SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes

Sophia Yoo
Princeton University
sophiayoo@princeton.edu

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

Jennifer Rexford
Princeton University
jrex@cs.princeton.edu

Abstract

Despite decades of mitigation efforts, SYN flooding attacks continue to increase in frequency and scale, and adaptive adversaries continue to evolve. Meanwhile, volumes of benign traffic in modern networks are also growing rampantly. As a result, network providers, which run thousands of servers and process 100s of Gbps of traffic, find themselves urgently requiring defenses that are secure against adaptive adversaries, scalable against large volumes of traffic, *and* highly performant for benign applications. Unfortunately, existing defenses local to a single device (e.g., purely software-based or hardware-based) are failing to keep up with growing attacks and struggle to provide performance, security, or both. In this paper, we present SmartCookie, the first system to run cryptographically secure SYN cookie checks on high-speed programmable switches, for both *security* and *performance*. Our novel split-proxy defense leverages emerging programmable switches to block 100% of SYN floods in the switch data plane and also uses state-of-the-art kernel technologies such as eBPF to enable *scalability* for serving benign traffic. SmartCookie defends against adaptive adversaries at two orders of magnitude greater attack traffic than traditional CPU-based software defenses, blocking attacks of 136.9 Mpps *without packet loss*. We also achieve 2x-6.5x lower end-to-end latency for benign traffic compared to existing switch-based hardware defenses.

1 Introduction

Distributed Denial-of-Service (DDoS) attacks have been studied for decades, but despite this rich history, volumetric attacks are still an important and unsolved problem today. In 2023, attacks increased by over 300% [5, 44], with downtime costing companies an average of $20,000-$40,000 hourly [25, 32]. One driving factor behind this growth is the widespread availability of DDoS-for-hire services, costing as little as $10 hourly and making it increasingly easy to launch attacks [34]. In recent years, DDoS attacks have also shaped political landscapes and played major roles in cyber warfare [23, 24, 29, 39].

One of the most common DDoS attacks, namely *SYN floods*, consume server memory until the server is forced to drop benign traffic [37, 38, 58]. SYN floods constituted up to 94.7% of all DDoS attacks in 2020 [49] and continue to remain a critical threat today [50]. Additionally, benign traffic volumes also continue to grow exponentially, reaching staggering loads of up to hundreds of Gbps in cloud-provider networks [63].

To respond to growing threats and traffic volumes, network providers (e.g., cloud providers, enterprise networks, ISPs) urgently require *scalable* defenses that block volumetric attacks *without compromising security* against adaptive adversaries or *degrading application performance* [21, 40, 58]. In other words, they need defenses with three key requirements: **security** (blocking attacks from adaptive adversaries), **scalability** (handling large amounts of benign and attack traffic), and **performance** (low latency for benign clients).

While state-of-the-art SYN-flooding defenses using *SYN cookies* have been proposed and standardized for many years, existing solutions have failed to simultaneously provide security, scalability, *and* performance [2, 3, 51, 67, 72]. Designing and practically implementing modern SYN-flooding defenses that meet each of these requirements is particularly challenging. Compared to defenses against many other volumetric attacks (e.g., ACK floods, RST floods, UDP amplification), SYN-flooding defenses cannot simply drop, rate-limit, or ignore unsolicited traffic. Doing so might cause denial of benign client requests, particularly since adversaries often spoof attack packets using legitimate addresses. Instead, defenses must identify and block large-scale attacks (requiring costly compute) while keeping per-flow state to track large numbers of verified connections (requiring large chunks of memory).

Server-Based Solutions. Server-based SYN cookie defenses are the time-honored way to provide security and can scalably handle *benign traffic* by default [2, 3, 67]. However, under large-scale attacks, software-based packet-processing and cryptographic cookie computation incur high overheads and lead to CPU exhaustion (§8). Ultimately, once the server's limited CPU capacity is overwhelmed by *volumetric attacks*, these defenses collapse and again result in DoS.

Switch-Based Solutions. High-speed programmable hardware switches present a unique opportunity to overcome this additional attack vector on end host CPU consumption. Switches provide packet-processing at orders of magnitude faster rates, and thus have the potential to execute the required cryptography more performantly than end hosts. However, purely switch-based solutions [51, 72] must operate under strict hardware constraints, including limited compute and limited memory. First, cryptographic primitives are not natively supported in switches and are computationally expensive. As a result, existing switch-based defenses use the insecure CRC32 to compute SYN cookies [51, 72], *abandoning security* and defeating the purpose of the SYN cookie check (§2.4). Additionally, because switches have only tens of MBs of memory [48], memory-intensive SYN-flooding defenses which track per-connection state *struggle to scale* [72] or *degrade application performance* [51] (§2.5).

Division of Labor to the Rescue. To overcome the limitations of existing solutions, we propose **SmartCookie**: a novel *split-proxy* defense that leverages collaborative programmable data planes on hardware and software targets. Our design *intelligently* partitions the defense workload to maximize the benefits of both switch-based and server-based defenses, while minimizing the limitations of each approach.

Key Insight. Our key insight for motivating SmartCookie's novel division of labor is two-fold. Switches are highly performant and have potential to quickly and *securely* block volumetric attacks, but they are memory-limited and scale poorly at keeping exact state for verified flows. This makes switches excellent as a first line of defense, but they should not be required to keep per-flow state. Meanwhile, servers enjoy superior memory resources, but are prohibitively slow at packet processing. This makes them ideal for exactly tracking *benign flows*, without the burden of blocking *attacks*.

At a high level, SmartCookie takes traditional SYN cookie defense elements, refactors them, splits them between a co-designed *switch agent* (running on the switch data plane) and *server agent* (running on the Linux kernel data plane), and stitches switch agent and server agent together with a collaborative protocol. We identify three key elements of existing defenses: **F1) SYN cookie checks**, **F2) TCP sequence number translations**, and **F3) keeping state for verified connections**. Following our key insight, we refactor and map defense elements **F1-F3** to switch and server as follows: SmartCookie switch agent *securely* performs cookie checks to quickly stop bad traffic (**F1**) and *approximately* tracks verified connections (**F3.A**) instead of keeping exact state, while SmartCookie server agent handles sequence number translations (**F2**) and *exactly* tracks verified connections (**F3.B**).

SmartCookie switch agent maintains approximate state using compact data structures with well-defined accuracy guarantees (§6.3). Note that in contrast to prior work, our defense does not require exact state at the switch, but *still achieves overall exact defense results*, due to its split design.

Contributions and Roadmap. This work makes the following contributions:

- The first system to run cryptographically secure SYN cookies on programmable switches, using robust hashes instead of the insecure CRC32 used in prior works (§5).
- The first split-proxy SYN-flooding defense for modern programmable data planes, using in-switch compact data structures for memory scalability and server-side eBPF for immediate deployability (§6).
- An end-to-end SmartCookie prototype with a *switch agent* implemented in P4 on Tofino switches and a *server agent* written in eBPF on Linux servers, which is resilient against attack rates of 136.9 Mpps (about 92Gbps) *without any packet loss*, with potential for easily reaching even higher rates (§8).
- 19x-105x throughput improvement over *software* solutions resulting from offloading of cookie computation and verification to high-speed switch hardware (§8.2) and 48-84% latency improvement with our novel split design over existing *hardware-only* designs (§8.3).

We present limitations of existing defenses in §2 and describe the threat model, problem setting, and architecture of SmartCookie in §3 and §4. We include a security analysis in §7, discuss additional features and future directions in §9, present related work in §10, and conclude in §11.

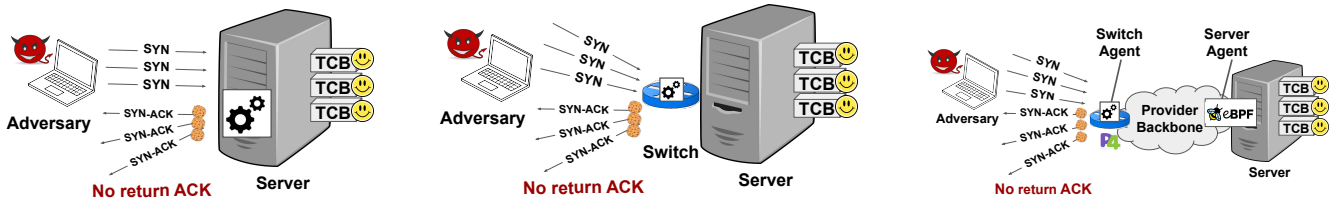
Ethics Statement. This work raises no ethical issues.

2 Limitations of Existing Defenses

2.1 Limitations of Server-based Defenses

Traditional server-based SYN-flooding defenses use *SYN cookies* to protect server memory, encoding state normally saved to server-side Transmission Control Blocks (TCB) during connection setup in a cryptographically computed cookie instead [2] (Figure 1a). The server uses the SYN cookie as the initial sequence number (ISN) that is placed in the sequence number field (`seq_no`) of the SYN-ACK packet sent to the client. Clients automatically return the cookie in the acknowledgment number of the final ACK of the TCP handshake, as $\texttt{ack\_no} = \texttt{seq\_no} + 1$. This returned cookie is then verified by the server.

In server-based defenses, the server cryptographically computes and verifies cookies and performs packet processing for every potential connection, introducing an added attack vector on server compute resources. Because these defenses run on general-purpose CPUs, they can easily be overwhelmed under heavy attack loads, due not only to cookie computation but also to software-based packet-processing overheads (§8, Table 1). Thus, server-based defenses struggle to scalably stop volumetric attacks, degrading application performance and again resulting in denial-of-service for benign clients.



(a) Server-based defense. (b) Switch-based defense. (c) SMARTCOOKIE: a split defense.

Figure 1: SMARTCOOKIE versus server-based and switch-based defenses.

2.2 Limitations of SYN Cookie Proxies

To protect the server from CPU exhaustion, the SYN cookie mechanism can run in a proxy placed in the NIC of the server or in a switch near the server (Figure 1b). Proxies perform cookie computation and verification in place of the server, only establishing a connection with the server for verified clients and thus defending against attacks that cause excessive consumption of server CPU cycles [33, 60]. Defense proxies also monitor previously verified connections to determine which packets require cookie checks.

However, this approach has fundamental drawbacks as well. First, proxies incur overheads related to connection setup. Specifically, after a proxy verifies a client, it must perform a second, separate proxy-to-server setup handshake to establish the connection from the server's point-of-view. This increases latency since the proxy must buffer any packets received from the client while it establishes the server-side connection. Additionally, proxies need to translate TCP sequence numbers between client and server, because there is a mismatch between the ISN chosen by the proxy as the SYN cookie value and the ISN chosen randomly by the server [60]. This forces the proxy to keep exact per-flow state to track verified connections (in addition to the per-flow state kept by the server), and the proxy must perform costly sequence number translations on every packet throughout the remainder of the connection. We note the SYN proxy design also relies on assumptions of symmetric routing, since all packets must pass through the proxy in both directions for sequence number translations.

2.3 Opportunities of Programmable Switches

Programmable switches are similar in price to fixed-function enterprise-grade switches. They are an attractive option for high-speed, flexible, and cost-efficient defenses against large-scale attacks [18, 19, 51, 72, 73]. While it is clear that switches have significantly more potential than servers for a performant and robust defense, we argue that programmable switches also showcase concrete benefits over SmartNICs as well.

