Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches

Menghao Zhang*, Guanyu Li*, Shicheng Wang*, Chang Liu*, Ang Chen†, Hongxin Hu°, Guofei Gu‡,
Qi Li*, Mingwei Xu*, Jianping Wu*
*Institute for Network Sciences and Cyberspace & Department of Computer Science and Technology, Tsinghua University
*Beijing National Research Center for Information Science and Technology (BNRist)
†Rice University °School of Computing, Clemson University ‡SUCCESS Lab, Texas A&M University

***Abstract*—Distributed Denial-of-Service (DDoS) attacks have become a critical threat to the Internet. Due to the increasing number of vulnerable Internet of Things (IoT) devices, attackers can easily compromise a large set of nodes and launch high-volume DDoS attacks from the botnets. State-of-the-art DDoS defenses, however, have not caught up with the fast development of the attacks. Middlebox-based defenses can achieve high performance with specialized hardware; however, these defenses incur a high cost, and deploying new defenses typically requires a device upgrade. On the other hand, software-based defenses are highly flexible, but software-based packet processing leads to high performance overheads. In this paper, we propose POSEIDON, a system that addresses these limitations in today's DDoS defenses. It leverages emerging programmable switches, which can be reconfigured in the field without additional hardware upgrade. Users of POSEIDON can specify their defense strategies in a modular fashion in the form of a set of defense primitives; this can be further customized easily for each network and extended to include new defenses. POSEIDON then maps the defense primitives to run on programmable switches—and when necessary, on server software—for effective defense. When attacks change, POSEIDON can reconfigure the underlying defense primitives to respond to the new attack patterns. Evaluations using our prototype demonstrate that POSEIDON can effectively defend against high-volume attacks, easily support customization of defense strategies, and adapt to dynamic attacks with low overheads.**

I. INTRODUCTION

Distributed Denial-of-Service (DDoS) attacks have been a longstanding threat. They have become even more so as an increasing number of vulnerable Internet of Things (IoT) devices are connected online. Over the past a few years, there has been a dramatic increase in the scale and diversity of DDoS attacks, many of which have frequently made the headlines [80], [59], [54], [66]. Recent surveys report 400,000 DDoS attacks every month [67], with peak volume reaching Tbps [5]. These attacks are also evolving quickly, leveraging new or mixed attack vectors [40], [62], [68], [20].

Today's defenses against large-scale DDoS attacks, however, have not caught up. One of the most widely adopted DDoS defenses is using a *traffic scrubbing center*, where a range of defense mechanisms are deployed near the destinations to mitigate DDoS "as-a-service" [17]. However, most of the devices deployed in the traffic scrubbing centers are expensive and proprietary hardware appliances, i.e., middleboxes [50], [57], [14]. Although these middleboxes deliver high performance, they tend to be inflexible in terms of functionality, capacity, and placement locations. As a result, whenever a new attack vector emerges, its corresponding defense would require an upgrade of the middleboxes, which in turn requires rounds of negotiations between customers and vendors. In addition to this lack of agility, hardware upgrades also incur significant economic costs.

Recent trends in networking—Software Defined Networking (SDN) and Network Function Virtualization (NFV)—can mitigate some concerns above by employing software-based network programmability. For instance, Bohatei [22] leverages NFV to elastically scale the number of defense virtual machines (VMs) based on attack composition, and it adopts SDN to steer the suspicious traffic to proper VMs. It also designs several efficient resource management mechanisms for scalability, responsiveness, and attack resilience. Despite these benefits, server-based packet processing incurs additional latency overheads and defense costs. These problems are deeply rooted in the nature of software-based platforms, where packets are processed on general-purpose CPUs rather than specialized network hardware customized to sustain Tbps traffic.

An ideal DDoS traffic scrubbing service should have low operational and capital cost; and at the same time, it should have high performance in packet processing and enable agile deployment of new defenses. These requirements are becoming more and more urgent with the increasing number of IoT botnets [3], [83], new variants of DDoS attacks [63], [76], and the stringent latency demands in today's network services [28], [92]. We observe that the emerging programmable switches [10] developed in the latest networking technology can provide an exciting opportunity to bridge this gap. First off, since programmable switches provide several orders of magnitude *higher throughput* than highly-optimized packet processing software [51], [35], a single switch could potentially replace hundreds of servers, significantly reducing per-capacity capital cost and operational expense. Moreover, such switches support *stateful* packet processing using domain-specific languages (e.g., P4 [9]), which can process packets with user-defined logic at *terabit line rate* in the switch pipeline. These potential benefits are particularly valuable for DDoS defense.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24007
www.ndss-symposium.org

While programmable switches are a promising candidate for DDoS defense, there are three challenges that we must address. First, we desire a high-level abstraction that can capture a wide range of DDoS defense policies. However, different DDoS attacks exploit a variety of protocol- and system-vulnerabilities. The corresponding DDoS defenses also need to be rather heterogeneous in order to handle different types of attacks. Such requirements make it challenging to describe the defense policies uniformly. Second, although programmable switches provide several orders of magnitude higher throughput and lower latency than commodity servers, they only have restrictive computational models and limited on-chip resources; this makes it challenging to implement sophisticated DDoS defenses (e.g., puzzle for HTTP Flood), and we also need to work within the switch resource limitations. Third, DDoS attacks are dynamic in terms of attack types and composition. This raises another requirement that the defense should be adaptive to attack dynamics. It is challenging to achieve this with high efficiency (i.e., switch resource utilization) and strong correctness guarantees (i.e., without interrupting legitimate flows).

To address the challenges above, in this paper, we propose POSEIDON, a performant, cost-efficient and agile DDoS defense system with programmable switches. First, we provide a modular policy abstraction for expressing defense policies, which can capture a wide range of DDoS defenses concisely. Second, we partition the defense primitives to run on programmable switches—and when necessary, on commodity servers—according to their properties, and map the high-level policies to the defense resources with an optimized orchestration mechanism. Third, we develop an effective runtime management mechanism to reconfigure POSEIDON for dynamic defense without interrupting legitimate flows. We stress that POSEIDON is not intended to provide a new algorithmic or theoretical contribution to DDoS defense, but rather to provide a practical and system-level solution leveraging the emerging programmable switches, which could potentially become a new platform for future DDoS defenses. Our implementation and evaluation demonstrate that POSEIDON is able to potentially defend against $\sim$Tbps attack traffic, capture a range of defense policies within tens of lines of code, adapt to policy changes in seconds, and handle dynamic attacks with negligible overheads.

In summary, we make the following contributions:

- We analyze the challenges of the current DDoS defense practices, identify new opportunities provided by programmable switches (§II), and discuss the design challenges in integrating programmable switches into the existing DDoS defense framework (§III).
- We provide a simple and modular abstraction to express DDoS defense policies, shielding the underlying hardware complexities from programmers (§IV).
- We develop an optimized resource orchestration mechanism to map the high-level policy primitives to the underlying hardware resources (§V).
- We develop a runtime management mechanism that can adapt to dynamic attacks with high resource utilization efficiency and strong correctness guarantees for legitimate flows (§VI).
- We implement a prototype of POSEIDON, and conduct extensive experiments to demonstrate the advantages of POSEIDON (§VII, §VIII).

Finally, we discuss several practical issues (§IX), summarize related work (§X), and then conclude the paper (§XI).

II. MOTIVATION AND OBSERVATION

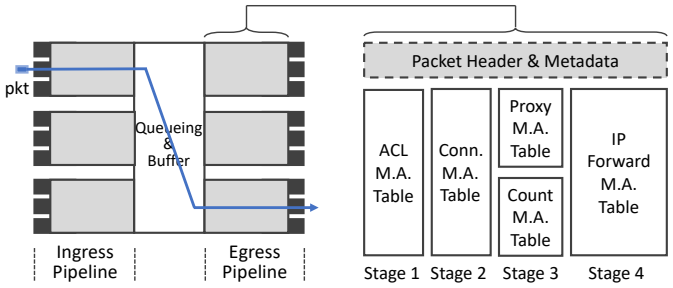
In this section, we further motivate the need for advanced DDoS defenses, and describe why the emergence of programmable switches is a promising enabler of new DDoS defense systems.

A. Challenges in DDoS defense

DDoS attacks have become a critical threat to the Internet services. To defend against DDoS attacks, one of the most deployed defenses is using a traffic scrubbing center, where a large cluster of commodity servers or proprietary middleboxes are organized to filter the malicious traffic. Two essential requirements are defense *cost* and *agility*. Unfortunately, today's defense systems are lacking in both regards.

First, DDoS defense should be cost-efficient. As DDoS attacks are challenging to eliminate without making fundamental changes to the Internet architecture, there will always be a "cat-and-mouse" game between attackers and victims. If one side could obtain more resources (attack traffic vs. defense devices) with lower cost, that side will win out. As a well-known fact in the operational security community, the costs for DDoS attackers and victims are determined by two separate markets, namely, botnet markets and defense markets [39]. As a result, it is important to increase the difficulty to obtain botnets and to reduce the costs to deploy defense countermeasures. Unfortunately, with the massive usage of vulnerable IoT devices and the emergence of various powerful botnets (e.g., Mirai [3], [13]), this balance is shifted towards attackers quickly and the Internet is stricken by storms of larger and larger DDoS attacks more and more frequently [63], [76], [83]. Although we can scale up the scrubbing capacity by adding more servers or proprietary middleboxes, doing so raises the capital cost and operational complexity, which is not symmetric to the rapid growth of attack traffic nowadays.

Second, DDoS defense should be agile in terms of *new defense deployment* and *traffic scrubbing procedure*. As discussed above, DDoS attacks are still evolving rapidly, and new attack vectors are emerging constantly [40], [62], [68], [20]. To address a new attack vector, hardware upgrades are necessary. However, proprietary middleboxes are extremely hard to upgrade, and even adding simple functionality such as modifying the statistic granularity is difficult to achieve without vendor support [57], [14]. Such inflexibility to deploy new defense mechanisms hinders our ability to quickly respond to new variants of DDoS attacks. To make matters worse, today's vendors usually deploy *all* known defense countermeasures into middleboxes to cope with attack dynamics [57], [14], which results in substantial processing resource waste and further raises the capital cost. Since it is unlikely to see all attacks simultaneously, most of the hardware resources are left unused during DDoS defense. Server-based solutions provide high programmability to solve the problem above, but this comes with high latency, high jitter and poor isolation



(a) Switch data plane architecture

(b) Ingress/Egress pipeline architecture

Match	Action	Action Data
src_ip:10.0.1.0/24	set_next_hop()	eport=4
src_ip:192.0.0.0/8, tcp_flag:0x10	validate_cookie()	cookie_seed=531
...	...	...
default	drop()	-

(c) Match-action table architecture

Fig. 1: Switch data plane architecture.

in packet processing. Software processing adds a latency of $50us$ to $1ms$ when handling as little as $100K$ packets per second [26], which is unacceptable for many latency-sensitive services [28], [92] common in today's data centers. When software experiences a flash crowd, legitimate traffic served by the server also experiences increased delays, even unexpected packet drops [26], [51], which makes scrubbing procedure challenging even for latency-insensitive services.

