On the Security Risks of Memory Adaptation and Augmentation in Data-plane DoS Mitigation

Hocheol Nam
KAIST
hcnam@kaist.ac.kr

Daehyun Lim
KAIST
dhlim1114@kaist.ac.kr

Huancheng Zhou
Texas A&M University
hczhou@tamu.edu

Guofei Gu
Texas A&M University
guofei@cse.tamu.edu

Min Suk Kang†
KAIST
minsukk@kaist.ac.kr

***Abstract*— Data-plane programmability in commodity switches is reshaping the landscape of denial-of-service (DoS) defense by enabling adaptive, line-rate mitigation strategies. Recent systems like Cerberus [1] augment limited switch memory with control-plane support to rapidly respond to evolving attacks. In this paper, we reveal a subtle yet critical vulnerability in this model; that is, the very mechanisms that enable the defense system's agility and scalability can be subverted by a new class of coordinated DoS attacks. We present HERACLES, the first attack to exploit hardware-level constraints in programmable switches to orchestrate precise resource contention across data-plane and control-plane memory. By leveraging side-channel timing signals, HERACLES triggers synchronized augmentation, memory squeezing, and time-window exploitation, which are three orthogonal contention strategies that significantly degrade or even completely disable the DoS mitigation capabilities. We implement and test HERACLES against real Tofino hardware and show that it can reliably disrupt DoS defenses across diverse DoS attack profiles, even when using loosely (1–2 second) time-synchronized attack sources. To mitigate this threat, we propose SHIELD, a multi-layered DoS mitigation sketch architecture that decouples memory operations across control- and data-plane layers, effectively mitigating the HERACLES attack while preserving both line-rate performance and detection accuracy.**

I. INTRODUCTION

Denial-of-Service (DoS) attacks remain persistent and evolving threat to Internet services and networks, with adversaries gaining significant advantages in scale and sophistication [2], [3]. By leveraging large botnets, for example, adversaries can rapidly scale up traffic volume to exhaust target network resources [4], [5], [6]. A core adversarial advantage lies in their adaptability; namely, a DoS coordinator can continuously monitor the effectiveness of an attack profile (e.g., attack types, volume, and traffic patterns) and promptly shift to a different profile to evade mitigation in response to observed mitigation. This *asymmetry* presents a long-standing challenge in the DoS landscape, as data-plane components in routers and servers lack the responsiveness required to counter such agile and reactive attack strategies.

The imbalance has recently begun to shift with advent of data-plane programmability in commodity switches. In particular, Cerberus [1] has demonstrated that a switch can swiftly adapt its DoS mitigation logic (i.e., how to assess the severity of a specific DoS attack profile) *entirely* within the data plane, thereby enabling line-rate response and mitigation against rapidly changing attack profiles. Furthermore, Cerberus augments the switch's limited memory by systematically offloading the high-order bits in its counting schemes to the control plane whenever possible. This first-ever line-rate adaptive DoS mitigation marks a significant step towards leveling the long-standing *asymmetry* between adversaries and defenders in the DoS landscape.

However, in this paper, we argue and demonstrate that the subtle limitations stemming from hardware-level constraints of programmable switches open up a new opportunity for a new class of DoS attacks that exploit the very flexibility and augmentation of memory resources celebrated in Cerberus. In particular, by leveraging the rigid hardware-level constraints (e.g., limited register access, strict staged pipeline) of programmable switches, we show that a remote adversary can infer the *precise timing* of the switch's memory augmentation, resizing, and flushing operations. This knowledge allows the adversary to orchestrate these operations in a manner that induces significant *contention* both among memory resources within the data plane and over the bandwidth between the data plane and control plane. The resulting contention on these critical resources can be exploited to disrupt the switch's overall DoS mitigation functionality, leading to a near-complete failure in detecting and mitigating DoS attacks.

We illustrate this threat through our HERACLES[1] attack, which strategically maximizes contention between in-data-plane memory and cross-plane (i.e., data-to-control plane) memory resources, ultimately causing significant degradation in Cerberus's DoS mitigation performance. The main attack strategies are illustrated in Figure 1. By carefully preplanning attack profiles, HERACLES orchestrates three orthogonal types

† Corresponding author

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.241857
www.ndss-symposium.org

[1] HERACLES: **H**ybrid **E**xploitation against **R**esource-sharing and **A**daptive **C**o-monitoring for **L**imited-resource **E**nvironment of Programmable **S**witches. In Greek mythology, Heracles defeated Cerberus.

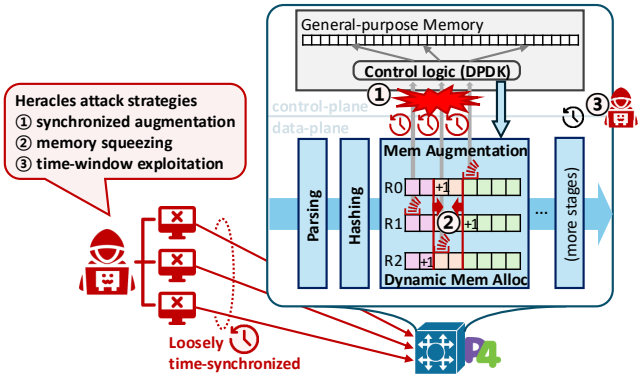


Fig. 1. The overview of the HERACLES attack.

of memory resource contention: (1) *Synchronized augmentation* triggers multiple memory offloads to the control plane (nearly) at the same time, overwhelming it with carry-bit messages that exceed its processing capacity and network bandwidth (see §III-B); (2) *Memory squeezing* manipulates the adaptive memory allocation logic by expanding in-data-plane memory for non-target tasks, thereby starving the memory available to specific DoS mitigation tasks at critical adversary-chosen moments (see §III-C); and (3) *Time-window exploitation* takes advantage of the periodic state-refresh behavior in the heavily packed data-plane memory, injecting traffic at precise times to degrade detection accuracy during refresh intervals (see §III-D).

We use Tofino switches [7] to evaluate the practicality of all three types of resource contention exploited in the HERACLES attack. We first leverage a set of new side-channel information leaks arising from the inflexible hardware-level constraints of Tofino switches and empirically demonstrate that accurate inference of the switch's timing of memory augmentation, adaptation, and flushing operations is indeed highly practical with loosely-synchronized (1–2 seconds) attack sources; see §III-A. Our evaluation of the HERACLES attack using four different DoS vectors (i.e., SYN flood, ICMP flood, DNS flood, UDP flood) demonstrates that a DoS coordinator, capable of controlling various loosely-synchronized attack sources (e.g., botnets), can successfully trigger all three forms of resource contention, ultimately rendering Cerberus's DoS mitigation logic almost completely ineffective. For example, HERACLES significantly increases the false negative rate of Cerberus's DoS mitigation logic, e.g., 78% with synchronized augmentation, 50% with memory squeezing, and 100% with time-window exploitation, which means that the vast majority of DoS attack traffic would bypass the DoS mitigation and reach its target servers.

To address this new class of detection bypass attacks, we devise a new data-plane DoS mitigation scheme that systematically *disassociates* the operations of the in-data-plane and control-plane memory resource adaptation and *minimizes* the data-to-control plane bandwidth usage. Our multi-layered sketch design, called SHIELD, decouples the timing of resource augmentation, adaptation, and flushing across distinct levels of the switch's data-plane and control-plane memory, degrading the performance of the timing information inference attack. Also, SHIELD significantly reduces the data-to-control plane bandwidth usage by carrying over bits across multiple layers of sketches within the data plane, rather than directly offloading them to the control plane. By significantly reducing the adversary's capability to learn the timing of memory adaptation in DoS mitigation and congest the data-to-control plane bandwidth with carry-bit messages, we effectively mitigate the HERACLES attack while preserving both the line-rate performance and the adaptive nature of the DoS mitigation logic.

We summarize our contributions as follows:

- We present HERACLES, the first coordinated attack that exploits flexible memory augmentation mechanisms in data-plane-programmable switches to induce three orthogonal types of memory resource contention, ultimately leading to detection bypass and thus DoS mitigation failure.
- We evaluate HERACLES on a real Tofino switch across 4 diverse DoS profiles, showing that an adversary with loosely synchronized (1–2 seconds) traffic sources can reliably trigger all three contention types and significantly degrade Cerberus's detection accuracy.
- We propose SHIELD, a multi-layered mitigation scheme that successfully mitigates the HERACLES attack while preserving line-rate performance and adaptive detection accuracy.

II. ON STATE-OF-THE-ART PROGRAMMABLE DATA-PLANE DOS MITIGATION

In this section, we first provide background on programmable data planes and several constraints in commodity programmable switches. Then, we introduce a state-of-the-art resource-sharing data plane application that lays the foundation for understanding the rest of the paper.

A. A Primer on Programmable Data Plane

Emerging data plane programmability, empowered by programmable switch ASICs and network programming language (i.e., P4 [8]), has introduced high-performance in-network network engineering terabit-scale *line-rate* packet processing capabilities in the switch pipeline. Such programmability enables network engineers to implement advanced features such as custom protocols, real-time monitoring, telemetry, and measurement [9], [10], [11], as well as enhanced security mechanisms [12], [13], [14], [1], [15] directly using programmable switches.

Resource constraints of programmable switches. While programmable switch ASICs present a promising alternative to traditional Software-Defined Networking (SDN) solutions, with their powerful performance, they also come with inherent and unavoidable limitations due to hardware-level constraints. These switches face strict resource limitations, such as a fixed number of stages and pipelines, limited memory capacity (e.g., SRAM, TCAM), and restricted computation resources (e.g., SALUs). For example, Intel Tofino [7], [16], a commodity programmable switch ASIC, follows the Tofino Native Architecture (TNA), which is a variant of Protocol Independent

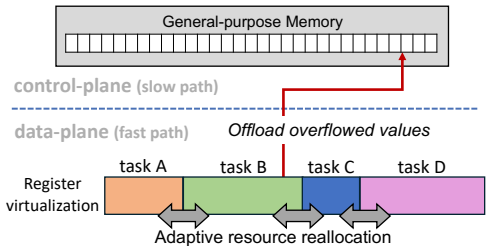


Fig. 2. Simplified overview of resource-sharing data-plane application design.

Switch Architecture (PISA) [17]. In PISA (including TNA), for each stage in the pipeline, developers can utilize a limited number of match-action tables and operations because resources are independently allocated for each stage. For each stage in the pipeline, developers can utilize a limited number of match-action tables and operations because the resources are independently allocated for each stage.

Programming data-plane applications under constraints. To overcome constraints of programmable switches, several sketch data structures are used in programmable switches while conducting lossy yet precise measurements [18], [19]. Sketches typically adopt fixed-size data structures such as counters or hash tables to extract the main characteristics of the traffic (e.g., packet count, heavy hitter, etc). For example, count-min sketch (CMS) and Bloom filter (BF) are widely adopted for counting and set membership, respectively. These approximate data structures become less accurate over time due to collisions and other noises. Therefore, they typically reset sketches at every pre-determined time interval [20], [21], [22], [23], [24], [14], [25], [1], [15].

B. Flexible Resource Sharing in Programmable Switches

To address the inherent resource constraints of programmable switches, several recent academic work have focused on *flexible resource sharing and augmentation* techniques. For instance, P4Visor [26] proposes a virtualization framework that enables the modular composition and simultaneous execution of multiple P4 programs through automated code merging and efficient resource sharing. NetVRM [27] introduces a virtual register memory abstraction that supports dynamic memory allocation across concurrent applications. FlyMon [11] also adopts a SRAM resource sharing mechanism with an address translation technique while providing runtime reconfigurable with a predefined unit without any downtime. Cerberus [1] applies resource-sharing techniques to in-network security tasks, specifically targeting the challenges of handling diverse, high-volume, and dynamic DoS attacks. To make efficient use of limited resources, data-plane resources are shared with other tasks. To facilitate a better understanding of the subsequent paper, we next provide a more detailed explanation of Cerberus's adaptive resource-sharing design.

Adaptive memory slicing and cross-layer augmentation. To support concurrent in-network security tasks under limited switch resources, Cerberus uses a *memory slicing* mechanism as illustrated in Figure 2. This technique allows multiple tasks to *share* a single register and ALU by concatenating multiple

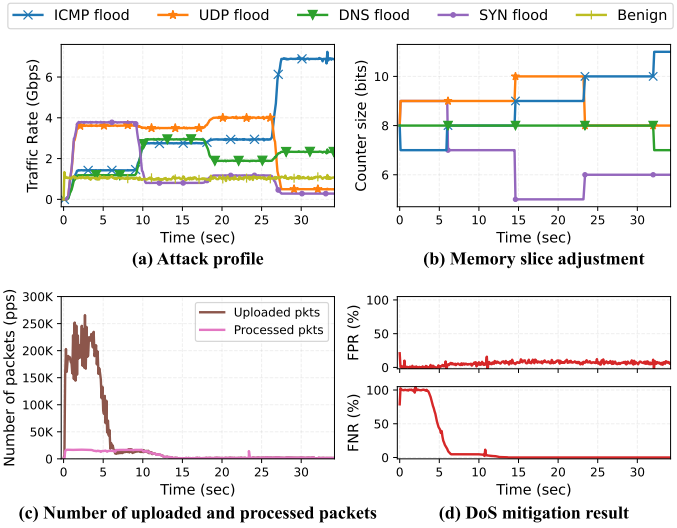


Fig. 3. Cerberus++'s defense performance against hybrid and dynamic DDoS attack, where attack profiles are randomly changed every 8 seconds.

values from different tasks. One downside of memory slicing is that it reduces the size of the counter available for each task, which can lead to frequent overflows. To handle high-volume traffic without exhausting on-chip memory, Cerberus uses an *offloading-based memory augmentation* mechanism, which is referred to as co-monitoring in [1]. Cerberus keeps the least significant bits in the data plane and offloads the most significant bits, which change infrequently, to the control plane. This also *dynamically adjusts* slice length based on the observed traffic, allocating more slices to tasks that frequently upload packets to the control plane.

