Hardware SYN Attack Protection For High Performance Load Balancers

Reuven Cohen[1], Matty Kadosh[2], Alan Lo[2], and Qasem Sayah[1]

[1]Department of Computer Science, Technion, Israel
[2]Nvidia

Abstract

SYN flooding is a simple and effective denial-of-service attack, in which an attacker sends many SYN requests to a target's server in an attempt to consume server resources and make it unresponsive to legitimate traffic. While SYN attacks have traditionally targeted web servers, they are also known to be very harmful to intermediate cloud devices, and in particular to stateful load balancers (LBs). Fighting against a SYN attack without negatively affecting legitimate connections is not easy, especially if the LB needs to perform frequent server pool updates during the attack, which is very likely since attacks can often last for many hours or even days.

We are the first to propose LB schemes that guarantee high throughput of one million connections per second, while supporting a high pool update rate without breaking connections, and fighting against a high rate SYN attack, of up to 10 million fake SYNs per second.

1 Introduction

A SYN flood attack is an easy, effective, and popular form of a distributed denial of service (DDoS) attack, which exploits the TCP three-way handshake connection establishment process [24]. This process starts when the client sends a SYN packet to the server. The server creates a new state for the to-be-established connection, sets this state to "half-open" and sends a SYN of its own to the client. Upon receiving an ACK to its SYN, the server changes the connection's state to active.

Servers have limited resources for half-open connections, and they place a threshold on the maximum number of such connections. When this threshold is reached, new SYNs are dropped. In a SYN flood attack, the attacker sends many fake SYN packets to the victim server, without completing the three-way handshake for these SYNs. These fake SYNs may consume all of the resources allocated by the server for half-open connections, and the server must then drop new SYNs, including those of legitimate connections.

SYN flood attacks have been known about for many years. Their popularity stems from their capacity to be performed distributedly, using spoofed source IP addresses, and how difficult it is to mitigate them. In this paper, we study the impact of this attack on state-of-the-art hardware load balancers employed by cloud operators.

In a cloud network, a cloud service can be identified by a virtual IP (VIP) address. Each VIP is mapped to a pool of servers, and each server is uniquely identified using a direct IP (DIP) address. Since there can be many DIPs associated with the same VIP, a load balancer (LB) is needed. The LB distributes the connections destined for a certain VIP to the various DIPs associated with this VIP. For each TCP connection, a "connection key" is calculated using a hash function on the packet header's 5-tuple. The key is identical for all the packets of the same connection, and the LB must ensure that all the packets with the same key are forwarded to the same DIP, as long as this DIP does not malfunction. This is known as per-connection consistency (PCC) [11].

PCC must be guaranteed even if the DIP pool changes. Such a change happens when more DIPs are added to the pool, when some DIPs have to be gracefully taken down, or when the assignment function has to be changed.

While SYN attacks have traditionally targeted web servers, they are known today to be very harmful also to intermediate cloud devices, and in particular to LBs. Most of the hardware load balancers known in the literature [10, 19] are susceptible to such attacks. SilkRoad [19] addresses SYN attacks by associating a rate-limiter to the SYNs forwarded to each VIP for detecting and dropping excessive traffic. If an upper rate threshold is crossed, SYN packets for the given VIP are dropped. A similar approach is used by firewall and proxy devices [6, 23]. However, with this approach, **the LB does not distinguish between real SYNs and fake SYNs, and drops them together**. This creates a huge negative impact on the setup time of real connections. In this paper, we propose schemes that use the LB's hardware to **discard fake SYNs only, with no impact on real connections**.

Fighting against a SYN attack without negatively affecting legitimate connections is not easy. But it is even more difficult if the LB needs to perform a pool update for a server under attack, which is very likely since attacks can often last for

many hours or even days. This is mainly because the same hardware resources needed for guaranteeing PCC during a pool update are also needed to fight SYN attacks.

This paper is the first to propose LB schemes that guarantee (a) a high connection rates of one million connections per second, and up to 100 million total connections, while (b) supporting a high pool update rate (more than 1 VIP per second) without breaking connections, and (c) fighting against a high rate SYN attacks (up to 10 million fake SYNs per second) without affecting real connections.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents background information. Section 4 discusses the challenges imposed by SYN attacks on the LB. Section 5 presents our cookie hardware table (CHT), which allows us to distinguish between real and fake connections. Section 6 discusses pool updates during a SYN attack. Section 7 presents a proof of concept. Section 8 describes how to mitigate almost any SYN attack rate using switch cryptography support, and Section 9 concludes the paper.

2 Related Work

SYN attack mitigation using SYN cookies was shown to perform better than other solutions [15, 17]. While it was believed that with SYN cookies, the server must reject all TCP options, some solutions were proposed to address this problem [17, 25].

SYN cache [15] is another technique that can be used to defend against SYN flood attacks. In contrast to using SYN cookies, which do not require the server to keep any state following the receipt of a SYN1 packet, SYN cache aims to reduce the size of a connection state. The idea is that instead of a per-socket queue, a global hash table is used for holding connection states. The hash table is implemented using a limited number of hash buckets. A new connection state is inserted into a specific hash bucket according to the connection 5-tuple and a secret hash key. This prevents the attacker from targeting a specific hash bucket.

Another approach for fighting against SYN attacks in datacenters is using a SYN proxy [6] – an intermediate network device that verifies the completion of the three-way handshake before the connection is forwarded to a server. This concept can be used by modern LBs, but its scalability is in no way comparable to the schemes presented in this paper.

Beamer [21] is a stateless software LB, which argues that stateful LBs are unable to cope with SYN flood attacks. In [21], it is shown that under a SYN flood attack of one million SYNs per second, Beamer is not affected, while stateful LBs start breaking connections.

Ananta [22] is Microsoft's software LB, deployed in the Azure public cloud. Ananta handles SYN flood attacks by isolating inbound traffic to the attacked VIP. The attack detection time can be more than 120 seconds for a busy VIP. Moreover, this solution completely shuts down the VIP service, unlike the solutions proposed in this paper.

Finally, SilkRoad [19] is a hardware stateful LB that addresses SYN attacks by associating a rate-limiter to the SYNs forwarded to each VIP. This rate-limiter is used for detecting excessive SYN rate. Like in Ananta, upon detecting an attack, SYNs received for the attacked VIP are dropped. Thus, the attacked VIP cannot serve legitimate connections until the attack stops.

3 Background

In TCP, connection establishment is a three-way handshake, which requires a SYN from the client, a SYN from the server, and an ACK from the client for the server's SYN. For the rest of this paper, these packets are referred to as SYN1, SYN2, and ACK3, respectively.

3.1 Prism LB

While the schemes proposed in this paper are applicable to many LB technologies, it is easier to present them in the context of a specific architecture. Prism is a newly proposed LB architecture that can handle an arrival rate of one million new connections per second, and a total number of more than 100 million simultaneous connections, while supporting more than one pool change per second and guaranteeing PCC [5].

Standard hardware LBs use a hardware equal-cost multi-path (ECMP) table, which forwards packets destined for the same logical address to different "next hops".[1] Each VIP has its own ECMP table, which holds the next hop information for each DIP in this pool. Packets are mapped based on their connection's signature to one of the ECMP table entries (also known as "bins"). For example, the ECMP table in Figure 1 holds the pool {DIP1, DIP2, DIP3, DIP4} for VIP1. This specific table aims to balance the load between the four DIPs evenly, and therefore 1/4 of the entries are allocated to each of the four DIPs.