Target Choice. Although some state-of-the-art SoC-based SmartNICs offer larger memory banks and cryptographic accelerators for specific primitives (e.g., MACsec/IPsec encryption) [55], switches have faster packet processing and offer better performance to cost/power ratios than alternative SmartNIC approaches [46, 51, 55]. For our target setting of large cloud enterprises, gateway switches can save bandwidth in the network core by blocking attacks directly at the network edge, before malicious traffic reaches the internal network and hosts' NICs. Still, SMARTCOOKIE could be deployed on SmartNICS, but given their lower capacity, these are better for smaller edge networks, and we choose to target switches.

2.4 Security Limitations of Switch Defenses

Prior works have leveraged high-speed switches as SYN cookie proxies in the network [51, 72]. Unfortunately, these systems have a crippling security flaw: they use the insecure CRC32 checksum as the "hash" for computing SYN cookies, abandoning security against adaptive adversaries.

The Need for a Secure Hash. SYN cookies are computed with a hash of the connection 4-tuple (source/destination IP addresses and port numbers), along with some secret key and a timestamp, so an adversary cannot perform a replay attack with the cookie at a later time. For security, this hash must be cryptographically robust, such that an adversary cannot easily craft a cookie that would pass as legitimate during a cookie check. Otherwise, the adversary could send many forged cookies to the victim, launching an attack more powerful than the original one: after verifying forged cookies, the victim must also prepare for new connections and allocate memory, wasting *both* compute and memory resources.

Insecurity of CRC. CRC is an error-detection checksum, not a cryptographic hash. When incorrectly used as a hash, it can be trivially cracked by an adaptive adversary performing key recovery and cookie forging attacks [70]. There are two broad attack vectors against CRC that can result in breaking the cookie defense: collision induction and nonce deduction.

Collision Induction. An adaptive adversary can exploit the fact that CRC is not collision-resistant to construct hash collisions. Without assuming security by obscurity, adversaries with knowledge of a CRC-based defense can simply send probe packets with different inputs to uncover a hash collision. Note the adversary has constant feedback on CRC outputs, as the defense must respond to all SYN packets.

	Secure Hash	High-Throughput	Non-Disruptive	Scales Beyond Switch Memory
Server-Based Cookies [2, 57, 60]	✓	✗	✓	–
Poseidon [72]	✗	✓	✓	✗
Jaqen [51]	✗	✓	✗	✓
SMARTCOOKIE	✓	✓	✓	✓

Table 1: Compared to existing defenses, SMARTCOOKIE is secure, high-throughput, non-disruptive, and scalable.

Nonce Deduction. Due to its linear nature, CRC is vulnerable to nonce deduction; an adaptive adversary can trivially crack CRC-based cookies in a few simple steps (we refer interested readers to §A for details). The simplicity and effectiveness of this attack leaves the door wide open for adaptive adversaries to completely break the CRC-based SYN cookie defenses of both [51] and [72]. In contrast, SMARTCOOKIE *securely* computes SYN cookies in the programmable data plane using a robust hash with strong security guarantees [8] (§5).

2.5 Performance Limitations of Switch Defenses

Existing switch-based defenses also struggle to provide memory scalability and good application performance [51, 72].

Poor Scalability. Poseidon [72] keeps per-flow state in the switch to perform sequence number translations for ongoing flows. Given limited switch memory, this simply cannot scale, as processing hundreds of thousands of flows at network speeds is a memory-intensive task (Table 1). In contrast, SMARTCOOKIE avoids keeping exact state in the switch by offloading it to the server, allowing the defense to scale (§6).

Disruptive Performance. Jaqen [51] also avoids keeping per-flow state on switches. However, it does so by disruptively forcing all benign clients to undergo a reset during connection setup, even after passing the SYN cookie check (Table 1). After this reset, clients eventually attempt a second handshake, which the Jaqen proxy allows through to the server, with some probability of false positive error. This enables server and client to directly choose sequence numbers without involving the Jaqen proxy. Unfortunately, Jaqen's design incurs extra latency (1-2 round-trip times) for benign connections, an especially undesirable performance penalty for clients across wide-area networks. In contrast, SMARTCOOKIE is transparent to clients, requiring a *single* handshake and maintaining good application-level performance (§8).

3 SMARTCOOKIE Problem Setting

The problems that state-of-the-art defenses experience against large-scale SYN floods are challenging to resolve, because they arise from the inherent hardware constraints of available targets and from the threat model experienced by network providers tasked with providing security, scalability, and performance. SMARTCOOKIE overcomes these challenges and presents a practical defense for large network operators, who control backbone switches and servers in their network. This setting opens unique opportunities for a division of labor that SMARTCOOKIE intelligently exploits (Figure 1c).

3.1 Threat Model

There are four key players in our threat model: clients, adversaries, switches, and servers. Switches and servers are controlled by the same network operator, which seeks to protect its servers from resource strain, while concurrently protecting network bandwidth. Under the same administrative authority, switches and servers can safely cooperate.

Clients. Client communications to any of the servers hosted by the provider backbone should be protected from SYN flooding disruption. Clients should also not experience degraded performance as a result of any deployed defense.

Adversaries. Our threat model focuses on *asymmetric* SYN-flooding attacks, where adversaries require orders of magnitude fewer resources than defenses. Asymmetric attacks are generally regarded as more challenging to defend against [17, 26], and their lower cost is likely what makes them so prevalent in the wild [50]. In our threat model, adversaries can spoof source IP addresses or utilize a limited number of compromised devices to send a flood of connection handshake requests that appear to be from unique clients and must be individually handled [7]. When spoofing, adversaries cannot gain feedback from the defense, as response packets to spoofed addresses never reach the adversary. Adversaries can also send some probe packets without spoofing to observe the resulting cookies and attempt to crack the cookie hash, as well as launch replay attacks using earlier cookies (§2.4). We assume the adversary can send attack traffic at up to Tbps rates. The adversary does not have physical access to the network switches or servers hosted by the provider backbone, and traffic from the adversary cannot reach the server without traversing a participating switch. In other words, the adversary cannot tamper with packets between the switches and servers internal to the provider backbone.

Switches. The switches are programmable, high-speed network hardware capable of Tbps processing rates. They are controlled by the same network provider and deployed at the network edge. Since the switches and servers are under the same *centralized* operator, we assume the communication channel between the switches and the servers is secure.

Servers. Physical servers, hereafter simply called servers, are owned, operated, and trusted by the network provider, and thus can be modified. Trusted defense modules run on the servers, but for immediate deployability, changes to the TCP/IP stack of the server's Linux kernel are not required. However, modest changes to the server TCP/IP stack can open further defense opportunities in the future (see §D). We

note that tenant VMs running on the physical servers are not trusted and thus are *unmodified* by the network provider. Also, in order to not affect the performance of applications running on the server, it is critical that any defense mechanism does not consume excessive CPU cycles.

3.2 Challenges

Modern programmable switches support flexible packet processing optimized for Tbps speeds [14,42,45]. They exert fine-grained control over packet forwarding in the data plane with line-rate throughput guarantees, but do so at the cost of strict constraints enforced by the underlying hardware [12, 13, 42]. Thus, there are fundamental challenges to overcome to realize the potential of a high-speed switch defense.

Challenge 1: Limited In-Switch Programming Model for Cryptographic Operations. Most programmable switches do not natively support any cryptographic primitives, and they can only process packets with a limited number of available operations (e.g., addition, subtraction, and XOR, but no multiplication or division). However, even more fundamentally, to guarantee high-throughput, switches use a pipeline model with only a limited number of stages for packet processing. Operations can also only be performed concurrently in a stage if there are no dependencies between operations. Finally, the output of one computation cannot be used until the following pipeline stage, making it difficult to fit all the necessary operations for cryptographic SYN cookie computation within the limited number of available stages. Thus, even if cryptographic accelerators were introduced on hardware switches, computing secure cookies within the computational constraints and limited stages is challenging, and could break the performance of the switch if done naively. We show how SMARTCOOKIE overcomes these challenges in §5.

Challenge 2: Limited In-Switch Memory for Per-Connection State. The amount of available data plane memory is limited, and it must be shared with other applications (e.g., routing tables) running on the switch. To perform sequence number translations and maintain the correctness of packet-processing, a naive defense would need to keep cumbersome per-flow state on the order of 6 bytes per connection (e.g., the 32-bit sequence number and hashed connection 5-tuple key). Unlike other data plane applications that require keeping significantly less state (e.g., a small 6-bit version number along with a hashed 5-tuple key) [52], the amount of state that must be kept by a defense cannot be further compressed with a hash digest, as this would cause information loss and break connections. Given available memory is on the order of tens of MBs, keeping this amount of state for every verified flow in a large network provider cannot scale to even hundreds of thousands of connections [51], and thus we simply cannot afford to allocate switch memory for each ongoing TCP flow. We show in §6 how SMARTCOOKIE intelligently partitions the defense to gracefully handle this challenge.

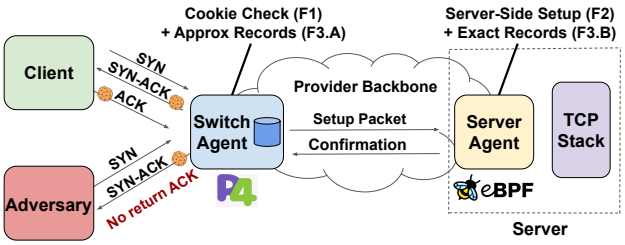


Figure 2: SMARTCOOKIE's split-proxy architecture.

4 SMARTCOOKIE Architecture

To overcome the challenges and existing limitations, SMARTCOOKIE proposes a novel split-proxy architecture (Figure 2).

Switch Agent. The SMARTCOOKIE switch agent performs cookie checks *securely* (F1) and tracks verified connections *approximately* (F3.A). This design is motivated by the underlying switch architecture, which is optimized for high-speed packet processing up to Tbps, an order of magnitude greater than the speeds of general-purpose CPUs. The switch agent is an ideal location for offloading of SYN cookie checks. It can securely and performantly block *100% of SYN floods*, without burdening the server (§5,§8). Recall that to achieve high throughput and low latency, however, the switch only has a limited amount of memory, and memory accesses are constrained. Thus, the switch agent should not be required to remember all the connections that have successfully passed the cookie check. In other words, *verified records* should be kept in an *approximate* data structure at the switch agent (§6.3).

Server Agent. Meanwhile, SMARTCOOKIE server agent is primarily responsible for handling benign connections, conducting sequence number translations (F2) and *exactly* tracking verified connections (F3.B). Servers offer greater compute flexibility and have fewer constraints on memory access and usage than switches. Thus, the server agent is ideal for maintaining exact information about verified connections and for performing sequence number translations on behalf of the switch agent. This offloading of unique per-connection sequence number translations is enabled by a special setup procedure, which the switch agent and server agent cooperatively perform using custom setup request and confirmation packets (§6). The server agent's sequence number translation mechanism ensures consistent sequence number progression, preserves the correctness of the TCP protocol, and is transparent to unmodified end hosts. We note that the greater compute flexibility at the server agent comes at the cost of lower packet-processing speeds as compared to hardware speeds, but this is a reasonable tradeoff since the amount of benign traffic the server agent must process is much smaller than the amount of attack traffic that the switch agent must identify and drop.