B. Opportunities of Programmable Switches

Current trends in SDN have extended the network programmability to the data plane through programmable switching ASICs (Application-Specific Integrated Circuits) and domain-specific languages (e.g., P4 [9]). In programmable switching ASICs, there are multiple ingress and egress pipelines, each with multiple ingress and egress ports (Fig. 1(a)). Packets are processed sequentially by each stage in a pipeline (Fig. 1(b)). Each stage has its own dedicated resources, such as registers (for storage and stateful packet processing), match-action units (for data plane customization), and stateful ALUs (for computation). Match-action tables match certain packet fields or metadata and apply actions on the packet (Fig. 1(c)). Each table modifies packet fields and generates metadata through which tables can share information. Using programmable switching ASICs, programmers can customize data plane logics with domain-specific languages like P4. In a P4 program, developers customize packet headers, build packet processing graphs, and specify entries in match-action tables. The compiler provided by switch vendors can compile the programs into binaries and generate interactive APIs. The binaries specify data plane contexts and are loaded into switches, while the APIs are used by control plane applications to interact with the data plane.

The programmable switching ASICs and P4 language make it possible to implement custom *terabit* packet processing devices, as long as the defined logics can be fitted into the match+action model of switching ASICs. Given the performance and flexibility, we briefly highlight some new opportunities that programmable switches bring for DDoS defenses:

TABLE I: Capital cost for different defense hardware.

Device	Capability	Equipment Cost	Power Cost
NSFOCUS ADS	48Gbps	$102,550 ($2,136/Gbps)	600Wtts (12.5Wtts/Gbps)
Commodity Server	40Gbps	$4,400 ($110/Gbps)	600Wtts (15Wtts/Gbps)
Programmable Switch	**3.3Tbps**	$10,500 (**$3/Gbps**)	450Wtts (**0.1Wtts/Gbps**)

Lower unit capital cost. The cost benefit when introducing programmable switches into DDoS defense framework includes two parts: equipment cost (in dollars) and power consumption (in Watts). As shown in TABLE I, according to our investigation, a typical 48Gbps DDoS defense middlebox costs about $102,550 and uses 600 Watts [57], a common server equipped with a 40Gbps NIC costs about $4,400 in 2018 and uses 600 Watts under full load, and a 3.3Tbps Barefoot Tofino switch costs about $10,500 and has a power consumption of around 450 Watts [74]. From this table, we can see that compared with the other two hardware devices, packet processing with programmable switches saves dollars by tens of to hundreds of times, which shows their potentials to reduce the cost for attack traffic scrubbing.

Flexibility to support future attacks. As newer and larger DDoS attacks emerge, enterprises today have to frequently purchase more capable hardware appliances and integrate them into the defense infrastructure. Proprietary middleboxes cannot easily support new attacks because of their limited programmability. Software-based defenses (e.g., Bohatei) are much more programmable, but they can only handle lower-speed traffic. In contrast, a single programmable switch could replace the deployment of numerous servers, and it can be programmed with, domain-specific languages like P4 to enable new defenses. These features provide higher performance and flexibility to defend against advanced DDoS attacks.

High packet processing performance. Switching ASICs are specifically designed and optimized for packet processing at line rate. They can achieve several orders of magnitude higher throughput and lower latency compared with highly-optimized software solutions [34]. Also, switching ASICs can provide strong performance isolation [51], which is essential for avoiding increased delays or packet drops for legitimate traffic during DDoS attacks. Other alternatives, such as Smart Network Interface Cards (NICs), Field Programmable Gate Arrays (FPGA) and Network Processing Units (NPUs) cannot match the performance of switching ASICs [34], [35], [78]. Such performance characteristics make switching ASICs a desirable platform for high-throughput and low-latency DDoS defenses, as the resulting defenses are a good match for the requirements of latency-sensitive services in today's data centers.

III. System Overview

In this section, we describe our defense scenario, workflow, and design challenges in more detail.

A. Problem Scope

Deployment Scenario: Our scenario focuses on the DDoS defense in traffic scrubbing centers, where an ISP or cloud net-

work provides DDoS defenses "as-a-service" for its customers, or builds its own traffic scrubbing center to mitigate DDoS attacks. As real-world examples, today's ISPs have already started to provide such value-added commercial services (e.g., AT&T [11]) to customers, and numerous cloud networks also have such scrubbing centers (e.g., Google, Alibaba, and Tencent [15], [16]). Of course, a customer network could also build a scrubbing center of its own. If desired, the ISP or cloud network could also allow its customers to deploy POSEIDON DDoS defense polices for their own traffic (e.g., the customer would monitor and scrub potential attack traffic to itself [61]). The scenario we focus on is complementary to CDN-based DDoS defenses, where users can offload their content to CDNs (Content Delivery Networks), and it can indeed co-exist with other defenses. For instances, before the attack traffic arrives at CDNs, it may be filtered out by some scrubbing center that sits in front of them.

Threat Model: We focus on volumetric DDoS attacks against victim destinations. We assume that attackers have a fixed budget to buy or rent a large number of bots in a botnet (e.g., a collection of compromised IoT devices), and aim to exhaust the bandwidth or computation resources of the victims [77], [79]. Attackers can choose a composition from a set of candidate DDoS attacks (e.g., Smurf attacks, SYN flood attacks, ICMP/UDP flood attacks, Elephant flow attacks, DNS reflection attacks, NTP amplification attacks, HTTP flood attacks, Slowloris attacks, and etc.) and launch multiple DDoS attacks simultaneously. During the attack, attackers can change the types and the mix of attacks dynamically.

B. POSEIDON *Workflow and Challenges*

We illustrate the general workflow of a classic DDoS defense, i.e., attack detection, traffic steering, policy declaration, and attack mitigation. First, the ISP or cloud network uses some in-band or out-of-band anomaly detection techniques to determine whether a customer is under attacks [7], [52]. The detection algorithms are out-of-scope for this paper. We assume that the detection procedure will produce some coarse-grained characterizations of the suspicious traffic, i.e., the attack types, the estimated volume of each type of suspicious traffic, and the suspicious IP prefixes. Alternatively, such information could also be obtained from the victim customers. Note that this is a common practice for many ISPs today [22]. In our scenarios, the estimation for attack traffic does not need to be very precise, and it is only used to help steer the suspicious traffic, to use the right defense polices, and to allocate the switch resources. The *monitor* primitives in the POSEIDON system will further obtain more fine-grained detection results for concrete attack responses. Second, the suspicious traffic is steered to the traffic scrubbing center, and operators specify the needed POSEIDON policies containing the estimated information to mitigate the attack. Third, POSEIDON orchestrates and manages the resource of the scrubbing center, including programmable switches and commodity servers, to coordinate them together for attack mitigation.

Fig. 2 shows this workflow. POSEIDON takes the DDoS defense policies as input, and maps the policies to the available pool of resources (i.e., switches and servers). Users of POSEIDON do not need to understand the details of the underlying resources; instead, they only need to focus on choosing the set

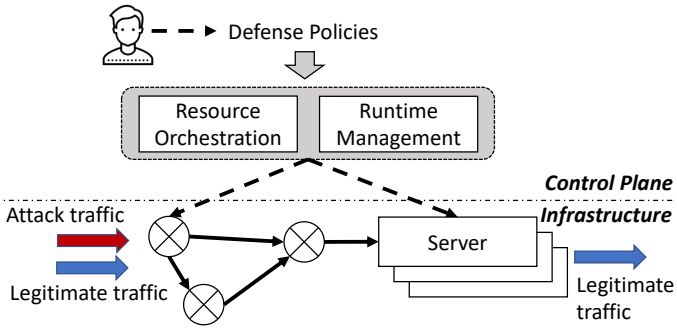


Fig. 2: The overall architecture of POSEIDON.

of primitives for attack mitigation. In order to achieve this goal, POSEIDON needs to address the following three challenges.

Simple, modular policy representation (§IV): Of course, one could always directly write the defense programs in P4 or C/C++, but this would be rather low-level and error-prone [4], [72]. As a result, we desire a simple and modular way to specify the defense policies. However, different DDoS attacks target different protocol- and system-vulnerabilities, so the defenses should necessarily differ in their working mechanisms. For example, a SYN flooding attack exploits the TCP three-way handshake to consume resources on the targeted server and render it unresponsive, a DNS reflection attack exploits vulnerabilities in DNS servers to turn initially small queries into much larger payloads to overwhelm victims, and some new IoT-based DDoS attacks exploit the vulnerabilities of IoT protocols to conduct attacks [64]. The heterogeneity in DDoS attacks requires heterogeneous DDoS defense mechanisms, which make it challenging to design a simple and modular representation to capture the defense policies.

Optimized, efficient defense orchestration (§V): Although programmable switches could achieve several orders of magnitude higher throughput and lower latency than commodity servers, they only have limited on-chip resources and restrictive computational models [10], [72], [74], [75], [44]. Therefore, it is necessary for us to utilize the resources on the switching ASICs as efficiently as possible. Moreover, some defenses may even go beyond the computational model of the switching ASICs, which is impossible to be fully implemented on the switches. The aforementioned points make it challenging to fully explore the potential of switching ASICs to mitigate multi-vectored DDoS attacks.

Handling dynamic attacks at runtime (§VI): Advanced DDoS attacks are usually dynamic, where attackers change its attack composition and the volume of each attack type over time. This requires that POSEIDON should be adaptive to the attack dynamics, i.e., POSEIDON should change the deployed defenses based on the attacks. However, some DDoS defense mechanisms (e.g., SYN proxy) are stateful, and naïvely recompiling the P4 programs for deployment would lead to state loss and flow interruption. A strawman solution is to update the defenses when all flow states are no longer needed. However, since some flows could be long-lived, it may be difficult to identify a single point in time for this update. During this unbound period, the precious resources on programmable switches cannot be used to scrub the attack traffic, which is a waste of precious and high-density defense resources. Achiev-

Expression
E ::= $\mathsf{v} \mid \mathsf{h} \mid M(\vec{\mathsf{v}}) \mid E \diamond E$
Predicate
P ::= $E \circ E \mid P \& P \mid P | P \mid \neg P$
Monitor
M ::= $\mathsf{count}(P, \vec{\mathsf{h}}, \mathsf{every}) \mid \mathsf{aggr}(P, \vec{\mathsf{h}}, \mathsf{every})$
Action
A ::= $\mathsf{drop} \mid \mathsf{pass} \mid \mathsf{log} \mid \mathsf{rlimit} \mid \mathsf{sproxy} \mid \mathsf{puzzle}$
Policy
C ::= $A \mid \mathsf{if}\ P\mathsf{:}\ C\ \mathsf{else:}\ C \mid (C|C)$

Fig. 3: The syntax for expressing POSEIDON defense policies. $\diamond$ describes calculative operators and $\circ$ describes logical operators.

ing an efficient (i.e., switch ASICs utilization) and correct (i.e., without legitimate flow interruption) DDoS defense with programmable switching ASICs is another challenge.

IV. EXPRESSING DEFENSE POLICIES

POSEIDON presents a high-level interface to programmers for developing DDoS defense policies. Instead of exposing low-level interfaces in P4 or C/C++, POSEIDON modularizes a set of defense primitives that can be shared across and composed by different policies. Users can also extend this set of primitives easily.

A. The POSEIDON Policy Language

At first glance, DDoS attacks may seem very different in nature, as they exploit different attack vectors and require different defense strategies. However, we observe that there are key components common to many volumetric attacks—detecting an attack typically requires a set of tests on packet headers or statistics, and responding to a detected attack eventually boils down to specific actions taken on network packets. Therefore, it should be feasible to capture these common components in a high-level abstraction.

In particular, we believe that the Frenetic (NetCore) family of SDN programming languages [53], [2], [4] provides a good starting point due to their high modularity [53]. A policy in these languages consists of a series of match/action statements over a selected set of packet headers. Since these languages are primarily targeted at specifying packet forwarding behaviors, we introduce several customizations for DDoS defense based on a prototype language named NetCore [53]. A summary of POSEIDON's syntax is shown in Fig. 3.

