runs classification algorithms on the traffic sample, and installs OpenFlow rules to reroute suspicious traffic. SDN-R, on the other hand, does not perform sampling or classification; rather, it collects link load data from all switches, and computes rerouting decisions for all flows at congested links. In addition, we have created a third baseline SDN++ to give SDN defenses an extra advantage—it enhances OpenFlow switches with an extra module that can run classification algorithms in the data plane without involving a controller. We use SDN++ as a baseline to demonstrate the "upperbound" of centralized defenses; in practice, such a module is only implementable in P4 switches. In all cases, the SDN controller uses SOL [33], a state-of-the-art traffic engineering framework, for traffic engineering and computing rerouting decisions.

Attacks. We use similar strategies as in Crossfire [40] for bot distribution, flow density, and attack target links. Attackers generate Crossfire, Coremelt, and SPIFFY flows in different experimental setups. Normal users employ regular TCP connections for file downloads. One of the main evaluation metrics is the ability for a defense to mitigate attacks and recover normal user throughput.

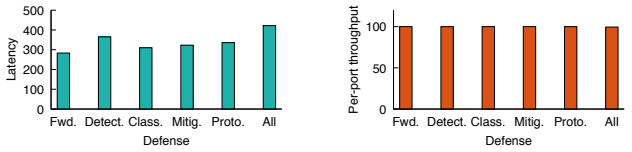
Experimental platforms. Most existing work on link-flooding defense [39, 40, 53] use *flow-level simulation*, where traffic patterns are simulated at a coarse, flow-level granularity for scalable evaluation. We adopt the same strategy by extending an existing flow-level simulator for Ripple [20]. In addition, we have also evaluated Ripple in two other platforms to understand the fine-grained behaviors that flow-level simulators cannot capture. Concretely, we have used *packet-level simulation* using a version of ns3 [8] that is integrated with `bmv2` support, which can faithfully simulate how P4 switches process every single packet. Since fine-grained simulation comes at the cost of higher simulation time, packet-level simulation is only feasible on smaller networks. Both packet- and flow-level simulators run in a Ubuntu 18.04 server with six Intel Xeon E5-2643 Quad-core 3.40 GHz CPUs and 128 GB RAM. To demonstrate real hardware feasibility, we have also used *a Wedge 100BF hardware switch*, whose bandwidth is 100Gbps per port and 1.6 Tbps in aggregate. We flash the switch hardware with the manually converted P4-Tofino program for this evaluation. In the following subsections, when reporting a set of results we also clarify which platform(s) the experiments have been conducted on.

6.2 Overhead

Our first set of experiments measures the overhead of Ripple defense programs. Most of the results are obtained using the P4-Tofino defense program on a real hardware switch.

Resources	Detection	Classification	Mitigation	Protocol	All
Stages	6	6	3	6	12
VLIWs (%)	2.86	5.99	1.82	3.65	10.68
ALU (%)	18.75	29.17	10.42	14.58	43.75
Hash unit (%)	4.17	15.28	4.17	2.78	25.00
SRAM (%)	4.17	11.98	5.38	4.48	15.62

Table 2: Resource utilization on the Tofino hardware switch (policy: Coremelt).



(a) Latency (nanoseconds) (b) Throughput (Gbps)

Figure 4: Ripple incurs extra latency on the order of nanoseconds, and it achieves linespeed throughput.

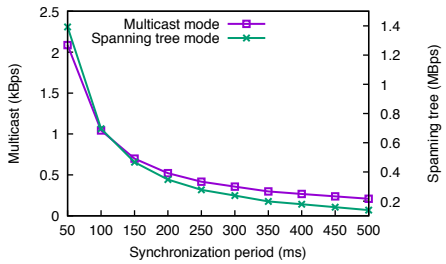


Figure 5: Traffic overheads of Ripple's distributed protocol at different synchronization periods.