5 Secure SYN Cookies in the Data Plane

SMARTCOOKIE switch agent is responsible for performantly computing and verifying SYN cookies in the data plane, but for any defense to be worthwhile, this must be done securely.

Choice of Hash Function. There are several potential choices of data-plane hash functions. Recall that because of the computational constraints of the switch, it is extremely challenging to compute a cryptographically secure hash function in the data plane. Hence, several prior works [51, 72] opted to use the CRC32 checksum as a "hash" function to compute SYN cookies, resulting in significant vulnerability [70]. This is because an adaptive adversary can always send crafted SYN packets and observe the resulting cookies (i.e., a chosen-plaintext attack), efficiently solve and extract the key used in hashing, and then forge cookies for any 4-tuple, bypassing the defense entirely (§2.4).

SMARTCOOKIE securely computes and verifies cookies using *HalfSipHash-2-4* [8], which is from the SipHash family of hashes used by Linux for computing SYN cookies [2]. We also choose HalfSipHash-2-4 for performance reasons, as it is faster than SipHash-2-4, while still sharing the same construction. We believe HalfSipHash-2-4 *with key rotation* offers acceptable security against even well-provisioned adversaries (e.g., key brute-forcing adversaries), achieving good security along with lightweight performance. The hash is cryptographically robust and designed for speed on short inputs, making it ideal for computing cookies in the data plane, where the input to the hash is just a few bytes from the packet header.

Securely Computing Cookies. HalfSipHash is seeded by secret keys, used to initialize four internal variables *v0-v3* (Figure 3). The hash input gets mixed with these internal variables using arithmetic operations (Add, Shift, XOR) across several computational rounds. HalfSipHash-*c*-*d* performs *c* rounds of computation (compression rounds) for each *w* 32-bit word of the hash input. Once all input bytes have been processed by compression rounds, an additional *d* rounds of computation are performed, called finalization rounds. Finally, the four internal variables are XORed, giving the hash output.

Each round of computation requires 14 arithmetic operations, including six circular left shifts. Since most of the arithmetic operations directly use the output of the previous operation (Figure 3), they create a long dependency chain. Since the P4 language [36] does not natively support the circular shift operation, naively implementing a circular left shift of *n* bits requires three intermediate operations: a left shift of *n* bits, a right shift of 32 - *n* bits, and then a bitwise OR of these two intermediate results. Because of the dependency chain between the bitwise OR and the intermediate shifts, calculating a single circular shift will require two pipeline stages, which quickly becomes unreasonably costly given the limited number of available stages in the switch.

Instead of using a multi-operation, multi-stage approach to circular bit shifts, we optimize the computation by using a

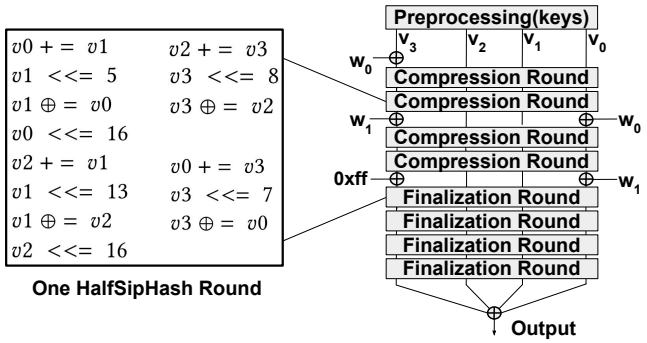


Figure 3: HalfSipHash-2-4 requires 2 compression rounds on each input word *w* and 4 finalization rounds at the end, with each round costing 14 operations.

built-in slicing primitive on the switch to slice out the desired upper and lower bits of the variable. We then use a supported concatenation primitive to stitch the newly relocated bits together in a new variable, fitting the full circular shift into one operation in a single pipeline stage. Additionally, we optimize the number of pipeline stages needed for the computation, by manually grouping the 14 arithmetic operations in a round into 4 stages based on their dependencies.

Recirculation. To calculate cookies, we need to operate HalfSipHash on a 16-byte ($w = 4$ word) input, requiring in total $2w+4 = 12$ compression and finalization rounds. Although we have optimized HalfSipHash for modern switch hardware, we still require several pipeline passes to perform all the rounds. Hardware switches support a packet recirculation feature that allows for computation across multiple pipeline passes. We note that naively using just the ingress pipeline of the switch for computing the hash would require $r = (2w+4)/2-1 = w+1 = 5$ recirculations. However, we integrate HalfSipHash with SMARTCOOKIE's switch agent logic to run in both the ingress and egress pipelines, reducing r to $(2w+4)/4-1 = w/2 = 2$ recirculations per hash.

In practice, network switches have dedicated recirculation ports, and additional ports can be reserved for recirculation. Dedicating additional ports for recirculation would reduce the bandwidth that could have been used for serving other traffic, but this is an acceptable performance tradeoff for the security gains of a robust hash. We show in §8 that even with recirculations, our HalfSipHash-based switch agent returns secure cookies 2.6x faster than the next fastest defense. Additionally, the recirculation limitations of our prototype result from the lack of cryptographic building blocks in the network hardware, which can be improved in the future.

Key Rotation. To defend against brute-force attacks, HalfSipHash keys are rotated periodically (e.g., every 5-30 seconds). This reflects our underestimation of the time needed to brute-force a key [8]. To ensure handshakes from clients are not accidentally blocked, cookies computed with an old key are still accepted for a short period after a key rotation.

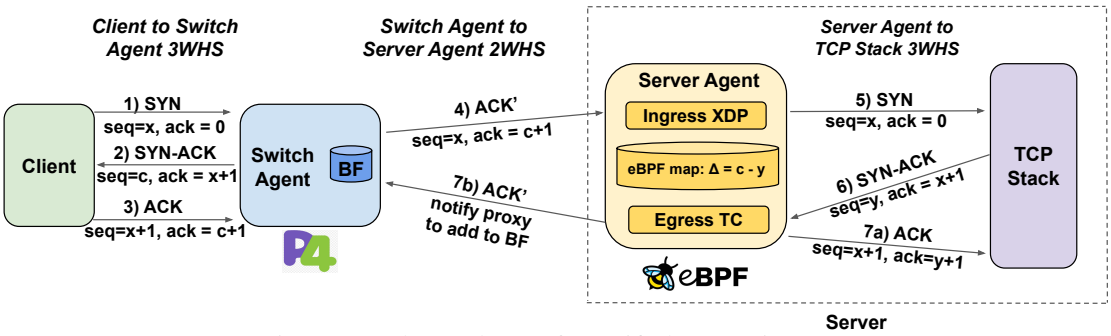


Figure 4: End-to-end setup for verified connections.

6 Split-Proxy Design

With our secure SYN cookie hash in the data plane, we can now safely offload cookie checks to a high-speed switch. However, we must still tackle the challenge of limited switch memory for handling benign flows. SmartCookie accomplishes this with a split-proxy design, where a switch agent and server agent cooperate to performantly stop attack traffic and correctly handle benign traffic. Our design avoids packet buffering during setup and bypasses sequence number translations at the switch agent, allowing it to *approximately* track verified connections and scale beyond available switch memory.

6.1 Switch to Server Two-Way Handshake

Figure 4 shows SmartCookie's complete setup procedure for verified connections. In `packets 1-3` SmartCookie switch agent verifies clients by performing a secure SYN cookie check as in the traditional defense. However, under the SmartCookie protocol, the switch agent does not buffer the final ACK of the TCP three-way handshake (3WHS) between the client and switch agent. Instead, the switch agent directly forwards this ACK packet to the SmartCookie server agent with an additional *setup tag*, notifying the server agent to bootstrap the connection setup (`packet 4`). The switch agent also uses this packet to instruct the server agent how to handle the difference in initial sequence numbers (ISNs) that were chosen by the switch agent and the server's network stack.

After receiving this tagged setup packet, the server agent sets up its side of the connection, shown with `packets 5-7a`. The server agent then sends a packet to the switch agent to confirm the connection establishment, completing the custom setup between switch agent and server agent as shown with `packet 7b`. Note that the server agent has been instructed by the switch agent how to handle sequence number deltas, so the client and server see their expected sequence numbers in both directions. This allows the switch agent to step into a passive forwarding role and avoid the expense of sequence number translations throughout the remainder of the connection.

SmartCookie converts the original 3WHS between switch and server into a custom two-way handshake (2WHS) between switch agent and server agent, notifying the server of the client connection, because at this point the server is still unaware of the client. We note that a 3WHS between switch agent and server agent is undesirable, causing overhead at the switch for buffering packets from the client while the second 3WHS is being conducted. Additionally, even after the end-to-end connection is established, per-packet processing at the switch for sequence number translations is undesirable. Instead, by explicitly informing the server agent of the connection and sending information for sequence number translations, the switch agent can safely forward packets from a verified connection without additional processing.

Handling Setup Latency and Packet Drops. SmartCookie reliably handles more complex scenarios introduced by connection setup latency, reordered packets, or packet drops. Consider the scenario where the 2WHS between the switch agent and server agent is not yet complete, either because it is still in progress or because the setup packet sent to the server agent has been lost. If the client sends additional data packets to the server during this state, the SmartCookie switch agent handles this gracefully by continuing to verify and tag these packets before forwarding them to the server agent (note that the switch agent never buffers client packets). Since the client has yet to receive any packets from the server, the server-side sequence numbers have not progressed and the client would still pass the cookie check at the switch agent. As long as the client's packets continue to pass the cookie check, the switch agent will tag and forward them to the server agent, and upon receipt of any tagged packet from the switch agent, the server agent will immediately set up the connection and send an explicit confirmation to the switch agent. Upon receiving this confirmation, the switch agent no longer tags any packets from the client and simply forwards packets in both directions.

6.2 Redesigned Server Setup

SmartCookie uses *eBPF* to facilitate division of labor between switch and server agents. Our design does not modify the network stack of end hosts, which is particularly beneficial for datacenter deployments where network operators may not control end host TCP behavior (e.g., tenant VMs).

eBPF Primer. eBPF (extended Berkeley Packet Filter) is a powerful and lightweight technology that allows for safe, fast, kernel-level execution of programs directly from user-space, without requiring programmers to rewrite kernel source code [10]. The Linux networking community has been taking advantage of eBPF for efficient packet filtering and safe kernel-level execution in many different security applications, including in the DDoS space [11, 57, 65].