Adaptability to dynamic attacks. To test the claimed adaptability of Cerberus, we conducted an experiment using a dynamic attack profile, totaling 10 Gbps, where the adversary randomly changes the attack profile every 8 seconds, as shown in Figure 3(a). Note that for fair comparison with all other experiments in this paper, we tested Cerberus++, which is an improved version of the open-source Cerberus with optimized control-plane performance; see §III for more details about the improvements. The data-plane memory slice inside Cerberus++ is read according to this attack profile as shown in Figure 3(b). As shown in Figure 3(c), the control plane of Cerberus++ temporarily receives a large number of packets (maximum 250K packets per second); nevertheless, it can effectively adapt to the changing attack profile and achieve near-0% of false negative rate within 8 seconds, as shown in Figure 3(d), confirming the effectiveness of the adaptive memory slicing and augmentation mechanisms.

III. The Heracles Attack

We present the Heracles attack and demonstrate its effectiveness against the state-of-the-art resource-sharing-based DoS defense system, Cerberus [1], in this section. Before we delve into the detailed attack strategies, we discuss the adversary model and the experiment setting we consider in this paper.

Adversary model. The adversaries we consider in this paper aim to *bypass* the DoS defense system deployed in a target network. The network that deploys the data-plane DoS defenses shares resources among multiple in-network monitoring and defense tasks, as done in Cerberus [1].

We assume the target network operates one or a small number of programmable switches for in-network DoS mitigation. This deployment model aligns with the typical architecture of programmable-switch-based defenses proposed in prior work [12], [28], [13], [24], [29], [14], [1]. These switches are strategically placed at logical chokepoints, such as in-house scrubbing centers [30], where suspicious, high-volume traffic is redirected for inspection and filtering [31]. The adversary generates DoS traffic aimed at final DoS targets (e.g., financial, gaming, or e-commerce platforms) *through* the target network, expecting that the traffic will traverse the network's programmable switches en route to the victim servers.

In some deployments, the target network may distribute DoS traffic across multiple switches through simple load balancing mechanisms, such as ingress routers redirecting traffic at line rate [30], [32]. While our experiments focus on a single programmable switch, the attacks and mitigation strategies we present naturally extend to multi-switch environments where each switch operates independently to mitigate load-balanced DoS traffic.[2]

We also assume that the adversary controls large-scale botnets that are geographically distributed across the globe, as is common in real-world botnets [4], [33], [34]. Also, we assume that the adversary can coordinate these botnets in a loosely synchronized (e.g., 1-2 seconds) manner to generate attack traffic. The same assumption has been widely adopted in prior academic works on pulsing DoS attacks and defenses [35], [36], [37], [38], [39], [40], [41], [29], [42], [43] and is supported by real-world botnet campaigns [44], [45].

Experiment setting. For all experiments, we deploy the open-sourced implementation of Cerberus [46] on the Tofino programmable switch (Edgercore Wedge-100BF-32X [47]). We develop our own implementation of the control-plane application on the switch because the open-sourced Cerberus only provides a Python version of the control plane that has poor packet processing performance, which is not suitable for our DoS attack evaluation. Specifically, for more reliable and fair evaluation, we implemented a *DPDK* [48] version of Cerberus control plane and name this improved version as *Cerberus++*. Thresholds and detection policy for each DDoS mitigation tasks (ICMP/SYN/DNS/UDP flooding) are borrowed from other work [12], [24], [13], [1], and adjusted to achieve about 1% false positive rate (FPR) with benign traffic [49]. Detailed parameters for the experiment can be found in [50].

In our experiment, we used 2 GB size of huge pages, 8,192 of `RX_RING_SIZE`, 32,768 of `NUM_MBUFS`, and 128 of `BURST_SIZE` for DPDK. Also, Intel P4 Studio [51] and the bf-sde (version 9.13.1 or higher) must be installed on the switch or can be simulated using the Tofino-model with Open P4 Studio [52]. The switch (or control plane server) operates with Ubuntu 22.04. We used the Cisco TRex traffic generator [53] via NVIDIA Mellanox ConnectX-4 100 Gbps NICs that are connected to the Wedge 100BF-32X programmable switch, to generate malicious traffic and send benign traffic (i.e., CAIDA Internet trace dataset [49]) to the switch. To emulate realistic loose synchronization among attack sources, we added zero-mean Gaussian noise with a standard deviation of 0.33 seconds to all generated traffic, while bounding within ± 1 second to prevent extreme deviations. Our code for the HERACLES attack is publicly available at [50], although we note that it, like many other programmable switch-based defense applications [1], [14], [13], [24], relies on specific hardware and software configurations.

[2]To the best of our knowledge, no coordinated, scale-out solution yet exists for handling DoS traffic volumes that exceed the capacity of a single programmable switch.

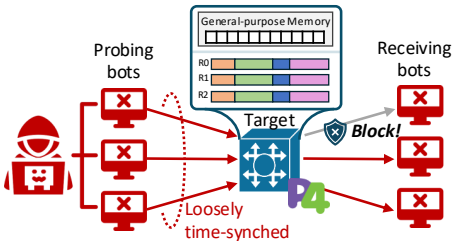


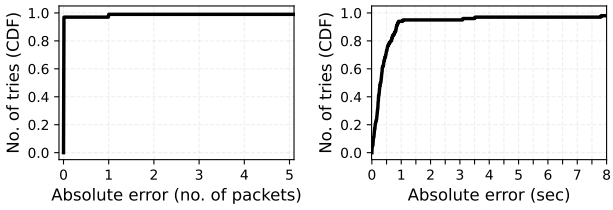
Fig. 4. Overview of the HERACLES's probing process.

In the following, we first show empirically how the adversary infers the internal switch parameters (including precise timing information) of Cerberus that are necessary for the HERACLES attack (§III-A). Then, we explain three specific strategies of the HERACLES attack: (1) Synchronizing updates in data-plane counters to overflow carry-bit messages to control-plane counters (§III-B), (2) Squeezing counter-bits allocated for a certain attack vector (§III-C), and (3) Flooding with new attack profile(s) immediately after the counter resets (§III-D), demonstrating their effectiveness in each separate section.

A. Attack Preparation: Inferring Switch Internal Parameters

For the HERACLES attack to be successfully mounted, one crucial required step that has to be taken first is to infer several internal parameters of a target switch. This inference step is critical because the inferred parameters are necessary to trigger memory resource contention and eventually cause resource exhaustion.

The inference begins with the probing process (see Figure 4), where the adversary uses a set of probing bots and receiving bots to measure the internal parameters of the target switch. *Probing bots* are controlled by the adversary to send probe packets through the target switch. *Receiving bots* are also controlled by the adversary to receive probe packets from the probing bots and count the number of packets they receive from the probing bots. This probing phase requires only a small number of bot pairs whose communication paths include the probed switch. By using a small number of probing bots (or even a single probing bot), an adversary can effectively



(a) Accuracy of threshold probing **(b) Accuracy of time window inference**

Fig. 5. Threshold probing and time-window inference accuracy. We plot the cumulative distribution of absolute errors over 100 trials each.

emulate a large number of logical sources with their IP addresses spoofed. Moreover, the probing does not risk exposing the adversary's botnet IP addresses to the target switch (thus avoid being blacklisted by IP-based filtering) because they can use spoofed source IP addresses in the probe packets.

At a high level, as shown in Figure 4, the HERACLES adversary needs to infer three parameters: (1) the filtering threshold of each task, (2) the time window of each task, and (3) the task combinations that share the same register. In this probing process, the probing bots can send probe packets that traverse the target switch at a varying, adversary-controlled rate, time, and duration. The receiving bots count the incoming packets to provide feedback on whether some packets are blocked or not. Guided by the feedback, the HERACLES adversary further adjusts the attack rate, time, and duration for high inference accuracy.

1) Inferring Detection Threshold: Knowledge of the filtering threshold for DoS mitigation is critical to our attack strategy. Specifically, it (i) allows the adversary to avoid triggering DoS defenses while still sending significant volumes of attack traffic, and (ii) enables inference of the monitoring time window and register-sharing tasks, as we elaborate on later in this section.

This is because thresholding-based filtering is typically implemented using match-action tables in the data plane, where packets exceeding the threshold are immediately dropped or rate-limited. Thus, the switch provides a direct and observable response to threshold violations. To infer the threshold, probing bots send controlled bursts of packets corresponding to a specific detection task within a short time interval. By monitoring when packets begin to get blocked or rate-limited, the attacker can approximate the threshold for that task.

Figure 5(a) shows how the proposed threshold inference is effective in our realistic evaluation setup. We set new, randomly-assigned thresholds at the target switch for each probing process and then test how our probing process can accurately measure the thresholds. We conduct 100 probing processes, each with a different threshold. We apply Gaussian noise (with standard deviation of $1/3$ seconds) to the latency of the probing packets to simulate realistic network conditions. The results show that the probing process can accurately measure the thresholds with an absolute error of 0 in 97 out of 100 probing processes. This confirms that our threshold inference process is capable of accurate threshold measurement with tolerable absolute error.

2) Inferring Windows Timing: To infer the time window, we leverage how timestamps are utilized to implement *sliding windows* in the Tofino Native Architecture (TNA) [54]. Tofino provides a 48-bit hardware timestamp, but due to the lack of native support for 48-bit arithmetic operations, developers typically use only a portion of this timestamp with slicing [55], [56], [57]. In particular, sliding window mechanisms are realized via match-action tables that detect changes in specific bit slices of the [1]. For instance, in a 48-bit timestamp `tstamp`, the single-bit slice `tstamp`[33:33] flips approximately every 8.59 seconds, while `tstamp`[35:35] flips every 34.36 seconds. This behavior allows us to infer the underlying windowing logic: by observing which bit transitions are used in the match-action table, we can deduce the periodicity of the time window. Even when the probing traffic is affected by timing noise, such as jitter introduced by the network, this bit-level predictability enables robust inference of the time window.

Specifically, we aim to recover the following three parameters: (1) the *period* of the time window, (2) the *start time* of the window, and (3) the *total number* of windows. Let T_n denote the threshold and P_n the period of the time window associated with task n. The threshold T_n is assumed to be known from prior analysis (§III-A1), leaving the estimation of P_n and the window alignment as our main focus.

To infer the *period* of the time window, the sender in a probing pair transmits packets at varying packet-per-second (PPS) rates and observes whether any of them are dropped. The key idea is to manipulate the traffic rate so that, depending on the size of the time window, the aggregate number of packets within that window either exceeds or stays below the blocking threshold. When the PPS is high, the number of packets sent within a time window can exceed the threshold T_n, triggering packet drops. This occurs when the inequality $P_n \times \text{PPS}_{\text{high}} \geq T_n$ holds. Conversely, when the PPS is low, the number of packets in any window remains below the threshold (i.e., $P_n \times \text{PPS}_{\text{low}} < T_n$), and all packets are accepted. To identify such a threshold-crossing point, the adversary can instruct a botnet to perform an *exponential search* over increasing PPS values to find a PPS_{high} that causes some packets to be blocked. Once this value is found, a *binary search* is used to refine the bounds between PPS_{low} (no packets blocked) and PPS_{high} (some packets blocked), thereby narrowing down the possible values of P_n. When the attacker finds such a pair where PPS_{high} results in blocked packets but PPS_{low} does not, it implies that the period of the time window must satisfy:

$$\frac{T_n}{\text{PPS}_{\text{high}}} \leq P_n < \frac{T_n}{\text{PPS}_{\text{low}}}. \tag{1}$$

This inequality provides a tight bound on P_n, enabling the attacker to accurately estimate the time window period using only external probing traffic.

Next, to determine the *start time* of the time window and the *total number* of windows, the sender in the probing pair transmits packets at a rate of PPS_{high} for a duration of $T_n/\text{PPS}_{\text{low}}$ seconds, while varying the start time of transmission. The key

observation is that if the transmission begins at the true start of a time window, the number of packets sent within that window will exceed the threshold T_n, resulting in some packets being blocked. Conversely, if the transmission does *not* align with the actual window start, all packets will fall below the threshold within each window, and no blocking will occur. With this insight, the off-path attacker can instruct botnets to try at different starting times and find the start of the time window. By repeating this procedure and identifying multiple time window start points, the attacker can then compute the number of time windows. This is achieved by analyzing the intervals between successive inferred window starts and dividing the total observation period by the estimated period P_n.

Figure 5(b) shows the effectiveness of windows timing inference. We set a new randomly configured time window setting (period, number, start time) for each probing process and conduct 100 probing processes, each with a different time window period, number, and start time. With the experiment, we confirmed that 96 out of 100 of our probing processes correctly measure the period and number of the time window. In the case of inferring the start time of the time window, our probing process can accurately measure the start time with an absolute error of less than 1 second for all cases where the period and number of the time window are correctly inferred (i.e., 96 out of 100). These results show that our window timing inference process is capable of accurate measurement with tolerable absolute error to launch the HERACLES attack.

3) Inferring Register-sharing Tasks: Last, to infer which tasks share the same register, we leverage a key constraint of the P4 programming model [58], [54]. In the Tofino Native Architecture, line-rate performance is maintained by allowing each packet to access a given register *only* once as it passes through the pipeline. As a result, tasks that share a single register must coordinate their access; that is, they must update the register simultaneously and use a common key to do so efficiently.

This constraint implies that co-located tasks also share the same time window slot. If these tasks were to use different keys, a packet would need to access the same register multiple times, which violates the single-access constraint. An alternative would be to recirculate the packet through the pipeline multiple times, each time targeting a different key or update action, but this approach treats each recirculation as a new packet, significantly degrading throughput.

Avoiding such performance penalties would require additional register resources and significantly more complex pipeline logic, which defeats the original purpose of resource sharing (i.e., to conserve limited hardware resources). As a result, developers who implement register sharing typically avoid complexity and extra resource allocation, reinforcing the likelihood that co-located tasks share both register and time window semantics.