Prism's main contribution is guaranteeing PCC while maintaining a hardware state only for connections affected by a pool update. These connections are inserted into a special hardware table, called MCT (migrated connection table). For example, suppose that the content of bin 3 in Figure 1 changes from DIP3 to DIP4. Every connection whose packets were forwarded to bin 3 before the change is added into the MCT, and its MCT entry indicates that packets of this connection must be forwarded to DIP3. Packets received by Prism from the clients are first checked against the MCT for a hit (Figure 1). If there is an MCT hit, the identity of the DIP to which the packet should be forwarded is taken from the corresponding MCT entry. If there is no MCT hit, the packet continues to

[1] A "next hop" can be a switch port, a tunnel, a MAC address, or an IP address.

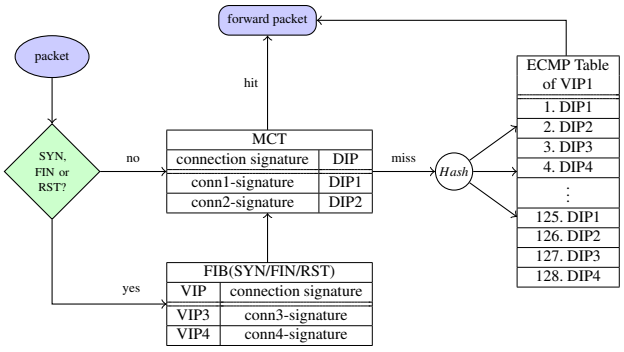


Figure 1: Packet flow for a specific VIP (VIP1) in Prism

the ECMP table, where it finds the default DIP corresponding to its connection signature.

The length of the ECMP table for each VIP is determined according to the expected traffic distribution, such that the number of connections routed through the same bin does not exceed a certain threshold. This is important because during a server pool update, all the connections associated with the same bin might have to be added into the MCT within a short time period.

When a SYN is received by Prism, the signature of the packet's 5-tuple is recorded by a hardware learning table, called FIB(SYN1). To make sure that the server agrees to establish the connection, Prism also records signatures of the SYN2 packets, coming from the servers, in a FIB(SYN2) table. The software periodically fetches these FIB tables, and inserts the learned signatures into the **software** connection table (SCT).

While the MCT maintains a state only for connections whose default bin has changed, the SCT is used for tracking the status of all open connections. Each signature fetched from FIB(SYN1) is inserted into the SCT with a pending (half-open) state. The state is updated to active upon capturing a corresponding server SYN2 in FIB(SYN2). Prism also uses a FIB(FIN1) table to learn about FIN packets sent by the clients, and a FIB(FIN2) table to learn about FIN packets sent by the servers. Upon receiving two FINs for a connection, Prism removes the connection signature from the SCT and from the MCT, if applicable.

Prism hashes the connection 5-tuple key into a shorter connection signature, because the 5-tuple is too long to be used in the hardware tables. Using signatures introduces collisions, i.e., connections with different 5-tuple keys, but with the same signature. In Prism, signatures are designed such that if two different connections have the same signature, they must have the same VIP. Thus, Prism deals with hash collisions by maintaining a counter that indicates the number of different connections currently mapped to the same SCT (and MCT, when applicable) entry.

3.2 SYN Cookies

SYN cookies [3, 7, 24] represent a technique used by servers to protect against SYN flood attacks. Without this technique, a server that receives a SYN packet constructs a SYN queue entry for the connection state, and sends a SYN packet back to the client. This SYN packet has a random initial sequence number. When the server receives an ACK for its SYN, it removes the connection from the pending connection queue, and creates an active regular connection.

With SYN cookies, the server does not maintain a state for pending connections. Rather, upon receiving a SYN1, it creates a SYN2 of its own, with a special initial sequence number known as a cookie. When the server receives an ACK that fits no existing connection, it checks if this ACK contains a sequence number that can be considered a legitimate cookie, and therefore could have been sent as a response to a SYN2. If this is the case, the server considers this ACK as a legitimate ACK3, and creates a new corresponding connection in the SCT. If the server uses SYN cookies whenever its queue of pending connections reaches a certain threshold, and in particular during a SYN attack, it can continue accepting new connections without allocating resources to pending connections.

A SYN cookie is a 32-bit number that consists of (see Figure 2): (a) a 5-bit timestamp with a resolution of 64 seconds, used for preventing replay attacks; (b) three bits derived from the maximum segment size (MSS) of the connection; (c) a 24-bit hash of the connection 5-tuple and a secret key. A SYN cookie is verified as follows. First, the timestamp is checked to make sure that this ACK falls into the correct time interval. Then, the packet's 5-tuple is hashed and checked against the first 24 bits of the cookie. If there is no match, the ACK is dropped. Otherwise, the cookie is considered legitimate and a corresponding connection is established. The MSS value for the established connection is derived from the the cookie's MSS 3-bit field.

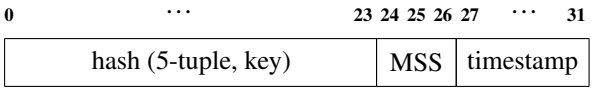


Figure 2: A standard 32-bit SYN cookie

4 Challenges Imposed by SYN Attacks on the LB

As explained in Section 3.1, Prism creates a connection state in the SCT when the client's SYN is fetched by the software from the FIB(SYN1) table. Then, the software inserts the connection's signature as a new SCT entry whose state is pending. Later, when the software fetches the corresponding

server's SYN from the FIB(SYN2) table, Prism updates the signature's state in the SCT to active.[2]

From the above description follows that there is a significant difference between Prism and the server when a new connection is established. Prism needs only SYN1 and SYN2 packets to set a connection as active, while the server also needs the ACK3 packet. This implies that Prism may consider fake connections as real, and will have to spend expensive MCT resources to get rid of them. **This is a problem of all hardware-based LBs.** For example, SilkRoad [19] inserts all fake connections into its hardware table and removes them only after a timeout.

Prism was designed to address the following challenges **using standard, programmable, switch hardware**:

- **Challenge-1** Performance: supporting an arrival rate of one million connections per second, and up to 100 million simultaneous active connections. This requires that all packets, including SYNs, be processed by the hardware.
- **Challenge-2** PCC: ensuring per-connection consistency under a heavy pool update rate of more than 1 VIP per second.

SYN attacks introduce a new challenge as follows:

- **Challenge-3** Coping with a very high rate of fake SYNs, while preventing any impact on real connections. When we started this research, we hoped to mitigate an attack of one million SYNs per second on a single VIP, while not affecting the one million connections per second received for all VIPs. But eventually we were able to mitigate a much higher rate, of 10 million fake SYNs per second.

With respect to Challenge-3, the following switch constraints play a critical role:

- **FIB-constraint** Keeping the size of the FIB tables small not only during normal operation but also during a SYN attack, **without missing real SYNs**. This constraint arises because the FIB(SYN1) and FIB(SYN2) tables record all SYN packets, including those of fake connections. Since these hardware tables have limited length and the software is limited to fetching only a few millions FIB entries per second, fake SYNs may prevent real SYNs from being recorded.
- **MCT-constraint** The MCT is used for guaranteeing PCC following a pool update. Like the FIB learning tables, the MCT is also very limited: it is designed for holding only a small fraction of the connections (up to 0.1%), and the rate of adding new connections into it is also limited.[3] If these scarce MCT resources are used for fake connections, the LB will not be able to use them for real connections during a pool update, and these real connections will be broken.