Hardware utilization. Table 2 shows the hardware resource utilization for each program component. As we can see, the classification pipeline incurs the highest resource utilization, because it is the most complex component of the policy. Overall, the defense program uses 10.68% VLIWs (Very Long Instruction Words) and 43.75% ALUs (Arithmetic Logical Units) for header computation, 25% of the CRC hash units, as well as 15.62% SRAM (Static RAM). All these hardware resources are spread across 12 hardware stages. We note that more recent switch models (e.g., Tofino 2) have higher resource provisions for all types of resources. Another important takeaway is that the defense program is implementable in today's programmable switch hardware.

Latency. Next, we evaluate the extra latency incurred by the Ripple defense, using a baseline switch program "Fwd", which is a minimal P4 program that only forwards traffic without any other processing. As Figure 4(a) shows, the Ripple defense program incurs 139 nanoseconds of latency compared with the baseline. Interestingly, we found that the classification component incurs the least latency overhead, and the detection component incurs the most overhead. This is because the classification component is dominated by a set of

Figure 6: SDN-R reroutes all traffic, and normal user flows experience an average path length increase of 31%.

register operations, which are parallelized by switch hardware; on the other hand, the detection component involves sequential processing. Overall, the extra latency is negligible, as network RTTs are typically on the order of milliseconds in the Internet core.

Throughput. Next, we evaluate the throughput of Ripple using the on-switch hardware packet generator, which can generate full linespeed traffic (100Gbps per port). Our baseline program is still "Fwd". As Figure 4(b) shows, the throughputs of Ripple and of the baseline are very close, at about 99.52 Gbps per port. This is because of the pipelined nature of the switch hardware, which is designed to mask small latency increases by massive parallelism.

The above results demonstrate that Ripple defenses are practical on today's hardware switches, and that they incur relatively low overhead. Next, we turn to measure the traffic overhead due to the Ripple distributed protocol using packet-level simulation:

Traffic overhead. Figure 5 presents the results for different synchronization periods for a single link. For both synchronization modes (spanning tree vs. multicast), the overheads are low enough to be practical. Concretely, the multicast mode only propagates link utilization metrics, and it incurs 2.1 KBps overhead at a period of 50 ms. The spanning tree mode propagates all other metric types and generates more traffic: the overhead is 1.4 MBps at 50 ms. More frequent synchronization also leads to higher overhead. Overall, the overhead is low since today's network linkspeeds are 40-100Gbps.

6.3 The Ripple Compiler

Table 3 shows the number of lines of code that Ripple uses to capture state-of-the-art policies. The policy programs are much more concise than the generated P4 programs. Ripple also works efficiently, generating switch programs within one second in all cases. We have manually verified that the programs can successfully mitigate Crossfire, Coremelt, and SPIFFY attacks by deploying them to the ns3 simulator and evaluating them against real attacks.

Policy	LoC of policies	LoC of P4	Compilation time
Crossfire	13	1509	68ms
Coremelt	9	924	37ms
SPIFFY	18	1516	69ms
Multi-vector	18	1910	85ms

Table 3: Ripple captures state-of-the-art defenses within 20 lines of code; the compiler works efficiently and generates P4 programs for each policy within one second. Multi-vector is a combination of Crossfire and Coremelt.

Topo Name	ANS	CRL	Bell Canada	SurfNet	UUNet
#switches	18	33	48	50	49
#links	25	38	65	68	84