Modified TCP Interface. With eBPF, we redesign the TCP interface between the network and the Linux kernel's network stack. This allows us to offload connection setup and sequence number translations to the server *without* modifying the kernel TCP stack, enabling immediate deployability. Our server agent deploys an eBPF ingress and egress program attached to different kernel hooks (XDP and TC, respectively), performing connection setup with the kernel network stack and sequence number translations on incoming and outgoing packets (see §B.2 for details). The eBPF programs act as a lightweight translation layer that converts custom packets received from the switch agent into proper TCP packets for regular TCP/IP processing by an unmodified kernel.

eBPF Map for Tracking State. eBPF maps can be used to communicate between user space and kernel space, and they are the only way to store and share state between eBPF programs. Maps are implemented as key-value stores, where values are defined by data type and size. The server agent uses an eBPF map to track the connection state of any given flow and to coordinate the behavior of the ingress and egress eBPF programs based on this state. The map key is the 4-tuple connection information (source and destination IP addresses and port numbers). The map value for each key stores the connection state of the connection and the sequence number delta to be applied for that connection. As shown in Figure 4, the sequence number delta for a given connection is determined to be $\Delta = c - y$, where c is the switch agent's original ISN (i.e., cookie value) and y is the ISN chosen by the kernel. At first, the server agent's ingress program extracts c from `packet 4` and stores it in the eBPF map temporarily. Later, the server agent's egress program extracts y from `packet 6` and then calculates and stores the true Δ in the eBPF map.

6.3 Compact Data Structure at Switch Agent

To determine which packets require cookie checks, the switch agent (like any standard defense) must maintain a record of verified connections. In prior designs, the proxy keeps a local record of *every* verified connection that has passed the cookie check, using this record to remember the sequence number deltas that must be applied to all packets in the flow. This is an unattractive solution because it requires per-flow state, and memory is a limited commodity in high-speed switches.

Bloom Filters for Approximate State. Since SmartCookie offloads sequence number translations to the server agent, the switch agent can avoid keeping exact state by using a resource-efficient compact data structure called a Bloom filter. The Bloom filter can *approximately* keep track of clients that have passed the cookie check and successfully established a connection with the server. Bloom filters are efficient and powerful approximate data structures, but they come at the expense of a small number of false positives. There are no false negatives in Bloom filters, which is an important feature that ensures benign traffic is never blocked. We note the purpose of the Bloom filter is *not* to stop an adversary's non-SYN packets from reaching the server, although this does happen as a bonus. Instead, the goal of this design feature is *to not keep per-flow state at the switch agent for verified flows*, reducing strain on the memory-constrained switch.

Implementing Bloom Filters in the Switch Agent. To approximately track the set of benign TCP connections, we use register memory arrays on the Tofino programmable switch to implement a Bloom filter. Since the switch's pipeline only allows accessing one index per array when processing a packet, we implement a Bloom filter variant called the Partitioned Bloom Filter (PBF) [6]. PBF splits its m-bit memory into k separate arrays $M_1[\cdot] \ldots, M_k[\cdot]$, each sized $\frac{m}{k}$, and uses k indexing hash functions $h_1, \ldots, h_k : \mathcal{F} \to [\frac{m}{k}]$ to map the input key $f \in \mathcal{F}$ (in our case, flow 4-tuple) into locations in the array. All arrays are initialized to zero. To insert a flow f, for each array i, we calculate the index $h_i(f)$ and mark 1 in the corresponding index, i.e., $M_i[h_i(f)] \leftarrow 1$. To query whether flow f has been added, we check the same indices and report positive if they are all 1, i.e., $M_i[h_i(f)] == 1, \forall i \in [k]$.

Automatically Cleaning the Bloom Filters. To prevent over-saturation and maintain the accuracy of the filters as time goes on, we can keep multiple Bloom filters at the switch agent in a rotating fashion, where at any given time window T_W, some filters are being written to, read from, or cleaned (i.e., emptied). A connection is considered active if any one of the Bloom filters reports a positive. For ongoing connections, we add to the most recent filter whenever we see the server sending a packet to the client; note that even for uni-directional upload traffic servers continuously send ACK packets to clients. Thus, when a filter is cleaned, ongoing connections automatically get carried over to the active filter.

We also note that T_W should be chosen such that we maintain a reasonable timeout window that reflects the characteristics of the majority of connections in the network [31]. A T_W value that is too small (e.g., a few seconds) will cause idling connections to be closed too early, while T_W values that are too large will require us to save too many active connections in the Bloom filter, causing increased memory usage and potentially higher false positive rates. Shorter T_W windows (also called "aggressive aging timeouts") are the security best practice for preserving stability and performance while under attack, and can be as short as 10 seconds [22, 68].

6.4 Handling False Positives at Server Agent

The server agent cooperates with the switch agent to perform a last-line-of-defense cookie check on the small fraction of packets that experience a false positive in the Bloom filter.

Identifying False Positives. The server agent knows to perform this cookie check on any packets that are not already part of an ongoing connection at the server and do not have a setup tag from the switch agent. We note that SYN packets are always stopped and handled at the switch agent, so the server agent only handles benign traffic and non-SYN packets (either adversarial or benign) that have triggered a false positive in the Bloom filter. If the false positive was the result of an adversary's non-SYN packet (e.g., if the adversary was attempting to perform an ACK flood), it would fail the cookie check at the server agent and get dropped (§7.2). However, if the false positive was triggered by a benign client (e.g., with the final ACK packet of the 3WHS between client and switch agent), then the packet would pass the cookie check at the server agent and the server agent would set up the connection with the kernel network stack.

Key Management. In existing SYN cookie defenses, only one party computes and verifies cookies. However, since both the switch and server agents verify cookies, they must share the same SYN cookie computation scheme so that cookies are consistent across the system. In other words, both switch and server agents must share the same hash functions and secret keys used to compute cookies; the network operator acts as the central controller for managing these keys across the system, installing the same initial key and performing periodic key rotation for both the switch and server agents. Furthermore, to ensure consistency when using timestamps to generate and verify cookies, SMARTCOOKIE relies on the network operator to synchronize clocks across the switch and server agents. We also note that on the server side, secret keys are only visible to the eBPF switch agent module and are not visible to applications or tenant VMs running on the server.

7 Security Analysis

We discuss possible attacks from adaptive adversaries against our system itself (above and beyond our initial SYN-flooding attack vector), and explain how we address them.

7.1 Attacks on the Cookie Check

Replay Attacks. When a cookie is first computed at the switch, it is computed with respect to the current epoch of time. When cookies are recomputed and verified, they are only accepted if they are from the current or immediately preceding epoch. While the Linux kernel uses 1 minute epochs [1], we choose 1 second epochs instead, to reflect round-trip times (RTTs) commonly seen on the modern Internet. This ensures that SMARTCOOKIE only accepts cookies returned within acceptable RTT delays, while defending against replay attacks with older cookies. For consistency, the cloud provider deploying SMARTCOOKIE should ensure that clocks are synced (e.g., NTP) between the switch agent and the server agent.

Cookie Forging and Key Recovery. An adversary can send probe packets to the switch agent and observe the returned cookies in order to crack the hashing mechanism and forge her own cookies; formally, this is performing a Chosen Plaintext Attack (CPA) against our hash function. The security of SMARTCOOKIE against cookie forging can be directly reduced to the security of the underlying HalfSipHash family of hash functions. Assuming the security properties of HalfSipHash, the adversary cannot recover the key without a brute-force search [8]. To make brute-forcing ineffective, SMARTCOOKIE rotate the key periodically (e.g., every 5-30 seconds). Thus, with the pseudorandom properties of HalfSipHash, the adversary's best attack strategy is reduced to simply guessing the cookie.

Lucky Cookies, and Saving TCP Options. The Linux kernel's default SYN cookie stores a quantized Maximum Segment Size (MSS) and discards all other TCP options. Our switch agent similarly encodes the MSS in cookies, for the server agent to later decode. This approach leaves 24 bits of entropy in the cookie, meaning that each adversary's attack packet has a negligible probability ($1/2^{24}$) of "luckily" guessing the right cookie by chance. If desired, it is possible to use 3-4 more bits of the cookie to save a few more TCP options, in exchange for slightly reduced entropy.

7.2 Attacks on the Bloom Filter

ACK Floods and TTL Expiry Attacks. The Bloom filter used by our switch agent to track verified connections has a small probability of reporting false positives; an adversary can send many ACK packets with randomized source IP addresses and port numbers, and a small fraction of these may trigger a false positive and be forwarded to the server. We refer to this as the ACK-flooding attack, and evaluate the performance of SMARTCOOKIE against such attacks in §8. We note that these false positive ACK packets will be silently dropped by the server agent after failing its last-line-of-defense cookie check (§6.4). We also filter TTLs of non-SYN packets that arrive at the switch agent, hiding from an adaptive adversary seeking to perform a TTL expiry attack whether her packets were dropped at the switch agent or at the server agent [62]. This prevents the adversary from getting feedback on which attack packets successfully triggered false positives. Thus, she cannot tailor attack packets for the Bloom filter's internal structure to achieve a higher false positive rate.

Opening Many "Legitimate" TCP Connections. The false positive rate of Bloom filters depends on the number of connections inserted. In our threat model (§3.1), adversaries have access to a moderate number of compromised devices to

launch *asymmetric* SYN-flooding attacks, but not enough to mount successful *symmetric* attacks, such as TCP connection floods. Connection-flooding adversaries would need to establish many "legitimate" TCP connections to raise the Bloom filter's false positive rate before launching a SYN-flooding or ACK-flooding attack, requiring significant resources (e.g., machines, memory, packet processing). Nevertheless, we note our filters are periodically cleaned and designed to achieve a reasonably low false positive rate under realistic traffic conditions (§6.3). Even with high rates of Bloom filter pollution, SMARTCOOKIE still blocks 100% of SYN floods in the switch and is highly resilient under ACK floods (§8.4).

8 Evaluation

We evaluate SMARTCOOKIE by running several experiments on a hardware testbed. We 1) demonstrate the performance of SMARTCOOKIE for securely computing SYN cookies with HalfSipHash in the switch data plane, compared to other approaches, and 2) show the overall performance of SMARTCOOKIE in comparison to Jaqen [51], the state-of-the-art switch-based SYN-flooding defense. We note that the authors own all testbed infrastructure, and attack traffic was only directed to dedicated testbed servers, raising no ethical issues.

8.1 Experiment Setup

Prototype Implementation. SMARTCOOKIE's switch agent is implemented in P4 targeted for an Intel Tofino Wedge100-32BF programmable switch, with approximately 1000 lines of code. The server agent is implemented with eBPF. For more details, we refer readers to Figure §4, §B.1, and §B.2. We have released our prototype source code on GitHub[1]. We also run prototypes of Jaqen SYN Cookie Proxy Mode 1 and Mode 2 [51], based on source code obtained from the authors.

Testbed. The testbed consists of four servers and an Intel Tofino Wedge32X-BF programmable switch. Two machines act as adversaries, each with a 20-core Intel Xeon Silver 4114 CPU and a Mellanox ConnectX-5 2x100Gbps NIC, generating attack traffic using DPDK 19.12.0 and pktgen-DPDK. Two other machines act as server and client, each with 8-core Intel Xeon D-1541 CPUs and Intel X552 2x10Gbps NICs. All machines run Ubuntu 21.10 with kernel v5.13.0, and the eBPF programs are built with BCC v0.24.0 and Clang v13.0.0-2. The server machines are all connected to the switch via Direct Attach Copper (DAC) cables, with under 0.1ms ping latency (round-trip time) between any pair of machines.