[2] If a signature remains in the SCT "too long", namely, longer than N times the average lifetime of a connection with the corresponding VIP (N is a per-VIP parameter whose default value is 2), the signature is copied into the MCT to verify that it is active. If it is found to be active, i.e., a packet from the client is received for this signature within a short timeout period, the signature is removed from the MCT and kept only in the SCT. Otherwise, it is removed from both tables.

Symbol	Meaning	Value
r_{real}	arrival rate of real SYNs	1 million/s
r_{fake}	arrival rate of fake SYNs	**to be maximized**
$r_{polling}$	polling rate of the FIB	100/s
$\|pool\|$	number of servers (DIPs) in the pool of a VIP	up to a few thousands
$\#bin$	number of ECMP entries (bins) allocated to a VIP	up to a few thousands
$\#binTotal$	total number of bins allocated to all VIPs	up to 100,000
$\|CHT\|$	number of cookies in CHT	up to 100,000
RTT_{max}	maximum tolerable RTT	10–50ms
T_{RST}	the time a server waits for ACK3 before sending a RST	5s
T_{CHT}	the timeout for removing a cookie from CHT	RTT_{max}
$T_{connection}$	average connection lifetime	100s
$T_{suspected}$	time after which an active signature is suspected to be inactive	N·$T_{connection}$ (default N=2)
T_{MCT}	the time a signature stays in the MCT for activity detection	5s
$T_{exposed}$	the time between inserting a fake signature into the SCT and removing it	T_{CHT}

Table 1: Notations

For the rest of the paper, the scheme described in Section 3.1 is referred to as Scheme-1. This scheme can work under a SYN attack if the server does not use SYN cookies, in the following way. When a server receives a fake SYN, it waits a short configurable timeout for the client to send an ACK3, and then resets the connection by sending a RST packet. Prism records RST packets coming from the servers in a FIB(RST) hardware table. It periodically reads this table and removes the corresponding connections from the SCT and, when applicable, also from the MCT.

However, Scheme-1 has the following problems (Table 1 presents notations used throughout the paper for analysis):

1. The scheme does not work when the servers use SYN cookies. This is because in such a case the servers do not send a RST packet for unestablished connections.
2. Violation of Challenges 1, 2, and 3: Under a heavy SYN

[3] In our prototype switch, this rate is ≈ 25,000 per second.

attack, the LB is unable to accommodate a rate of one million real connections per second, and it cannot support a high rate of pool update while ensuring PCC. Our prototype shows that the LB is not even close to achieving these challenges.

Scheme-1 does not address Challenges 1, 2, and 3 due to hardware limitations. With respect to the FIB-constraint, under a SYN attack, the length of the FIB tables significantly increases. With no attack, FIB(SYN1) needs to maintain $\frac{r_{real}}{r_{polling}}$ entries, where r_{real} is the arrival rate from real clients and $r_{polling}$ is the FIB polling rate. Under a SYN attack, FIB(SYN1) needs to maintain $\frac{r_{real}+r_{fake}}{r_{polling}}$ entries – an increase by a factor of $\frac{r_{real}+r_{fake}}{r_{real}}$. For example, if r_{real} = 1 million per second, r_{fake} = 10 million per second, and $r_{polling}$ = 100 per second, the number of entries needed to be recorded by FIB(SYN1) and FIB(SYN2) increases by a factor of 11, from 10,000 to 110,000.

With respect to the MCT-constraint, under a SYN attack, the length of the MCT during a pool update also significantly increases. Using Scheme-1, a VIP under attack requires the MCT to maintain additional $[\frac{1}{|pool|} \cdot r_{fake} \cdot T_{exposed}]$ fake signatures[4], where $|pool|$ is the number of servers in the pool of the attacked VIP. As shown in [5], the MCT holds at most 70,000 entries during the most aggressive update rate (1 bin per second, each holding $\approx$ 1,000 connections). With $T_{exposed}$ = 5 seconds, $|pool|$ = 100, and a SYN attack rate of r_{fake} = 10 million per second, the MCT would need an additional 500K entries – an increase by a factor of 8.

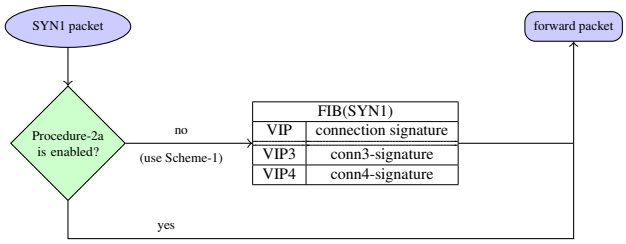
State-of-the-art LB architectures address SYN attacks using two thresholds for the number of pending connections associated with each VIP [19]. When the upper threshold is crossed, the LB drops SYN1 packets that are destined for the given VIP until the number of pending connections reaches the lower threshold. But this approach does not distinguish between real and fake connections, and it therefore violates Challenge-1 and Challenge-3. It also violates PCC (Challenge-2), because fake SYNs consume the MCT resources that are needed during pool updates.

5 Using a Cookie Hardware Table (CHT)

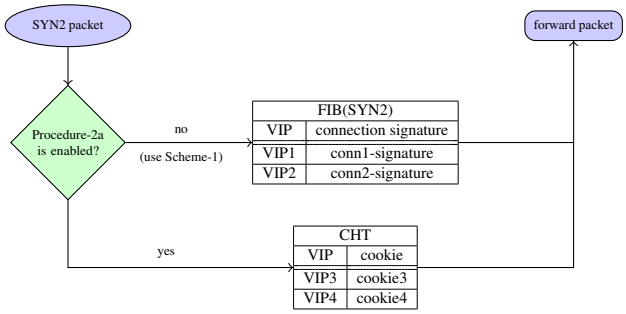
5.1 Procedure-2a

We now introduce a new LB procedure, called Procedure-2a, whose most important property is that connection signatures are added into the SCT only when their ACK3 packet is received. Using Procedure-2a, SYN1 packets are completely ignored by the LB, and SYN2 packets are only processed by the hardware. Thus, the software and the SCT are never exposed to fake SYNs.

[4]Upon adding or removing a DIP, approximately $\frac{1}{|pool|}$ of the bins of the given VIP are modified. Thus, $\frac{1}{|pool|}$ of the signatures of this VIP are expected to be added into the MCT.



(a) SYNs coming from the clients (SYN1)



(b) SYNs coming from the servers (SYN2)

Figure 3: The flow of SYNs for Procedure-2a

When Prism suspects that there is a SYN attack on some VIP[5], it executes Procedure-2a rather than Scheme-1 on packets sent to and received from this VIP. In Procedure-2a, Prism uses a new hardware table called a CHT (Cookie Hardware Table) for recording SYN cookies from the SYN2 packets coming from the servers of the attacked VIP's pool.

Figure 3 shows the packet flow for Procedure-2a. As already said, this procedure completely ignores SYN1 packets (Figure 3(a)). When a SYN2 packet is received, the sequence number field of this packet plus 1 is considered as a SYN cookie and is recorded by the CHT (Figure 3(b)). If Prism does not consider a VIP to be under attack, it runs Scheme-1 on all the packets sent and received for this VIP.

Figure 4 presents the flow of every ACK packet coming from the clients in Procedure-2a. When Prism receives such a packet, it does not know if this is an ACK for a server's SYN, namely, ACK3, or an ACK for a "regular" non-SYN packet. Thus, it must process all ACKs in the same way. It first checks the acknowledgment sequence number field against the CHT. If there is no match, the packet is considered a regular ACK (not ACK3), and is forwarded to the server. If there is a match, the packet is considered an ACK3 and its signature is recorded by a FIB(ACK3) table. This is a new table, not used by Scheme-1. When FIB(ACK3) is fetched by the software, a new SCT entry is created for this signature, and its state is set to "active". Thus, the FIB(ACK3) table of Procedure-2a

[5]When exactly to start running Procedure-2a is an important question, addressed in Section 6.4.