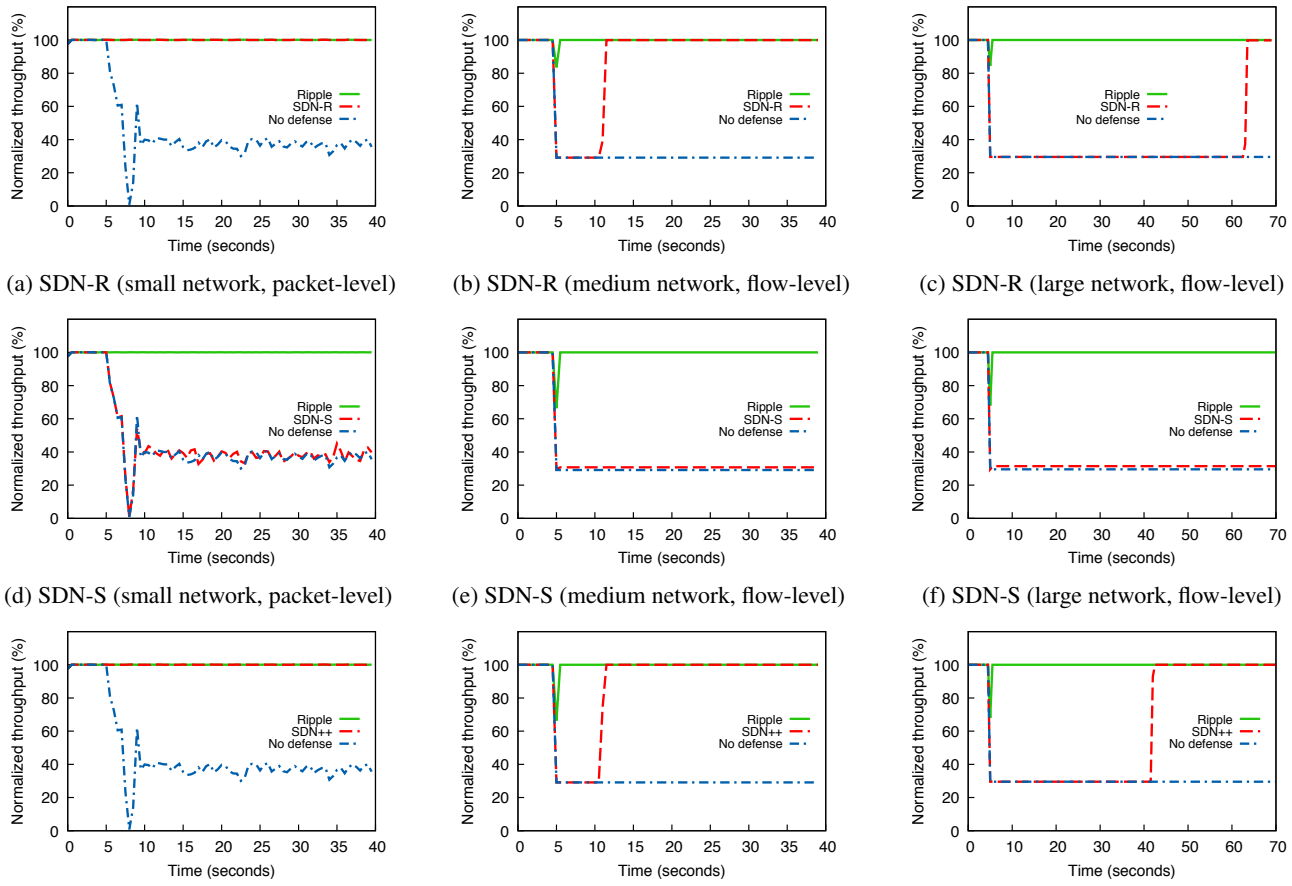
Table 4: Topology setups used in large-scale simulation. All topologies are from Topology Zoo [7].

6.4 Defense effectiveness

Next, we evaluate the effectiveness of the defenses on three topologies with increasing sizes and traffic complexities. We use the packet-level simulator for the small network; we use flow-level simulator for medium and large networks because fine-grained simulation does not scale to large setups. Table 4 shows the topology setups that we have used for evaluation.