Based on the observation and insight that register-sharing tasks must use the same key for access, we infer which tasks are mapped to the same register. In Cerberus, measurement tasks rely on key-based data structures such as count-min sketches and Bloom filters. These structures use hashed representations of packet tuples, typically 2-tuples (e.g., source/destination IP addresses), as their access keys (or flow keys). The use of specific key-feature pairs for DoS detection and mitigation has been extensively studied over the past decades [12], [13], [1]. As a result, the tuple formats that are effective for defending against particular attack types have become well established and widely adopted. For instance, ICMP and UDP flood attacks are typically tracked using 2-tuple keys, while more complex reflection-based attacks, such as DNS amplification and NTP amplification, require finer-grained identification via 4- or 5-tuple keys to uniquely distinguish individual flows.

By analyzing the key associated with each task, we can cluster tasks into candidate groups that are likely to share the same register. Furthermore, since the time window for each task has already been inferred in the previous step, we can refine these groupings by identifying tasks that not only use the same key but also operate within the same time window. This cross-referencing allows us to narrow down the set of plausible co-location combinations.

B. Attack Strategy-I: Synchronized Augmentation

With the measured timing information, we demonstrate the first attack that exploits the memory resource augmentation mechanism (called co-monitoring) of Cerberus [1]. Specifically, in this attack, the adversary sends multiple attack patterns in the form of packets with different flow keys at a *synchronized rate*, so that the counters of each task in the sketch overflow together nearly at the same time. As a result, the control plane receives a large number of packets at once, overwhelming its processing capacity and leading to *undercounting*.

Let us explain why this rather simple attack strategy is effective. In sketch-based counting, each packet's flow ID, derived by hashing tuples like (src_ip, dst_ip), is used as a key to store measurement values in the sketch. Thus, when packets arrive uniformly from diverse sources, each updates its own counter based on a distinct flow key. Since all flows are subject to the same maximum counter capacity, uniformly increasing counters will reach the counter limit and trigger overflow simultaneously. In Cerberus [1], such overflows trigger packet mirroring to the control plane, causing a burst of mirrored packets and forcing the control-plane application to process a large number of traffic at once. Consequently, the control plane (either residing on the switch or an external server) receives excessive packets at once. Unfortunately, handling near millions of packets per second can be challenging [59].

The overflow packets in the data-to-control plane link causes not only some packet loss but critically also undercounting in the control plane. In DoS defense systems that augment memory resources, such as Cerberus [1], high-order bits of the packet count are mirrored to the control plane when a counter overflows. Thus, a single packet drop in the control plane is equivalent to missing multiple packet counts in the data plane, leading to significant undercounting.

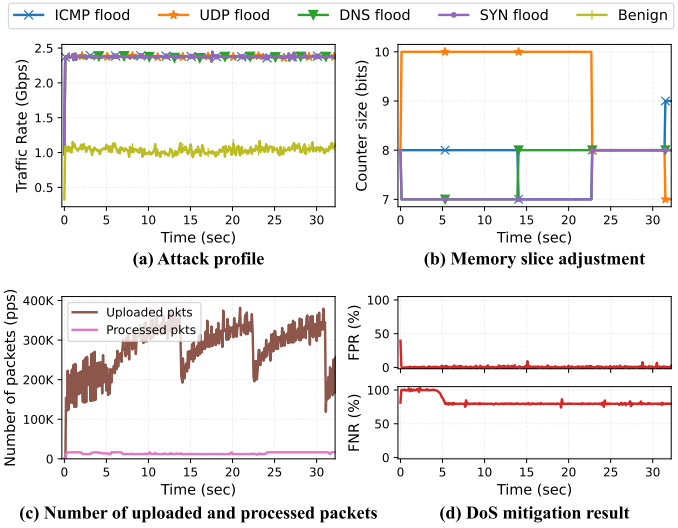


Fig. 6. Synchronized augmentation attack against Cerberus++.

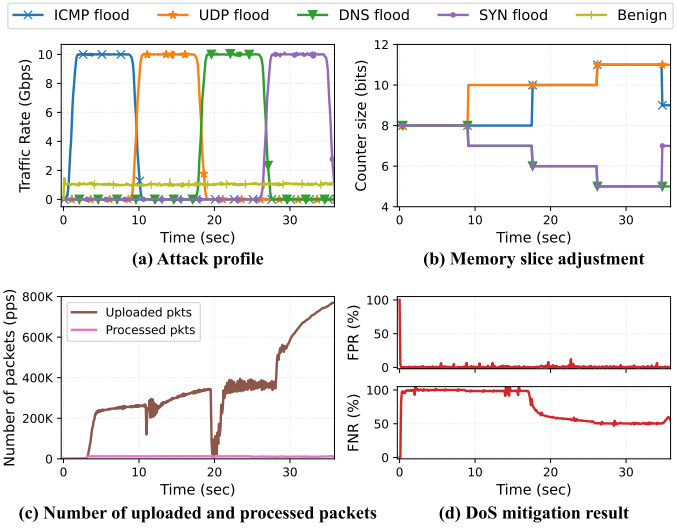


Fig. 7. Memory squeezing attack against Cerberus++.

Figure 6 presents the results of a synchronized augmentation attack against Cerberus++, which is our improved implementation of Cerberus [1]; see §III. The attack profile consists of uniformly distributed traffic, with four distinct attack vectors each transmitted at a constant rate of 2.5 Gbps, as shown in Figure 6(a). This simulates a botnet of approximately 10,000 IPs, where each bot sends low-volume packets (e.g., header-only) at a fixed rate, maintaining a constant inter-packet delay.

We consider a set of loosely synchronized botnets (see our assumption in §III), where the adversary synchronizes these flows by measuring round-trip times (e.g., via ping) to the router [35]. Since all attack profiles transmit at identical rates, Cerberus++'s resource-sharing mechanism attempts to evenly allocate memory slices. Cerberus fails to keep memory slices in the same ratio due to the control-plane failure to process all packets, as illustrated in Figure 6(b).

Figure 6(c) illustrates the packet processing performance of the control plane under a synchronized augmentation attack. Unlike the naive DoS attack demonstrated in Figure 3(c), this attack successfully maintains a persistent large overflow. The control plane of the DoS defense fails to handle all incoming packets from the data plane due to a combination of factors: the mismatch between theoretical bandwidth specifications and actual processing capabilities [59], [60], and a bottleneck caused by a lock during atomic updates to the data structures in the control plane. Under the synchronized attack profile, the packet rate forwarded from the data plane to the control plane surges to approximately 380K packets per second (pps) while the control plane processes forwarded packets from the data plane only about 15K pps on average, as shown in Figure 6(c).

This causes the control plane to handle only a fraction (only about 4%) of the packets received, which causes a serious *undercounting* problem in the control-plane memory. As we already mentioned, since overflowed messages carry *high-order* bits of information in the data plane, a single message drop in the control plane is actually equivalent to missing multiple packet counts in resource augmentation design. Such significant undercounting and processing delays due to the bottlenecks eventually disrupt threshold-based defense systems. This results in about 78% of the attack traffic being forwarded to the outsourcing server (see FNR in Figure 6(d)), as simultaneous overflows occur in counter buckets hit by multiple IP pairs at the same time.

C. Attack Strategy-II: Memory Squeezing

The memory squeezing attack strategy exploits the flexibility of Cerberus's adaptive memory-slicing mechanism. With this attack strategy, the adversary artificially *inflates the memory demands* of all tasks except the intended target task(s), causing them to appear memory-starved. These tasks then request additional memory from the control plane, triggering resource reallocation by reducing the memory assigned to the target task, which has not shown signs of starvation. Once the target task's memory is (inferred to be) sufficiently reduced, the adversary enters an attack phase, triggering severe overflows by overwhelming the reduced memory slice of the target task.

Figure 7 presents the results of the memory squeezing attack. As shown in Figure 7(a), the adversary changes different attack profiles every 8.59 seconds, strategically sending non-target attack traffic to manipulate Cerberus++'s adaptive memory slice mechanism. This attack profile is designed to reduce the memory slices allocated to two specific target tasks, DNS and SYN flood detection by inflating the memory demands of other co-located tasks.

In response, Cerberus++'s control plane adjusts memory slices based on current memory requirements. As illustrated in Figure 7(b), the memory allocation for non-target tasks increases to more than 10 bits, while the allocation for the targeted DNS and SYN tasks drops to 5 bits. When the actual DNS and SYN flood attacks begin, as shown in Figure 7(c), the reduced counter size lead to frequent overflows, causing the data-to-control plane traffic to spike about 800K packets per second (pps). This overwhelms the link, resulting in significant

packet loss. Consequently, the control plane fails to observe enough overflows to hit the detection threshold, leading to persistent undercounting. As a result, the false negative rate (FNR) remains high, averaging around 50%, as shown in Figure 7(d), which means it filters only 50% of malicious traffic.

D. Attack Strategy-III: Time-window Exploitation

In this attack, the attacker exploits the window-based sketch refresh mechanism to evade threshold-based detection. Specifically, since the sketch *periodically resets* measurement values at the start of each new time window, an adversary who knows both the threshold and the window interval can send many flows with distinct keys, each staying below the detection threshold within a single window. This prevents the sketch from accumulating enough per-flow evidence to trigger an alert. The number of transmitted packets per IP pair triggered by this attack strategy is smaller than that of other two attack strategies because it does not cause undercounting. To compensate and still cause harm, the adversary increases the packet size, thereby maximizing bandwidth consumption. Although the attacker could increase the number of distinct flow keys by generating more unique source IP addresses, doing so incurs additional cost, which makes this option less attractive for low-cost adversaries.

This vulnerability arises from a common design choice in sketch-based monitoring systems. Sketches provide approximate flow tracking but suffer from accuracy degradation over time due to hash collisions and limited counter space. To address this, developers typically implement one of two refresh strategies: manual resets triggered by the control plane, or window-based refresh mechanisms [23]. Considering the need for maintaining long-term measurement reliability, most systems adopt the window-based refresh approach [13], [24], [14], [15], [1], [61]. However, we show that this mechanism can be exploited by adversaries who have inferred internal states, specifically, the timing and structure of the window transitions, to repeatedly evade detection by remaining below threshold within each window.

Figure 8 demonstrates the effectiveness of the time-window exploitation strategy against Cerberus++. As shown in Figure 8(a), the adversary sends malicious traffic at the same rate, but not reaching the threshold of Cerberus. To maintain the same traffic rate as in previous attacks, larger packet sizes are used, resulting in a 20 times reduction in the number of transmitted malicious packets. Despite this, the overall traffic volume remains high. Figures 8(b) and (c) show that the attack still causes counter overflows; however, unlike previous attacks, approximately 63.1% of the mirrored packets are successfully processed by the control plane, enabling more accurate measurement. Nevertheless, as shown in Figure 8(d), no individual IP pair surpasses the configured threshold, resulting in a false negative rate (FNR) close to 100%, and allowing the attack to bypass Cerberus++'s defense mechanisms.

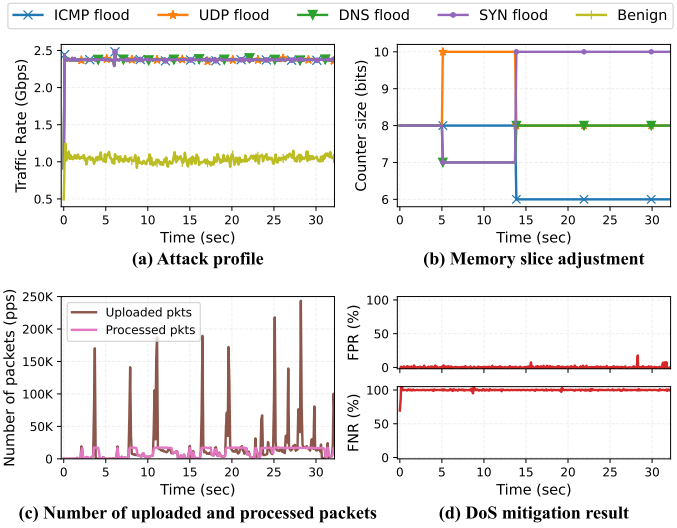


Fig. 8. Time-window exploitation attack against Cerberus++.

IV. The Shield Sketch Data Structure

In this section, we introduce Shield[3], a novel sketch data structure designed to handle dynamically changing DoS attack profiles in programmable data planes, with a particular focus on mitigating the Heracles attack. Shield is engineered to match the resource-sharing efficiency and DoS mitigation performance of state-of-the-art systems like Cerberus [1], while fully neutralizing the vulnerabilities exploited by the Heracles attack.

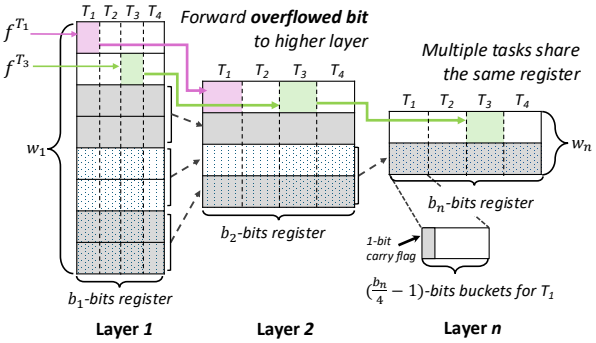
The core idea behind Shield is to *decouple* timing information across multiple *hierarchical layers*, rendering it significantly harder for adversaries to infer the timing patterns essential to launching Heracles. Each layer in the hierarchy independently performs sketch reset or decay operations at *different time intervals*, thereby obfuscating the timing signals that adversaries rely on.

A. Data Structure

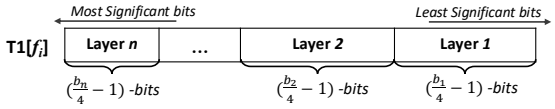
The design goals of Shield are threefold: (1) Enabling *concurrent DoS defense tasks* by supporting multiple mitigation tasks within a single sketch; (2) *Obfuscating the timing information* by distributing sensitive sub-timing information across multiple layers, preventing adversaries from measuring it; and (3) *Minimizing data-to-control plane communication*, thereby significantly mitigating control-plane flooding attacks. The first goal ensures flexible and high-performance DoS defense, similar to Cerberus. The latter two goals directly counter the Heracles attack. To be specific, timing obfuscation disrupts the attack preparation step by degrading inference of scheduling parameters, and communication minimization mitigates control-plane flooding caused by synchronized augmentation or memory squeezing.