Similar as NetCore, *expressions* could be formed over value (v), header fields (h) or *monitor* instances ($M(\vec{\mathsf{v}})$). Multiple *expressions* can also be composed with different calculative operators together ($E \diamond E$). *predicates* are constructed over *expressions* with diverse logical operators ($E \circ E$), which are used by policies to perform tests and decide on actions. There are also differences from NetCore, however. We allow the definition of attack detection logic using *monitors*, which collect aggregate statistics over certain header fields and use them as indicators of specific attacks. The defense *actions* also go beyond forwarding packets to switch ports—they might, for instance, record needed states across packets (e.g., for SYN flooding defense), invoke more sophisticated actions supported in software (e.g., client puzzles), or combine multiple actions together for mixed-vector attacks. In the following discussion, we mainly illustrate several distinct primitives from NetCore.

First, the detection of DDoS attacks typically relies on traffic statistics instead of per-packet information. Thus, we introduce our *monitor* abstractions. Intuitively, a *monitor* primitive should aggregate some certain packets during a period into a block of memory, with a certain statistic granularity. The statistic granularity is a set of header fields that collectively defines the granularity for monitoring. For instance, some attacks may be detected using statistics over source IP addresses (e.g., UDP flood), whereas others may need a finer granularity (e.g., detecting elephant flows). We provide two monitor elements here, count and aggr. As its name indicates, $\mathsf{count}(P, \vec{\mathsf{h}}, \mathsf{every})$ selects the packets that satisfy the predicate P, and counts the number of the packets with granularity $\vec{\mathsf{h}}$ every every seconds. For example, $\mathsf{count}(\mathsf{pkt.tcp.flag} == \mathsf{SYN}, [\mathsf{ip.src}], 5)$ denotes counting all the TCP SYN packets for each source IP address every 5 seconds. As another example, $\mathsf{count}(\mathsf{pkt.udp.dport} == 53, [\mathsf{ip.src}], 3600)$ would monitor the source addresses that have sent DNS requests. The period every is set as one hour so that the DNS replies that reply too late is regarded as invalid. Different from count, which focuses on the number of elements, $\mathsf{aggr}(P, \vec{\mathsf{h}}, \mathsf{every})$ increments the counter with packet sizes. It can be used as a powerful primitive to collect statistics over the number of bytes for certain packets. For instance, $\mathsf{aggr}(\mathsf{True}, [\mathsf{ip.src}], 5)$ would count the size of all the packets for each source IP address every 5 seconds.

Second, primitive *actions* are central building blocks for DDoS response: an action receives a set of packets and conducts the corresponding processing for these packets. We observe that although DDoS defenses are heterogeneous for different attacks, the defense actions have many similarities, and there is only a limited set of basic building blocks. Once a malicious action is detected, some defense mechanisms simply drop the packets with a specific predicate (drop). While for benign identifies (e.g., source IP address), most defense mechanisms let their packets pass (pass). Rate limiter (rlimit) rate-limits the packets that satisfy certain conditions. And for most TCP-related DDoS attacks, SYN proxy (sproxy) is a powerful defense element, and puzzle is effective for HTTP-based flood attacks. Meanwhile, for many attacks, operators may have the need to log certain packets for forensic use, so we also introduce a log primitive. Importantly, the set of building blocks presented here is not meant to be an exhaustive list—programmers can easily add new ones to the library of defense primitives using our policy language (more discussion in §IX).

Finally, for policy declaration, POSEIDON is very similar to NetCore/NetKAT family of languages, which allows branches (if ... else ...) and policy composition (|). Users can include conditional branches that invoke different defense primitives based on certain conditions. They can also compose multiple policies together using the composition operator |. We will illustrate these primitives with concrete examples in the next subsection.

B. Policies by Examples

Now, we describe DDoS defense policies for six typical DDoS attacks, where the first two are adapted from

```
1  syn_count = count(pkt.tcp.flag == SYN, [ip.src], 5)
2  ack_count = count(pkt.tcp.flag == ACK, [ip.src], 5)
3
4  if syn_count([pkt.ip.src]) - ack_count([pkt.ip.src]) > T:
5      drop
6  else if syn_count([pkt.ip.src]) == ack_count([pkt.ip.src]):
7      pass
8  else:
9      sproxy
```

Fig. 4: SYN flood defense.

```
1  dns_query = count(pkt.tcp.dport == 53, [ip.src], 3600)
2
3  if pkt.tcp.sport == 53:
4      if dns_query([pkt.ip.dst]) > 0:
5          pass
6      else:
7          drop
```

Fig. 5: DNS amplification defense.

Bohatei [22], and the third is a new policy supported by POSEIDON. Please refer to Appendix A for the other three defense examples. Our goal here is not to develop new defense mechanisms, but to illustrate the flexibility and simplicity of POSEIDON policy language in dealing with a diverse set of DDoS attacks. Once the characteristics of one attack are identified, operators can easily and simply express the defense policy for the attack. It is worth mentioning that our primitives do not have to be purely implemented in the programmable switches, while some sophisticated primitives may need the assistance of the servers.

SYN flood attack. As shown in Fig. 4, we first track the number of open TCP sessions for each source IP every 5 seconds, which is implemented by counting SYNs and ACKs with the count primitive (first 2 lines). Based on the statistics of the previous period, if a source IP has much more SYNs than ACKs, we mark its future packets as attacks and drop them (line 4, line 5). If a source IP has no asymmetry between SYNs and ACKs, we mark it as benign and let its packets pass (line 6, line 7). Otherwise, we mark this as a gray area and send packets from these source IPs to a sproxy defense module (line 8 and line 9).

DNS amplification. In DNS amplification attacks, attackers use numerous spoofed protected servers' IPs to request many DNS queries that result in a large number of answers to overwhelm the protected servers' bandwidth. To defend against this kind of attacks, as shown in Fig. 5, we first track all the protected servers' DNS queries (line 1). Only the incoming DNS replies which have been queried by protected servers within an hour are allowed to enter the network (lines 3–7).

HTTP flood attack. In HTTP flood attacks, each attacker generates a large number of legitimate HTTP requests and sends them to victim servers, which can easily overload the web servers and make the service unavailable. To mitigate this kind of attack, as illustrated in Fig. 6, a simple approach is to track the number of HTTP requests for each source IP (line 1). If the number of client sessions exceeds a threshold (T) during the previous period, adopt the puzzle mechanism for this source IP (lines 3–4). Otherwise, forward the packets from this source IP as normal (lines 5–6). Note that the puzzle defense cannot be implemented within the programmable switch. Rather, the switch redirects the flow to an HTTP server that implements the defense (e.g., a CAPTCHA). We discuss puzzle defense further in §V-A.

```
1  http_get_counter = count(pkt.http == GET, [ip.src], 5)
2
3  if http_get_counter([pkt.ip.src]) >= T:
4      puzzle
5  else:
6      pass
```

Fig. 6: HTTP flood defense.

From these examples and programs, we can see that POSEIDON policy language is easy to understand and expressive enough to convey operators' defense intents.

V. ORCHESTRATING THE DEFENSE

POSEIDON has a resource orchestration component that analyzes each primitive in a given policy, and partitions the needed functions across the switches and the servers for effective defense. At a high level, POSEIDON first constructs a directed graph of defense primitives, and computes an optimal placement of the graph by solving several sets of constraints.

A. Analysis of Defense Primitives

As described before, POSEIDON has three classes of defense primitives: a) *monitors* collect statistics over the network traffic (e.g., lines 1–2 in Fig. 4), b) *actions* specify the defense decisions taken on a particular kind of packets (e.g., lines 5, 7, and 9 in Fig. 4), and c) *branches* express the control flow of the defense (e.g., lines 4, 6, and 8 in Fig. 4). Before delving into the details about primitive placement, we describe how POSEIDON supports each kind of primitives, and how much resource each primitive requires on the switch and/or server. TABLE II contains a summary.

Monitors. The detection of DDoS attacks relies on collecting traffic statistics over packet headers. The *monitors* in POSEIDON can be fully implemented in the switches for these purposes. Under the hood, POSEIDON implements the *monitors* using *sketches* [88], which are resource-efficient data structures that can approximate the needed statistics with well-known error bounds. Instead of storing precise per-flow information, a sketch uses hash functions to compute several indexes of a flow ID, and then accesses the corresponding values in register arrays. All these operations can be performed at line rate. A sketch requires two match-action tables, one for hashing a key (e.g., a flow ID) to compute the indexes to the register arrays, and another for updating the values in the arrays using stateful ALUs. Since the index is computed by hashing the packet header fields, it is unavoidable that different packet headers may be mapped to the same index, resulting in collisions. Nevertheless, it has been shown that the resulting error bound is small enough to be practical for DDoS defense [27], [43].

Specifically, our $\mathsf{count}(P, \vec{\mathsf{h}}, \mathsf{every})$ and $\mathsf{aggr}(P, \vec{\mathsf{h}}, \mathsf{every})$ primitives are implemented using a) *count-min sketches* [18] as the primary data structure, and b) a match-action table to select packets based on the given predicate P. To maintain timing-related information for count and aggr, similar as NetHCF [6], we use two register arrays in the switch SRAM. One array C records information about the current time period, and another P stores information about the most recent period in the past. When packets come in, the first array is updated to collect the statistics; at the end of a period, we copy the content of C to P using the switch control plane.

TABLE II: Implementation details and resource utilization of POSEIDON primitives.

Primitives	Switch Component	Switch Resource Usage	Server Component
monitors			
count($P, \vec{\mathsf{h}}$, every)	match-action entry + count-min sketch	stages: 2, hash functions: $\lceil \log_{1/2} \delta \rceil$, stateful ALUs: 6, SRAM: for the ϕ biggest elements in a set, in order to achieve a relative error bound of ε with probability δ, usage = $\frac{64\lceil \log_{1/2} \delta \rceil}{\varepsilon \phi}$	N/A
aggr($P, \vec{\mathsf{h}}$, every)	match-action entry + count-min sketch	stages: 2, hash functions: $\lceil \log_{1/2} \delta \rceil$, stateful ALUs: 6, SRAM: for the ϕ biggest elements in a set, in order to achieve a relative error bound of ε with probability δ, usage = $\frac{64\lceil \log_{1/2} \delta \rceil}{\varepsilon \phi}$	N/A
actions			
drop	flow entry	stages: 1, hash functions: 0, stateful ALUs: 0, SRAM: negligible	N/A
pass	flow entry	stages: 1, hash functions: 0, stateful ALUs: 0, SRAM: negligible	N/A
rlimit	meter + flow entry	stages: 3, hash functions: 1, stateful ALUs: 0, SRAM: in order to achieve a false positive rate of ε, usage = $\frac{8n}{ln(1/(1-\varepsilon))}$	N/A
sproxy	handshake proxy + session relay	stages: 3, hash functions: 2, stateful ALUs: 4, SRAM: in order to achieve a false positive rate of ε, usage = $\frac{32n}{ln(1/(1-\varepsilon))}$	N/A
puzzle	-	-	CAPTCHA
log	selecting, grouping	stages: 3, hash functions: 2, stateful ALUs: 2, SRAM: in order to achieve a false positive rate of ε, usage = $\frac{32n}{ln(1/(1-\varepsilon))}$	aggregation
branches			
if . . . else . . .	tag-based match action	stages: 1, hash functions: 0, stateful ALUs: 0, SRAM: negligible	N/A

Actions. POSEIDON has a set of defense primitives that take actions on network traffic based on the statistical results. POSEIDON's framework is general enough to capture a range of defense actions, including a) the class of defenses that can be supported entirely in the switch ("switch only"), b) the class of defenses that require some level of server involvement ("switch assisted"), and c) the class of defenses that needs to run entirely on the servers ("server only"). Defenses in switch-only class can fit into the programming model of the switching ASIC. The current version of POSEIDON supports drop, pass—which can be mapped to the corresponding match-action table entries easily, as well as rlimit and sproxy—which are more complex and require more resources in the switch pipeline; this set can be easily extended to include more defenses.