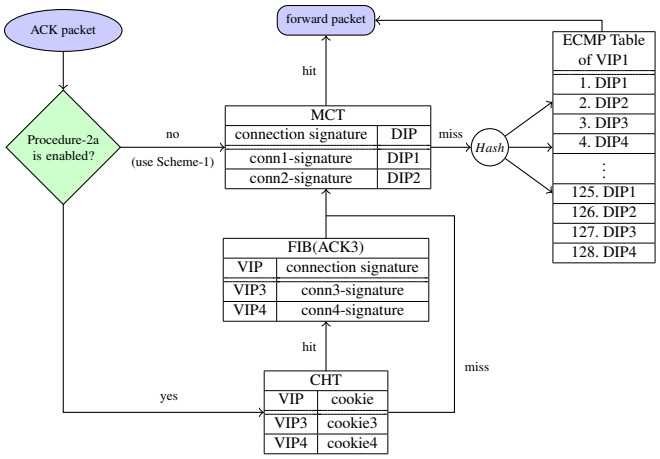


Figure 4: The flow of ACK packets coming from the clients in Procedure-2a

actually replaces the FIB(SYN1) and FIB(SYN2) tables of Scheme-1.

The CHT is implemented as a hardware-based cuckoo hash table [14, 16]. Every ACK packet sent to a VIP for which Procedure-2a is enabled is checked in $O(1)$ against every CHT entry associated with this VIP. Each cookie is maintained by the CHT for a timeout period, whose value is equal to RTT_{max}, a constant that indicates the maximum tolerable RTT in the system. After the timeout period, the entry is removed by the hardware.

Prism's algorithm for Procedure-2a can be summarized as follows:

- Ignore SYN1 packets.
- Upon receiving a SYN2 packet, insert the packet's cookie into the CHT.
- Upon receiving a client ACK packet, check its acknowledgment number against the CHT. If there is a hit (this is an ACK3), add the ACK's signature into the FIB(ACK3) table and forward the packet to the server. If there is no hit, forward the packet to the server.
- Read FIB(ACK3) periodically. Consider all signatures as new active connections and add them into the SCT.

Procedure-2a has two important properties:

1. SYN2 packets for which no ACK3 is received are never considered as pending or active connections and are never added to the SCT. Moreover, the cookies of such SYNs are quickly removed from Prism. This makes Prism completely transparent to fake SYNs.
2. Procedure-2a works correctly also for a VIP whose servers do not use SYN cookies. This is because regardless of whether a server uses or does not use SYN cookies, the client is always required to send an ACK3 with a legitimate acknowledgment number.

r_{fake} \ T_{CHT}	$10ms$	$20ms$	$50ms$
0/s	10K	20K	50K
1 million/s	20K	40K	100K
10 million/s	110K	220K	550K

Table 2: The maximum number of CHT entries needed for Procedure-2a, as a function of r_{fake} and T_{CHT}

Procedure-2a addresses Challenges 1 and 3 by processing all packets in hardware and not dropping legitimate SYN1 packets. However, as shown in Section 6, it does not guarantee PCC when a VIP's pool is updated during a SYN attack, and it therefore violates Challenge-2. Section 6 also shows how to solve this problem.

In Procedure-2a, the length of the CHT grows linearly with r_{fake} and T_{CHT}, namely, the rate of the attack, and the timeout after which a cookie is removed from the table. Table 2 shows the number of entries needed for certain r_{fake} and T_{CHT} values. Not all these combinations can be supported by today's off-the-shelf switches. For example, in Spectrum 2 [18], there are only 512K available hardware entries, which must be shared between all tables.

Thus, with today's hardware, Procedure-2a can handle 10 million fake SYNs per second only if $T_{CHT} = RTT_{max} = 10ms$. But, RTT_{max} can be $10ms$ only if the cloud uses regional datacenters that are located close to the users [1, 20].

5.2 False Positive Analysis

Procedure-2a may introduce false positive CHT hits. A false positive here is an ACK that hits a CHT entry not because it was sent as a response to a SYN2, but because it accidentally contains an acknowledgment number that appears in the CHT. False positives can be divided into two categories: (1) an ACK of an active connection that hits a real cookie or (2) an ACK of an active connection that hits a fake cookie (a cookie generated from a fake SYN1). In both cases, the ACK's connection signature is learned by FIB(ACK3), and then fetched by the software and added into the SCT as active. Such an anomaly, where the SCT contains a connection that does not actually exist, may happen not only due to CHT false positives, but also due to other exceptions, such as a failure of a client before sending a FIN. Prism knows to handle such exceptions with a small overhead [5].

Since the length of a SYN cookie is 32 bits, the probability for a "regular ACK" (not ACK3) to falsely hit a cookie is $\frac{|CHT|}{2^{32}}$. Assuming that at most 100 million normal ACKs pass through Prism every second[6], the expected number of

[6] The switch can forward tens of millions of ACKs per second, depending

false positive signatures per second is 2,328, assuming that the CHT holds 100,000 cookies at any time. Each of these signatures stays in the SCT for $T_{suspected}$ seconds, after which the signature is copied into the MCT for activity detection. This means that the total number of false positive signatures held by the SCT is $2{,}328 \cdot (T_{suspected} + T_{MCT}) = 710{,}000$, an overhead of only 0.71%. After T_{MCT} seconds in the MCT, a false positive signature is considered inactive and removed from the MCT and SCT. This means that approximately $5 \cdot 2{,}328 = 11{,}640$ false positive signatures are expected to be in the MCT simultaneously for activity detection. Since the maximum MCT length (with a pool update rate of 1 per second) is $\approx 70{,}000$ [5], false positive signatures impose an overhead of $\approx 16\%$ on the MCT.

To reduce this overhead, we add an extra 16-bit field to each CHT entry (Figure 5). These bits store a hash of the SYN2 packet's 5-tuple.[7] Using this "extended cookie", the probability for a regular ACK to falsely hit an extended cookie is only $\frac{|CHT|}{2^{48}}$. With a rate of 100 million regular ACKs per second, the expected false positive rate drops to approximately one every 28.15 seconds, which is translated into at most 1 false positive signature in the MCT at any given time.

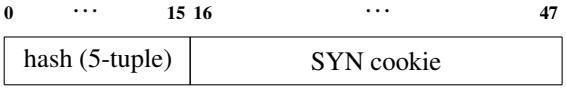


Figure 5: A 48-bit extended CHT entry for reducing false positive probability

6 Pool Updates During a SYN Attack

6.1 Procedure-2a May Violate PCC

The importance of PCC during pool updates has been a well-known issue for years [8, 11, 12]. Performing a pool update during a SYN attack is even more important than performing an update during regular operation, because the VIP operator may wish to mitigate the attack by adding more DIPs, or by taking down some DIPs for software update.

When Scheme-1 is used, Prism can support up to one million new connections per second, while accommodating a high pool update and guaranteeing PCC [5]. To explain how Scheme-1 guarantees PCC, we define "affected bins" as ECMP bins whose content should be modified during the considered pool update, and "affected signatures" as connection signatures that have been mapped to affected bins.

on its throughput and on the packet MTU. For example, with a throughput of 1Tb/s and an MTU of 1,500 bytes, the number of ACKs is ≈ 92 million per second.