To meet these goals, Shield adopts a hierarchical layered register design. This architecture enhances timing obfuscation and improves memory efficiency by allocating numerous small

[3]**S**hared **Hie**rarchical Registers for **L**ayered **D**ecay

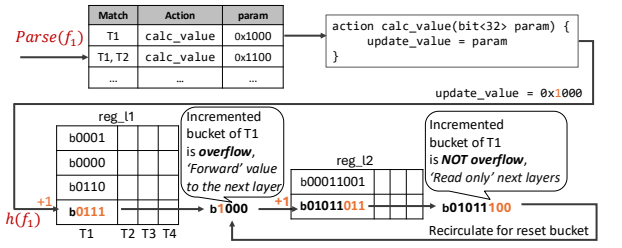


(a) Simplified overview of the SHIELD.

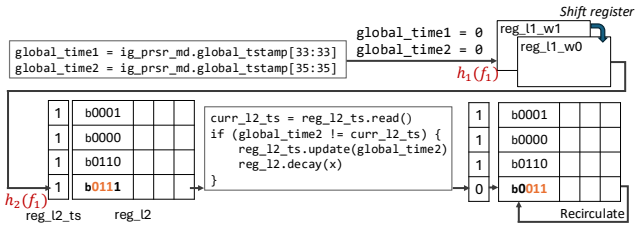


(b) Logical counter bucket for task T1.

Fig. 9. Overview of SHIELD. Each layer stores counters of different tasks (T_1-T_4) in the same register. Overflowed bits at lower layers are transferred to higher layers.



(a) Update operation of a flow key f_1.



(b) Decay operation of a flow key f_1.

Fig. 10. Example operations of SHIELD.

counter buckets for frequent, low-volume (or mouse) flows and fewer larger buckets for high-volume (or elephant) flows, reflecting the typical skewed distribution of network traffic [62], [63], [64].

As shown in Figure 9, SHIELD consists of n layers of registers. Each register is implemented as an array of size w_n, and each array element contains a bucket of b_n bits. These buckets are shared across four different in-network counting tasks (T_1 to T_4) in this example. For illustration, Fig. 9 uses four tasks with equal-sized memory slices, but both the number of tasks and slice sizes are developer-configurable. Each task within a register is isolated by fixed-size memory slices and carry flags. Unlike Cerberus [1], these slices in SHIELD are immutable at runtime.

SHIELD also differs from Cerberus in how it handles counter overflows. When a lower-layer counter overflows, the excess is forwarded to the *next higher layer*, not directly to the control plane, enabling hierarchical bit propagation. Lower layers store lower-order bits with fine granularity for tracking frequent, low-impact flows, while higher layers accumulate overflowed higher-order bits to enable accurate tracking of heavy hitters without overestimation, as shown in Figure 9(a).

This design is formalized using a hierarchical bit-encoding scheme. In traditional layered sketches [63], [64], the estimated flow count $\hat{f}_i$ is computed as:

$$\hat{f}_i = \sum_{l=1}^{L} C_l[i], \tag{2}$$

where $C_l[i]$ is the counter value for flow i in layer l.

In contrast, SHIELD encodes the flow count as:

$$\hat{f}_i = \sum_{l=1}^{L} (C_l[i] \times 2^{\sum_{j=1}^{l-1}(b_j-1)}), \tag{3}$$

where b_j is the bit-width of counters in layer j. This ensures that each layer contributes to the final count at the correct bit position, preserving numerical accuracy across layers.

Figure 9(b) illustrates the logical counter buckets for each task in SHIELD. By promoting only the most significant bits to higher layers, SHIELD provides the illusion of a large, unified counter with effective bit-width equal to the sum of all layer widths. This design reduces the frequency of control-plane updates by avoiding premature overflows that would otherwise trigger reporting in shared-memory environments. A theoretical analysis of SHIELD's estimation error and the effects of decay is provided in Appendix A.

B. Operations

At its core, SHIELD employs a hierarchical structure, where each layer functions similarly to a counting Bloom filter (CBF) [65]. We describe two key operations that support multiple mitigation tasks and enable automatic sketch refresh: each layer operates like a CBF, so we describe two main unique operations for multiple task support and automatic sketch refresh: (1) *Update* and (2) *Refresh*.

Update. SHIELD is designed to support multiple concurrent tasks within a shared register. When an incoming packet with a flow key f_1 arrives, as illustrated in Figure 10(a), the packet is classified into one or more tasks based on the parsed results. SHIELD then determines the appropriate `update_value` based on the task and its predefined slice within the register. SHIELD saves these increments (i.e., `update_value`) in the match-action table in advance: For example, in Figure 10(a), Task 1 occupies the highest bit, so we need to add 0x1000 to increase the counter of Task 1. The index of register is defined with the hash $i = h_i(f_1)$ mod w_l, and SHIELD supports using multiple hashes like a CBF.

After the increment, SHIELD checks for overflow. If the carry flag is set (i.e., overflow), the overflowed bit is propagated to the next layer to increment the task's value in the next layer by 1. If no overflow occurs, upper layers are accessed in read-only mode, without being incremented. If all layers, including the

highest layer n, overflow SHIELD generates a mirror packet and forwards it to the control plane, similar to the co-monitoring mechanism in Cerberus [1]. Since each register in Tofino can be accessed only once per packet as it moves through the pipeline, SHIELD resets the carry flag via *recirculating* the packet.

Refresh. SHIELD offers two refresh mechanisms for each layer to decouple the timing information with the sketch: *time window-based* reset and *bit shift-based decaying*.

Figure 10(b) illustrates the decay operation. Upon receiving a packet, SHIELD extracts `global_time` value from the hardware timestamp in the parser metadata. Each layer can have different `global_time`, which means different layers can use different time decay period. For example, layer 1 register for window 1 (`reg_l1_w1`) shifts to the next window register (`reg_l1_w0`) based on `global_time1` value. This results in periodic full reset of layer 1, which is beneficial for handling short-lived mice flows, as layer 1 contains small-sized buckets.

In contrast, layer 2 or higher layers employ decaying rather than window shifting. Each register in these layers is associated with a 1-bit timestamp register (`reg_l2_ts` in Figure 10(b)). If the hardware timestamp value in `global_time` differs from the value from `reg_l2_ts`, SHIELD triggers a decay operation; that is, a bit shift followed by an `AND` operation (using a predefined mask) (through recirculating) and initializes all unnecessary bits to zero, to eliminate values that invade each task boundary.

One notable implementation challenge arises from a fundamental architectural limitation of programmable switches; namely, it is not possible to update all register entries concurrently. As shown in Figure 10(b), the decay operation in SHIELD is only triggered when a packet hits the corresponding register entry after the `global_time` has transitioned. Yet, it is undesirable to rely solely on natural traffic to ensure that every entry is decayed within the decay interval. Without full coverage, many entries would retain stale values, compromising the effectiveness of the decay mechanism.

We solve this problem by proactively generating and injecting specially crafted helper packets that are designed to access all register entries in a given layer and expedite decay for entries that have not yet decayed once the new window starts. Thus, once decay has occurred within a given window, no further decay operations can be triggered. Because the hash functions used in SHIELD are known at design time, we can precompute the necessary flow keys to ensure complete coverage of all register indices. We then periodically inject these helper packets at every decay or reset interval, ensuring that all entries undergo the appropriate decay operation, regardless of actual traffic patterns. This subtle yet effective approach enables SHIELD to maintain fresh and accurate counters without resorting to costly bulk updates from the control plane, preserving both correctness and line-rate performance.

V. EVALUATING SHIELD

In this section, we evaluate the performance of SHIELD. We first show the robustness of SHIELD against the HERACLES attack (§V-A). Then, we evaluate its accuracy (§V-B) and resource overheads (§V-C).

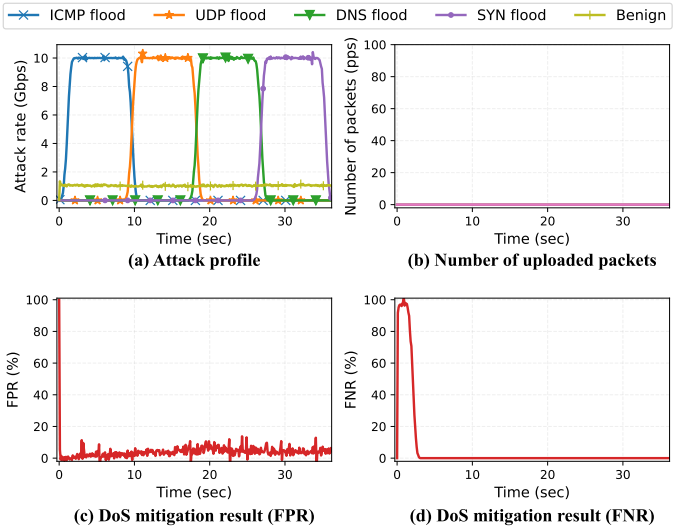


Fig. 11. HERACLES attack against SHIELD.

A. Robustness against HERACLES

We evaluate how SHIELD is robust against HERACLES attacks. The experimental setup for the evaluation is the same as in Section III, except that the target system is changed to SHIELD. For this, we implemented the Tofino version of SHIELD and launched the HERACLES attack. In this evaluation, we use three-layer structured SHIELD and use two hashes for entry key. In our implementation, each layer uses a 32-bit register of size 2^{16}, two 32-bit register of size 2^{15}, and two 32-bit register of size 2^{14} to deploy four in-network defense tasks. Our $P4_{16}$ implementation of SHIELD for Intel Tofino programmable switch is available at [50].

We begin by evaluating whether SHIELD effectively disrupts the timing inference phase (i.e., the attack preparation step) of the HERACLES attack. Using the best-effort inference strategy described in §III-A, we observe that SHIELD significantly reduces the accuracy of time-window inference compared to Cerberus. For time-window inference, SHIELD produces a median absolute error of 4.29 seconds (which is the same as the maximum error in our experiment setup of 8.59-second window size), compared to 0.29 seconds in Cerberus, which shows significant degradation. In contrast, SHIELD demonstrates marginal degradation of the threshold inference (a median absolute error of 1 packet, whereas Cerberus yields only 0 packets) because sending bursts of attack packets that exceeds a threshold is possible even against SHIELD. These substantially increased overall errors demonstrate that SHIELD degrades the adversary's ability to infer internal sketch parameters, thereby effectively neutralizing a critical prerequisite of the HERACLES attack.

We now present the result of the HERACLES attack against SHIELD, as shown in Figure 11. To rigorously evaluate its robustness, we assume a powerful adversary that has full knowledge of SHIELD's internal counter size of each task and

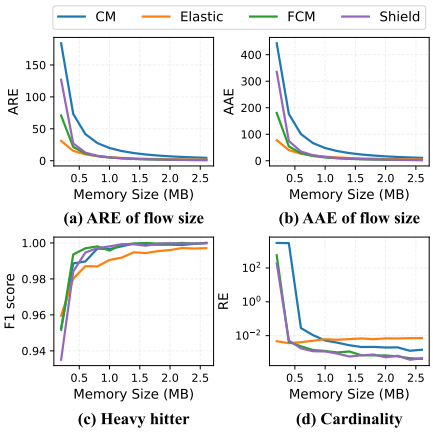


Fig. 12. Accuracy comparison with CAIDA-18 internet trace traffic [66].

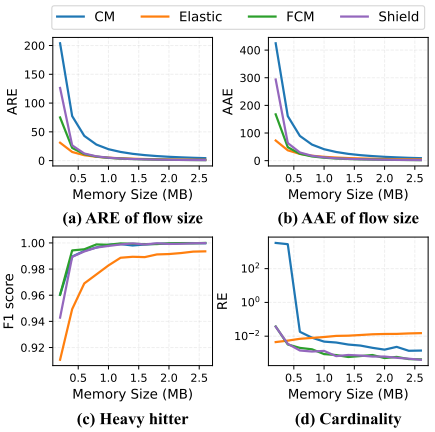


Fig. 13. Accuracy comparison with CAIDA-19 internet trace traffic [67].

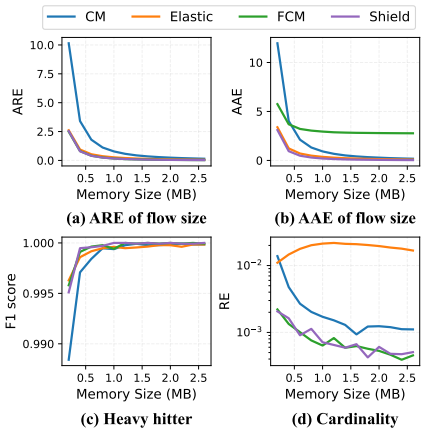


Fig. 14. Accuracy comparison with MAWI upstream ISP traffic [68].

time-window parameters. Although such precise inference is unlikely in practice (as demonstrated above), this conservative assumption enables a more stringent test of SHIELD's defenses. Figure 11(a) describes the attack profile that we used; the adversary launches hybrid attack profiles that dynamically change their pattern based on the time window of layer 1 of SHIELD. Figure 11(b) shows that the number of packets mirrored to the control plane is close to zero even when an attacker uses a synchronized attack pattern. This is because SHIELD uses some of the SRAM space to provide an abstraction that each counter bucket is 37-bit wide: 7 bits from layer 1, 15 bits from layer 2 and layer 3, respectively. This abstraction allows HERACLES to achieve near-zero communication between the data and control planes.

Figure 11(c) and (d) demonstrate that SHIELD effectively blocks both hybrid and dynamic attacks, despite relying on fixed-sized memory slices, unlike Cerberus, which employs dynamic memory resizing. Notably, SHIELD achieves 0% FNR (i.e., successfully blocking all malicious traffic) within just 3 seconds, whereas Cerberus fails to reduce FNR below 60% across all HERACLES attack strategies; see Figure 6, 7, and 8. In addition, SHIELD maintains a lower FPR (3.85% on average), compared to Cerberus, which exhibits an average FPR of 5.80% under the same attack.