Switch-assisted defenses need to be carefully partitioned into two separate components: a switch component that is offloaded to hardware, and a server component that runs in software. POSEIDON aims to carve out as much as logic possible for hardware offloading, since this would translate to higher performance. For instance, consider the log primitive with three steps. It first selects the kind of traffic to be logged, then groups the packets of interest based on certain keys (e.g., flow IDs), and finally aggregates the results for logging. Similar as Marple [55] and *Flow [75], POSEIDON uses match-action tables to implement the select step, uses stateful registers to group the results, and performs the aggregation step on servers since it involves more complex logic. In this example, the servers only need to do minimal amount of work, since the switch component has filtered out most of the irrelevant data.

Server-only defenses require sophisticated actions that go beyond the capability of the switching ASIC, such as those that require complex arithmetic operations, loops, or application-layer processing. Offloading these operations to the switch is not possible at least with today's switching hardware. A representative case is puzzle [37], [82], [38], which is often used to defend against HTTP-based flood. Puzzle forces each client to solve a cryptographic puzzle (e.g., graphical puzzles) for each request before the server provides its resources, thereby imposing a large computational task on attackers bent on generating legitimate service requests to consume server resources. We use *CAPTCHA* as an implementation of puzzle.

Policy declaration. First, DDoS defense usually takes different actions for different types of traffic, and this can be supported using branches to specify the control flow. An if . . . else . . . branch could be implemented as a tag-based match-action table, which classifies incoming packets that match different predicates using different tags. For example, in Fig. 4, we generate different tags for packets that satisfy different predicates, e.g., tags 1, 2, and 3 for the predicates in lines 4, 6, and 8, respectively. Each branch is then mapped into a tag-based match-action entry, and the following code block would identify the packets based on their tags. Second, composition operator | are very useful when operators want to compose multiple policies, which allows operators to apply different polices to different packet group together. Currently, if two policies have different actions for the same packet, we simply adopt the stricter one. For example, if policy 1 would like to drop the incoming packet while policy 2 lets it pass, we will drop it finally.

Flow affinity. In addition, some defenses need to be stateful and have bidirectional semantics. For example, sproxy requires that the inbound and outbound traffic of the same flow are always steered to the same instance; similarly, DNS requests and responses should also be processed by the same instance. To achieve this, we design our hash function as $hash_1(pkt.src) + hash_2(pkt.dst)$, in which way exchanging source and destination fields does not affect the final hash value.

B. Placing Defense Primitives

Next, we describe the algorithm that POSEIDON uses to place the various defense primitives to the network.

Similar as [36], [72], POSEIDON extracts a graph structure from the defense policy, where the nodes are the defense primitives and the edges represent the traffic flow. Note, however, each defense primitive has self-contained state, and for modularity, it does not expose internal states to other

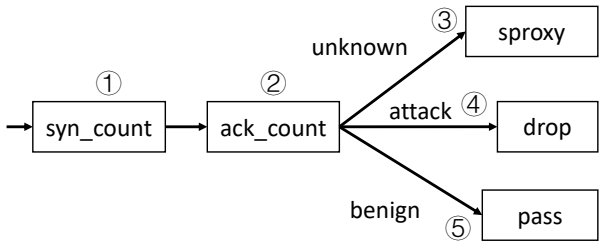


Fig. 7: SYN flood defense graph.

primitives explicitly. Therefore, POSEIDON uses a topological sort to transform the graph into an ordered list of primitives. For instance, as shown in Fig. 7, the graph for syn flood defense could be transformed into a list of nodes ①②③④⑤. When there are multiple defenses that need to be deployed in conjunction, POSEIDON would obtain a list of primitives for each. It then computes the resource usage of the primitives based on the analysis in §V-A, and uses the information for placement.

POSEIDON then places the lists of nodes into the network, including programmable switches and commodity servers. Since programmable switches can achieve orders of magnitude higher performance, our goal for our placement is to maximize the amount of processing offloaded to the switches. Of course, the resource limitations of the switches pose constraints to our problem, most prominently in terms of the number of stages in a switch, and the amount of SRAM (for registers) and stateful ALUs per stage. Our placement algorithm takes these constraints into account while optimizing for maximal offloading.

To reduce the switch-server traffic transfer, the placement algorithm partitions each list once and only once—the first part is offloaded to the switch and the second to the servers. To mitigate the resource limitations of a single switch, POSEIDON can also leverage resources from multiple switches for processing. Concretely, several switches can be organized together sequentially, which can provide more processing stages and memory resources as traffic flows through. This effectively abstracts a path that consists of multiple switches into a much larger switch—for instance, the number of usable stages has increased to $S = \sum_{switch} Num_{stage}$. As future work, we are also planning to support parallelism across multiple paths. The sequential and parallel placement can also be used together to achieve even higher performance. POSEIDON then formulates the placement problem as an Integer Linear Program (ILP).

Input. Assume that POSEIDON needs to place P defense programs, and that each program has N_p nodes. We further assume that the estimated volume of each type of attacks is available to POSEIDON with a certain expected error probability. Using the above information, we can compute the switch resources each defense primitive would require. We use the following notations for the various types of resources: for the n-th node of program p, it uses a stage count of $STAGE_{p,n}$, SRAM in the tth stage $SRAM_{p,n,t}$, and the stateful ALUs in the tth stage $ACTION_{p,n,t}$ ($1 \leq t \leq STAGE_{p,n}$). Furthermore, the amount of traffic after passing through this node is $T_{p,n}$. Every node in the processing would reduce the traffic volume, so that the amount of traffic received by the servers would be minimized.

Output. We define $X^j_{p,n} = 1$ if and only if the n-th node of the p-th program starts at the j-th stage of the "abstracted" switch (i.e., a path of switches), otherwise $X^j_{p,n} = 0$. So for each program P, the last node on the switch would be $LastN_p = \sum_J \sum_N X^j_{p,n}$. As a result, our objective can be written as

$$max \sum_{P} \sum_{n=1}^{n=LastN_p} T_{p,n} \tag{1}$$

Constraints. There are several types of constraints that we need to consider.

Register memory per stage. For each stage, the amount of SRAM allocated for packet processing cannot exceed $SRAM$. Thus we have

$$\forall j,t, \sum_{P} \sum_{Np} SRAM_{p,n,t} \cdot X^j_{p,n} \leq SRAM \tag{2}$$

Number of stateful ALUs per stage. Similarly, for each stage, the total number of stateful ALUs allocated for packet processing cannot exceed $ACTION$. Thus we have

$$\forall j,t, \sum_{P} \sum_{Np} ACTION_{p,n,t} \cdot X^j_{p,n} \leq ACTION \tag{3}$$

Number of stages. The total number of stages for all defense programs cannot exceed the upper limit of stage count S. Here, we use $Z_{p,n}$ to denote the start stage of node n for program p. $Z_{p,n}$ and $X^j_{p,n}$ are related: if $X^j_{p,n} = 1$, then $Z_{p,n} = j$. Then we have

$$\forall p,n, Z_{p,n} + STAGE_{p,n} \leq S \tag{4}$$

Node ordering. The placement should respect the ordering of the nodes, i.e., for each program P, if the node $n1$ precedes node $n2$, then the start stage of node $n1$ should appear earlier than the start stage of node $n2$ subtracting $STAGE_{p,n1}$. Then we have

$$\forall p, if\ n1 < n2, Z_{p,n1} + STAGE_{p,n1} < Z_{p,n2} \tag{5}$$

Using the above constraints, we can solve the 0-1 Integer Linear Programming (ILP) problem and obtain the optimal placement using existing optimization toolboxes [31]. The result would specify which primitives should be placed in the switch, as well as the amount of allocated resources to each primitive.

VI. HANDLING DYNAMIC ATTACKS

Next, we discuss how POSEIDON handles dynamic attacks at runtime. To ensure defense correctness, we need to replicate state in the programmable switches and use server memory as a temporary store. When a switch is being reconfigured with a new P4 program, traffic is steered to the relevant servers that contain defense state for processing. To achieve this, POSEIDON uses a central controller to coordinate the switches and the servers (see Fig. 9). During a policy update, the controller generates a new defense strategy, i.e., deploying a new P4 program to the switches and a new configuration for the servers. The new P4 program would be loaded to the switches directly, replacing the previous defense strategy.

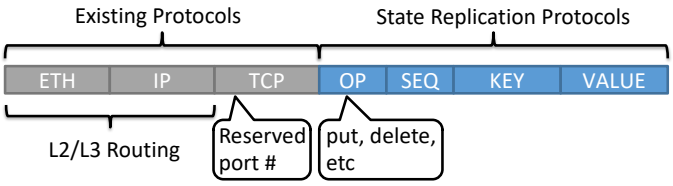


Fig. 8: Format of the state replication packets.

During this update, flows are sent to the relevant servers for processing. The servers always implement logic for all types of defenses, but the switch-only defenses are never activated unless in this transition state. In order to replicate state at runtime, there are several issues that need to be solved for efficient and consistent replication.

States requiring replication. An intuitive approach is to identify all states using program analysis techniques [41], [45] (e.g., the *registers* in P4), and replicate these states to servers when they are modified. However, on one hand, some states can be automatically recovered after the traffic is steered to the servers, which means replicating these states is not necessary. On the other hand, some states are no longer useful when an attack finishes, such as monitor statistics, so they do not need to be replicated. Our principle here is *to identify the states which will still take effect for legitimate traffic even when attacks finish*, with the goal of ensuring correctness for legitimate flows. In the current version of our primitives, sproxy is a good example. It maintains the difference between sequence numbers for the synproxy-source connection (generated as SYN cookies by the synproxy) and the destination-synproxy connection (chosen by the destination when a connection is established by the synproxy on behalf of a verified source). This state is needed for sequence number translation for each packet after the connection is established. Therefore, it must be replicated to the servers for flows from legitimate hosts.

Approach to replication. The networking community has developed several approaches [29], [41], [84] to migrating state across virtual machines, which we can leverage for our problem. However, these approaches are not directly applicable. First, packet processing at switches is at line rate, and its performance is several orders of magnitude higher than that of commodity servers / virtual machines. As a result, we cannot apply a one-time cost operation (e.g., move states from switches to servers via expert/import APIs when scaling) when we want to recompile the switch with new P4 programs, since it is almost impossible to infer all the exact state locations immediately. Moreover, different from the scenario where the source and destination for state migration have similar processing power, state on the switches comes in much higher volume. Simply replicating the states from the switch to the server would easily overwhelm the server.