Figures 7(a)-(i) present the defense effectiveness of all tested systems, as measured by the throughput degradation the attack causes over time. We normalize the aggregate throughput of normal users over that before the attack, so a higher percentage indicates a stronger defense, and 100% means a full recovery. We also plot a "no defense" baseline that shows the attack impact without deploying any defense. There are four key takeaways: *(1)* Compared to SDN-R, which reroutes all flows from the congested links, Ripple achieves a similar level of throughput recovery but it acts much faster. This is because Ripple directly reroutes traffic in the data plane without a central controller. As the network becomes larger, the advantage of Ripple also becomes more prominent. *(2)* SDN-S only samples and reroutes 1% flows, so it acts faster than SDN-R; compared to SDN-S, Ripple recovers throughput much more effectively. This is because the SDN controller only sees heavily downsampled traffic. The defense decisions cannot take action on the majority of malicious flows, as they are not included in the samples. *(3)* SDN++ is the most powerful SDN variant, and it can recover throughput with similar effectiveness as SDN-R. It also responds faster, as classification is done in the extra switch module and the controller performs traffic engineering on reported suspicious flows. *(4)* Overall, Ripple outperforms all three SDN baselines.

We quantify the effectiveness of a defense system by measuring the attack impact on normal user throughput. For each defense, we measure the throughput degradation ratio per unit time, and compute the aggregate degradation until throughput recovers to a stable state. This aggregate A denotes the attack



(a) SDN-R (small network, packet-level) (b) SDN-R (medium network, flow-level) (c) SDN-R (large network, flow-level)

(d) SDN-S (small network, packet-level) (e) SDN-S (medium network, flow-level) (f) SDN-S (large network, flow-level)

(g) SDN++ (small network, packet-level) (h) SDN++ (medium network, flow-level) (i) SDN++ (large network, flow-level)

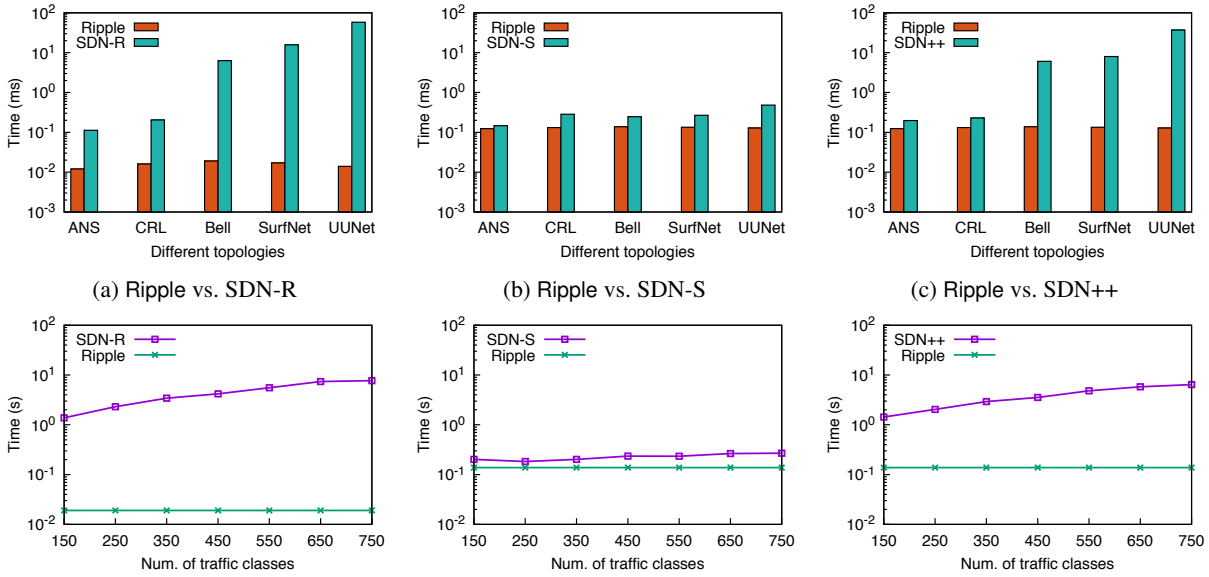
Figure 7: Ripple can mitigate attacks faster than all SDN baselines, and it recovers normal throughputs effectively. Small: customized topology with 10 switches and 15 links; Medium: Bell Canada; Large: UUNet. Table 4 summarizes the topologies.

