BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

HEURISTIC METHODS FOR THE MITIGATION OF DDOS ATTACKS THAT ABUSE TCP PROTOCOL

HEURISTICKÉ METODY PRO POTLAČENÍ DDOS ÚTOKŮ ZNEUŽÍVAJÍCÍCH PROTOKOL TCP

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PATRIK GOLDSCHMIDT**
AUTOR PRÁCE

SUPERVISOR **Ing. JAN KUČERA**
VEDOUCÍ PRÁCE

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Goldschmidt Patrik**
Program: Informační technologie
Název: **Heuristické metody pro potlačení DDoS útoků zneužívajících protokol TCP**
Heuristic Methods for the Mitigation of DDoS Attacks that Abuse TCP Protocol
Kategorie: Počítačové sítě

Zadání:

1. Seznamte se s problematikou útoků typu odepření služby a zařízením vyvíjeným v rámci sdružení CESNET pro ochranu před těmito útoky.
2. Nastudujte možnosti potlačení útoků zneužívajících protokol TCP.
3. Navrhněte adaptivní přístup volby parametrů a strategie mitigace DoS v závislosti na vlastnostech probíhajícího útoku (síla, úspěšnost mitigace).
4. Takto navržený přístup implementujte.
5. Pro potřeby vyhodnocení rozhodovacího mechanismu implementujte také vybranou heuristickou metodu/y pro potlačení DoS útoku zneužívajícího protokol TCP.
6. Vyhodnoťte implementované řešení z hlediska dosažených vlastností.
7. V závěru diskutujte výsledky a možnosti dalšího pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Kučera Jan, Ing.**
Konzultant: Žádník Martin, Ing., Ph.D., UPSY FIT VUT
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 26. října 2018

Abstract

TCP SYN Flood is one of the most wide-spread DoS attack types used on computer networks nowadays. As a possible countermeasure, this thesis proposes a network-based mitigation method TCP Reset Cookies. The method utilizes the TCP three-way-handshake mechanism to establish a security association with a client before forwarding its SYN data. The algorithm can effectively mitigate even more sophisticated SYN flood attacks at the cost of 1-second delay for the first established connection. However, the method may not be suitable for all the scenarios, so decision-making algorithm to switch between different SYN Flood mitigation methods according to discovered traffic patterns was also developed. The project was conducted as a part of security research by CESNET. The discussed implementation of TCP Reset Cookies is already integrated into a DDoS protection solution deployed in CESNET's backbone network and Czech Internet exchange point at NIX.CZ.

Abstrakt

TCP SYN Flood sa v súčasnosti radí medzi najpopulárnejšie útoky typu DoS. Táto práca popisuje sieťovú mitigačnú metódu TCP Reset Cookies ako jeden z možných spôsobov ochrany. Spomínaná metóda je založená na zahadzovaní všetkých prijatých pokusov o nadviazanie spojenia, až pokým s daným klientom nie je uzatvorená bezpečnostná asociácia na základe využitia mechanizmu TCP three-way-handshake. Tento prístup dokáže efektívne odraziť aj sofistikovanejšie útoky, avšak za cenu sekundového oneskorenia pri prvom nadväzovanom spojení daného klienta. Metóda však nie je vhodná vo všetkých prípadoch. Z tohto dôvodu táto práca ďalej navrhuje a implementuje spôsob dynamického prepínania rôznych mitigačných metód na základe aktuálne prebiehajúcej komunikácie. Tento projekt bol vykonaný ako súčasť bezpečnostného výskumu spoločnosti CESNET. Spomínaná implementácia metódy TCP Reset Cookies je už v čase písania tejto práce integrovaná do DDoS riešenia nasadeného na hlavnej sieti spoločnosti CESNET, ako aj v českom národnom peeringovom uzle NIX.CZ.

Keywords

DDoS, DDoS mitigation, Heuristic DDoS mitigation, TCP abuse, TCP SYN Flood, TCP Reset Cookies

Klíčová slova

DDoS, DDoS mitigace, Heuristická mitigate DDoS, TCP zneužití, TCP SYN Flood, TCP Reset Cookies

Reference

GOLDSCHMIDT, Patrik. *Heuristic Methods for the Mitigation of DDoS Attacks that Abuse TCP Protocol*. Brno, 2019. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jan Kučera

Rozšířený abstrakt

Transmission Control Protocol (TCP) je jednou zo základných súčastí sady Internetových protokolov. Všetky webové, súborové, e-mailové služby a množstvo iných na tejto sade priamo závisia. Kvôli jeho dôležitosti je však TCP terčom množstva kybernetických útokov, ktoré v posledných rokoch naberajú na počte a sile. Na základe predpovede od spoločnosti Cisco má byť v roku 2022 uskutočnených 14.5 milióna útokov, pričom na základe aktuálneho trendu je väčšina útokov vykonávaná ako TCP SYN Flood.

Aktuálne spôsoby mitigácie týchto útokov sú voči pokročilejším variantom často neefektívne, prípadne nie sú vhodné pre aplikáciu na sieťových zariadeniach. Táto práca z tohto dôvodu navrhuje a implementuje metódu *TCP Reset Cookies*, ktorá sa snaží tieto nedostatky odstrániť. Hlavným účelom práce je teda vytvoriť efektívnu metódu pre mitigáciu pokročilejších útokov, ktorú je možné jednoducho aplikovať na sieťové zariadenia ako hardvérové firewally alebo IDS/IPS systémy.