[7] This is similar to the connection signature, except that the full signature contains 64 bits and not only 16.

Scheme-1's pool update algorithm is separately invoked for each affected bin in the following way. First, all signatures of the given bin are copied into the MCT in one or more iterations. Iteration $i+1$ is needed only if new SYNs are received for the given affected bin during iteration i. In most cases, this process converges after 1-2 iterations [5]. Only then can the bin's content be updated. In a system with 1,000 VIPs, each with 100 DIPs serving roughly the same number of active connections, Prism can perform at least 20 bin updates per second, each with 1,000 signatures on average. This is roughly equivalent to adding or removing one DIP per second.

We have seen that in Procedure-2a, the SYN1 and SYN2 packets are not recorded, and only ACK3 packets trigger the insertion of a connection into the SCT. While this allows Prism to address Challenges 1 and 3, it violates PCC during a pool update. To understand why, consider a connection whose SYN1 is directed to bin_i at t^{SYN1}, and its ACK3 is received by the LB at t^{ACK3}. Let t be the time when the content of bin_i is updated, say from DIP1 to DIP2, and consider the following possible relationships between t, t^{SYN1} and t^{ACK3}:

1. $t < t^{SYN1}$: in this case PCC is guaranteed, since both SYN1 and ACK3 are forwarded to DIP2.
2. $t > t^{ACK3}$: in this case PCC is guaranteed in the following way. First, both SYN1 and ACK3 are forwarded to DIP1. Then, between t^{ACK3} and t, the connection's signature is added to the MCT. This guarantees that all future packets of this connection continue to be forwarded to DIP1 until the connection ends.
3. $t^{SYN1} < t < t^{ACK3}$: this is a faulty case. First, the SYN1 packet is forwarded to DIP1. Then, the content of bin_i is modified while Prism is unaware of this connection, and has no entry for this connection in the MCT or the SCT. Finally, the ACK3 is sent to DIP2 and the connection is broken.

6.2 Procedure-2b

From the previous discussion, we can conclude that to guarantee PCC, we must make Prism aware of SYN1 packets. We now present Procedure-2b. The new procedure is similar to Procedure-2a, except that the signatures of SYN1 packets are also recorded by FIB(SYN1). The software periodically fetches these signatures, adds them into the SCT, and sets the state of each signature to pending. This state is later modified to active upon receiving the corresponding ACK3 packet. An SCT entry that remains pending for a timeout period is considered fake and is removed.

The pool update algorithm for Procedure-2b is the same as for Scheme-1. For each affected bin, all affected signatures are copied into the MCT, including those whose state is pending. Some of these pending signatures are fake, but their timeout has not yet expired, so they cannot yet be distinguished from real ones.

Like in Scheme-1, the SCT in Procedure-2b needs to hold approximately $\lceil r_{fake} \cdot T_{exposed} \rceil$ fake pending signatures. However, while in Scheme-1 $T_{exposed}$ is equal to T_{RST} = 5s, in Procedure-2b, it is equal to $T_{CHT} = RTT_{max}$. This reduces the overhead of fake signatures on SCT by a factor of 100-500, as shown in Table 3. For example, with r_{fake} = 10 million SYNs per second and $RTT_{max} = 10ms$, the overhead in Scheme-1 is 50%, as discussed in Section 4, while the overhead in Procedure-2b is $\lceil r_{fake} \cdot T_{CHT} \rceil$ = 100K entries from 100 million, namely, 0.1%.

r_{fake} \ LB algorithm	Scheme-1	Procedure-2b (RTT_{max}= 10ms)	Procedure-2b (RTT_{max}= 50ms)
1 million/s	5%	0.01%	0.05%
5 million/s	25%	0.05%	0.25%
10 million/s	50%	0.1%	0.5%

Table 3: The expected overhead of fake signatures in the SCT as a function of r_{fake} and RTT_{max}

To analyze Procedure-2b with respect to the MCT-constraint, we note that upon adding or removing a server from the pool of a VIP under attack, the MCT holds approximately $\lceil \frac{1}{|pool|} \cdot r_{fake} \cdot T_{exposed} \rceil$ fake signatures. This is also true for Scheme-1, but in Scheme-1 $T_{exposed} = T_{RST}$ while in Procedure-2b $T_{exposed} = T_{CHT}$. This implies that the number of MCT entries in Procedure-2b is smaller by a factor of $\frac{T_{RST}}{T_{CHT}}$ = 500 than in Scheme-1. Table 4 compares the two schemes for the case where $|pool|$ = 10.

r_{fake} \ Scheme	Scheme-1	Procedure-2b
1 million/s	714%	1.42%
5 million/s	3570%	7.1%
10 million/s	7140%	14.2%

Table 4: The expected overhead of fake signatures copied into the MCT during an update of a VIP, for $|pool|$ = 10, T_{RST} = 5s and $T_{CHT} = 10ms$

Figure 6 depicts the overhead of recording fake signatures by FIB(SYN1) in Procedure-2b, as a function of r_{fake}. We can see that the length of FIB(SYN1) grows linearly, and with existing FIB technology, the maximum tolerant r_{fake} is limited to 1 million SYNs per second.

6.3 Scheme-2: Putting It All Together

To address Challenge-3 better, we observe that there is no need to track SYN1 packets during normal operation, namely, when the pool of a VIP is not updated. Thus, it is better to use Procedure-2a as long as there is no need to update the pool, and to switch to Procedure-2b when a pool update is needed. This is what Scheme-2 does.

As shown in Figure 7, the LB uses Scheme-1 as long as the VIP is not under attack. When there is an attack, Prism

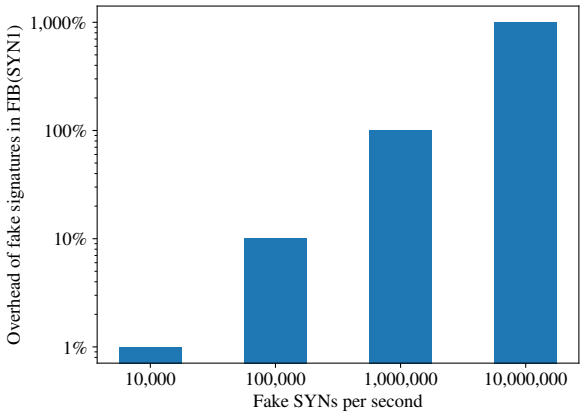


Figure 6: The overhead of recording fake signatures by FIB(SYN1) in Procedure-2b, as a function of r_{fake}

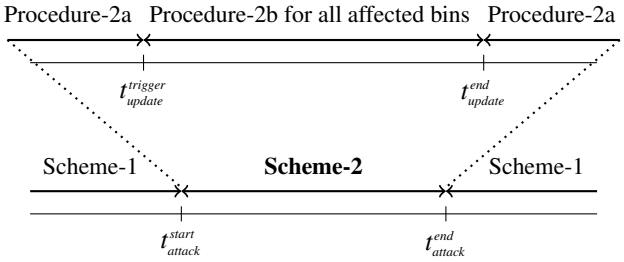


Figure 7: Scheme-2 consists of 0 or more time periods during which Procedure-2b is invoked (the figure shows only one such period). During the rest of the time, Scheme-2 executes Procedure-2a and SYN1 packets are therefore not recorded.

switches to Scheme-2. While using Scheme-2, Prism runs Procedure-2a as long as a pool update is not needed for the VIP under attack. When a pool update is requested, at time $t^{trigger}_{update}$, Prism switches to Procedure-2b and all affected bins are updated one by one. When the pool update ends, at t^{end}_{update}, Prism continues to run Procedure-2a. When the SYN attack ends, Prism switches to Scheme-1.