To address these problems, our first step is to amortize the traffic overhead across a period: when state is created or modified in switches, we replicate it to the servers. Some states may be out-of-date in the switch if a flow is terminated. This signal is also transferred to the servers so that the relevant states can be removed. We provide a unified interface between the switch and server to keep the states consistent and up-to-date, using a state replication protocol as shown in Fig. 8. `OP` stands for operator, which can be a `Put` (state creation

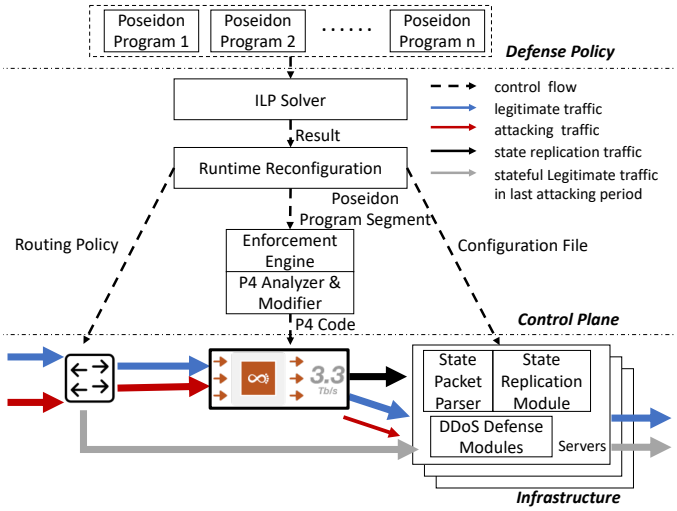


Fig. 9: POSEIDON Implementation.

or update), `Delete` (state deletion), or other types of state synchronization operations. `SEQ` is used as a sequence number for reliable transmissions. `KEY` records the packet headers to index the state, and `VALUE` records the value of the state. For example, in a typical sproxy, the `KEY` should be the five-tuples and the `VALUE` should be the sequence number difference.

Second, to avoid overwhelming the servers, we spread the traffic from a switch across a set of servers. During runtime, the state replication traffic is distributed across these servers, which is achieved by embedding the server's IP address into the IP field of the state replication packets. The mapping between the destination server's IP address and the `KEY` is stored in our controller. When the reloading starts, the controller updates the upstream routing table with this mapping to guarantee the corresponding traffic is steered to the correct server instances. Note that there is a small time gap (hundreds of milliseconds) for the P4 program to be successfully loaded. During this time period, the suspicious traffic is steered to the server clusters for traffic scrubbing, and these servers also constantly report the established legitimate flow information to the controller. After the new P4 program takes effect, the controller updates the routing tables again to steer the traffic to the switches.

Summary. To summarize, the runtime state replication follows the following steps: (1) When operators specify a policy, POSEIDON identifies the states that need replication. (2) At runtime, if states are created/updated/deleted, POSEIDON generates state replication packets and steers such traffic across a set of servers. It also records the mapping between the server's IP address and the `KEY` in the controller. (3) When operators changes the defense policy to handle new attacks, POSEIDON updates the upstream routing according to the mapping, so as to ensure that the legitimate traffic is steered to the correct servers. It also recompiles and reloads the new P4 programs. Note that the entire procedure runs automatically when the POSEIDON system starts, and operators only need make changes to the high-level policies when there are dynamic attacks.

VII. IMPLEMENTATION

We have developed a prototype implementation of POSEIDON, including all the primitives in §IV, a policy enforcement

engine in §V, and the switch/server interface and the state replication mechanisms in §VI, as shown in Fig. 9. The primitives that can be offloaded to switches in POSEIDON are implemented in P4 on Barefoot Tofino [56] switches, using ∼1800 lines of code. The corresponding parts that run on servers are implemented in DPDK [19], with ∼3600 lines of code in C/C++. For the primitives that cannot be offloaded into switches (e.g., puzzle), we reuse the state-of-the-art defenses adapted from open source systems, such as CAPTCHAs.

The policy enforcement engine is implemented in Python with ∼600 lines of code. To translate a POSEIDON policy into a P4 program, the engine first partitions the POSEIDON policy into the stateful monitors (e.g., count, aggr) and the packet processing logic. Then these two components are translated into different P4 program segments (e.g., registers, match-action tables, control flow) separately. Finally, the two program segments are spliced into a complete P4 program. The library of defense primitives (e.g., sproxy, rlimit, pass) can be directly accessed by the policy enforcement engine when translating the code. To support future extensions, we have also implemented a script that can transform new defense actions into the format of the library of POSEIDON actions. New actions can therefore be added easily. The script also avoids name conflicts by adding a prefix to variable names in the original action code.

The switch/server interface is implemented in P4 for the switch component and in C/C++ (using DPDK) for the server component. On the switch side, we have a P4 *analyzer* module and a P4 *modifier* module, with ∼400 lines of Python code. The P4 *analyzer* module first extracts the states that need to be replicated, then the P4 *modifier* module augments the original P4 program to support state replication. On the server side, we implement a state parser module and a state replication module in a separated thread using DPDK in ∼500 lines of code. The state parser module first parses the keys and values from the packets, then the state replication module updates the corresponding states in the servers.

VIII. EVALUATION

In this section, we evaluate POSEIDON with respect to the following key questions:

- How expressive is the POSEIDON language in supporting different defense policies (§VIII-B)?
- How efficient is the POSEIDON policy placement mechanism in terms of resource utilization (§VIII-C)?
- How effective is the POSEIDON runtime management mechanism for adapting to dynamic attacks (§VIII-D)?
- How well can POSEIDON mitigate attacks, in terms of defense effectiveness, performance and cost (§VIII-E)?

A. Experimental Setup

[1]http://mawi.wide.ad.jp/mawi/samplepoint-G/2019/201908281400.html
[2]http://mawi.wide.ad.jp/mawi/samplepoint-F/2019/201909011400.html

TABLE III: Replayed workload traffic.

#	Traffic Trace	Average Flow Length	Average Packet Size
T1	ToIXP Traffic[1]	165.7 packets/flow	1253B/packet
T2	ToISP Traffic[2]	75.1 packets/flow	564B/packet
T3	Enterprise Traffic	9.5 packets/flow	622B/packet

We use a combination of a real-world testbed and trace-driven evaluations to demonstrate the aforementioned advantages. Our testbed has 10 Dell R730 servers, a Barefoot Tofino switch (33 x 100 GbE) and an H3C switch. Each server is equipped with Intel(R) Xeon(R) E5-2600 v4 CPUs (2.4 GHz, 2 NUMAs, each with 6 physical cores and 12 logic cores), 15360K L3 cache, 64G RAM and two Intel XL710 40GbE NICs. Fig. 9 shows the setup of the eight servers, the Tofino switch, and the H3C switch, which compromise the defense infrastructure; in addition, one server acts as the controller that translates defense policies for deployment; and another server hosts the traffic generator, which can generate normal workloads and different types of attack traffic.

The normal workload traffic is collected from an online trace dataset [60] and an enterprise, including three types of traffic traces to cover different scenarios, as shown in TABLE III. These traces have different flow length and packet sizes. We also use two public real-world attack traces, a SYN flood attack trace [81] and a UDP flood attack trace [23]. For the other four types of attack traffic, i.e., DNS amplification, HTTP flood, Slowloris and Elephant flow, we use a specialized DDoS traffic generating tool, `UFONet` [21], to generate the corresponding attack traffic traces. In our experiments, we replay these traces with `DPDK Pktgen` to generate high-volume workload traffic and attack traffic. On our testbed, we can ramp up the attack volume up to 40 Gbps. For larger attacks, we use simulations.

B. Policy Expressiveness

To demonstrate the expressiveness of the POSEIDON primitives, we have summarized state-of-the-art DDoS attacks and their defense mechanisms, and presented the results in TABLE IV. We further categorize them by different protocols. For each protocol, there are a variety of DDoS attacks, each targeting some specific vulnerabilities. Next, we present a typical defense solution using POSEIDON primitives for each DDoS attack.

ICMP Protocol. ICMP-based DDoS attacks include ICMP flood attacks and Smurf attacks. To defend against ICMP flood attacks, we can first use the count primitive to identify suspicious IPs that send too many ICMP echo-request packets, and then use the rlimit primitive to rate-limit the packets from these IPs. For other IPs, we can simply let their packets pass. For Smurf attacks, we can use the count primitive to track all the protected servers' ICMP echo-request packets within a period, and only allow ICMP echo-reply packets that have been queried by protected servers to enter the protected network.

TCP Protocol. For TCP-based DDoS attacks, we have already discussed typical defenses with POSEIDON primitives for SYN flood attacks and Elephant Flow attacks in §IV-B. For SYN-ACK flood attacks and ACK flood attacks, we can use the

TABLE IV: State-of-the-art DDoS attacks and their corresponding defense mechanisms.

Protocol	DDoS attack	Description	Typical defense solution	Poseidon defense
ICMP	ICMP Flood	The victim servers are flooded with fabricated ICMP echo-request packets from a wide range of IP addresses	Rate-limit received ICMP packets from the same address or subnet	count + rlimit/pass
	Smurf Attack	A large number of fake ICMP echo-request packets with the victim severs' IP address are broadcast to a large network using an IP broadcast address	Filter ICMP echo-reply packets that are not queried by the victim servers	count + drop/pass
TCP	SYN Flood	The victim servers are bombarded with fabricated SYN requests containing fake source IP addresses	SYN Cookie/Proxy	count + sproxy/pass/drop
	SYN-ACK Flood	The victim servers are flooded with a large number of fake SYN-ACK packets	Filter SYN-ACK packets that are not queried by the victim servers	count + pass/drop
	ACK Flood	The victim servers are flooded with fabricated ACK packets from a wide range of IP addresses	Filter ACK packets that have not been responded by the victim servers with SYN-ACK packets	count + pass/drop
	FIN/RST Flood	The victim servers are bombarded with fake RST or FIN packets that do not belong to any of active connections	Filter FIN/ACK packets that do not belong to any action connections, then rate-limit received FIN/RST packets from the same connection	count + rlimit/pass/drop
	Elephant Flow	The attacker generates elephant flows to overwhelm the bandwidth of the victim servers	Rate-limit flows that send too many bytes	aggr + rlimit/pass
UDP	UDP Flood	The victim servers receive a large rate of fake UDP packets from a wide range of IP addresses	Rate-limit received UDP packets from the same address or subnet	count + rlimit/pass
	DNS Flood	The victim DNS servers are bombarded with a flood of requests from a wide range of IP addresses	Rate-limit received DNS requests from the same address or subnet	count + rlimit/pass
	DNS Amplification Attack	The attacker requests data about a domain from public DNS servers, and directs the reply to the victim servers	Filter DNS replies that are not queried by the victim servers	count + pass/drop
	SSDP DDoS Attack	The attacker spoofs discovery packets with the victim servers' IP address to each plug-and-play device, to request for as much data as possible by setting certain flags	Filter SSDP replies that are not queried by the victim servers	count + pass/drop
	QUIC Reflection Attack	By spoofing the victims' IP address and sending a "hello" message to QUIC servers, the attacker tricks the servers into sending large amounts of unwanted data to the victim servers	Filter QUIC replies that are not queried by the victim servers	count + pass/drop
	NTP Amplification Attack	The attacker sends numerous NTP requests providing the victim servers' IP address	Filter NTP replies that are not queried by the victim servers	count + pass/drop
	Memcached DDoS Attack	The attacker spoofs requests to a vulnerable UDP memcached server, which then floods a targeted victims with large amount of traffic	Filter Memcached replies that are not queried by the victim servers	count + pass/drop
HTTP	HTTP Flood	The attacker generates large numbers of HTTP requests and sends them to the victim servers	Set limits for client sessions, CAPTCHA	count + pass/puzzle
	SlowLoris Attack	The victim servers are bombarded with too many open connections	Rate limit IP sources that establish numerous connections but send a few bytes	count/aggr + rlimit/pass

TABLE V: Lines of code to implement different defense intents in POSEIDON, P4, and C/C++ (using DPDK).