Setup: Realistic Traffic Load. To simulate a real-world setting with realistic benign traffic loads, we write customized client and server Go scripts to replay CAIDA anonymized Internet traffic trace 2018 [15]. We did not modify the Linux kernel's default TCP retransmission behavior.

[1] https://github.com/Princeton-Cabernet/p4-projects/tree/master/SmartCookie

Setup: Estimating the False Positive Rate. Our switch agent implementation uses Partitioned Bloom Filters with $k = 3$ arrays of 2^{20} bits each, with total memory size $m = 3 \times 2^{20}$ bits. For our experiments, we choose a connection timeout window T_W of 15 seconds for Bloom filter cleaning (§6.3), which reflects the default keepalive timeout of many applications [31]. We choose a 15-second trace window of benign traffic to match T_W, with $n = 579{,}600$ flows. Our trace represents heavier flow loads than the average from CAIDA trace statistics (386,000 flows over the same time window) [16]. Thus, we generously estimate our false positive rate as:

$$F_p(n,m,k) = \left(1 - \left(1 - \frac{k}{m}\right)^n\right)^k \approx 7.66\%. \qquad (1)$$

We replay each flow as an HTTP request in real time (approximately 38,500 requests per second), where the request starting time corresponds to the timestamp offset of the flow's first packet. We also measure the number of attack packets received at the server with this setup and verify that the measured false positive rate matches our expectations. Meanwhile, since Jaqen's prototype uses a smaller Bloom filter ($2^{16} \times 4$), we replay fewer flows ($48,800$) in Jaqen's experiments, such that the defense exhibits the same false positive rate of 7.66%. Since the number of connections in both experiments exceed the number of available ports, we add 65,536 IP addresses to both client and server; we use the connection 4-tuple as keys for Bloom filter lookups for both SMARTCOOKIE and Jaqen.

More generally, under different traffic conditions, time windows, and number of connections, we can use Equation §1 to estimate corresponding false positive rates. We also note that our current implementation did not exhaust the memory available on the Tofino 1 switch, and we can build larger Bloom filters to support more connections, trading off more memory for a reduction in the false positive rate. Other switch models (e.g., Tofino 2) also provide more onboard memory to further increase Bloom filter size and lower false positive rates.

Setup: DDoS Attacks. We run pktgen-DPDK on the adversary machines to generate SYN floods with randomized source IPs and ports. Due to false positives, Jaqen will allow 7.66% of SYN-flooding traffic to reach the server and trigger a *connection setup and kernel SYN cookie computation*. Meanwhile, since our switch agent handles 100% of SYN packets without passing any to the server, we run a separate experiment where we subject SMARTCOOKIE to ACK flood traffic (§7.2), 7.66% of which will reach the server and trigger *only eBPF SYN cookie verification* at the server agent.

8.2 Hashing Throughput

Under the experiment setup described above, we compare SMARTCOOKIE's performance against kernel-based SYN cookies and XDP-based cookies. Since prior work proposed running AES, a secure encryption algorithm, on Tofino

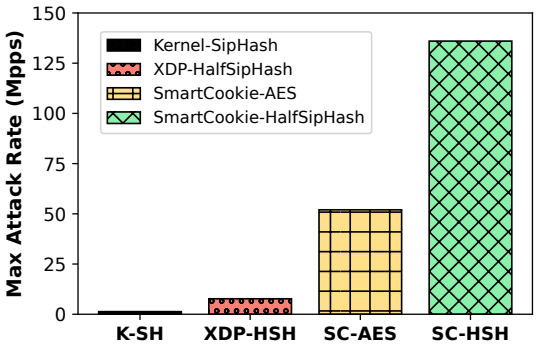


Figure 5: **Throughput.** SMARTCOOKIE-HalfSipHash defends against attacks *without packet loss* until a rate of 136.9Mpps, outperforming the next fastest defense by 2.6x.

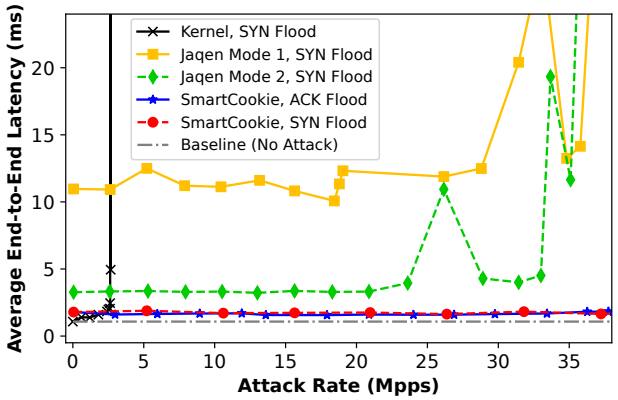


Figure 6: **Latency.** SMARTCOOKIE reduces latency by 48-84% compared to Jaqen, protecting client performance.

switches [20], we also implement and benchmark a variant of SMARTCOOKIE switch agent using AES to compute cookies.

Measuring Throughput. We measure the maximum attack rate in Mpps each defense can handle *before any packet loss*. Since our benchmark performs one hash calculation per SYN packet, we effectively measure maximum *hashing* throughput.

Results. As shown in Figure 5, SMARTCOOKIE-HalfSipHash significantly outperforms all other defenses, achieving a throughput of 136.9Mpps on high-speed switch hardware. This is *two orders of magnitude greater* than the throughput achieved by the software-based kernel defense, which uses SipHash and can only serve 1.3Mpps before the server's CPUs are exhausted. Using XDP to compute cookies can bypass the overhead of the kernel network stack and achieve 6x speedup (7.3Mpps), but this is still much slower than SMARTCOOKIE-HalfSipHash. Meanwhile, although SMARTCOOKIE-AES is faster than XDP, it requires more recirculations and only achieves 52 Mpps; SMARTCOOKIE-HalfSipHash outperforms it by 2.6x.

We note that Tofino switches have dedicated recirculation ports, *which operate without affecting capacity on other ports*. Our prototype switch agent achieves 136.9Mpps using a single Tofino 1 switch under its vanilla setup (pre-configured ports with 200Gbps recirculation throughput). To achieve even higher throughput, we can simply load balance between multiple switches to multiplex their throughput, or configure the switch to convert unused physical ports into extra recirculation bandwidth. For example, as a back-of-the-envelope calculation, using 20 recirculation ports, we could achieve 2Tbps recirculation throughput and serve roughly 1.4 billion requests per second. We also note that while repurposing additional ports for recirculation would reduce overall switch capacity, it will not affect latency. Finally, other switches exist (e.g., Tofino 2) with higher per-port throughput and more pipeline stages. Such switches, along with future hardware with native cryptographic support, can further reduce recirculation needs, additionally boosting throughput.

8.3 Latency

In this experiment, we launch both benign and attack traffic against a server as described in §8.1, and measure the end-to-end latency of benign application traffic while the server is under attacks of different magnitudes. We compare SMARTCOOKIE against the vanilla kernel-based SYN cookie defense and Jaqen's two SYN proxy modes. SMARTCOOKIE's design is transparent to clients and does not cause connection reset, leading to 48%-84% lower latency than Jaqen.

Measuring Latency. We measure the end-to-end application latency by initiating HTTP requests from the client to the server, with response size of 14.5KB (corresponding to the average flow size in the CAIDA 2018 trace). For each attack rate, we send 30 requests and calculate the average latency. Since Jaqen's Proxy Mode 2 triggers an application-layer reset, most applications (`curl`, `wget`, etc.) will wait for at least one second before retrying the connection. This timeout is an unreasonably large penalty for benign clients, and so we write a customized Go client script that immediately retries after a reset, waiting for only 1ms between connection attempts. This showcases the best possible performance of Jaqen, as it avoids the long default reset timeout, but we note that reconfiguring client applications is not always possible.

Results. As shown in Figure 6, SMARTCOOKIE has consistently low end-to-end latency (1.71ms), a 48%-84% reduction compared to Jaqen, even with the reconfigured fast retry client that bypasses Jaqen's default 1-second timeout. Jaqen Mode 1 incurs a reset and one additional round trip, with a minimum latency of 11.12ms. Jaqen Mode 2 incurs two additional round trips and an application-layer retry, requiring at least 3.31ms for end-to-end setup with our fast 1ms retry client and *over one second* for default TCP applications that have not been reconfigured. Meanwhile, SMARTCOOKIE's latency, which is only 1.71ms, is close to the baseline latency of the vanilla kernel without any attack (1.08ms).

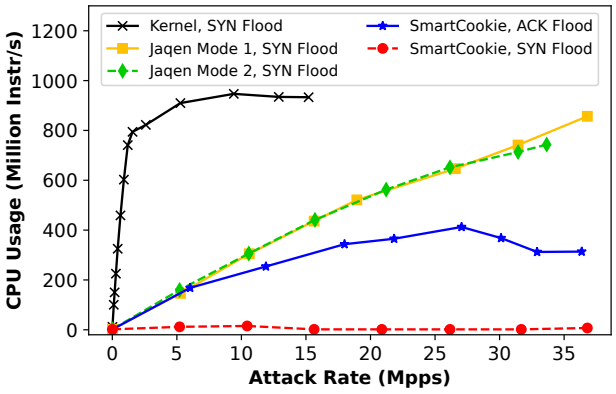


Figure 7: **CPU Usage.** SMARTCOOKIE has *no* server CPU overhead against SYN floods and reduces overhead against ACK floods by 33%-36% compared to Jaqen.

8.4 Server CPU Usage

Using the same setup from §8.1, we replay trace-based benign traffic while launching attack traffic at increasing rates. We measure the server's CPU overhead across various defenses. SMARTCOOKIE has *no overhead* for SYN floods and reduces overhead for ACK floods by 33-36% compared to Jaqen.

Measuring CPU. To measure CPU overhead, we use the `perf stat` command to read CPU performance counters and collect the number of instructions executed per second for each CPU core, taking the average across cores. We also repeat each measurement ten times and take this average. For more stable results, we turned off frequency scaling (Turbo Boost) and fixed all cores' frequency at 2.1GHz.

Results. Both modes of Jaqen exhibit similar server CPU overhead, as they both allow the same rate of false positive SYN packets to reach the server, where the defense degrades to SYN cookie computation on server software (Figure 7). We note that SMARTCOOKIE switch agent processes 100% of SYN packets directly on network hardware, completely protecting server CPUs from resource exhaustion for SYN cookie computation. With SMARTCOOKIE, servers experience *no CPU overhead during SYN floods*. Thus, in order to see the effect of false positives on CPU usage with SMARTCOOKIE, we also measure the overhead of the server agent handling map lookup and cookie verification for false positive ACK packets (§6.4): SMARTCOOKIE under an ACK flood exhibits approximately 33%-36% lower CPU overhead for the same attack rate and same false positive rate as Jaqen under a SYN flood. This is because Jaqen is a purely switch-based defense that degrades to the vanilla kernel's defense on false positives, which begins to suffer from attack rates as low as 1.3 Mpps. Meanwhile, SMARTCOOKIE is designed as a split-proxy system that cooperates with an XDP-based server agent for false positives (§B.2). XDP operates on raw packets early in the kernel stack before any socket buffer is allocated, lowering the overhead significantly compared to the kernel defense.