impact, and a larger A means that the attack is more effective. We found that under the Ripple defense, we have $A = 0.17$ in the worst-case scenario; this can be interpreted as "the attack degrades the throughput for 17% for 1 second". For SDN baselines, on the other hand, we have $A = 10$, 30, and 15 for SDN++, SDN-S, and SDN-R on average, respectively, which are orders of magnitude larger. As another interesting finding, SDN-R performs worse than SDN++ and SDN-S in terms of protecting normal user flows. This is because the latter two defenses use a classifier to identify and then only reroute potentially suspicious flows; user flows still follow the original routing. In contrast, SDN-R reroutes all flows, which leads to higher hop counts and increased latency for user flows (Figure 6). This means that even an "best-effort" classifier is still useful for increasing defense effectiveness. The modularity of Ripple language allows the defense to incorporate such defense optimizations very easily.

6.5 Mitigating rolling attacks

The next set of experiments are designed to evaluate how well the defenses can handle adaptive adversaries. Before evaluating rolling attacks, we start by performing a set of microbenchmarks on attack response time, which is defined as the time for a defense to take effect after the attack begins. We further use a wider range of of topologies (Figures 8(a)-(c)) and traffic complexities (Figures 8(d)-(e)). As the microbenchmark shows, Ripple always produces the fastest response.

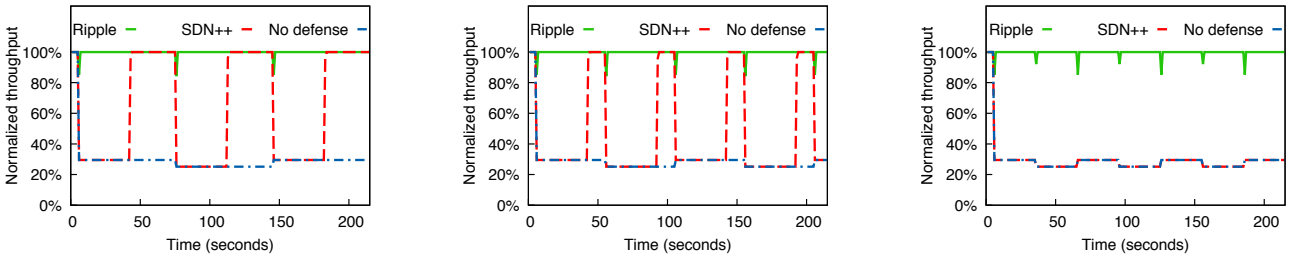
Faster response time is a key enabler for Ripple to mitigate rolling attacks. We launch rolling attacks using Crossfire flows in the largest topology, and compare Ripple with SDN++ as the baseline defense. Concretely, the adversary dynamically shifts the attack traffic to different links to evade mitigation. Figures 9(a)-(c) present the normal user throughput under the attack, and they further test different rolling attack strengths as measured by the frequency for shifting attacks. As we can see, Ripple can always detect the changing targets very quickly, and recover the throughput soon afterwards using a suitable defense strategy. However, for SDN++, the defense decisions are always lagging behind. For fast-changing attacks, the SDN defense experiences a constant throughput degradation during the attack. This confirms the effectiveness of rolling attacks for increasing attack persistence (as first identified by Crossfire) [40]; it also shows that Ripple can effectively mitigate rolling attacks and break such persistence.



(a) Ripple vs. SDN-R (b) Ripple vs. SDN-S (c) Ripple vs. SDN++

(d) Ripple vs. SDN-R (e) Ripple vs. SDN-S (f) Ripple vs. SDN++

Figure 8: The attack response time of the defense systems with different topology sizes and traffic complexities as measured by the number of traffic classes. A traffic class is a collection of flows that arrive at the same ingress and are routed by the network in the same way to the same egress.



(a) Rolling attacks (slow) (b) Rolling attacks (medium) (c) Rolling attacks (fast)

Figure 9: Ripple is effective against fast-changing rolling attacks. SDN baselines constantly lag behind.