Policy	Attack	POSEIDON	P4	DPDK
1	SYN flood	9	939	1070
2	DNS amplification	7	255	898
3	HTTP flood	6	354	1184
4	Slowloris	8	513	995
5	UDP flood	6	376	911
6	Elephant flow	6	373	903

count primitive to track whether the SYN-ACK packets or ACK packets have been generated by protected servers within a period, and only allow legitimate packets to enter the protected network. For FIN/RST flood attacks, we can first use the count primitive to track the connections between the protected servers and the external network, and only allow FIN or RST packets in the active connections. Furthermore, for these remaining FIN or ACK packets, we use the count primitive again to count the number of FIN/RST packets in each connection and use the rlimit primitive for rate limiting.

UDP Protocol. UDP-based DDoS attacks (especially UDP-based amplification attacks) are one of the most popular DDoS attacks today [64]. We have already discussed the corresponding defenses with POSEIDON primitives for UDP flood attacks and DNS amplification attacks in §IV-B. For DNS flood attacks, we can use a similar approach as UDP flood attacks, rate-limiting the DNS request packets from the same address or subnet. For the other four UDP-based amplification attacks (SSDP DDoS attacks, QUIC Reflection attacks, NTP amplification attacks, and Memcached DDoS attacks), we can use a similar defense as in DNS amplification attacks, filtering replies that are not triggered by the victim servers.

HTTP Protocol. We have already discussed the corresponding defense solutions in POSEIDON for HTTP flood attacks and SlowLoris attacks in detail in §IV-B.

From the discussions on each categorization above, we can see that although the attacks require different defense solutions, almost all these DDoS defenses require monitoring primitives to identify the malicious packet groups, as well as a reusable set of defense actions for packet processing. Regarding this point, our *monitor* primitives provide a useful abstraction to aggregate certain packets and collect statistics, and our *action* primitives offer a powerful packet processing abstraction for DDoS defenses. From TABLE IV, we can also see the current set of primitives are expressive enough to cover a wide range of state-of-the-art DDoS defense mechanisms.

We have provided six policy examples in §IV-B to demon-

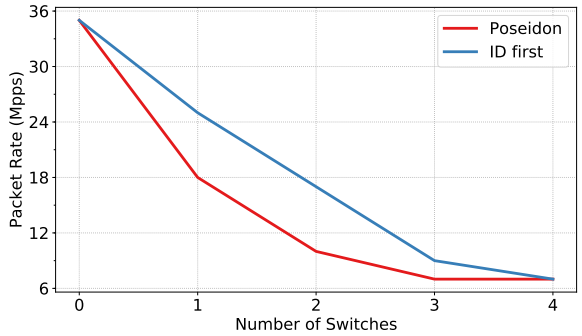


Fig. 10: Traffic arriving at servers.

strate the the expressiveness of our POSEIDON policy language. We further summarize the number of lines of code in POSEIDON, in P4, and in C/C++ when implementing these six policy examples, as shown in TABLE V. From this table, we can see that POSEIDON allows concise specifications of the defense policies, without having to burden network operators with the task of writing low-level code. Moreover, POSEIDON also shields the complexities of the underlying hardware. Policies #1, #2, #4, #5, and #6 can be fully implemented in the switches, whereas #3 requires the assistance of the servers. But a programmer does not have to be aware of the implementation details. In particular, programming in P4 is akin to "assembly-level" programming and usually requires hand optimizations. We have also included sample code in P4 for implementing policy #1 in Appendix B as a concrete example.

C. Policy Placement Mechanism

To demonstrate the efficiency of POSEIDON policy placement mechanism, we compare it with a strawman solution, which simply places the programs using their policy IDs, the smallest ID first. We assume that attackers launch three attacks simultaneously, 10 Mpps SYN flood, 20 Mpps DNS amplification and 15Mpps HTTP flood. The corresponding resource utilization for each primitive could be obtained from TABLE II. We assume that a switch has 12 stages, each with 5Mb register array and 4 stateful actions; these constraints are much more strict than most Barefoot Tofino switches. We use the rate of packets arriving at servers as the metric to evaluate the effectiveness of our policy placement mechanism.

As we can see from Fig. 10, the more switches there are, the more traffic is filtered at the switch and the fewer packets are sent to the servers. Comparing with the strawman placement mechanism, the placement of POSEIDON can mitigate larger attacks with the same number of switches. Note that the curves for both approaches would finally converge to the same points. This is because when there are enough programmable switches, all primitives that can run on the switches have been offloaded; the rest of the primitives need to run on the servers, and this leads to a constant number of packets to be sent to the servers for processing. In our scenario, it is impossible for attackers to launch tens of DDoS attacks simultaneously. As a result, our ILP problem has a relatively small size, and it can be solved within seconds. This also indicates that our system is able to orchestrate the defense resources in a pretty fast manner, which can accommodate to policy changes quickly.

D. Dynamic DDoS Attacks

To evaluate the effectiveness of runtime state replication against attack dynamics, we mix normal workload traces with attack traces and replay them from the packet generator. At runtime, we change the attack from #1 to #2, and adapt the defense policy accordingly. As we can see in Fig. 11 and Fig. 12, POSEIDON ensures that legitimate traffic goes through the scrubbing center normally without broken connections, even without packet loss. In contrast, without runtime state replication, connections will be broken and packets will be dropped, since no state can be found on the servers. This would interrupt the legitimate flows and lead to significant performance degradation. In addition, from the trends of the three traces, we can conclude that the more elephant flows the trace contains, the worse the flow interruption and packet loss will be.

To evaluate the overhead of runtime state replication, we replay three attack traffic traces at an increased rate. As shown in Fig. 13, the ratio between control message traffic and workload traffic is constant even when the attack traffic is multiplied. This is because POSEIDON only replicates runtime states for legitimate traffic. In this figure, the control message ratio for `Enterprise` traffic is a bit higher than other two traces. This is because the `Enterprise` trace contains a large number of mouse flows (about 9 packets per flow), and for each flow, we have to generate a state replication packet. Even in this case, the overhead is still very low (less than 4%), which indicates that our runtime management scheme incurs negligible overheads.

E. Overall Effectiveness

To demonstrate the effectiveness of POSEIDON during attacks, we measure the bandwidth of legitimate TCP flows under the six types of DDoS attacks, and count the number of sent/received packets at the traffic generator. We use a simple time series anomaly detection tool `nfdump` for the coarse-grained detection; the detection results would further trigger the loading of different defense policies, and the allocation of switch resources. Fig. 14 shows the defense effect for legitimate TCP flows during the UDP flood attack. The defense effect for the other types of DDoS attacks is similar, since most attack traffic is filtered by POSEIDON before it reaches the traffic generator[3]. As we can see from the figure, POSEIDON can respond to the attack rapidly and restore the throughput of legitimate flows quickly, which indicates the effectiveness of POSEIDON in coping with DDoS attacks. Note that there is a small time gap between the attack onset and the defense taking effect (in seconds), which mainly results from the DDoS detection time, the execution time of our resource orchestration module and the program loading time of the Tofino switch.

To demonstrate the performance of POSEIDON, we measure the end-to-end latency for workload traffic at the traffic generator, based on these six typical DDoS attacks, and compare it with an NSFOCUS ADS 4020 middlebox [57] and an NFV system similar to Bohatei [22]. As we can see from Fig. 15, for

[3]A closer look into our experiments also shows that the IP addresses of these real-world traces are not very dispersed; the switching ASICs has enough memory to support the *monitor* modules. We discuss further on this issue in Discussion(§IX).

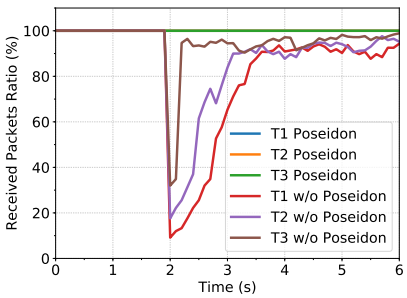


Fig. 11: Received packets before/after policy transition.

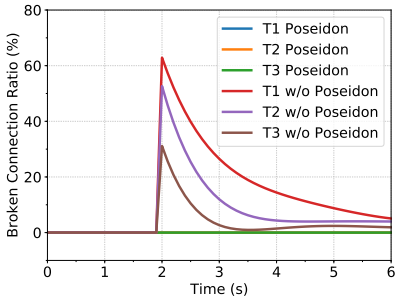


Fig. 12: Broken connections before/after policy transition.

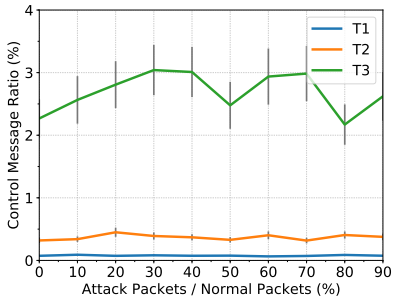


Fig. 13: Control traffic/workload traffic ratio.

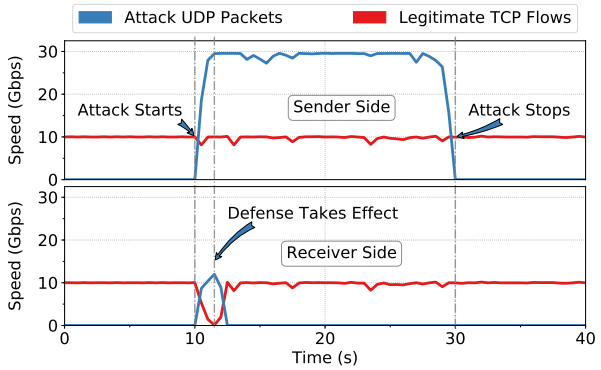


Fig. 14: Throughput restoration for legitimate flows.

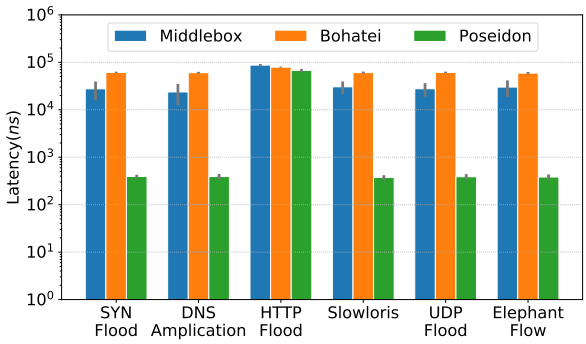


Fig. 15: Latency in traffic scrubbing center.

most types of attacks, POSEIDON reduces packet processing latency in scrubbing centers by two orders of magnitude compared with the middlebox and the NFV system. In particular, POSEIDON processes packets within hundreds of nanoseconds while the middlebox or the NFV system requires tens of microseconds. This demonstrates a significant performance improvement, which is crucial for the requirements of latency-sensitive services in today's datacenters. For HTTP flood, we use puzzle, which can only be implemented on the servers; so the latency benefit of POSEIDON is not as obvious and latency results are comparable. Since we do not have access to terabit-level traffic generators at this moment, we are not able to evaluate the throughput of our prototype using extreme pressure tests. However, in principle, POSEIDON can defend against ∼Tbps attack traffic with a small number of devices (including programmable switches and commodity servers). This is because a compiled P4 program on a programmable switch is guaranteed to run at terabit line rate; otherwise it would already be rejected by the compiler at the compilation stage [10], [78]. For terabit DDoS attacks, the other two approaches would require an extremely large number of devices. In contrast, POSEIDON can achieve this with much lower device count and much lower cost.

To show the cost reduction of POSEIDON compared with the other two solutions, TABLE I serves as a good starting point. As we can see, programmable switches can reduce the equipment cost by nearly two orders of magnitude, and reduce the power consumption by nearly three orders of magnitude. Although POSEIDON requires a small number of servers to assist programmable switches, the order of magnitude will not change drastically. Similar results also been obtained by a recent project that evaluates the power consumption of in-network computing [78], which shows that switching ASICs can reduce power consumption by 1000x compared with commodity CPU.