	SRAM	TCAM	HashUnit	Instr.
Jaqen Mode 1 (CRC)	4.6%	0.0%	19.4%	4.9%
Jaqen Mode 2 (CRC)	4.6%	0.0%	19.4%	6.3%
SMARTCOOKIE-AES	13.8%	0.1%	30.6%	9.4%
SMARTCOOKIE-HSH	6.8%	1.4%	56.9%	9.9%

Table 2: Resource usage on programmable switch.

	SRAM	TCAM	HashUnit	Instr.
tna_simple_switch.p4	6.3%	9.0%	0.0%	9.6%
+SMARTCOOKIE	12.6%	11.5%	54.2%	18.0%
switch.p4	33.6%	31.6%	19.4%	13.5%
+SMARTCOOKIE	40.1%	37.5%	73.6%	20.3%

Table 3: Adding SMARTCOOKIE to complex P4 programs.

8.5 Switch Resource Usage & Compatibility

In Table 2 we report SMARTCOOKIE's utilization of switch hardware resources with two variants of hashes (HalfSipHash and AES), as reported by Intel's P4i tool. SMARTCOOKIE has a trivial footprint for important shared resources (TCAM:1.4%, SRAM:6.8%). Notably, the HashUnit usage is highest (57%), but HashUnits are used less by other network functions. We also report resource utilization of Jaqen SYN Proxy (Modes 1 and 2), which use CRC as its cookie hash.

Furthermore, we successfully integrated SMARTCOOKIE-HalfSipHash into two feature-rich, complex P4 programs: `tna_simple_switch.p4` and `switch.p4`, demonstrating SMARTCOOKIE can co-exist with other sophisticated network functions. Table 3 presents resource utilization metrics for each base program and the variant with SMARTCOOKIE added, highlighting SMARTCOOKIE's efficient footprint and compatibility with other switch functions.

9 Discussion

Routing Considerations. We envision our defense deployed in a provider network with multiple edge switches, motivating the need to be robust to both asymmetric routing and potential routing changes. SMARTCOOKIE handles asymmetric routing by default with several clever design decisions, and it can handle routing changes using two high-level approaches, each with their own tradeoffs. Cooperating switch agents can maintain synchronized Bloom filters with state for all verified connections. Alternatively, Bloom filters across individual switch agents can dynamically adapt to routing changes using a packet sampling approach. Please see §C for more details.

Serving Tenants Transparently. SMARTCOOKIE does not modify the server's network stack, and can serve unmodified tenant VMs running on servers. From the tenant's point of view, the TCP protocol is unchanged. Today's high-performance VM hypervisors and container hosts often run specialized software switches (e.g., eBPF-based Cilium [4])

to handle tenant traffic, and SmartCookie 's server agent can be integrated into these software switches.

Handling Other DDoS Attacks. Although SmartCookie was explicitly designed for large-scale SYN-flooding attacks, by default it also handles other TCP-based volumetric attacks (e.g., ACK floods, SYN-ACK floods, RST/FIN floods), quickly dropping attack traffic on behalf of the server. We believe our design can also be generalized to UDP-based volumetric attacks, with mechanisms like those proposed in [69].

Transport Protocols. We note that SmartCookie makes assumptions about TCP and must be upgraded when clients use new features, like MPTCP. To avoid protocol ossification, SmartCookie should not be applied by default on all traffic. Instead, it should be an opt-in feature for tenants enabled only during attacks, like the Linux kernel's SYN cookie defense. Future servers and tenants using newer transport protocols would require updated designs; split-design defenses supporting newer protocols (e.g., MPTCP) and connection-oriented UDP traffic (e.g., QUIC) are interesting future works.

Further Improvements. Our eBPF server agent is independent of the kernel network stack, enabling direct deployability without kernel updates. In the future, the switch agent can synchronize ISNs with the kernel to entirely avoid sequence number translations and further boost performance (§D).

10 Related Work

SYN Cookie Defenses. State-of-the-art SYN cookie defenses, including standard practices such as DDoS scrubbing centers, struggle to capitalize on the unique capabilities of programmable switches. They either have the data plane do *too much* (e.g., performing the complete TCP handshake or keeping per-flow state) [51, 72] or *too little* (e.g., simply forwarding attack traffic to a server for software-based packet-processing) [27, 28]. These works miss opportunities to refactor the server, under-optimize switch resource usage, incur performance penalties on benign clients, and most importantly create vulnerabilities in the defense with insecure hashes.

The most closely related such work is Jaqen [51]. Unlike Jaqen, SmartCookie uses a cryptographically robust hash that provides strong security guarantees and does not sacrifice performance for scalability. [60] also proposed a SYN cookie proxy, but their proxy must similarly keep per-flow state and is implemented in software [9]. SmartCookie overcomes strict resource constraints to efficiently run SYN cookies directly on switch hardware.

Split Functionality. Poseidon [72] presents a two-part DDoS defense: a switch component on hardware and a server component running in software. However, Poseidon requires the majority of the defense to reside in software, which is orders of magnitude slower at packet-processing than switch hardware. More importantly, Poseidon's SYN cookie proxy uses an insecure hash, and it must also keep per-flow state. SmartCookie overcomes these limitations and presents a novel split proxy that preserves switch resources, optimizes performance, and computes cookies with a *secure* hash.

XDP. GateBot [11] is a DDoS defense system developed by Cloudflare that drops traffic based on iptable rules implemented with XDP. Recently, Google also used XDP to implement SYN cookies, taking advantage of XDP's early execution hook to bypass the kernel TCP stack [57]. SmartCookie provides an order of magnitude higher performance than XDP-based SYN cookies, stopping all SYN flood traffic in switches at the network edge and only relying on XDP-based cookies to handle a small number of false positives.

Cryptography in the Data Plane. Prior works have explored cryptographic functionality in the data plane, although some targeted software models instead of hardware environments. Scholz et al. [61] implemented prototypes of cryptographic hash functions, including SipHash, targeted for CPU backends, NICs, and FPGAs. [20] and [71] first implemented the AES cipher and HalfSipHash for Tofino switches. Our work leverages such cryptographic building blocks to design and implement a complete split defense system with end-to-end evaluation results. NeoBFT [66] further optimized HalfSipHash on switches by computing multiple message signatures in one batch. PINOT [69] implemented a lightweight 2-round Even-Mansour (2EM) cipher for switches for privacy-protecting IP address encryption; however, their implementation uses fixed permutations that cannot be quickly rotated, which introduces a potential risk for brute-force attacks.

General In-Network Defenses. Previous works used software-defined networking (SDN) to implement SYN-flooding defenses, outside of SYN cookies [28, 30, 52, 53, 59, 64]. Approaches include rate-limiting based on an upper-bound threshold of SYN packets [52], enforcing whitelists or blacklists based on completion of the TCP handshake [30, 53], and using switches to identify and steer suspicious traffic to software platforms for further handling [28]. More generally, other recent works proposed switch-based defenses against link-flooding attacks [73] and pulse-wave DDoS attacks [35]. Surveys of switch-based DDoS mitigations and other switch-based network security applications are presented in [18, 19].

11 Conclusion

SmartCookie is the first SYN-flooding defense to provide cryptographically secure SYN cookies on high-speed switches. With its novel split-proxy design leveraging programmable data planes, SmartCookie is robust against adaptive adversaries and scalably serves large loads of traffic while under attack, without disrupting benign flows or degrading performance. We show that, contrary to common belief, a hardware-software codesign performs better than a purely hardware design. With networks that are now end-to-end programmable, we believe many security applications would benefit from similar division-of-labor design principles.

Acknowledgments

We sincerely thank Alan Zaoxing Liu for his assistance in our evaluation. We are grateful to John Sonchack, Oliver Michel, Henry Birge-Lee, Hyojoon Kim, and Maria Apostolaki for their detailed and valuable feedback on earlier versions of our work, and we also thank the anonymous reviewers and shepherd for their insightful comments. This work was supported in part by DARPA Grant HR0011-20-C-0160. Sophia Yoo was supported by NSF GRFP Grant DGE-2039656.

References

[1] Linux SYN cookie epoch time. https://elixir.bootlin.com/linux/v6.0/source/include/net/tcp.h#L497, 2005.

[2] tcp(7) - Linux man page. https://linux.die.net/man/7/tcp, 2005.

[3] FreeBSD Manual Pages. https://www.freebsd.org/cgi/man.cgi?syncookies, 2008.

[4] Cilium: ebpf-based networking, observability, security. https://cilium.io/, 2017.

[5] AARNet. The rise of DDoS attacks in 2023: what you need to know . https://www.aarnet.edu.au/the-rise-of-ddos-attacks-in-2023-what-you-need-to-know#, 2023.

[6] Paulo Sérgio Almeida. A case for partitioned bloom filters. *IEEE Transactions on Computers*, 2022.

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai botnet. In *USENIX Security Symposium*, 2017.

[8] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A Fast Short-Input PRF. *Lecture Notes in Computer Science*, 7668, 2012.

[9] Bmv2 authors. Behavioral model (bmv2). https://github.com/p4lang/behavioral-model, 2019.

[10] Suricata authors. Suricata - eBPF and XDP. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html, 2018.

[11] Gilberto Bertin. XDP in practice: Integrating XDP into our DDoS mitigation pipeline. In *Netdev: The Technical Conference on Linux Networking*, 2017.

[12] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. In *ACM SIGCOMM Computer Communication Review*, 2014.

[13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM*, 2013.

[14] Broadcom. BCM56870 Series. https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56870-series.

[15] CAIDA. The CAIDA UCSD Anonymized Internet Traces 2018 - July 19th, equinix-nyc.dirA.20180719-130000, 2018.

[16] CAIDA. Trace Statistics for CAIDA Passive OC48 and OC192 Traces, 2018.

[17] Ang Chen, Wenchao Zhou, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Loo, Linh Phan, Micah Sherr, and Clay Shields. Dispersing Asymmetric DDoS Attacks with SplitStack. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2016.

[18] Xiang Chen, Hongyan Liu, Dong Zhang, Qun Huang, Haifeng Zhou, Chunming Wu, and Qiang Yang. Empowering ddos attack mitigation with programmable switches. *IEEE Network*, 2022.

[19] Xiang Chen, Chunming Wu, Xuan Liu, Qun Huang, Dong Zhang, Haifeng Zhou, Qiang Yang, and Muhammad Khurram Khan. Empowering network security with programmable switches: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 2023.

[20] Xiaoqi Chen. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. In *ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure (SPIN)*, 2020.

[21] Catalin Cimpanu. AWS said it mitigated a 2.3 Tbps DDoS attack, the largest ever. https://www.zdnet.com/article/aws-said-it-mitigated-a-2-3-tbps-ddos-attack-the-largest-ever/, 2020.

[22] Cisco. Security Configuration Guide: Zone-Based Policy Firewall, Cisco IOS XE Fuji 16.7.x. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/sec_data_zbf/configuration/xe-16-7/sec-data-zbf-xe-16-7-book/sec-ddos-attack-prevn.html, 2017.

[23] James Coker. Finland Government Sites Forced Offline by DDOS Attacks. https://www.infosecurity-magazine.com/news/finland-government-sites-offline/, 2022.