In the specific example of Figure 7, Prism is requested to perform only one pool update during the SYN attack. But in general, there might be many pool updates during the execution of Scheme-2, and each update requires Prism to switch from running Procedure-2a to running Procedure-2b, and then back to running Procedure-2a.

The advantage of switching between Procedure-2a and Procedure-2b, rather than always running Procedure-2b, is that FIB(SYN1) is used only during pool updates. This implies that the FIB and MCT are exposed to fake SYNs only during very short time intervals. Thus, less hardware resources are needed. However, during the time periods when Scheme-2 is carrying out Procedure-2b, both fake and real SYNs are recorded by FIB(SYN1), fetched by the software, and must be inserted into the MCT. Thus, r_{fake} is still limited to 1 million SYNs per second.

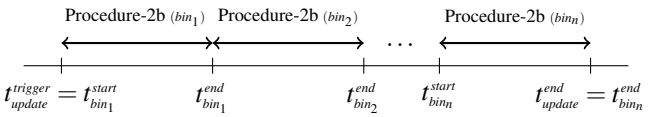


Figure 8: Improving Scheme-2 by invoking Procedure-2b multiple times, once per each bin, during a pool update. All these executions reflect one instance of Procedure-2b in Figure 7.

To improve Scheme-2 with respect to r_{fake}, we propose that during a pool update, Procedure-2b is performed separately per each affected bin, and not for all bins together. Upon updating each bin_i, the LB configures FIB(SYN1) to record only SYNs destined for this bin. This is illustrated by Figure 8. The update process for bin_i starts at $t^{start}_{bin_i}$ and ends at $t^{end}_{bin_i}$.

Let $\#bin$ be the number of bins allocated to the attacked VIP and $\#binTotal$ be the total number of bins. Assuming that signatures are uniformly divided among the bins, $\frac{r_{fake}}{\#bin} + \frac{r_{real}}{\#binTotal}$ is the expected rate of SYN1 packets the LB should record during the update of each bin. Since $r_{fake} \gg r_{real}$ and $\#binTotal \gg \#bin$ hold for the attacked VIP, this improvement decreases the hardware resources needed for FIB(SYN1) by a factor of $\#bin$. The same improvement factor is obtained with respect to the MCT-constraint.

To analyze the maximum tolerable r_{fake} during a pool update, let ρ be the rate of copying signatures into the MCT, and δ be the rate of SYNs (real and fake) directed to the updated bin. For the update algorithm to run correctly and guarantee PCC for all real connections, $\frac{\delta}{\rho} < 1$ must hold. For a VIP under attack, $\delta = \frac{r_{fake}}{\#bin} + \frac{r_{real}}{\#binTotal}$. Thus $(\frac{r_{fake}}{\#bin} + \frac{r_{real}}{\#binTotal}) \cdot \frac{1}{\rho} < 1$ must hold. In our prototype switch, r_{real} = 1 million SYNs per second, ρ = 25,000 per second, and #binTotal = 100,000. Thus, r_{fake} is limited only by $24,990 \cdot \#bin$. For example, with $\#bin$ = 400, r_{fake} is $\approx$ 10 million SYNs per second.

The attacked VIP might have only a few bins when the attack starts. In such a case, when the attack is detected, the number of bins allocated for this VIP is gradually increased, as discussed in Section 3.1. But increasing $\#bin$ beyond 400 does not improve r_{fake} due to CHT constraints. As discussed in Section 5.1, with RTT_{max} = 10ms, the CHT resources can handle a maximum r_{fake} of $\approx$ 10 million SYNs per second. With an RTT_{max} = 20ms, the maximum r_{fake} rate decreases to 5 million SYNs per second.

6.4 Synchronizing the Servers and the LB

The coordination between the LB and the the servers with respect to SYN attacks is important. If the LB is unaware of an attack, and it continues running Scheme-1 while the servers are using SYN cookies, it will almost immediately collapse, as discussed in Section 4. Hence, an approach where the LB runs an independent scheme for detecting a SYN attack is unacceptable. Another option is that the LB will always run Scheme-2, because this scheme works whether or not SYN cookies are used by the servers. But this approach is also unacceptable because it reduces the pool update rate for all the VIPs, even when there is no attack.

Another option is to use a centralized controller for the synchronization. When a server starts using cookies, it notifies the controller, and the controller notifies the LB to switch to Scheme-2 for the considered VIP. When all the servers stop using cookies, the controller tells the LB to switch back to Scheme-1 for the considered VIP.

We now propose a simple distributed scheme that does not require any centralized entity. When a server uses SYN cookies, it sets the unused URG flag in the TCP header of the SYN2 packets it sends to the clients. To prevent any client confusion, the LB hardware clears this bit after inspecting it. Since the decision whether to use Scheme-1 or Scheme-2 is made by the LB per VIP, and not per server (it is possible that DIP1 detects an attack and starts using cookies, while DIP2 does not), the LB hardware maintains a bit vector for every VIP, where each bit indicates the status of one server in the VIP's pool. If a SYN2 packet is received with URG=1 from a given server, the corresponding status bit is set to 1. If a SYN2 packet is received with URG=0, the corresponding bit is not modified. The switch software reads this vector once a while, e.g., every 10ms, and then resets all bits to 0. For each VIP, if the LB is running Scheme-1 and the read vector is $\neq 0$, the LB immediately switches to running Scheme-2. If the LB is running Scheme-2, then after M=5 consecutive times during which the vector is 0, the LB moves to running Scheme-1.

7 Proof of Concept

The most important component in Scheme-2 is the CHT, which allows the LB to keep a software state only for real connections. The purpose of this proof of concept (POC) is to implement the CHT and to prove that it can filter a high rate of fake SYNs. As described by Figure 3, the CHT records sequence numbers of SYN2 packets and then checks each ACK packet received from a client to assess if it is a valid ACK3. We implement the CHT by leveraging the hardware FIB logic used for MAC address learning and forwarding. When used for MAC address learning and forwarding, the FIB searches for a match between the destination MAC address of a received packet and its existing list of addresses. In addition, it adds the source MAC address to this list. In the POC, for each SYN packet coming from the server, the sequence number plus 1 is learned in the same way a source MAC address is learned. For each ACK packet coming from a client, the acknowledgement number is checked against the list of all numbers, in the same way a destination MAC address is checked against a list of addresses, to decide if this is a valid ACK3. If this is the case, the signature of this ACK is remembered as a valid connection.

7.1 P4 Implementation

The POC is implemented on top of a Spectrum 2 ASIC [18] running SONiC [2] on an x86 Pentium CPU, with 8GB DRAM and 4 2.20GHz cores. We use the P4 language [4] and the Mellanox P4 compiler to extend the hybrid programmable pipeline with the SYN filtering functionality. In the ingress pipeline stage, after the parser, a P4 table called `tcp_learn` is defined in the following way:

```
// learn SYN2 sequence number (cookie)
action learn_sequence() {
    set_hardware_learning();
    set_src_mac((bit<48>) (headers.tcp.seq_no + 1));
}

// trigger ACK3 validation
action check_ack() {
    set_dst_mac((bit<48>) headers.tcp.ack_no);
}

table tcp_learn {
    key = {
           headers.tcp.flags : ternary;
           standard_metadata.ingress_port: exact;
    }
    actions = {
        NoAction; learn_sequence; check_ack;
    }
}
```

When used for MAC address learning, this table is responsible for triggering the FIB learning mechanism. In the POC, this table applies the `learn_sequence` action on SYN2 packets, which sets the value of the sequence number plus 1 in the packet source MAC address, and sets the remaining upper 16 bits to zero (instead of the 5-tuple hash, just to keep the POC simple). This triggers an insertion of a cookie into the CHT. For ACK packets coming from the clients, the table applies the `check_ack` action, which sets the value of the acknowledgment field in the destination MAC address of the packet. This triggers a match check against the values stored in the CHT. If there is a match, the hardware sets a register called `is_cht_hit`.