B. Accuracy of SHIELD

Next, we provide accuracy comparison of different measurement tasks and provide the performance metrics corresponding to each task. To evaluate the accuracy of SHIELD, we used 32 traffic traces from each of the CAIDA [49] and MAWI [68] datasets. We use the source IP address as the flow key for accuracy comparison [63]. Each CAIDA and MAWI trace contains about (28M, 5.4M) packets and (680K, 140K) distinct flows with Zipf skewness of (1.5, 1.1) in a 15s window, respectively. For the performance evaluation of SHIELD, we used 32 continuous traffic traces.

We use 4 metrics to evaluate SHIELD. We explain how the metrics are derived as follows:

- *Relative Error (RE)*: $|1 - \frac{\hat{f}_i}{f_i}|$, where f_i and $\hat{f}_i$ are the actual and estimated statistics, respectively. We use RE to evaluate cardinality estimation.
- *Average Relative Error (ARE)*: $\frac{1}{N}\sum_{i=1}^{N}\frac{|f_i - \hat{f}_i|}{f_i}$, where N is the number of flows, and f_i and $\hat{f}_i$ are actual and estimated flow sizes. We use ARE to evaluate flow size estimation.
- *Average Absolute Error (AAE)*: $\frac{1}{N}\sum_{i=1}^{N}|f_i - \hat{f}_i|$, where N is the number of flows, and f_i and $\hat{f}_i$ are actual and estimated flow sizes. We use AAE to evaluate flow size estimation.
- *F1-score*: $\frac{2\times PR\times RR}{PR+RR}$, where precision rate ($PR$) is the fraction of relevant instances among the retrieved instances, and recall rate (RR) is the fraction of relevant instances that were retrieved. We use F1-score to evaluate heavy hitter detection performance.

Figures 12–14 compare the accuracy of SHIELD against Count-Min (CM) and FCM-Sketch [63] across a range of network measurement tasks relevant to security applications, including flow size estimation, cardinality, and heavy hitter detection. The evaluation is conducted under varying memory limits, from 0.2 to 2.6 MB. Figures 12 and 13 present results using CAIDA internet trace datasets 2018 and 2019 [49], while Figure 14 shows corresponding results on the MAWI upstream ISP trace [68].

Flow size estimation. Flow size estimation involves counting the packets of all individual flows in a time window. As shown in Figures 12, 13, and 14(a)–(b), the ARE and AAE of SHIELD are comparable to those of other sketches. While CM exhibits a relatively high average relative error of around 5 (except on the MAWI dataset, which contains fewer packets), Elastic, FCM, and SHIELD consistently achieve an ARE close to 1 when allocated more than 1.6 MB of memory. For average absolute error, FCM records a value near 3 on the MAWI dataset, whereas Elastic and SHIELD remain below 1, indicating more precise flow size estimation.

Heavy hitter detection. In our experiments, a heavy hitter is defined as a flow that accounts for more than 0.05% of the total packet volume within a given time window. Figures 12, 13,

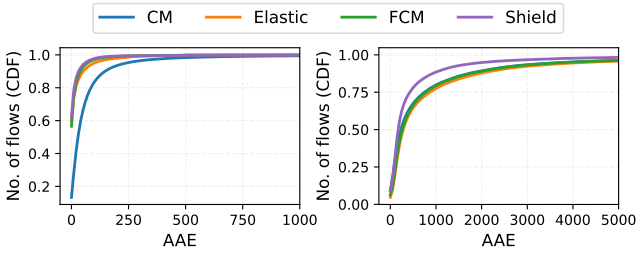


Fig. 15. CDF of average absolute error comparison for mice (flow size ≤ 255) and elephant flow (flow size > 255).

and 14(c) present a comparison of F1-scores for SHIELD, CM, Elastic, and FCM. Across all traces, every sketch—including SHIELD —achieves an F1-score exceeding 0.99 even with just 0.6 MB of memory. These results show that SHIELD is highly effective at identifying heavy hitters, making it well-suited for DDoS detection and mitigation scenarios.

Cardinality estimation. In our evaluation, cardinality refers to the number of distinct flows observed within the traffic dataset. Figure 12–14(d) presents a comparison of relative error (RE) for cardinality estimation among SHIELD, CM, Elastic, and FCM. On the CAIDA datasets (Figures 12 and 13), SHIELD and FCM achieve low RE when given more than 0.4 MB of memory, while CM requires over 0.6 MB to reach a comparable level of accuracy. In contrast, Elastic maintains a consistently low RE in the range of 0.001 to 0.002, regardless of memory size. In the MAWI dataset (Figure 14), CM and Elastic appear to have higher RE due to the y-axis scale, but their actual estimation performance is similar to results observed in the CAIDA datasets.

Effectiveness of decay. To evaluate the effectiveness of SHIELD's decay mechanism, we compared the AAE of flow size estimation results after a few window reset periods passed for mice and elephant flow. In this experiment, we transmitted traffic from the CAIDA19 dataset for 1 minute. SHIELD performs the decay operation on layer 2 and 3, while others reset their sketch in 15-second cycles.

Figure 15 shows the CDF plots for AAE from flow size estimation for mice and elephant flows. As shown in Figure 15(a), Elastic, FCM, and SHIELD do not show significant difference in AAE for mouse flows. This is because SHIELD also uses a reset mechanism in the first layer. On the other hand, the elephant flow, which is stored in a higher layer using decay mechanism, shows a difference in AAE of about 250+ compared to the other sketches, as shown in Figure 15(b). However, SHIELD also loses some value through the decay mecahnsim, which is why it loses AAE.

C. Resource Overhead

Table I presents the hardware resource utilization of SHIELD in comparison with CM, FCM [63], and Cerberus [1], all evaluated on the Intel Tofino architecture [7], with resource usage statistics collected using Intel P4 Insight (`p4i`) [69]. In this evaluation, all sketches (except SHIELD) use a traditional time-window-based refresh mechanism to periodically clear outdated measurement data. And, all other sketches are configured to run four different DoS defense tasks.

TABLE I
HARDWARE RESOURCE CONSUMPTION OF BASIC SWITCH FUNCTIONALITY AND 4 DOS DEFENSE TASKS ON TOFINO SWITCH.

Resource	CM	FCM	Cerberus	SHIELD
SRAM	34.9%	41.6%	16.7%	24.4%
TCAM	0.0%	5.6%	0.0%	0.0%
SALU	37.5%	62.0%	20.8%	45.8%
Hash units	18.9%	15.4%	14.8%	22.4%
Pipeline stages	12	12	9	11

Unlike conventional sketches, only Cerberus and SHIELD are designed to support multiple tasks under resource-sharing constraints while keeping the pipeline usage within the 12-stage limit per pipeline of Tofino 1 ASIC. Therefore, as illustrated in Table I, both SHIELD and Cerberus require a smaller number of pipeline stages than CM and FCM, which are not designed for concurrency. In our evaluation, SHIELD consumes 7.7%p more SRAM and 25%p more SALU than Cerberus. This overhead stems from the increased use of registers and ALUs for the layered structure and time-decaying mechanism of SHIELD. These results demonstrate that SHIELD achieves comparable concurrency and scalability to Cerberus while offering additional robustness against the HERACLES attack.

VI. DISCUSSION

Programmable switches promise unprecedented performance and flexibility for in-network processing, but they are fundamentally shaped, and constrained, by their underlying hardware and programming abstractions. This section discusses how design challenges in programmable switch architectures can lead to security challenges, and how HERACLES and SHIELD reveal both the consequences of these design decisions and the limits of current mitigation strategies. Then, we discuss task-specific trade-offs between SHIELD and Cerberus.

A. Systemic Challenges in Programmable Switch Design

We list two challenges that commonly appear across the literature on programmable switches and novel defense applications.

Performance-driven trade-offs and insecure defaults. Developers building data-plane applications operate under strict constraints; e.g., line-rate performance, limited memory, and a rigid match-action pipeline. To meet these requirements, developers often adopt design choices that prioritize efficiency. Common examples include using CRC-based hash functions with weak entropy [70], [71], [72], [12], [73], [1], [14], minimizing register widths to reduce memory usage [1], [11], [27], [14], and omitting protections for slow-path processing [55], [74]. These optimizations are well-known and widely used, yet they may introduce potential problems, such as hash collisions or inaccurate counters. However, secure alternatives, such as cryptographic hashes [15], [75] or larger sketches, typically incur unacceptable performance costs that may limit practical adoption.

Control plane bottlenecks. While the data plane is optimized for Tbps-level throughput, the control plane can become a bottleneck. Reading large register arrays or updating match-action table (MAT) entries introduces latencies in the millisecond range, making timely reaction to attacks challenging [60]. Moreover, control planes are sometimes deployed on remote servers with minimal rate limiting or overload protection. These architectural asymmetries persist across many systems and can be difficult to address efficiently. Although research has advanced the state of the art (e.g., SmartNIC offloading [76] or DPDK-based control paths [60]), such solutions are not yet widely adopted, and may still be vulnerable to inference-based attacks, as demonstrated in our DPDK-based version of Cerberus (or Cerberus++).

B. Heracles and Shield as Case Studies

Heracles demonstrates how structural aspects of switch design can be exploited. The attack does not rely on obscure bugs or misconfigurations, but instead targets expected behaviors such as resource sharing, periodic sketch resets, and static filtering thresholds that result from developer trade-offs under hardware constraints. Heracles highlights how vulnerable these "default" or widely-adopted designs can be when adversaries are aware of the system's operational model.

Shield offers a defense that increases robustness within these constraints by introducing layered sketching and more explicit task partitioning. It demonstrates that side-channel resistance can be improved with better abstractions and modularity. Heracles is not a one-off exploit, but rather illustrates broader architectural tensions in programmable switch design. Addressing these challenges, especially the balance between performance, flexibility, and resilience, will be important as programmable data planes become more widely adopted.

C. Task-specific trade-offs between Shield and Cerberus

As discussed in §V-C, Shield incurs higher resource usage than Cerberus (e.g., 25.0%p and 7.7%p more SALU and SRAM consumption). This overhead, however, represents a deliberate trade-off to provide robust and stable mitigation against sophisticated, multi-vector attacks that Cerberus fails to handle. Nevertheless, not all in-network DoS defense tasks require such elevated resource budgets. For instance, query–existence–based defenses (e.g., DNS amplification) typically rely on small counters and simple existence tests rather than high-precision aggregation.

Resource sharing of query existence-based tasks. Several amplification-attack defenses operate by checking the existence of outstanding queries, often using approximate data structures such as counting Bloom filters (CBFs). Our evaluation shows that the Heracles attack does not effectively induce large-scale overflows in Cerberus when CBFs are used. This stems from that CBF-based detection focuses on membership existence rather than threshold-based rate discrimination, making overflow-based manipulation significantly more difficult. Thus, for resource efficiency, we recommend retaining the original Cerberus design for the query-existence–based defense tasks, even though Shield also supports cardinality estimation.

Resource sharing of byte counting task. Other forms of DDoS mitigation require byte-count–based thresholds, as in the Coremelt defenses in Mew [14] and Cerberus [1]. Unlike simple flow-counting, however, byte-count measurements increase much more rapidly because each update adds the packet's full payload size. When such byte-count tasks share registers with other measurements, we find that it becomes substantially easier for an adversary to trigger large bursts of overflows, resulting in excessive flooding toward the control plane. This rapid growth of the counter also accelerates the memory squeezing attack, enabling a malicious task to quickly consume resources at the expense of others (see Appendix B). Robust resource sharing for byte-count–based measurements is feasible in Shield, as it provides a large, fixed-size counter abstraction that prevents such pathological escalation.

VII. Related Work

In this section, we review related work on programmable data-plane applications and their systematic constraints (§VII-A), as well as how our Heracles attack relates to existing in-data-plane DoS defense systems (§VII-B).

A. Programmable Switch Applications and Systemic Constraints

The emergence of programmable switches has enabled high-performance packet processing at line rate, forstering a wide range of innovations in networking. These include applications in load balancing [77], [70], consensus acceleration [78], [79], and in-network telemetry and monitoring [9], [10], [11]. Programmable data planes have also shown promise in the security domain, powering systems for covert channel detection and volumetric DoS attack mitigation [80], [55], [12], [13], [24], [14], [1], [15].

However, such systems are inherently constrained by the architectural limitations of programmable switches. Recent work has proposed various methods to work within or around these constraints. For example, sketch-based systems like UnivMon [9] and Elastic [81] support multi-feature measurements, while multi-key sketches such as BeauCoup [71] and CoCoSketch [82] enhance query expressiveness. Others, like FCM [63] and CL [64], optimize memory usage by exploiting traffic skewness. SketchLib [18] analyzes common sketch bottlenecks in RMT-based switches and offers optimized implementations without sacrificing accuracy. Runtime reconfigurability has also been explored through systems such as FlyMon [11], FlexCore [83], and P4runpro [84], which allow dynamic task updates without downtime.

As programmable data-plane applications grow in sophistication, so do the potential attack surfaces. Prior work has explored attacks targeting programmable switch infrastructure. Dumitru et al. [85] analyzed how bugs in P4 programs could lead to security vulnerabilities, such as DoS or privilege escalation, though their evaluation assumes complete access to the switch internals. Wang et al. [86] discussed theoretical

TABLE II
APPLICABILITY OF THE HERACLES ATTACK STRATEGIES ACROSS IN-DATA-PLANE DOS DEFENSES.

	SHIELD	Cerberus [1]	Jaqen [13]	Poseidon [12]	Ripple [24]	Mew [14]
Inference Capabilities						
Inferring Detection Threshold	○	○	○	○	○[a]	○[a]
Inferring Windows Timing	△	○	○	○	○[a]	○[a]
Inferring Co-located Tasks	×	○	N/A	N/A	N/A	○[a]
Attack Feasibility						
Synchronized Augmentation	×	○	N/A	N/A	N/A	N/A
Memory Squeezing	N/A	○	N/A	N/A	N/A	○[a]
Time-window Exploitation	×	○	○	○	○[a]	○[a]

[a] Adversaries should generate attack traffic along specific paths of choice among multiple switches.

attack surfaces in P4-based data-plane applications but did not empirically validate their findings. Other studies highlight weaknesses in specific algorithmic components, such as flow selection logic that uses 3rd-order features (e.g., inter packet delay, packet size distribution) [74], or the absence of authentication in control interfaces like P4Runtime [87].