[24] Kevin Collier, Shanshan Dong, and Ali Arouzi. Hacktivists, new and veteran, target Russia with one of cyber's oldest tools. https://www.nbcnews.com/tech/security/hacktivists-new-veteran-target-russia-one-cybers-oldest-tools-rcna20652, 2022.

[25] Information Technology Intelligence Consulting. Hourly Downtime Costs Rise. https://itic-corp.com/blog/2019/05/, 2019.

[26] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer Denial-of-Service attacks In-Flight with FineLame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.

[27] Marinos Dimolianis, Adam Pavlidis, and Vasilis Maglaris. Syn flood attack detection and mitigation using machine learning traffic classification and programmable data plane filtering, 2021.

[28] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and Elastic DDoS Defense. In *USENIX Security Symposium*, 2015.

[29] Lauren Feiner. Cyberattack hits Ukrainian banks and government websites. https://www.cnbc.com/2022/02/23/cyberattack-hits-ukrainian-banks-and-government-websites.html, 2022.

[30] Silvia Fichera, Laura Galluccio, Salvatore C. Grancagnolo, Giacomo Morabito, and Sergio Palazzo. OPERETTA: An OPEnflow-based REmedy to mitigate TCP SYNFLOOD Attacks against Web Servers. In *The International Journal of Computer and Telecommunications Networking*, 2015.

[31] Gabriel. What is Apache Keepalive Timeout? How to optimize this critical setting. https://ioflood.com/blog/2020/02/21/what-is-apache-keepalive-timeout-how-to-optimize-this-critical-setting/, 2020.

[32] Nick Galov. 39 Jaw-Dropping DDoS Statistics to Keep in Mind for 2022. https://hostingtribunal.com/blog/ddos-statistics/#gref, 2022.

[33] Patrik Goldschmidt and Jan Kučera. Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques. In *International Symposium on Integrated Network Management*. IEEE, 2021.

[34] Miguel Gomez. Dark Web Price Index 2020. https://www.privacyaffairs.com/dark-web-price-index-2020/, 2022.

[35] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. Aggregate-based congestion control for pulse-wave ddos defense. In *ACM SIGCOMM*. ACM, 08 2022.

[36] P4.org Architecture Working Group. P416 Portable Switch Architecture (PSA). https://p4.org/p4-spec/docs/PSA.html.

[37] Alexander Gutnikov, Oleg Kupreev, and Yaroslav Shmelev. DDoS Attacks in Q1 2022, Kapersky Lab Technical report. https://securelist.com/ddos-attacks-in-q1-2022, 2022.

[38] Alexander Gutnikov, Oleg Kupreev, and Yaroslav Shmelev. DDoS Attacks in Q4 2021, Kapersky Lab Technical report. https://securelist.com/ddos-attacks-in-q4-2021, 2022.

[39] Jessica Haworth. Israeli government websites temporarily knocked offline by 'massive' cyber-attack. https://portswigger.net/daily-swig/israeli-government-websites-temporarily-knocked-offline-by-massive-cyber-attack, 2022.

[40] Richard Hummel, Carol Hildebrand, Hardik Modi, Gary Sockrider, Roland Dobbins, Steinthor Bjarnason, Jill Sopko, Suweera DeSouza, Ivan Bondar, and Oliver Daff. NETSCOUT Threat Intelligence report for 2H 2019. https://www.netscout.com/sites/default/files/2020-02/SECR_001_EN-2001_Web.pdf, 2019.

[41] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2018.

[42] Intel. Barefoot Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html.

[43] Andreas Iselt, Andreas Kirstädter, Antoine Pardigon, and Thomas Schwabe. Resilient Routing Using MPLS and ECMP. In *IEEE Xplore*, 2004.

[44] Nivedita James. 45 Global DDOS Attack Statistics 2023. https://www.getastra.com/blog/security-audit/ddos-attack-statistics/, 2023.

[45] Juniper. Juniper Networks' MX480 Universal Routing Platform. https://www.juniper.net/us/en/products/routers/mx-series/mx480-universal-routing-platform.html, 2022.

[46] Patrick Kennedy. Intel Tofino2 Next-Gen Programmable Switch Detailed. https://www.servethehome.com/intel-tofino2-next-gen-programmable-switch-detailed/, 2020.

[47] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. RedPlane: Enabling Fault-Tolerant Stateful In-Switch Applications. In *ACM SIGCOMM*, 2021.

[48] Daehyeok Kim, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, and Srinivasan Seshan Liu. Generic External Memory for Switch Data Planes. In *ACM Workshop on Hot Topics in Networks (HotNets)*, 2018.

[49] Oleg Kupreev, Ekaterina Badovskaya, and Alexander Gutnikov. DDoS Attacks in Q2 2020, Kapersky Lab Technical report. https://securelist.com/ddos-attacks-in-q2-2020, 2020.

[50] Oleg Kupreev, Alexander Gutnikov, and Yaroslav Shmelev. DDoS Attacks in Q3 2022, Kapersky Lab Technical report. https://securelist.com/ddos-report-q3-2022/107860/, 2022.

[51] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In *USENIX Security Symposium*, 2021.

[52] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *ACM SIGCOMM*, 2017.

[53] Reza Mohammadi, Reza Javidan, and Conti Mauro. SLICOTS: An SDN-Based Lightweight Countermeasure for TCP SYN Flooding Attacks. In *IEEE Transactions on Network and Service Management*, volume 14. IEEE, June 2017.

[54] Netronome. BPF, eBPF, XDP and Bpfilter... What are These Things and What do They Mean for the Enterprise? https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/, 2018.

[55] Nvidia. Nvidia BlueField-3 DPU Programmable Data Center Infrastructure On-a-Chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2022.

[56] Vern Paxson. End-to-End Routing Behavior in the Internet. In *IEEE/ACM Transactions on Networking*, volume 5, 1997.

[57] Petar Penkov, Eric Dumazet, and Stanislav Fomichev. Issuing SYN Cookies in XDP. In *Netdev, The Technical Conference on Linux Networking*, 2020.

[58] Mário Pinho. AWS Shield threat landscape review: 2020 year-in-review. https://aws.amazon.com/blogs/security/aws-shield-threat-landscape-review-2020-year-in-review/, 2021.

[59] Mohamed Rahouti, Kaiqi Xiong, Nasir Ghani, and Farooq Shaikh. SYNGuard: Dynamic threshold-based SYN flood attack detection and mitigation in software-defined networks. In *The Institution of Engineering and Technology Networks*, 2020.

[60] Dominik Scholz, Sebastian Gallenmuller, Henning Stubbe, Bassam Jaber, Minoo Rouhi, and Georg Carle. Me Love (SYN-)Cookies: SYN Flood Mitigation in Programmable Data Planes. In *P4 Workshop in Europe (EUROP4)*. Open Networking Foundation, 2020.

[61] Dominik Scholz, Andreas Oeldemann, Fabien Geyer, Sebastian Gallenmuller, Henning Stubbe, Thomas Wild, Andreas Herkersdorf, and Georg Carle. Cryptographic Hashing in P4 Data Planes. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2019.

[62] Cisco Security. TTL Expiry Attack Identification and Mitigation . https://sec.cloudapps.cisco.com/security/center/resources/ttl_expiry_attack.html#2, 2023.

[63] Amazon Web Services. AWS Best Practices for DDoS Resiliency. In *AWS Whitepaper*, 2022.

[64] Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-Defined Networks. In *ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[65] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/, 2018.

[66] Guangda Sun, Mingliang Jiang, Xin Zhe Khooi, Yunfan Li, and Jialin Li. Neobft: Accelerating byzantine fault tolerance using authenticated in-network ordering. In *ACM SIGCOMM*, pages 239–254, 2023.

[67] Microsoft TechNet. Syn attack protection on Windows Vista, Windows 2008, Windows 7, Windows 2008 R2, Windows 8/8.1, Windows 2012 and Windows 2012 R2. https://docs.microsoft.com/en-us/answers/questions/144446/synattackprotect.html, 2014.

[68] Check Point Software Technologies. Understanding Aggressive Aging. https://sc1.checkpoint.com/documents/R80.20/SmartConsole_OLH/EN/html_frameset.htm?topic=documents/R80.20/SmartConsole_OLH/EN/Wh_4163Q-r2uASm5pwt7Iw2, 2021.

[69] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Programmable in-network obfuscation of DNS traffic. In *NDSS: DNS Privacy Workshop*, 2021.

[70] Liang Wang, Prateek Mittal, and Jennifer Rexford. Data-plane security applications in adversarial settings. In *ACM SIGCOMM Computer Communication Review*, 2022.

[71] Sophia Yoo and Xiaoqi Chen. Secure Keyed Hashing on Programmable Switches. In *ACM SIGCOMM Workshop on Secure Programmable network Infrastructure (SPIN'21)*. ACM, 2021.

[72] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric DDoS attacks with programmable switches. In *Network and Distributed System Security Symposium*, 2020.

[73] Huancheng Zhou, Sungmin Hong, Yangyang Liu, Xiapu Luo, Weichao Li, and Guofei Gu. Mew: Enabling large-scale and dynamic link-flooding defenses on programmable switches. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

Appendix

A Breaking CRC-Based Cookies

Given the connection 4-tuple m and the secret nonce n, the CRC-based cookie is typically generated using $\mathrm{CRC}(m\|n)$. We note CRC is linear and affine: for any A, B, and C we have

$$\mathrm{CRC}(A)\oplus\mathrm{CRC}(B)\oplus\mathrm{CRC}(C)=\mathrm{CRC}(A\oplus B\oplus C). \quad (2)$$

The adversary can send a SYN packet with known input m_0 and observe the output $\mathrm{CRC}(m_0\|n)$. Subsequently, the adversary can compute $\mathrm{CRC}(m_0\|0)$ herself, and derive

$$\mathrm{CRC}(0\|n)=\mathrm{CRC}(m_0\|0)\oplus\mathrm{CRC}(m_0\|n)\oplus\mathrm{CRC}(0). \quad (3)$$

Now that the adversary knows $\mathrm{CRC}(0\|n)$, she can compute new cookie value for any arbitrary 4-tuple m_x, as follows:

$$\mathrm{CRC}(m_x\|n)=\mathrm{CRC}(m_x\|0)\oplus\mathrm{CRC}(0\|n)\oplus\mathrm{CRC}(0). \quad (4)$$

We also note that the same attack can be applied even if the nonce is placed at different input locations.

B Additional Implementation Details

B.1 Switch Agent: Processing Details

The switch agent processes two classes of packets from the client and two classes of packets from the server agent:

- *SYN packets from the client*: the switch agent computes a SYN cookie and returns this to the client in the ISN of the SYN-ACK response packet.
- *Non-SYN packets from the client*: the switch agent checks its Bloom filter to determine validity. If the Bloom filter reports a positive, the packet is forwarded to the server. Otherwise, the switch agent performs a cookie check, forwarding the packet to the server agent with a setup tag if it passes the check and dropping the packet otherwise.