In the egress pipeline stage, a P4 table called `tcp_filter` is defined in the following way:

```
action validate_and_learn_signature() {
    trap(0, 1 /*trap_id*/);
}

table tcp_filter {
    key = {
           headers.tcp.flags : ternary;
           standard_metadata.ingress_port: exact;
           standard_metadata.is_cht_hit: exact;
    }
    actions = {
        NoAction; validate_and_learn_signature;
    }
}
```

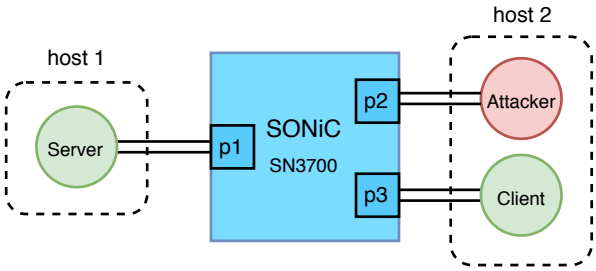


Figure 9: POC testbed topology

This table is responsible for remembering the signatures of verified ACK3 packets. This action is applicable for ACK packets coming from any client ports for which `is_cht_hit` was set in the previous pipeline stage (i.e., there was a match between their sequence number and a previous SYN2 sequence number+1). For this POC, the signatures of confirmed ACK3 packets are remembered by trapping and forwarding them to the software.

7.2 POC Topology

Figure 9 shows the testbed topology for the POC. The client and the attacker send traffic to different ports from the same host, while the target server is connected to another port. The client runs normal send and receive scripts that emulate a standard TCP three-way handshake. To emulate a heavy SYN attack, the MoonGen packet generator [9] is used. MoonGen runs on top of DPDK [13], and it can saturate a 10 Gb/s link with minimum-sized packets. This enables the attacking host to generate millions of fake SYNs per second.

The server runs on a host with a ConnectX-4 NIC. The client and the attacker run on another host with a dual-port ConnectX-4 NIC, such that each of them is connected to a different port (see Figure 9). The server host has an Intel 8-core 3.4GHz i7-3770 CPU. The client host has an Intel 8-core 3.07GHz i7-950 CPU. The server also runs MoonGen to respond to the high SYN rate. It replies to each received SYN1 with a corresponding SYN2 whose sequence number is randomly chosen.

The MoonGen server and attacker scripts are executed without rate limiting. The traffic rate varies between 5-7 million packets per second. During the experiment, all the information related to ACK packets is collected, including their transmission times, and whether they are confirmed ACK3 packets or not.

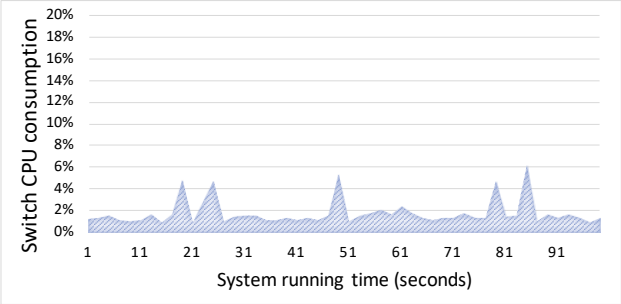


Figure 10: The switch's CPU consumption over the system running time

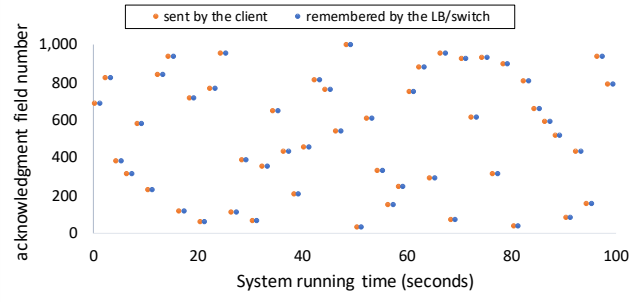


Figure 11: The acknowledgment field number of ACKs considered by the LB as ACK3 (representing legitimate connections) and of ACK3 packets sent by the client throughout the system running time

7.3 POC Experiment Results

Figure 10 presents the statistics for the switch's CPU consumption during an attack of seven million SYNs per second. It is evident that CPU consumption is under 2% during most of the time, and it never exceeds 6%.

Figure 11 shows the acknowledgment number field of ACK packets learned by the LB and the ACK3 packets sent by the client over the system running time. Since confirmed ACK3 packets are forwarded to the software and are added into the SCT as new connections, this experiment proves that the software is not exposed to fake SYNs.

We perform another experiment to verify that client's regular ACK packets (as opposed to ACK3 packets) do not mistakenly establish new connections. To this end, the client script is modified such that it sends 100 ACK packets after sending an ACK3. The acknowledgment number of each ACK increases by 100 compared to the previous ACK. Figure 12 presents the collected results. As can be seen, the client manages to complete each TCP handshake after which it sends 100 ACKs every 5 seconds. It is also evident that the switch learns only the first ACK of the connection (ACK3).

Next, we test the scalability of the POC. The objective of this experiment is to verify that even if the LB receives a high volume of SYN1 packets and regular ACK packets, it does

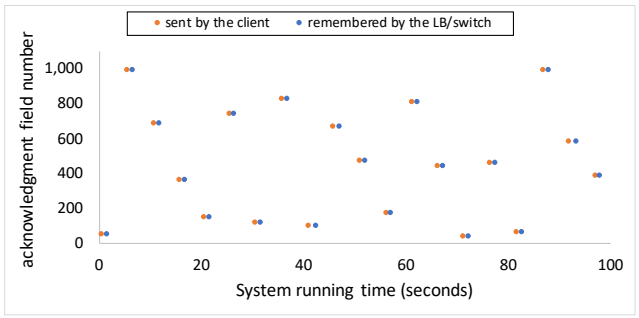


Figure 12: The acknowledgment field number of ACKs considered by the LB as ACK3s (representing legitimate connections) and ACK3 packets sent by the client over the system running time

not confuse between regular ACK and ACK3 packets. In this experiment, one host runs a MoonGen script that generates one million SYN2 packets every second and assigns a random sequence number to each packet. This host emulates a server replying to one million SYNs per second. The other host also runs a MoonGen script, which emulates sending a high volume of random regular ACK packets passing through the LB. For the latter MoonGen script, the traffic rate is not limited.

This experiment runs for about 30 minutes. During this time, statistics are collected from the switch CPU, and the number of ACK and SYN2 packets that trigger P4 actions (`check_ack` and `learn_sequence`) is collected as well. This is done using hardware counters that are hit when the relevant actions are triggered. Figure 14 presents the collected results. First, the CPU-wide curve is 1% on the average and never exceeds 3-4%. Second, the rate of SYN2 packets (the bottom thin curve) is indeed one million per second. Third, the rate of regular ACKs (top thin curve) is 13 million per second. Finally, not shown by the graph, the switch CPU does not receive any ACKs during the entire 30-minute interval, which proves that there are no false positives.