IX. DISCUSSION

Security of POSEIDON. POSEIDON shares a similar two-layer architecture as classic SDN, a control layer and an infrastructure layer. However, it is resilient to attack vectors in classic SDNs that target the control channel [71], [33], [91], [12], because it does not adopt the reactive event processing paradigm in OpenFlow-based SDN. However, there are still several potential vulnerabilities. First, attackers may use spoofed traffic to mislead *monitor* modules to invoke the wrong *action* modules, or overwhelm the stateful memory in switching ASICs (e.g., hash table in `count` and `aggr` primitives). This may further lead to statistical inaccuracy and unexpected hash collisions. Actually, spoofed IP traffic is a challenge that is not specific to POSEIDON; it also affects a number of other switch-based systems as listed in §X. We observe that a recent research effort, NetHCF [6], [44], tries to filter spoofed IP traffic with programmable switches, which can be a good starting point to prevent statistic pollution and reduce unnecessary false positive. It should be easy to integrate such mechanisms into POSEIDON. Without spoofed traffic, the *monitor* modules in POSEIDON (5∼10 MB SRAM per stage in the latest programmable switch) can guarantee a very low false positive (e.g., less than 1%) for a few million buckets. This could potentially accommodate millions of IPs and billions of packets per second. This has also been validated by several other recent research projects [49], [86], which have developed various sketches with programmable switches to

conduct network monitoring under terabit traffic. In addition, we can also leverage external DRAM in servers to alleviate the memory pressure [42] in programmable switches. This would make much more memory available for sketches and achieve much lower hash collision rates (or false positives).

Second, attackers may change the attack composition dynamically within seconds so that POSEIDON cannot respond in a timely manner. A potential solution is to further optimize the performance of our orchestration component using more powerful servers, and to leverage more advanced heuristics to solve the ILP problem. In addition, we can also use the switch-only primitives for long-lived attacks, and only involve the servers for short-lived attacks. In this way, defenses against short-lived attacks will not need to recompile the switch programs, avoiding the need to trigger frequent policy changes. We leave the detailed exploration of these security problems to our future work.

Extensibility of POSEIDON. POSEIDON has a set of modular primitives for monitoring, analysis, and attack response. Operators could easily develop more defense primitives in this framework. To integrate a new defense primitive into the existing defense library, operators should define the new primitive, analyze its implementation with respect to switching ASICs constraints, calculate its resource usage, and extend the defense library with this new primitive. Then, the new primitive could be loaded into our POSEIDON framework and used with other primitives. Note that although POSEIDON cannot handle zero-day DDoS attacks directly, the programmability and modularity properties of POSEIDON would accelerate the deployment of new defense mechanisms significantly. This benefit cannot be achieved with traditional proprietary middleboxes, even NFV-based defense systems.

Automation of POSEIDON. Current POSEIDON requires some human intervention for writing the defense policies. This can be further automated if there are no zero-day DDoS attacks. Operators can set a defense policy for each DDoS attack beforehand, and POSEIDON would load the corresponding policies when DDoS attacks are detected. Nevertheless, for zero-day DDoS attacks, human intervention is unavoidable. Operators need to analyze the characteristics of the new DDoS attacks, and may potentially need to extend the POSEIDON primitives with more defense strategies.

X. RELATED WORK

There is a long body of works on DDoS attacks and defenses, for which comprehensive surveys exist [90], [52]. Here, we briefly discuss the other most related topics.

SDN/NFV-based DDoS Defense. Some works have been devoted to defending against DDoS attacks with SDN/NFV from various perspectives. Bohatei [22] leverages NFV and SDN to achieve flexible and elastic DDoS defense. Xu et al. [85] propose an adaptive approach using limited switch TCAM to balance the coverage and granularity of DDoS detection. Afek et al. [1] propose to filter the spoofing traffic with OpenFlow switches. By contrast, POSEIDON proposes a cost-efficient and agile DDoS defense framework that leverages the new opportunities provided by programmable switches.

Programmable Switches. Researchers have looked at accelerating various applications in networking and distributed systems using programmable switches. Examples include layer-4 load balancing [51], [58], network resource allocation mechanisms [69], key-value stores [35], [48], coordination services [34], [46], [47], fair queueing [70], fast connectivity recovery [32], network monitoring and measurement tasks [73], [55], [74], [75], [30], [87]. These applications achieve far better performance with lower costs than their software counterparts that run on commodity servers. POSEIDON is inspired by these works, but focuses on a different problem, DDoS defense, and provides a systematic approach to integrating programmable switches into the current DDoS defense framework.

Policy Languages. There are many domain-specific languages in networking and security communities which aim to simplify policy expression, such as Chimera [8], NetCore/NetKAT [2], [65], [4], [25], [24], PSI [89]. Although our key idea of software-defined programmable security is not tied to any specific language, to hide underlying hardware complexity and reduce operator burden, we extend POSEIDON policy language based on Pyretic NetCore [53], and provide a high-level abstraction tailored for DDoS defenses.

XI. CONCLUSION

In this paper, we highlight the challenges for today's DDoS defense and identify new opportunities that programmable switches bring for mitigating volumetric DDoS attacks. We introduce POSEIDON, a performant, cost-efficient and agile DDoS defense system, which addresses the key limitations in today's DDoS defense. The POSEIDON language provides a simple, modular DDoS policy abstraction that can support a range of policies, shielding the low-level hardware complexity. The POSEIDON orchestration component provides an optimized, efficient resource orchestration mechanism to map the high-level policy primitives to the underlying hardware resources. The POSEIDON runtime manager provides a transparent, effective scheme to adapt to the attack dynamics while achieving resource utilization efficiency and guaranteeing correctness for legitimate flows. Our implementation and evaluation demonstrate that POSEIDON is highly effective in attack mitigation, and only incurs negligible overheads. These results show that POSEIDON is an effective system for mitigating modern advanced DDoS attacks.

ACKNOWLEDGMENT

We thank our shepherd, William Enck, and the anonymous NDSS reviewers for their valuable comments. We would also like to thank Ennan Zhai and Yifei Yuan for their feedback on the policy language, and Douglas Everson from Clemson University for proofreading this paper. We also thank Xiao Kong, Yi Qiao, Jiasong Bai, Zili Meng and Haixin Duan from Tsinghua University, Lei Xu from Texas A&M University and Palo Alto Networks, Kai Gao from Sichuan University, Zhaogeng Li for Baidu Inc. for joining some discussions on this paper. Menghao, Guanyu, Shicheng and Chang are also sincerely grateful for their former Ph.D. advisor, Jun Bi from Tsinghua University, for his strong support. This work is supported in part by the National Key R&D Program of China (2017YFB0801701), the National Science Foundation of China (No. 61625203, No.61872426, No. 61832013, No. 61572278, U1736209), the BNRist Network and Software Security Research Program (No. BNR2019TD01004), ONR grant

N00014-20-1-2734, and the US National Science Foundation (No. 1617985, No. 1642129, No. 1700544, No. 1740791, No. 1846291, No. 1700499, No. 1642143). Mingwei Xu and Qi Li are the corresponding authors.

REFERENCES

[1] Y. Afek, A. Bremler-Barr, and L. Shafir, "Network anti-spoofing with SDN data plane," in *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*. IEEE, 2017, pp. 1–9.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker, "Netkat: Semantic foundations for networks," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 113–126.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *USENIX Security Symposium*, 2017, pp. 1092–1110.

[4] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, "Snap: Stateful network-wide abstractions for packet processing," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 29–43.

[5] A. T. ARTICLES, "5 most famous DDoS attacks," https://www.a10networks.com/resources/articles/5-most-famous-ddos-attacks, 2018, [Online; accessed Aug. 19, 2019].

[6] J. Bai, J. Bi, M. Zhang, and G. Li, "Filtering spoofed ip traffic using switching asics," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. ACM, 2018, pp. 51–53.

[7] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. ACM, 2002, pp. 71–82.

[8] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis." in *USENIX Security Symposium*, 2012, pp. 365–379.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[10] 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," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 99–110.

[11] A. Business, "Distributed denial of service (ddos) defense," https://www.business.att.com/products/ddos-protection.html, 2018, [Online; accessed Aug. 13, 2019].

[12] 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 *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 19–36.

[13] O. Cetin, C. Ganán, L. Altena, T. Kasama, D. Inoue, K. Tamiya, Y. Tie, K. Yoshioka, and M. van Eeten, "Cleaning up the internet of evil things: Real-world evidence on isp and consumer efforts to remove mirai," 2019.

[14] Cisco, "Cisco guard xt 5650," https://www.cisco.com/c/en/us/products/collateral/security/guard-xt-5650a/product_data_sheet0900aecd800fa55e.html, 2018, [Online; accessed Jul. 8, 2019].

[15] A. Cloud, "Anti-ddos basic," https://www.alibabacloud.com/products/ddosdip, 2018, [Online; accessed Jul. 19, 2019].

[16] T. Cloud, "Dayu anti-ddos," https://intl.cloud.tencent.com/product/bad, 2018, [Online; accessed Aug. 19, 2019].

[17] Corero, "The evolution of ddos protection," http://info.corero.com/the-evolution-of-ddos-protection.html, 2018, [Online; accessed Jun. 19, 2018].

[18] 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.

[19] I. DPDK, "Learn how to get involved with dpdk," https://www.dpdk.org/, 2017, [Online; accessed Jun. 13, 2019].

[20] Engadget, "New dos attack exploits algorithms to knock sites offline," https://www.engadget.com/2019/08/09/new-ddos-attack-algorithms/, 2019, [Online; accessed Aug. 19, 2019].

[21] Epsylon, "Ufonet - denial of service toolkit," https://ufonet.03c8.net, 2019, [Online; accessed Aug. 19, 2019].

[22] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense." in *USENIX Security Symposium*, 2015, pp. 817–832.

[23] C. I. for Cybersecurity, "A realistic cyber defense dataset," https://www.unb.ca/cic/datasets/ids-2018.html, 2019, [Online; accessed Aug. 19, 2019].

[24] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[25] N. Foster, D. Kozen, K. Mamouras, M. Reitblatt, and A. Silva, "Probabilistic netkat," in *European Symposium on Programming Languages and Systems*. Springer, 2016, pp. 282–309.

[26] R. Gandhi, H. H. Liu, Y. C. Hu, G. Lu, J. Padhye, L. Yuan, and M. Zhang, "Duet: Cloud scale load balancing with hardware and software," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 27–38, 2015.

[27] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani, "Streaming algorithms for robust, real-time detection of ddos attacks," in *27th International Conference on Distributed Computing Systems (ICDCS'07)*. IEEE, 2007, pp. 4–4.

[28] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation." in *OSDI*, vol. 16, 2016, pp. 249–264.

[29] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 163–174.

[30] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 357–371.

[31] Gurobi, "The fastest mathematical programming solver," http://www.gurobi.com/, 2019, [Online; accessed Jun. 19, 2019].

[32] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 161–176.

[33] S. Jero, W. Koch, R. Skowyra, H. Okhravi, C. Nita-Rotaru, and D. Bigelow, "Identifier binding attacks and defenses in software-defined networks," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 415–432.

[34] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[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 *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 121–136.

[36] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches." in *NSDI*, 2015, pp. 103–115.

[37] A. Juels and J. G. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks." in *NDSS*, vol. 99, 1999, pp. 151–165.

[38] S. Kandula, D. Katabi, M. Jacob, and A. Berger, "Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 287–300.

[39] M. S. Kang, V. D. Gligor, and V. Sekar, "SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks." in *NDSS*, 2016.

[40] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 127–141.