In contrast, the HERACLES attack demonstrates a practical and remote exploitation strategy that targets the systemic assumptions embedded in real-world data-plane defenses. Unlike earlier attacks that depend on software bugs or control-plane compromise, HERACLES exploits only architectural constraints and observable behavior, requiring no priviledged access.

B. *HERACLES and Existing DoS Mitigations*

The HERACLES attack reveals a new class of vulnerabilities that arise from adaptive memory management and shared resource semantics in programmable switches. While primarily designed to target Cerberus [1], the attack strategies generalize to other threshold-based, window-driven DoS defenses.

Table II summarizes the applicability of the HERACLES strategies to several prior defenses. Jaqen [13] and Poseidon [12] employ periodic threshold-based mitigation, making them susceptible to probing-based inference of filtering thresholds and time windows. Ripple [24] and Mew [14], which defend against link-flooding attacks [5], [6], also rely on periodic sketches and may become susceptible to the two probing-based inferences, albeit requiring the adversary to direct attack traffic along certain paths of choice among multiple switches. Note that the synchronized augmentation and memory squeezing strategies are not directly applicable to many of these systems, as they do not perform dynamic memory adjustment.

We have empirically validated the applicability of HERACLES against these prior defenses; see Appendix B. Testing all these defenses on real hardware was infeasible due to several practical constraints (e.g., limited access to multiple programmable switches, proprietary P4 programs), thus we resorted to high-fidelity simulation that accurately mimics the Tofino switch's hardware-specific constraints.

While not every HERACLES attack strategy applies equally, the underlying design patterns (such as static thresholding, shared memory, and predictable refresh cycles) are widely adopted across prior defenses. These patterns, while effective for resource efficiency, may open potential avenues to inference-based attacks like HERACLES, particularly in adversarial environments.

VIII. CONCLUSION

Highly flexible and dynamic memory management has been a long-sought objective in in-data-plane DoS mitigation. Yet, when implemented atop commodity switches with rigid hardware constraints, such flexibility becomes a liability, where innocuous design choices create new attack surfaces, as demonstrated by our HERACLES attack, ultimately undermining the mitigation itself. Our SHIELD takes the first step toward hardware-constraint-conscious system design for resilient DoS defenses, which we believe warrants systematic exploration in future work.

ETHICAL CONSIDERATIONS

Our work does not raise ethical concerns because all experiments were conducted in controlled environments using an isolated programmable switch and server, and the background real-world traffic used in the experiments is anonymized. The particular implementation of the HERACLES attack is specific to Cerberus [1], an academic prototype rather than a deployed commercial system, and we have disclosed our findings and modifications to the authors of [1], who confirmed that our modified control plane conforms to their intended design. Disclosing this attack and its underlying root cause is intended to prevent similar vulnerable P4 design patterns from being adopted in future production environments.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their insightful comments and invaluable feedback. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00441762, Global Advanced Cybersecurity Human Resources Development). This material is based upon work supported in part by the National Science Foundation (NSF) under Grant No. 2148374. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] H. Zhou and G. Gu, "Cerberus: Enabling efficient and effective in-network monitoring on programmable switches," in *Proc. IEEE S&P*, 2024, pp. 4424–4439.

[2] O. Yoachimik and J. Pacheco, "Record-breaking 5.6 Tbps DDoS attack and global DDoS trends for 2024 Q4," 2025. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-for-2024-q4/

[3] A. V. Vu, B. Collier, D. R. Thomas, J. Kristoff, R. Clayton, and A. Hutchings, "Assessing the Aftermath: the Effects of a Global Takedown against DDoS-for-hire Services," in *Proc. USENIX Security*, 2025.

[4] 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 *Proc. USENIX Security*, 2017, pp. 1093–1110.

[5] A. Studer and A. Perrig, "The Coremelt Attack," in *Proc. ESORICS*, 2009, pp. 37–52.

[6] M. S. Kang, S. B. Lee, and V. D. Gligor, "The Crossfire Attack," in *Proc. IEEE S&P*, 2013, pp. 127–141.

[7] Intel Corporation, "Intel® tofino™ series," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, 2014.

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

[10] Y. Zhou, C. Sun, H. H. Liu, R. Miao, S. Bai, B. Li, Z. Zheng, L. Zhu, Z. Shen, Y. Xi, P. Zhang, D. Cai, M. Zhang, and M. Xu, "Flow Event Telemetry on Programmable Data Plane," in *Proc. ACM SIGCOMM*, 2020, p. 76–89.

[11] H. Zheng, C. Tian, T. Yang, H. Lin, C. Liu, Z. Zhang, W. Dou, and G. Chen, "Flymon: enabling on-the-fly task reconfiguration for network measurement," in *Proc. ACM SIGCOMM*, 2022, p. 486–502.

[12] M. Zhang, G. Li, S. Wang, C. Liu, A. Chen, H. Hu, G. Gu, Q. Li, M. Xu, and J. Wu, "Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches," in *Proc. NDSS*, 2020.

[13] Z. Liu, H. Namkung, G. Nikolaidis, J. Lee, C. Kim, X. Jin, V. Braverman, M. Yu, and V. Sekar, "Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches," in *Proc. USENIX Security*, 2021, pp. 3829–3846.

[14] H. Zhou, S. Hong, Y. Liu, X. Luo, W. Li, and G. Gu, "Mew: Enabling Large-Scale and Dynamic Link-Flooding Defenses on Programmable Switches," in *Proc. IEEE S&P*, 2023, pp. 3178–3192.

[15] S. Yoo, X. Chen, and J. Rexford, "SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes," in *Proc. USENIX Security*, 2024, pp. 217–234.

[16] A. Agrawal and C. Kim, "Intel Tofino2 – A 12.9 Tbps P4-Programmable Ethernet Switch," in *Proc. IEEE HotChips*, 2020, pp. 1–32.

[17] 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 *Proc. ACM SIGCOMM*, 2013, p. 99–110.

[18] H. Namkung, Z. Liu, D. Kim, V. Sekar, and P. Steenkiste, "SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches," in *Proc. USENIX NSDI*, 2022, pp. 743–759.

[19] S. Landau-Feibish, Z. Liu, and J. Rexford, "Compact Data Structures for Network Telemetry," *ACM Comput. Surv.*, vol. 57, no. 8, Mar. 2025.

[20] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: query-driven streaming network telemetry," in *Proc. ACM SIGCOMM*, 2018, p. 357–371.

[21] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, "Heavy hitters in streams and sliding windows," in *Proc. IEEE INFOCOM*, 2016, pp. 1–9.

[22] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, "Memento: Making Sliding Windows Efficient for Heavy Hitters," in *Proc. ACM CoNEXT*, 2018, p. 254–266.

[23] X. Z. Khooi, L. Csikor, J. Li, M. S. Kang, and D. M. Divakaran, "Revisiting heavy-hitter detection on commodity programmable switches," in *Proc. IEEE NetSoft*, 2021, pp. 79–87.

[24] J. Xing, W. Wu, and A. Chen, "Ripple: A Programmable, Decentralized Link-Flooding Defense Against Adaptive Adversaries," in *Proc. USENIX Security*, 2021, pp. 3865–3881.

[25] Z. Zeng, L. Cui, M. Qian, Z. Zhang, and K. Wei, "A survey on sliding window sketch for network measurement," *Computer Networks*, vol. 226, p. 109696, 2023.

[26] P. Zheng, T. Benson, and C. Hu, "P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs," in *Proc ACM CoNEXT*, 2018, pp. 98–111.

[27] H. Zhu, T. Wang, Y. Hong, D. R. Ports, A. Sivaraman, and X. Jin, "NetVRM: Virtual Register Memory for Programmable Networks," in *Proc. USENIX NSDI*, 2022, pp. 155–170.

[28] X. Z. Khooi, L. Csikor, D. M. Divakaran, and M. S. Kang, "DIDA: Distributed In-Network Defense Architecture Against Amplified Reflection DDoS Attacks," in *Proc. IEEE NetSoft*, 2020, pp. 277–281.

[29] A. G. Alcoz, M. Strohmeier, V. Lenders, and L. Vanbever, "Aggregate-Based Congestion Control for Pulse-Wave DDoS Defense," in *Proc. ACM SIGCOMM*, 2022, pp. 693–706.

[30] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and Elastic DDoS Defense," in *Proc. USENIX Security*, 2015, pp. 817–832.

[31] P. Zilberman, R. Puzis, and Y. Elovici, "On Network Footprint of Traffic Inspection and Filtering at Global Scrubbing Centers," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 14, no. 5, pp. 521–534, 2015.

[32] S. B. Lee, M. S. Kang, and V. D. Gligor, "CoDef: Collaborative Defense Against Large-Scale Link-Flooding Attacks," in *Proc. ACM CoNEXT*, 2013, pp. 417–428.

[33] H. Griffioen and C. Doerr, "Examining Mirai's battle over the Internet of Things," in *Proc. ACM CCS*, 2020, pp. 743–756.

[34] Mirai tracker, "Mirai Botnet IP list," 2025, https://mirai.security.gives/data/ip_list.txt and http://www.sanyal.org/mirai-ips.txt.

[35] A. Kuzmanovic and E. W. Knightly, "Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. the Mice and Elephants)," in *Proc. ACM SIGCOMM*, 2003, pp. 75–86.

[36] X. Luo and R. K. Chang, "On a new class of pulsing denial-of-service attacks and the defense." in *Proc. NDSS*, 2005, pp. 1–19.

[37] Y. Zhang, Z. M. Mao, and J. Wang, "Low-Rate TCP-Targeted DoS Attack Disrupts Internet Routing," in *Proc. NDSS*, 2007, pp. 1–15.

[38] R. Rasti, M. Murthy, N. Weaver, and V. Paxson, "Temporal Lensing and its Application in Pulsing Denial-of-Service Attacks," in *Proc. IEEE S&P*, 2015, pp. 187–198.

[39] Y.-M. Ke, C.-W. Chen, H.-C. Hsiao, A. Perrig, and V. Sekar, "CICADAS: Congesting the Internet with Coordinated And Decentralized Pulsating Attacks," in *Proc. ACM AsiaCCS*, 2016, pp. 699–710.

[40] H. Shan, Q. Wang, and C. Pu, "Tail Attacks on Web Applications," in *Proc. ACM CCS*, 2017, pp. 1725–1739.

[41] J. Park, D. Nyang, and A. Mohaisen, "Timing is Almost Everything: Realistic Evaluation of the Very Short Intermittent DDoS Attacks," in *Proc. PST*, 2018, pp. 1–10.

[42] R. Guo, J. Chen, Y. Wang, K. Mu, B. Liu, X. Li, C. Zhang, H. Duan, and J. Wu, "Temporal CDN-Convex Lens: A CDN-Assisted Practical Pulsing DDoS Attack," in *Proc. USENIX Security*, 2023, pp. 6185–6202.

[43] X. Li, D. Wu, H. Duan, and Q. Li, "DNSBomb: A New Practical-and-Powerful Pulsing DoS Attack Exploiting DNS Queries-and-Responses," in *Proc. IEEE S&P*, 2024, pp. 4478–4496.

[44] E. Kovacs, "Pulse Wave DDoS Attacks Disrupt Hybrid Defenses," 2017. [Online]. Available: https://www.securityweek.com/pulse-wave-ddos-attacks-disrupt-hybrid-defenses/

[45] Imperva, "Attackers Use DDoS Pulses to Pin Down Multiple Targets," 2018. [Online]. Available: https://www.imperva.com/blog/archive/pulse-wave-ddos-pins-down-multiple-targets/

[46] H. Zhou and G. Gu, "Cerberus-prototype," 2024. [Online]. Available: https://github.com/successlab/Cerberus

[47] Edgecore Networks, "DCS800 Data Center Switch - Wedge100BF-32X," 2023. [Online]. Available: https://www.edge-core.com/wp-content/uploads/2023/08/DCS800-Wedge100BF-32X-R11.pdf

[48] DPDK Project, "DPDK – the open source data plane development kit accelerating network performance," 2025. [Online]. Available: https://www.dpdk.org/

[49] The CAIDA UCSD, "The caida anonymized internet traces dataset," 2008. [Online]. Available: https://www.caida.org/catalog/datasets/passive_sampler_dataset/

[50] H. Nam and D. Lim, "GitHub repository for Heracles and Shield," 2025. [Online]. Available: https://github.com/NetSP-KAIST/shield

[51] Intel Corporation, "Intel® p4 studio," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html

[52] P4 Language Consortium, "open-p4studio," 2025. [Online]. Available: https://github.com/p4lang/open-p4studio

[53] TRex Team, "TRex - Realistic Traffic Generator," 2025. [Online]. Available: https://trex-tgn.cisco.com/

[54] V. Gurevich and A. Fingerhut, "$P4_{16}$ programming for intel® tofino™ using intel p4 studio™," in *Proceedings of the 2021 P4 Workshop*, 2021, pp. 1–40, retrived December 9, 2024 from https://opennetworking.org/wp-content/uploads/2021/05/2021-P4-WS-Vladimir-Gurevich-Slides.pdf.

[55] J. Xing, Q. Kang, and A. Chen, "NetWarden: Mitigating network covert channels while preserving performance," in *Proc. USENIX Security*, 2020, pp. 2039–2056.

[56] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise Time-synchronization in the Data-Plane using Programmable Switching ASICs," in *Proc. ACM SOSR*, 2019, pp. 8–20.