8 Hardware Authentication of Cookies

If the switch does not need to use a CHT, it would be able to mitigate almost any rate of fake SYNs. But this requires the switch hardware to check the validity of cookies in every ACK packet it receives, without storing cookies in the CHT. For this approach to work, the following requirements must be fulfilled:

1. All the servers of the same VIP should use the same cryptographic hash function and the same key to calculate their SYN cookies.

2. The LB should know the cryptographic function and key used by each VIP.

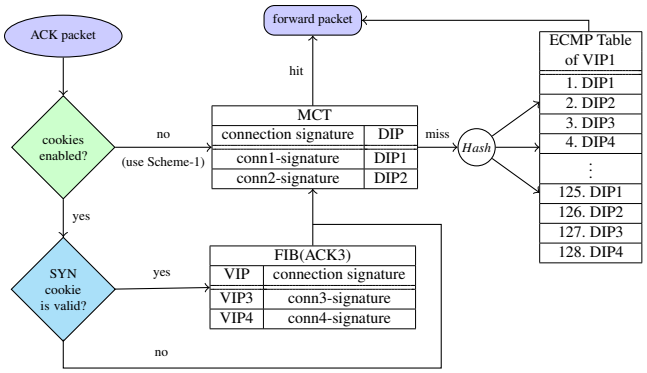


Figure 13: The flow of clients' ACKs when Hardware Authentication of Cookies is used

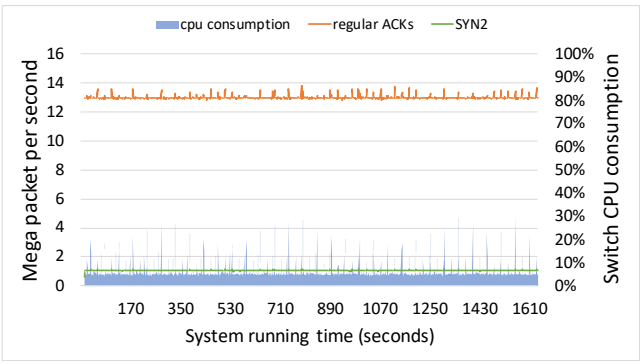


Figure 14: The switch's packet rate and CPU consumption over the system running time

3. The switch hardware should be able to execute the cryptographic function on every received ACK packet.

As far as we know, requirement (3) is currently not satisfied by off-the-shelf switches.

Let H_k be the cryptographic hash function used by a certain VIP for the encrypted part of the cookie, where k is a secret key. For each ACK packet sent to this VIP, the LB computes H_k on the ACK's 5-tuple, and compares it to the packet's acknowledgment number minus 1. If there is a match, the packet is considered a valid ACK3, and is therefore recorded by the FIB(ACK3) table of the LB. Otherwise, the packet is forwarded to the server as a regular ACK. This new flow of ACK packets is depicted in Figure 13.

During the cookie validation process, the LB is supposed to verify the cookie's timestamp field (Figure 2). To this end, the LB must know the timestamp used by every server in the pool of the given VIP. This can be done by inspecting the SYN2 packets sent by every VIP's server and remembering the last timestamp each server uses.

To analyze the rate of false positives, let r be the rate of regular ACKs passing through the LB for the given VIP. The probability for a false positive validation is $\frac{4}{2^{29}}$ since the values of the 3-bit field of the MSS are ignored and there are four acceptable timestamp values. For r = 100 million ACKs per second, there are ≈ 0.75 false positives per second. Suppose that each false positive signature is copied into the MCT for an activity detection of five seconds, and is then removed from the MCT and the SCT. Therefore, the MCT is expected to simultaneously hold less than four false positive signatures, which is a negligible overhead.

9 Conclusions

This paper showed that fighting against a SYN attack is difficult if a hardware LB needs to perform frequent server pool updates. We proposed several hardware-based schemes, which take advantage of a new Cookie Hardware Table (CHT). This table is used by the LB for recording SYN cookies from the SYN packets coming from the attacked servers. When the LB receives an ACK from the client, it checks the acknowledgment sequence number field against the CHT to deceide if this ACK will open a new legitimate connection.

Using analysis, simulations and a proof of concept, we showed that with the proposed schemes, the LB can support a high connection rate of one million new connections per second, together with a frequent pool update rate, without breaking connections and without being affected by a high rate of SYN attacks of up to 10 million fake SYNs per second.

References

[1] Aviatrix. Azure Inter Region Latency. https://docs.aviatrix.com/HowTos/arm_inter_region_latency.html, 2019.

[2] Microsoft Azure. Sonic. https://azure.github.io/SONiC/, 2020.

[3] D. J. Bernstein. SYN Cookies. http://cr.yp.to/syncookies.html, 1996.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. In *ACM SIGCOMM*, 2014.

[5] Reuven Cohen, Matty Kadosh, Alan Lo, and Qasem Sayah. LB Scalability: Achieving the Right Balance Between Being Stateful and Stateless. In *Arxiv 2020, https://arxiv.org/pdf/2010.13385.pdf*.

[6] Van Tuyen Dang, Truong Thu Huong, Nguyen Huu Thanh, Pham Ngoc Nam, Nguyen Ngoc Thanh, and Alan Marshall. SDN-Based SYN Proxy—A Solution to Enhance Performance of Attack Mitigation Under TCP SYN Flood. *The Computer Journal*, 2019.

[7] Wesley M Eddy. Defenses against TCP SYN flooding attacks. *The Internet Protocol Journal*, 2006.

[8] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2016.

[9] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proceedings of the Internet Measurement Conference*, 2015.

[10] Rohan Gandhi, Hongqiang Harry Liu, Y Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *ACM SIGCOMM*, 2015.

[11] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from google's network infrastructure. In *ACM SIGCOMM*, 2016.

[12] Albert Greenberg, James Hamilton, David A Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. In *ACM SIGCOMM*, 2018.

[13] DPDK Intel. Data plane development kit, 2014.

[14] Adam Kirsch and Michael Mitzenmacher. Using a queue to de-amortize cuckoo hashing in hardware. In *Allerton*, 2007.

[15] Jonathan Lemon et al. Resisting SYN Flood DoS Attacks with a SYN Cache. In *BSDCon*, volume 2002, pages 89–97, 2002.

[16] Gil Levy, Salvatore Pontarelli, and Pedro Reviriego. Flexible packet matching with single double cuckoo hash. *IEEE Communications Magazine*, 2017.

[17] Patrick McManus. Improving SYN Cookies. https://lwn.net/Articles/277146/, 2008.

[18] Mellanox. Spectrum-2 Ethernet Switch ASIC. https://www.mellanox.com/products/ethernet-switch-ic/spectrum-2, 2019.

[19] 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.

[20] Microsoft. Azure network round trip latency statistics. https://docs.microsoft.com/en-us/azure/networking/azure-network-latency, 2019.

[21] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *USENIX NSDI*, 2018.

[22] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. In *ACM SIGCOMM*, 2013.

[23] Christoph L Schuba, Ivan V Krsul, Markus G Kuhn, Eugene H Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. IEEE Symposium on Security and Privacy, 1997.

[24] The Internet Society. Transmission Control Protocol. RFC 4987. https://tools.ietf.org/html/rfc4987, 2007.

[25] Andre Zuquete. Improving the functionality of SYN cookies. In *Advanced Communications and Multimedia Security*. Springer, 2002.