[41] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for nfv: Simplifying middlebox modifications using statealyzr." in *NSDI*, 2016, pp. 239–253.

[42] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*. ACM, 2018, pp. 1–7.

[43] R. R. Kompella, S. Singh, and G. Varghese, "On scalable attack detection in the network," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*. ACM, 2004, pp. 187–200.

[44] G. Li, M. Zhang, C. Liu, X. Kong, A. Chen, G. Gu, and H. Duan, "Nethcf: Enabling line-rate and adaptive spoofed ip traffic filtering," in *ICNP*. IEEE, 2019.

[45] H. Li, H. Hu, G. Gu, G.-J. Ahn, and F. Zhang, "vNIDS: Towards elastic security with safe and efficient virtualization of network intrusion detection systems," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 17–34.

[46] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 104–120.

[47] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say no to paxos overhead: Replacing consensus with network ordering." in *OSDI*, 2016, pp. 467–483.

[48] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 795–809, 2017.

[49] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with univmon," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 101–114.

[50] A. Mahimkar, J. Dange, V. Shmatikov, H. M. Vin, and Y. Zhang, "dfence: Transparent network-based denial of service mitigation." in *NSDI*, vol. 7, 2007, pp. 327–340.

[51] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.

[52] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[53] C. Monsanto, J. Reich, N. Foster, J. Rexford, D. Walker *et al.*, "Composing software defined networks." in *NSDI*, vol. 13, 2013, pp. 1–13.

[54] S. Moss, "Major DDoS attack on dyn disrupts aws, twitter, spotify and more," http://www.datacenterdynamics.com/content-tracks/security-risk/major-ddos-attack-on-dyn-disrupts-aws-twitter-spotify-and-more/97176.fullarticle, 2016, [Online; accessed Jul. 15, 2018].

[55] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 85–98.

[56] B. Networks, "Tofino: World's fastest p4-programmable ethernet switch asics," https://barefootnetworks.com/products/brief-tofino/, 2017, [Online; accessed Jun. 13, 2019].

[57] NSFOCUS, "Nsfocus anti ddos solution," https://nsfocusglobal.com/wp-content/uploads/2018/05/Anti-DDoS-Solution.pdf, 2018, [Online; accessed Jul. 8, 2019].

[58] V. Olteanu, A. Agache, A. Voinescu, and C. Raiciu, "Stateless datacenter load-balancing with beamer," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, vol. 18, 2018, pp. 125–139.

[59] D. Pauli, "Chinese gambling site served near record-breaking complex DDoS," https://www.dailydot.com/debug/lizard-squad-hackers/, 2016, [Online; accessed Jul. 15, 2019].

[60] W. Project, "Mawi working group traffic archive," http://mawi.wide.ad.jp/mawi/, 2019, [Online; accessed Aug. 19, 2019].

[61] S. Ramanathan, J. Mirkovic, M. Yu, and Y. Zhang, "SENSS against volumetric DDoS attacks," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 266–277.

[62] R. Rasti, M. Murthy, N. Weaver, and V. Paxson, "Temporal lensing and its application in pulsing denial-of-service attacks," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 187–198.

[63] A. D. Rayome, "Ddos attacks increased 91% in 2017 thanks to iot," https://www.techrepublic.com/article/ddos-attacks-increased-91-in-2017-thanks-to-iot/, 2018, [Online; accessed Jul. 23, 2019].

[64] C. Rossow, "Amplification hell: Revisiting network protocols for DDoS abuse," in *NDSS*, 2014.

[65] C. Schlesinger, M. Greenberg, and D. Walker, "Concurrent NetCore: From policies to pipelines," in *ACM SIGPLAN Notices*, vol. 49, no. 9. ACM, 2014, pp. 11–24.

[66] A. Scroxton, "Dyn reveals details of complex and sophisticated iot botnet attack," https://www.computerweekly.com/news/450401857/Dyn-reveals-details-of-complex-and-sophisticated-IoT-botnet-attack, 2016, [Online; accessed Jul. 15, 2018].

[67] C. Security, "Ddos attacks 2018: New records and trends," https://www.calyptix.com/top-threats/ddos-attacks-2018-new-records-and-trends/, 2019, [Online; accessed Mar. 19, 2019].

[68] H. Shan, Q. Wang, and C. Pu, "Tail attacks on web applications," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 1725–1739.

[69] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the power of flexible packet processing for network resource allocation." in *NSDI*, 2017, pp. 67–82.

[70] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy, "Approximating fair queueing on reconfigurable switches," in *USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[71] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: Scalable and vigilant switch flow management in software-defined networks," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 413–424.

[72] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 2016, pp. 15–28.

[73] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*. ACM, 2017, pp. 164–176.

[74] J. Sonchack, A. J. Aviv, E. Keller, and J. M. Smith, "Turboflow: information rich flow record generation on commodity switches," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 11.

[75] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow," in *2018 USENIX Annual Technical Conference USENIX ATC 18)*. USENIX Association, 2018.

[76] T. Spring, "Mirai variant targets financial sector with iot ddos attacks," https://threatpost.com/mirai-variant-targets-financial-sector-with-iot-ddos-attacks/131056/, 2018, [Online; accessed Jul. 29, 2019].

[77] StressThem, "The next generation ip stresser," https://www.stressthem.to/, 2019, [Online; accessed Aug. 19, 2019].

[78] Y. Tokusashi, H. T. Dang, F. Pedone, R. Soulé, and N. Zilberman, "The case for in-network computing on demand," in *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 2019, p. 21.

[79] TS3Booter, "ts3booter.net," https://ts3booter.net/, 2019, [Online; accessed Aug. 19, 2019].

[80] W. Turton, "An interview with lizard squad, the hackers who took down xbox live," https://www.dailydot.com/debug/lizard-squad-hackers/, 2017, [Online; accessed Jul. 15, 2019].

[81] C. S. University, "Darpa 2009 intrusion detection dataset," http://www.darpa2009.netsec.colostate.edu/, 2019, [Online; accessed Aug. 19, 2019].

[82] X. Wang and M. K. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 78–92.

[83] S. Weagle, "The rise of iot botnet threats and ddos attacks," https://www.corero.com/blog/870-the-rise-of-iot-botnet-threats-and-ddos-attacks.html, 2018, [Online; accessed Jul. 30, 2019].

[84] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[85] Y. Xu and Y. Liu, "DDoS attack detection under SDN context," in *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*. IEEE, 2016, pp. 1–9.

[86] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 561–575.

[87] N. Yaseen, J. Sonchack, and V. Liu, "Synchronized network snapshots," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 2018, pp. 402–416.

[88] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch." in *NSDI*, vol. 13, 2013, pp. 29–42.

[89] T. Yu, S. K. Fayaz, M. P. Collins, V. Sekar, and S. Seshan, "PSI: Precise security instrumentation for enterprise networks." in *NDSS*, 2017.

[90] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks," *IEEE communications surveys & tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[91] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai, "Control plane reflection attacks in SDNs: new attacks and countermeasures," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 161–183.

[92] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion control for large-scale RDMA deployments," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 523–536.

APPENDIX

A. Defense Examples by POSEIDON language

Here are several extra defense examples in POSEIDON language.

SlowLoris attack. In SlowLoris attacks, TCP sessions are established between the bots and the victim server. After a session has been established, the bot does not respond with ACK packets, so this keeps the session open until timeout. Empty sessions occupy CPU and RAM resources, so the victim servers do not have enough resources to accept new requests. To mitigate this attack, a potential defense the operators could use is to count the number of bytes sent by each source IP and the number of TCP sessions established by each source IP (line 1, line 2). During the previous period, if the average number of bytes per connection is smaller than a threshold (T), then we can rate-limit the session establishment rate for this source IP (line 4, line 5, line 6). Otherwise, we allow its packets to pass (line 7, line 8).

```
packet_byte = aggr(True, [ip.src], 5)
connection_number = count(pkt.tcp.flag == SYN, [ip.src], 5)

if packet_byte([pkt.ip.src]) /
        connection_number([pkt.ip.src]) <= T:
    rlimit
else:
    pass
```

UDP flood attack. For UDP flood attacks, we track the number of UDP packets for each source IP every 5 seconds (line 1). During the previous period, if a source IP sends an anomalously high number of UDP packets, we mark it as attacks and rate-limit this source IP (line 3, line 4). Otherwise, we allow its packets to pass (line 5, line 6).

```
udp_counter = count(pkt.ip.protocol == UDP, [ip.src], 5)

if udp_counter([pkt.ip.src]) >= T:
    rlimit
else:
    pass
```

Elephant flow. To detect very large flows from some source IP addresses, we can first count the number of bytes each flow sends every 5 seconds (line 1, line 2). During the previous period, if the total bytes for a flow exceeds a threshold, we rate-limit this flow (line 4—6). Otherwise, we allow its packet to pass (line 7, line 8).

```
packet_byte_counter = aggr(True, [ip.src, ip.dst,
        ip.protocol, tcp.sport, tcp.dport], 5)

if packet_byte_count([ip.src, ip.dst, ip.protocol,
        tcp.sport, tcp.dport]) >= T:
    rlimit
else:
    pass
```

B. The P4 implementation of a sample POSEIDON policy

Here we include the P4 implementation for one sample POSEIDON policy, SYN flood defense, as shown in Fig. 4. The P4 snippet is simplified for clarity of presentation.

```
/* Header declaration */
struct headers {
    ether_t ether;
    ipv4_t  ipv4;
    tcp_t   tcp;
}
// Definitions of ether_t, ipv4_t and tcp_t are omitted

/* Metadata declaration */
header_type syn_proxy_meta_t {
    fields { ... }
}
metadata syn_proxy_meta_t meta;
// We remove the specific fields of metadata

/* Parser declaration */
parser parse_ether {
    extract(ether);
    return select(latest.etherType) {
        ETHERTYPE_IPV4: parse_ipv4;
        default: ingresss;
    }
}
parser parse_ipv4 {
    extract(ipv4);
    return select(latest.protocol) {
        IP_PROTOCOLS_TCP : parse_tcp;
        default: ingress;
    }
}
parser parse_tcp {
        extract(tcp);
        return ingress;
}
// Calculation of checksum is ignored

/* Monitor (counter) declaration */
register syn_count_cm_sketch_row1 {
    width : WIDTH;
    instance_count : COLUMN;
}
register syn_count_cm_sketch_row1_last_period {
    width : WIDTH;
    instance_count : COLUMN;
}
register ack_count_cm_sketch_row1 {
    width : WIDTH;
    instance_count : COLUMN;
}
register ack_count_cm_sketch_row1_last_period {
    width : WIDTH;
    instance_count : COLUMN;
}
// We omit the other rows of two count-min sketches

/* Match-Action Table declaration */
table syn_count_update_table {
    read {
        tcp.syn : exact;
    }
```

```
    actions {
        update_syn_count;
        _nop;
    }
}
action update_syn_count() {
    index_calculate(meta.syn_count_index, ipv4.srcAddr);
    syn_count_cm_sketch_add(meta.syn_count_index, 1);
}
// Definitions of other tables and actions are skipped

/* Control flow declaration */
control ingress {
    // Monitoring period
    apply(syn_count_update_table);
    apply(ack_count_update_table);
    apply(syn_count_read_table);
    apply(ack_count_read_table);
    // Packet processing period
    if (meta.ack_count_value == 0) {
        apply(drop_table);
    }
    else if (meta.syn_count_value == meta.ack_count_value){
        apply(forward_table);
    }
    else {
        apply(syn_proxy_table);
    }
}
// We ignore some details in the ingress control
// and the definition of egress control
```