7 Related Work

Link-flooding attacks. Existing work has demonstrated a range of effective link-flooding attacks [39, 40, 53], and similar real-world incidents have been reported in the wild [3, 6]. State-of-the-art defenses are based on OpenFlow SDN, which run defense algorithms as software SDN apps at a centralized controller [39, 43, 51, 61]. Ripple is the first *decentralized* defense based on programmable switches, and it achieves similar programmability as existing SDN defenses while outperforming them on fast-changing attacks.

Programmable switches. Programmable switches have found use in network measurement [27, 30, 48, 52, 59], load balancing [35, 42, 47], application-level acceleration [24, 37, 38, 44], and security [41, 57]. Recent work has also considered synchronizing or replicating switch states across the network [45, 54, 56]. Ripple is inspired by these work, but uses programmable switches to design a decentralized defense against link-flooding attacks. A position paper has argued for the advantage of programmable switches for link-flooding defense, but it only outlines a design sketch [58].

8 Conclusion

In this paper, we have presented Ripple, a decentralized defense against adaptive link-flooding attacks using programmable switches. Ripple has a policy language that specifies a defense panorama, and its compiler can generate switch-local programs in P4 that extract attack signals from network traffic. Moreover, the Ripple runtime uses a distributed protocol to synchronize local views and construct a network-wide panorama. Our evaluation shows that Ripple can be programmed for a range of defenses, and that it can outperform SDN defenses significantly in mitigating adaptive adversaries.

9 Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

References

[1] Add crypto extern to behavioral-model. https://github.com/p4lang/behavioral-model/pull/834.

[2] Attackers Use DDoS Pulses to Pin Down Multiple Targets. https://www.imperva.com/blog/pulse-wave-ddos-pins-down-multiple-targets/.

[3] Can a DDoS break the Internet? Sure, just not all of it. https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/.

[4] Detecting and mitigating target link-flooding attacks using SDN.

[5] Dyn analysis summary of Friday October 21 attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[6] How extorted e-mail provider got back online after crippling DDoS attack. https://arstechnica.com/information-technology/2015/11/how-extorted-e-mail-provider-got-back-online-after-crippling-ddos-attack/.

[7] The Internet Topology Zoo. http://www.topology-zoo.org/.

[8] NS-3 simulator. https://www.nsnam.org/.

[9] Nsfocus identifies DDoS attack trends in new 2018 insights report. https://nsfocusglobal.com/nsfocus-identifies-ddos-attack-trends-new-2018-insights-report/.

[10] OVH hosting hit by 1Tbps DDoS attack, the largest one ever seen. https://securityaffairs.co/wordpress/51640/cyber-crime/tbps-ddos-attack.html.

[11] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[12] The P4 language repositories. https://github.com/p4lang.

[13] Nox. https://github.com/noxrepo/nox, 2012.

[14] Beacon. https://www.sdxcentral.com/projects/beacon/, 2013.

[15] Pox. https://noxrepo.github.io/pox-doc/html/, 2017.

[16] OpenDaylight. https://www.opendaylight.org/, 2018.

[17] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13, 1970.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR*, 44(3), 2014.

[19] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM CCR*, 43(4):99–110, 2013.

[20] Kuan-yin Chen, Anudeep Reddy Junuthula, Ishant Kumar Siddhrau, Yang Xu, and H Jonathan Chao. SDNShield: Towards more comprehensive defense against DDoS attacks on SDN control plane. In *Proc. CNS*, 2016.

[21] Xiaoqi Chen. Implementing AES encryption on programmable switches via scrambled lookup tables. In *Proc. SIGCOMM SPIN Workshop*, 2020.

[22] Graham Cormode. Count-min sketches. *Encyclopedia of Database Systems*, 2009.

[23] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, April 2005.

[24] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. NetPaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[25] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. SPINE: Surveillance protection in the network elements. In *Proc. FOCI*, 2019.

[26] Seyed K Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic DDoS defense. In *Proc. USENIX Security*, 2015.

[27] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of TCP. In *Proc. SOSR*. ACM, 2017.