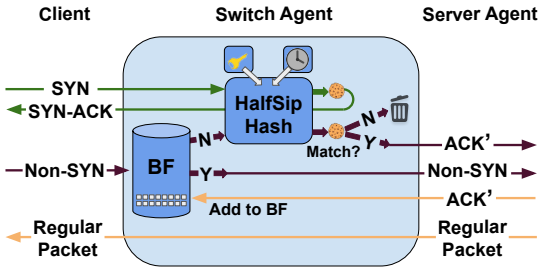


Figure 8: Switch agent logic flow.

- *Tagged confirmation packets from the server agent*: the switch agent adds the connection information to its Bloom filter. Note that instead of doing this directly after the cookie check, the switch agent waits for explicit confirmation from the second packet of the 2WHS, avoiding connection loss in scenarios where setup packets are lost between the switch agent and server agent (§6.1).
- *Regular packets from the server agent*: the switch agent passively forwards the packet to the client.

B.2 Server Agent: Linux eBPF Modules

The server agent is composed of eBPF modules that execute on the ingress and egress path of the kernel stack. Figure 9 demonstrates the tightly-knit functionality of the two modules, which can be understood as a progression of eBPF map `connection_states`. Entrance into these states is triggered by packets interpreted as *events*, and the associated *actions* for each state determine how packets are processed.

XDP Ingress Processing. eBPF programs can attach and execute at different kernel hooks, many of them occurring early on the path to the kernel network stack. eXpress Data Path (XDP) is a hook in the network driver that allows for execution of eBPF programs immediately after a packet is received off a network interface, before it even reaches the kernel TCP stack [41]. XDP operates on raw packet data before any socket buffer allocation. It offers the fastest in-kernel packet-processing without offloading to hardware [54].

The server agent's ingress program is attached at the XDP hook. As shown in in Figure 9, the ingress program initiates the 3WHS with the kernel network stack (**A1**) upon receipt of packets that are not already part of an ongoing connection and have a `setup` tag included (**E1**). The ingress program properly converts these setup packets into SYN packets and forwards them up the kernel stack. The sequence number delta for the connection is temporarily initialized and added to the map, and the `connection_state` stored in the map is set to `SYN_SENT` (**S1**). The ingress program also handles sequence number translations and checksum updates on all other incoming packets, before forwarding them up the network stack (**A4**). The module only processes TCP packets and simply passes non-TCP packets up the network stack.

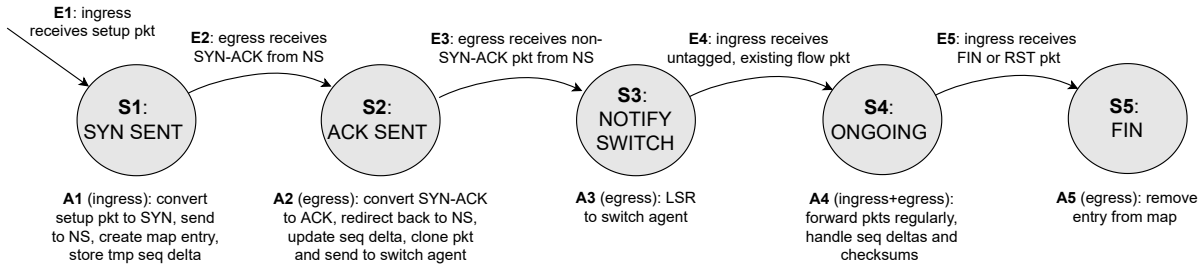


Figure 9: The server agent's eBPF map coordinates packet-processing between XDP ingress and TC egress modules.

TC Egress Processing. XDP is not yet supported on the kernel's egress traffic path, which is why we choose to implement egress processing at the traffic control (TC) hook, the next earliest hook in the stack. The TC hook offers access to the packet's socket buffer, providing greater processing flexibility at the cost of slightly lower processing speeds. If support for XDP on the egress path is added, SMARTCOOKIE's egress functionality can easily be ported from TC to XDP to further boost egress packet-processing performance.

SMARTCOOKIE's server agent uses a TC egress program to perform server-side setup and sequence number translations on outgoing packets. When the module receives an outgoing TCP packet from the network stack, it checks the stored `connection_state` and handles the packet accordingly (Figure 9). Generally, the server agent processes three types of packets from the switch agent:

- *Packets that are already part of an established connection*: the server agent performs sequence number and checksum updates and then passes the packet along to the kernel network stack for normal processing.
- *Packets with a setup tag that are not already part of an established connection*: the server agent knows the packet has successfully passed the SYN cookie check at the switch agent and bootstraps the connection establishment with the kernel network stack, saves the sequence number delta required for future packets, and sends the switch agent an explicit confirmation upon completion.
- *Packets that are not part of an established connection and also have no setup tag (false positives)*: the server agent performs a last-line-of-defense SYN cookie check on the packet, setting up the connection if it passes the check and dropping the packet otherwise.

C Routing Considerations

All traffic between clients and servers traverses a network switch (i.e., the edge switches are a 'cut' of the network between clients and servers). In a distributed defense with an upstream switch it is not guaranteed that all traffic will pass through the *same* switch in both directions (asymmetric routing) and traffic flowing in a single direction may also traverse different switches (routing changes). Asymmetric routing is very common due to hot-potato routing, a phenomenon where networks seek to pass traffic as quickly as possible out of their borders, resulting in traffic for one connection not taking the same path in both directions [56]. Routing changes, while less common at the connection level, can still occur and in particular affect longer-lived connections [56]. With an upstream defense, traffic from an ongoing flow could traverse a switch that did not perform the initial cookie check. This must be handled gracefully so the client can avoid additional verification or connection disruption.

C.1 Handling Asymmetric Routing

The SMARTCOOKIE protocol has removed the need for the switch agent to remain actively involved (i.e., with sequence number translations) in a connection once it has been established. This ensures that the switch agent need only *forward* return packets, especially since this traffic is coming from the servers and can be trusted without undergoing verification. Thus, once the setup phase has been completed, *any* switch in the provider backbone can safely forward the rest of the return traffic, supporting asymmetric routing and direct server return (DSR), regardless of the extent of asymmetry.

During connection setup, however, the switch agent must see the return traffic from the server to properly add the connection to its local record of verified connections. To accomplish this, the SMARTCOOKIE server agent ensures response traffic is routed through a single switch agent during setup by using loose source routing (LSR) to explicitly route response traffic from the server to the client through the *same* switch agent, until the switch agent has created a local record. In practice, when the switch agent sends a tagged packet to the server agent, it includes a switch agent identification (i.e., the IP address of the switch) that the server agent can use for routing relevant return packets. The server agent knows when the switch agent has updated its local record since the switch agent will stop tagging packets once the update is complete.

C.2 Handling Routing Changes

Routing changes are less common at the TCP connection level, especially when flow-aware load balancing (e.g., ECMP) is used instead of packet spraying [43]. Still, our switch agent must gracefully handle routing changes caused by configuration updates, equipment failures, or other unexpected reasons. We consider a scenario where a client has passed the cookie check at a given switch agent, `SwitchAgent_1`, and has been added as a verified connection to its local Bloom filter (BF), but has some packets routed to the server through a different switch agent, `SwitchAgent_2`, which does not have records of the connection. In this case, SMARTCOOKIE should ensure that a verified client would not get penalized by experiencing connection disruption (e.g., packet drops or connection reset) at the new switch agent.

BF Synchronization. For provider backbones that expect routing changes more regularly (e.g., with packet spraying), one solution could be to replicate BF state across the different switch agents, maintaining a copy of the set of all verified connections at each switch agent in a fault-tolerant manner [47]. However, if routing changes are expected infrequently (e.g., all connections are short-lived), synchronizing BF state could be avoided, as this would fill up the BFs unnecessarily and degrade the membership-reporting accuracy of the filters.

Packet Sampling. To avoid synchronizing state across BFs, the SMARTCOOKIE protocol could allow individual BFs to dynamically adapt to routing changes. To accomplish this, the switch agent could probabilistically allow a small sampling of non-SYN packets through to the server agent even if the packets are not in the BF and have not passed the SYN cookie check. By performing stateless rate limiting with probabilistic sampling, SMARTCOOKIE would trade off the risk of a routing change causing benign traffic to be dropped and the risk of allowing attack traffic through to the server agent.

More concretely, we consider a case where an adversary sends attack traffic with the same connection 4-tuple (spoofed from multiple vantage points) in order to cause traffic to enter the provider backbone through more than one switch agent. If the attack traffic consisted of SYN packets, each of the switch agents would initiate the SYN cookie check and the attack would be stopped at the network edge. For non-SYN packets, the switch agent would probabilistically let through a small number of unverified packets, placing a special tag on these packets before forwarding them to the server agent.

Upon receiving a packet with this special tag, the server agent would check to see if the packet was part of an active connection. If it was, the server agent would use LSR to respond to the specific proxy that sent the packet, signifying all is well and the switch agent can add the connection to its BF. However, if the packet was not part of an active connection at the server, the server agent would simply drop the packet and no further action would be taken. We note that this design requires minimal effort from the server agent, as it would simply perform connection lookups and would *not* perform any cookie checks for these special packets.

D Miscellaneous and Future Work

Header Field Compatibility and MTU. Setup tags added to packet headers are only visible within the cloud provider's internal network, between switch agent and server agent. Any such modifications are removed before unmodified packets are passed to the server's unmodified TCP stack. SMARTCOOKIE mostly tags handshake packets during connection setup, when packets are only 40 - 60 bytes. However, in special cases (setup packets dropped or routing changes) SMARTCOOKIE might tag a client's data packets, so the internal network's MTU should be roughly 10-20 bytes larger.

Integration with the Network Stack. The server agent interfaces with an unmodified TCP stack, which is useful in scenarios such as serving tenant VMs with customized kernels, or even VMs running a different guest operating system. However, the server agent needs to pay a small overhead for translating sequence numbers on every TCP packet. Without changing the trust model or requiring tenant VMs to have access to secret keys, a tighter integration with the Linux kernel of trusted physical servers can eliminate this performance penalty. Specifically, the server agent can explicitly instruct the kernel to choose the desired initial sequence number (ISN) and synchronize numbers between the switch and server.

Synchronize Initial Sequence Number. An unmodified Linux kernel chooses its ISN at random. By integrating the server agent 2WHS with the kernel's TCP connection setup, we could explicitly instruct the kernel to adopt the ISN chosen by the switch agent. This minor change in the kernel network stack would remove the need to perform sequence number translations in the server agent altogether, further improving its performance and automatically allowing the switch agent to avoid keeping per-flow state.

Avoid Duplicate Per-Flow State at Server Agent. We also note that currently the eBPF-based server agent maintains per-flow state for all TCP connections in an eBPF map, duplicating the kernel-maintained list of all TCP sockets. This leads to a small but non-negligible memory overhead. Future implementations of the server agent could query the connection state from the kernel's list of active TCP connections directly. This would allow the server agent to become stateless after connection setup, acting as an extremely lightweight module that only handles the 3WHS with the TCP stack and does not keep per-flow state.

Offloading the Server Agent. We can push the server agent to run directly on a NIC that supports eBPF hardware offloading, removing the overhead of running on the server's CPU. We can also run the server agent on a SmartNIC (e.g. NetFPGA), or even on a top-of-rack (ToR) switch in front of the server. These techniques would allow the server's CPU to be dedicated to only running application logic.