[57] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, and M. C. Chan, "Network Load Balancing with In-network Reordering Support for RDMA," in *Proc. ACM SIGCOMM*, 2023, pp. 816–831.
[58] M. Budiu and C. Dodd, "The $P4_{16}$ Programming Language," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 1, p. 5–14, Sep. 2017.
[59] M. Majkowski, "How to receive a million packets per second," 2015. [Online]. Available: https://blog.cloudflare.com/how-to-receive-a-million-packets
[60] C. H. Song, X. Z. Khooi, D. M. Divakaran, and M. C. Chan, "Revisiting Application Offloads on Programmable Switches," in *Proc. IFIP Networking*, 2022, pp. 1–9.
[61] J. Wu, H. Pan, P. Cui, Y. Huang, J. Zhou, P. He, Y. Li, Z. Li, and G. Xie, "Patronum: In-network Volumetric DDoS Detection and Mitigation with Programmable Switches," in *Proc. ESORICS*, 2024, pp. 187–207.
[62] T. Yang, Y. Zhou, H. Jin, S. Chen, and X. Li, "Pyramid sketch: a sketch framework for frequency estimation of data streams," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1442–1453, Aug. 2017.
[63] C. H. Song, P. G. Kannan, B. K. H. Low, and M. C. Chan, "FCM-Sketch: Generic Network Measurements with Data Plane Support," in *Proc. ACM CoNEXT*, 2020, p. 78–92.
[64] S. Kim, C. Jung, R. Jang, D. Mohaisen, and D. H. Nyang, "A Robust Counting Sketch for Data Plane Intrusion Detection," in *Proc. NDSS*, 2023.
[65] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 254–265, 1998.
[66] The CAIDA UCSD, "Anonymized Internet Traces 2018," https://catalog.caida.org/dataset/passive_2018_pcap, accessed: Jan 10, 2025. [Online]. Available: https://catalog.caida.org/dataset/passive_2018_pcap
[67] ——, "Anonymized Internet Traces 2019," https://catalog.caida.org/dataset/passive_2019_pcap, accessed: Jan 10, 2025. [Online]. Available: https://catalog.caida.org/dataset/passive_2019_pcap
[68] MAWI Working Group, "Mawi working group traffic archive - packet traces from wide backbone," 2024. [Online]. Available: http://mawi.wide.ad.jp/mawi/
[69] Intel Corporation, "Intel® P4 Insight," 2017. [Online]. Available: https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html
[70] 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 *Proc. ACM SIGCOMM*, 2017, p. 15–28.
[71] X. Chen, S. Landau-Feibish, M. Braverman, and J. Rexford, "BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time," in *Proc. ACM SIGCOMM*, 2020, pp. 226–239.
[72] R. B. Basat, X. Chen, G. Einziger, and O. Rottenstreich, "Designing Heavy-Hitter Detection Algorithms for Programmable Switches," *IEEE/ACM Transactions on Networking (ToN)*, vol. 28, no. 3, pp. 1172–1185, 2020.
[73] X. Chen, S. L. Feibish, Y. Koral, J. Rexford, O. Rottenstreich, S. A. Monetti, and T.-Y. Wang, "Fine-Grained Queue Measurement in the Data Plane," in *Proc. ACM CoNEXT*, 2019, p. 15–29.
[74] S. Kim, S. M. M. Mirnajafizadeh, R. Jang, and D. Nyang, "SketchFeature: High-Quality Per-Flow Feature Extractor Towards Security-Aware Data Plane," in *Proc. NDSS*, 2025, p. 1–16.
[75] M. Francisco, B. Ferreira, F. M. Ramos, E. Marin, and S. Signorello, "P4Chaskey: An Efficient MAC Algorithm for PISA Switches," in *Proc. IEEE ICNP*, 2024, pp. 1–6.
[76] C. Wei, S. Tu, T. Hasegawa, Y. Koizumi, K. Ramakrishnan, J. Takemasa, and T. Wood, "Envisioning a Unified Programmable Dataplane to Monitor Slow Attacks," in *Proc. IEEE ICNP*. IEEE, 2024, pp. 1–6.
[77] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *Proc. ACM CoNEXT*, 2015.
[78] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering," in *Proc. USENIX OSDI*, 2016, pp. 467–483.
[79] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "NetChain: Scale-Free Sub-RTT coordination," in *Proc. USENIX NSDI*, 2018, pp. 35–49.
[80] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. Ramos, and A. Madeira, "FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications," in *Proc. NDSS*, 2021.
[81] 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 *Proc. ACM SIGCOMM*, 2018, pp. 561–575.
[82] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "CocoSketch: High-performance Sketch-based Measurement over Arbitrary Partial Key Query," in *Proc. ACM SIGCOMM*, 2021, pp. 207–222.
[83] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen, "Runtime Programmable Switches," in *Proc. USENIX NSDI*, 2022, pp. 651–665.
[84] Y. Yang, L. He, J. Zhou, X. Shi, J. Cao, and Y. Liu, "P4runpro: Enabling Runtime Programmability for RMT Programmable Switches," in *Proc. ACM SIGCOMM*, 2024, pp. 921–937.
[85] M. V. Dumitru, D. Dumitrescu, and C. Raiciu, "Can we exploit buggy p4 programs?" in *Proc. ACM SOSR*, 2020, p. 62–68.
[86] L. Wang, P. Mittal, and J. Rexford, "Data-Plane Security Applications in Adversarial Settings," *SIGCOMM Comput. Commun. Rev.*, vol. 52, no. 2, p. 2–9, 2022.
[87] C. Black and S. Scott-Hayward, "Adversarial Exploitation of P4 Data Planes," in *Proc. IFIP/IEEE IM*, 2021, pp. 508–514.
[88] H. Nam and D. Lim, "Cerberus simulator," 2025. [Online]. Available: https://github.com/hcnam/sim-cerberus
[89] Z. Liu, "Jaqen p4 codebase," 2022. [Online]. Available: https://github.com/Froot-NetSys/Jaqen
[90] J. Xing, "Ripple GitHub Repository," 2021. [Online]. Available: https://github.com/jiarong0907/Ripple
[91] H. Zhou, "Mew-prototype," 2022. [Online]. Available: https://github.com/successlab/Mew-prototype
[92] H. Nam, D. Lim, H. Zhou, G. Gu, and M. S. Kang, "Zenodo repository for Heracles and Shield," 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17490769

Appendix A
Theoretical Analysis of the Shield

In this section, we provide a theoretical analysis of the data structure and decay mechanism of Shield. In particular, we analyze the error bound of the Shield and the effect of time-decaying. Before starting the formal analysis, we first summarize the necessary notation in Table III.

TABLE III
Notation used in the analysis of Shield

Notation	Description
n	total number of flows
i	flow id
f_i	the actual flow size of flow i
$\hat{f}_i$	the estimated flow size of flow i
$C_l[i]$	the counter value for flow i at layer l
L	the number of layers in the sketch
w_l	width (number of counters) in layer l
b_l	bit-width of counters in layer l
r	the ratio of width between layers ($w_i = rw_{i+1}$)
T_l	overflow threshold in layer l, where $T_l = 2^{b_l} - 1$
$h_l(i)$	hash function mapping flow i in layer l
ϵ_l	error bound factor at layer l
t_l	time interval for layer l
k_l	shift amount per layer l
2^{k_l}	decay factor in layer l

A. Analysis of Data Structure of Shield

Let $n_l (1 \leq l \leq L)$ denote the number of distinct flows whose corresponding update operation stops exactly at layer l. Let $\phi_l = n_l/n$ be the ratio of those flows.

Theorem 1. *Let $w_1 = e/\epsilon$ be the number of leaf nodes in Shield and $d = \ln 1/\delta$. Given d pairwise independent hash functions, the count-query $\hat{f}_i$ is bounded by*

$$\hat{f}_i \leq f_i + \epsilon ||f||_1 \tag{4}$$

with probability at least $1 - (\sum_{k=1}^{L} \phi_k r^{k-1})^d \delta$.

Proof. For simplicity, we first prove for SHIELD using one hash function and then extend the result to using multiple hashes.

We define an indicator variable $I_{i,j,k}$, which is 1 if $h_k(i) = h_k(j)$, and 0 otherwise. Due to the pairwise independent hashing, for $i \neq j$,

$$\mathbb{E}[I_{i,j,k}] = \frac{1}{w_k} = \frac{r^{k-1}}{w_1}. \tag{5}$$

We define the variable E_i as follows:

$$E_i = \sum_{k=1}^{L} \frac{n_k}{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j I_{i,j,k} = \sum_{k=1}^{L} \phi_k \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j I_{i,j,k}. \tag{6}$$

Obviously, E_i is a nonnegative random variable. E_i reflects the expectation of the error caused by the collisions happening at all layers when querying f_i (i.e., $\hat{f}_i = f_i + E_i$). The expectation of E_i is calculated as follows:

$$\mathbb{E}[E_i] = \mathbb{E}\left[\sum_{k=1}^{L} \phi_k \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j I_{i,j,k}\right] \tag{7}$$

$$= \sum_{k=1}^{L} \phi_k \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j \, \mathbb{E}[I_{i,j,k}] \tag{8}$$

$$= \sum_{k=1}^{L} \phi_k \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j \frac{r^{k-1}}{w_1} \tag{9}$$

$$= \frac{1}{w_1} \sum_{k=1}^{L} \phi_k r^{k-1} \sum_{\substack{j=1 \\ i \neq j}}^{n} f_j \tag{10}$$

$$\leq \frac{1}{w_1} ||f||_1 \sum_{k=1}^{L} \phi_k r^{k-1}. \tag{11}$$

Then, by the Markov inequality, we get

$$\Pr(E_i \geq \epsilon ||f||_1) \leq \Pr\left(E_i \geq \epsilon \frac{w_1}{\sum_{k=1}^{L} \phi_k r^{k-1}} \mathbb{E}[E_i]\right) \tag{12}$$

$$\leq \frac{\sum_{k=1}^{L} \phi_k r^{k-1}}{\epsilon w_1} \tag{13}$$

$$= \frac{\sum_{k=1}^{L} \phi_k r^{k-1}}{e}. \tag{14}$$

Extension of the result to multiple hashes is trivial because all hashes are independent. Using independence,

$$\Pr(E_i \geq \epsilon ||f||_1) \leq \left(\frac{\sum_{k=1}^{L} \phi_k r^{k-1}}{e}\right)^d = \left(\sum_{k=1}^{L} \phi_k r^{k-1}\right)^d \delta. \tag{15}$$

□

B. Analysis of Decay function of SHIELD

The decay mechanism of SHIELD follows *discrete exponential decay* with base 2, implemented via bit shifts. The exponential decay can be represented as

$$\frac{dN(t)}{dt} = -\lambda N(t), \tag{16}$$

where $N(t)$ is the value at time t and $\lambda (\lambda > 0)$ is a decay constant (or rate constant). The solution to this equation is

$$N(t) = N(0) \cdot e^{-\lambda t}, \tag{17}$$

where $N(0)$ is the initial value.

From this solution, at each decay interval, the counter value C of SHIELD is updated as:

$$C_l[i] = \left\lfloor \frac{C_l[i]}{2^k} \right\rfloor, \tag{18}$$

where k is the shift amount per layer and $C_l[i]$ is the counter value for flow i in layer l. Then, SHIELD follows the behavior of exponential decay with base 2:

$$C_l[i](t) = C_l[i](0) \cdot 2^{-kt}. \tag{19}$$

APPENDIX B
SIMULATION OF HERACLES AGAINST OTHER DOS DEFENSES

We have developed a new attack-defense simulation framework that adheres to the implementation and constraints of the Tofino hardware, enabling the systematic evaluation of the generalizability of HERACLES attack to other defense techniques designed for programmable switches. We open-source our simulation framework at [88]. With this evaluation framework, we implemented Poseidon [12], Jaqen [13], Ripple [24], and Mew [14] based on their original paper and open-sourced implementation, when available. Note that our simulation-based evaluation is an inevitable approach due to the limited access to the P4 programs or multi-switch hardware setups of some existing DoS mitigation systems; e.g., Poseidon is not open-sourced, Jaqen is not fully open-source yet [89]; Ripple [90] and Mew [91] are open-sourced but they require multiple programmable switches to run their implementation. Our simulation is confirmed to faithfully reproduce the behavior of the original P4 programs on Tofino hardware; see how the simulated attack results against Cerberus Figure 16(a) and Figure 17(a) well align with the real hardware experiment results in Figure 5(a) and Figure 5(b), respectively.

In our simulation framework, we tested the inference capabilities and attack feasibility of HERACLES attack. We conducted 100 probing processes, each with a different threshold and time window configuration. Also, we used a setting that added Gaussian noise (standard deviations of 1/3 seconds) to the average propagation time observed in the experiment in §III-A.

Figure 16(a) shows the threshold inference accuracy of Poseidon and Jaqen (Cerberus too, for comparison with its hardware experiment in Figure 5(a)), and Figure 16(b) shows Ripple and Mew in our simulation framework, respectively. We

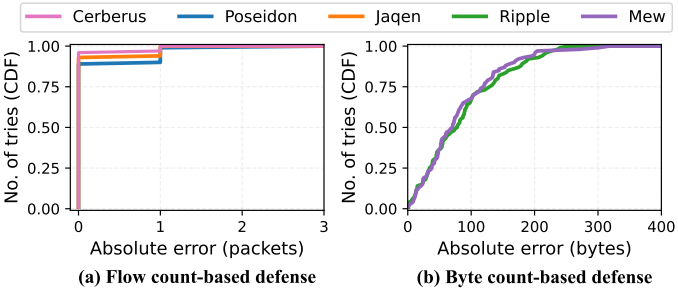


Fig. 16. Accuracy of threshold inference phase of the HERACLES attack against several DoS defense techniques in simulation.

configured the defense mechanism for the UDP flood attack of Poseidon and Jaqen as `rlimit()`, and `BlockList()`, respectively. For link flooding attack (LFA) defense systems (i.e., Ripple and Mew), we configured byte count-based threshold for the Coremelt attack detector. Also, for byte count threshold detection, we conducted two distinct probing processes: (1) infer approximate threshold with 512-byte packets and (2) infer accurate threshold with 64-byte packets. Figure 16(a) and (b) show the effectiveness of our threshold inference process. Our results show that the probing process still accurately measures threshold with an absolute error of 0 in 88 out of 100 processes in the flow count-based defense systems, as shown in Figure 16(a), and under 200 bytes of absolute error with 92 out of 100 as illustrated in Figure 16(b). This confirms that our threshold inference process of HERACLES is effective against existing DoS mitigation systems.