[28] Dimitrios Gkounis, Vasileios Kotronis, and Xenofontas Dimitropoulos. Towards defeating the crossfire attack using SDN. *arXiv preprint arXiv:1412.2013*, 2014.

[29] Garegin Grigoryan and Yaoqing Liu. LAMP: Prompt layer 7 attack mitigation with programmable data planes. In *Proc. ANCS*, 2018.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proc. SIGCOMM*, 2018.

[31] F. Hauser, M. Schmidt, M. Häberle, and M. Menth. P4-MACsec: Dynamic topology monitoring and data layer protection with MACsec in P4-based SDN. *IEEE Access*, 8, 2020.

[32] Frederik Hauser, Marco Häberle, Mark Schmidt, and Michael Menth. P4-IPsec: Implementation of IPsec gateways in P4 with SDN control for host-to-site scenarios. *arXiv preprint arXiv:1907.03593*, 2019.

[33] Victor Heorhiadi, Michael K Reiter, and Vyas Sekar. Simplifying software-defined network optimization using SOL. In *Proc. NSDI*, 2016.

[34] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven WAN. In *Proc. SIGCOMM*, 2013.

[35] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, Praveen Tammana, and David Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[36] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined WAN. In *Proc. SIGCOMM*, 2013.

[37] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-free sub-rtt coordination. In *Proc. NSDI*, 2018.

[38] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proc. SOSP*, 2017.

[39] Min Suk Kang, Virgil D Gligor, and Vyas Sekar. SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *Proc. NDSS*, 2016.

[40] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. The crossfire attack. In *Proc. S&P*, 2013.

[41] Qiao Kang, Lei Xue, Adam Morrison, Yuxin Tang, Ang Chen, and Xiapu Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[42] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proc. SOSR*, 2016.

[43] Soo Bum Lee, Min Suk Kang, and Virgil D Gligor. CoDef: collaborative defense against large-scale link-flooding attacks. In *Proc. CoNEXT*, 2013.

[44] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proc. FAST*, 2019.

[45] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing State: Consistent updates for stateful and programmable data planes. In *Proc. SOSR*, 2017.

[46] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.

[47] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proc. SIGCOMM*, 2017.

[48] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[49] Ryan Rasti, Mukul Murthy, Nicholas Weaver, and Vern Paxson. Temporal lensing and its application in pulsing denial-of-service attacks. In *Proc. S&P*, 2015.

[50] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. Heavy-hitter detection entirely in the data plane. In *Proc. SOSR*, 2017.

[51] Jared M Smith and Max Schuchard. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. In *Proc. S&P*, 2018.

[52] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow. In *Proc. USENIX ATC*, 2018.

[53] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proc. ESORICS*, 2009.

[54] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. LOcAl DEcisions on Replicated states (LOADER) in programmable data planes: programming abstraction and experimental evaluation. *arXiv preprint arXiv:2001.07670*, 2020.

[55] Lei Wang, Qing Li, Yong Jiang, Xuya Jia, and Jianping Wu. Woodpecker: Detecting and mitigating link-flooding attacks via SDN. *Computer Networks*, 147:1–13, 2018.

[56] Jiarong Xing, Ang Chen, and T.S. Eugene Ng. Secure state migration in the data plane. In *Proc. SIGCOMM SPIN Workshop*, 2020.

[57] Jiarong Xing, Qiao Kang, and Ang Chen. NetWarden: Mitigating network covert channels while preserving performance. In *Proc. USENIX Security*, 2020.

[58] Jiarong Xing, Wenqing Wu, and Ang Chen. Architecting programmable data plane defenses into the network with FastFlex. In *Proc. HotNets*, 2019.

[59] Nofel Yaseen, John Sonchack, and Vincent Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[60] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. NSDI*, 2012.

[61] Jing Zheng, Qi Li, Guofei Gu, Jiahao Cao, David KY Yau, and Jianping Wu. Realtime ddos defense using cots sdn switches via adaptive correlation analysis. *IEEE Transactions on Information Forensics and Security*, 13(7):1838–1853, 2018.