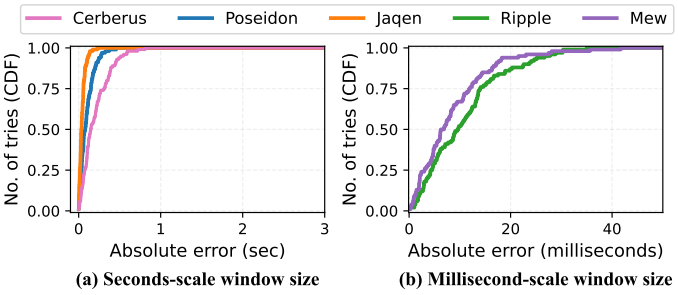


Fig. 17. Accuracy of time window inference phase of the HERACLES attack against several DoS defense techniques in simulation.

Figure 17(a) shows the time window inference attack accuracy of Poseidon and (Cerberus too, for comparison with its hardware experiment in Figure 5(b)), and Figure 17(b) shows Ripple and Mew in our simulation framework, respectively. Note that the time window of the link flooding attack defense system is sub-second (e.g., 100 to 500 ms). Therefore, we conduct more aggressive packet sending with high PPS when testing Ripple and Mew. We confirmed that 98 out of 100 in a seconds-scale time window configuration, and 86 out of 100 in a milliseconds-scale time window configuration correctly measure the period and number of the time window. In the case of inferring the start time of the time window, as shown in Figure 17, our probing process can accurately measure the start time with an absolute error of less than 1 second in seconds-scale configuration and 50 milliseconds in milliseconds-scale time window configuration for all cases where the period

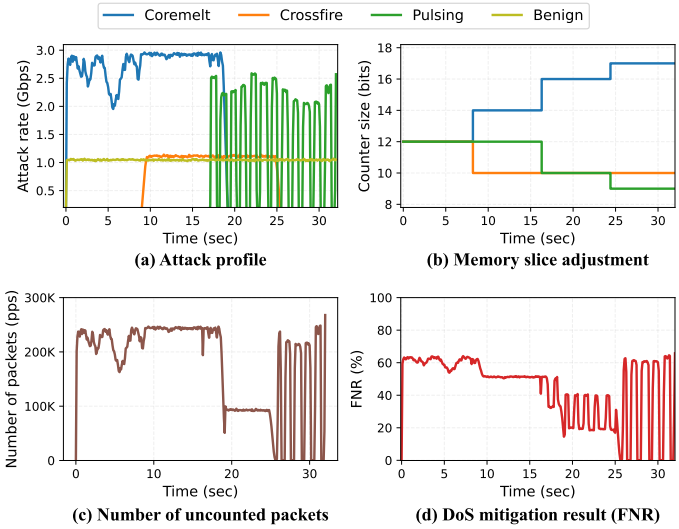


Fig. 18. Simulated memory squeezing attack against Mew [14].

and number of the time window are correctly inferred. This confirms that our threshold inference process is applicable to existing DoS mitigation systems.

Additionally, for Mew [14], we tested the memory squeezing attack feasibility against the attack detector mechanism. Figure 18 shows the attack-defense simulation result in Mew. In our simulation, Mew supports 3 types of defenses (Coremelt [5], Crossfire [6], and Pulsing [36]) in the same register. For this, the `pattern_state` register of Mew shares 3 different types of per-flow in-network measurement tasks: (1) '*byte count*' for Coremelt detection, (2) '*low rate flow count*' for Crossfire detection and (3) '*fluctuation count*' for Pulsing attack detection (see [14] and [91] for their implementation details). As shown in Figure 18(a), the adversary changes different attack profiles to manipulate Mew's adaptive memory slice mechanism. The attack profile in Figure 18(a) is designed to reduce slices allocated to the Crossfire and Pulsing attacks.

Figure 18(b) shows that the memory slice adjustment mechanism of Mew reduces the allocated counter size of the target task (Crossfire/Pulsing) while the memory size of non-target tasks (Coremelt) increases. Figure 18(c) and (d) illustrate the effectiveness of memory squeezing against Mew. As shown in Figure 18(c), due to the squeezed counter size, Mew undercounts malicious traffic. Consequently, Mew fails to fully defend against attacks, blocking only 30% of malicious traffic, as shown in Figure 18(d). This simulation result shows that our memory squeezing attack strategy is applicable to other adaptive resource-adjusting data-plane applications.

APPENDIX C
ARTIFACT APPENDIX

A. *Description & Requirements*

Our artifact contains several proof-of-concept implementations of DPDK-enabled Cerberus, and SHIELD that demonstrate the functionality of the techniques described in the paper. For the simple test, we provide the *Packet Testing Framework (PTF)* – a Python-based dataplane test framework – for SHIELD.

1) How to access: The artifact is available on both our GitHub [50] and Zenodo [92]. In this artifact appendix, we explain based on the GitHub version.

2) Hardware dependencies: Testing the functionality of SHIELD using the PTF test only requires a commodity desktop machine (with an x86-64 CPU with 8 cores and 16 GB of RAM) and software dependencies. However, testing full line-rate performance of SHIELD with large-volume traffic requires a Tofino programmable switch (such as Edgercore Wedge-100BF-32X). Also, one (with 2 ports) or two NIC cards (with 1 port) with a minimum data rate of 40Gbps is required, wired with two QSFP (e.g., QSFP28) cables with a programmable switch. Note that the QSFP port of the switch and the NIC connect with compatible transceivers with each vendor, or use cables that support dual-compatibility for both the switch and NIC vendors.

3) Software dependencies: We provide two distinct software dependencies for testing:

Testing with Tofino model. This option emulates the compilation and execution of P4 code through the Tofino Model (Intel-provided Tofino emulator), without an actual programmable switch. The Tofino model is a tool for developing and debugging TNA-based P4 programs. In most cases, the P4 program can operate on real hardware if it is executable on the Tofino model. However, the Tofino model can only validate the functionality due to the slow packet processing speed. Therefore, certain PTF tests are not possible due to the packet processing performance limitations. Also, the Tofino Native Architecture (TNA)-based $P4_{16}$ code is usually tailored to specific software environments. Thus, we recommend running the Tofino Model of Open P4 Studio on the Ubuntu 22.04 LTS system that we already tested. The PTF test requires Python 3.10 (will be installed with Open P4 Studio) and the `scapy` package.

Testing with a real programmable switch. This option requires an actual Tofino ASIC-powered programmable switch. In this case, proper Board Support Packages (BSP) and P4 Studio (bf-sde), which are under Intel's NDA, are required. The operating system of our switch is Ubuntu 22.04 LTS, and we recommend Intel P4 Studio (and bf-sde) version 9.13.1 (or higher), because the TNA-based P4 code is highly tailored to a specific version of bf-sde.

We also use DPDK (v24.11) on a programmable switch with `vfio-pci`. It is possible to use `igb_uio`, but we recommend using `vfio-pci`. To use `vfio-pci`, enable IOMMU in the switch BIOS, and then make sure to load the VFIO kernel module. Note that we are using `bf_kpkt` driver for using DPDK with the Tofino programmable switch.

4) Benchmarks: None

B. Artifact Installation & Configuration

Testing with Tofino model. Begin by cloning the Open Tofino GitHub repository,[4] because our artifact requires the Tofino Model to emulate the behavior of the Tofino ASIC. Installing P4 Studio takes a lot of time. SDE installation and setup takes more than 2 hours, depending on the hardware.

```
# cd ~
# git clone https://github.com/P4ica/tools.git
# git clone https://github.com/p4lang/open-p4studio.git
# git clone https://github.com/NetSP-KAIST/shield.git
# cd open-p4studio
# git submodule update --init --recursive
# cp ../shield/model-profile.yaml ./p4studio/profiles/
# ./p4studio/p4studio profile apply ./p4studio/profiles/model-profile.yaml
```

Then, set environment variables and create virtual network interfaces for the Tofino model.

```
# ./create-setup-script.sh > ~/setup-open-p4studio.bash
# source ~/setup-open-p4studio.bash
# $SDE_INSTALL/bin/pip3.10 install scapy
# cd $SDE_INSTALL/bin
# ln -s p4c bf-p4c
# cd $SDE
# echo "-DP4C=$SDE_INSTALL/bin/p4c" > ./bf-sde-open-p4studio.manifest
# sudo ${SDE_INSTALL}/bin/veth_setup.sh 128
```

Use '`sudo ${SDE_INSTALL}/bin/veth_teardown.sh`' command to remove virtual interfaces after the experiment finishes.

Testing SHIELD with a real programmable switch. Install P4 Studio and proper bf-sde based on Intel's instruction with proper `bsp` software and setup path with `set_sde.bash` script that Intel provided. Then load `bf_kpkt` module.

```
# sudo $SDE_INSTALL/bin/bf_kpkt_mod_load $SDE_INSTALL
# sudo ip link set `basename /sys/module/bf_kpkt/drivers/pci\:bf/*/net/*` up
```

Then build and run the P4 program and set the port configuration. Note that the port configuration can differ from below:

```
# cd $SDE
# ../tools/p4_build.sh ../shield-skech/shield.p4
# ./run_switchd.sh -p shield
bfshell> ucli
bf-sde> port-add 11/- 100G RS
bf-sde> port-add 12/- 100G RS
bf-sde> port-enb 11/-
bf-sde> port-enb 12/-
bf-sde> port-add 33/- 10G NONE
bf-sde> an-set 33/- NONE
bf-sde> port-enb 33/-
bf-sde> pm show
```

The command output of '`pm show`' should be like below:

```
-----+----+---+----+------+----+--+--+---+---+---+------+----------+----------+-
PORT |MAC |D_P|P/PT|SPEED |FEC |AN|KR|RDY|ADM|OPR|LPBK  |FRAMES RX |FRAMES TX |E
-----+----+---+----+------+----+--+--+---+---+---+------+----------+----------+-
11/0 |13/0| 44|1/44|100G  | RS |Au|Au|YES|ENB|DWN| NONE |         0|         0|
12/0 |12/0| 36|1/36|100G  | RS |Au|Au|YES|ENB|DWN| NONE |         0|         0|
33/0 |32/0| 64|1/64|10G   |NONE|Au|Au|YES|ENB|DWN| NONE |         0|         0|
33/1 |32/1| 65|1/65|10G   |NONE|Au|Au|YES|ENB|DWN| NONE |         0|         0|
33/2 |32/2| 66|1/66|10G   |NONE|Au|Au|YES|ENB|DWN| NONE |         0|         0|
33/3 |32/3| 67|1/67|10G   |NONE|Au|Au|YES|ENB|DWN| NONE |         0|         0|
```

Next, run the initial setup script and the control plane with another shell:

```
# cd ./shield-sketch
# sudo -E $SDE_INSTALL/bin/python3.10 ./setup.py
# sudo -E $SDE_INSTALL/bin/python3.10 ./run_cp_scapy.py
```

C. Experiment Workflow

Our repository is organized as follows:

- `cerberus`: data/control-plane code for reproduced Cerberus and Cerberus++. The P4 codebase and Python/DPDK versions of control plane implementations are included. Testing this artifact must require real Tofino hardware.

[4] https://github.com/p4lang/open-p4studio

- `shield-sketch`: data/control-plane code for SHIELD. P4 code, Python control plane, and PTF tests are included. Testing this artifact must require real Tofino hardware.
- `shield-model`: data/control-plane code for SHIELD for Tofino model. The P4 codebase, Python control plane, and PTF tests are included. This artifact appendix mainly focuses on this directory.

D. Major Claims

- **(C1):** The SHIELD uses a hierarchical layered register design. When a lower-layer counter overflows, the excess is forwarded to the next higher layer. This is proven by the experiment (E1) that verifies the functionality of SHIELD with PTF tests.
- **(C2):** The SHIELD refreshes or decays the values inside the register as the time window changes. The SHIELD allows each layer to have a different time window to refresh the register value. This is proven by the experiment (E1) that verifies the functionality of SHIELD with PTF tests.

E. Evaluation

1) Experiment (E1): [Verify Functionality] [10 human-minutes + 5 compute-minutes]: after setting up our environment, build the P4 program and run the Tofino model. Then execute PTF tests to validate the functionality of SHIELD. Our PTF test is composed as follows:

- `overflow.py`: Send packets affecting each layers. This PTF test will verify our major claim (C1).
- `decay.py`: Send packets affecting layer 1 and 2, and see whether the values on layer 2 and 3 are well decayed. This PTF test will verify our major claim (C2).

[How to] We provide Python scripts for the PTF test.

[Preparation] First, build and run the Tofino model:

```
# source ~/setup-open-p4studio.bash
# cd $SDE
# ../tools/p4_build.sh ../shield/shield-model/layered_cms.p4
# ./run_tofino_model.sh -p layered_cms \
  -f ../shield/shield-model/ptf-tests/ports_model.json
```

The output of this command explains the behavior of the P4 program. The terminal output of the Tofino model may contain errors, but this is not a problem. These errors occur because our script sends multiple batch commands – that include some unsupported operations (e.g., clear all) against specific tables or registers – to all tables and registers.

Then, type the following in another terminal. This runs the P4 program by launching the driver:

```
# source ~/setup-open-p4studio.bash
# cd $SDE
# ./run_switchd.sh -p layered_cms
```

The output of this command is `bf-shell`.

[Execution] Run the PTF test in another terminal:

```
# source ~/setup-open-p4studio.bash
# cd $SDE
# export PKTPY=false
# ./run_p4_tests.sh -p layered_cms \
  -f ../shield/shield-model/ptf-tests/ports_model.json \
  -t ../shield/shield-model/ptf-tests
```

[Results] The PTF test script will test the functionality of our implementation. Our PTF test script will test:

- `decay` : `DecayLayer2`
- `overflow.SendToLayer1`
- `overflow.SendToLayer2`

If it executes without any errors, it implies our code works on the real Tofino switch hardware as well. Due to the packet processing speed of the Tofino model, tests such as `DecayLayer3` or `overflow.SendToLayer3` requires real hardware.

F. Notes

To evaluate the hardware implementation version of Cerberus/Cerberus++ and SHIELD, please refer to `README` in each directory. Each directory includes implementation and steps that are necessary for executing Cerberus and SHIELD, except for the attack traffic generator. The attack traffic generator is not open-sourced for ethical reasons. Again, note that the hardware implementation version has strict hardware and software dependencies and requirements.