Metóda TCP Reset Cookies sa zakladá na princípe vytvorenia bezpečnostnej asociácie s klientom pred tým, ako sú jeho žiadosti o uzatvorenie spojenia (SYN správy) preposielané ich určenému adresátovi. Proces asociácie sa zakladá na využití mechanizmu TCP three-way-handshake, počas ktorého je medzi klientom uzatváraný komunikačný kanál. Štandard RFC 793 definuje presné hodnoty, ktoré musia byť obsiahnuté v TCP segmentoch pre správne uzatvorenie relácie. Štandard takisto definuje, ako sa má strana prijímajúca segment s neočakávanými hodnotami zachovať. Na základe týchto informácii môžeme definovať predpoklad, že útočník zasielajúci veľké množstvo SYN správ pomocou špecializovaného softvéru (typicky z falošných IP adries) nemá implementovaný algoritmus TCP podľa štandardu, a tým pádom požadovanú odpoveď na nevalidnú správu neodošle.

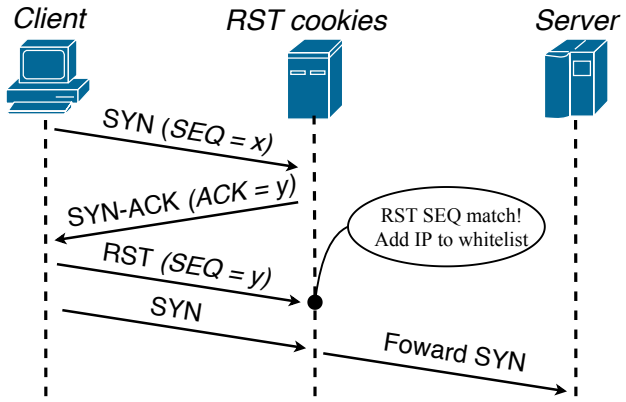


Figure 1: Funkcionalita algoritmu TCP Reset Cookies

Funkcionalita algoritmu RST Cookies je znázornená na obrázku 1. Ako môžeme vidieť, klient sa pokúša o uzatvorenie spojenia prostredníctvom TCP správy s príznakom SYN. Tejto správe je automaticky vygenerovaná pseudo-náhodná hodnota sekvenčného čísla (SEQ]) s hodnotou x. Správa smerujúca serveru je odchytená predradeným sieťovým mitigačným zariadením využívajúcim metódu RST Cookies. Metóda analyzuje prijatý segment a zašle odpoveď s príznakmi SYN + ACK ako definuje štandard, avšak namiesto očakávanej hodnoty vloží do poľa pre potvrdenie (ACK) hodnotu y, pričom očakávaná hodnota je $x + 1$. Správa s takouto hodnotou je odoslaná klientovi, ktorý podľa štandardu musí odpovedať správou s príznakom RST a sekvenčným číslom rovným hodnote y. Pri spracovaní takejto správy algoritmom je zdrojová IP adresa odosielateľa pridaná do

asociačnej tabuľky a všetky ostatné SYN správy od daného klienta sú preposielané bez intervencie algoritmu.

Pre zaistenie bezpečnosti takejto metódy je nutné zaručiť, aby boli zasielané nevalidné čísla *ACK* generované náhodne bez možnosti ich predikcie. Z tohto dôvodu je implementovaný algoritmus na dynamickú generáciu a validáciu hodnôt. Pre tento účel sú podporované 2 režimy – režim náhodných hodnôt v časových oknách a hash režim. Režim náhodných hodnôt v časových oknách generuje náhodnú hodnotu pre každé časové okno a vracia *ACK* hodnotu na základe času. Hash režim generuje unikátnu *ACK* hodnotu pre každé spojenie na základe počítania hash funkcie pre hodnoty, ktoré dané spojenie definujú.

Metóda RST Cookies poskytuje silnú ochranu proti bežným a pokročilým SYN Flood útokom, avšak jej použitie spôsobuje značné obmedzenia priepustnosti a citeľné navýšenie času uzatvárania prvého spojenia. Na základe našich testov je oneskorenie tohto spojenia zvýšené až o 1 sekundu z dôvodu nutnosti jeho resetovania a následného opätovného zaslania SYN správy. Použitie režimu generácie náhodných hodnôt prostredníctvom časových okien zníži priepustnosť packetov zhruba o 57%, zatiaľ čo hash varianta až o 87%.

Na základe týchto zistených nedostatkov môžeme usúdiť, že využívanie metódy pre mitigáciu bežných útokov nemusí byť vzhľadom na jej negatívny vplyv na sieťovú prevádzku efektívne. Z tohto dôvodu je ďalej navrhnutý a implementovaný systém na dynamické prepínanie rôznych mitigačných metód na základe aktuálnej sieťovej prevádzky, ale aj iných faktorov ako úspešnosť mitigácie, využívanie systémových zdrojov a pod.

Systém na prepínanie metód bol vyvinutý špeciálne pre účely použitia v riešení CESNET DDoS Protector, ktoré okrem metódy *RST Cookies* obsahuje aj iné mitigačné stratégie ako *SYN Drop* a *ACK Spoofing*. Tieto algoritmy je následne nutné pre použitie v rozhodovacom module registrovať. Registráciou sa systému na dynamické prepínanie oznámi ich existencia, ale aj definujú ich parametre. Rozhodovací modul na prepínanie následne vyhodnotí kvalitu jednotlivých mitigačných metód pomocou fitness jadra. Jadro určené na analýzu prevádzky zaznamenáva informácie o počte SYN, ACK a RST správ, ako aj počet unikátnych IP adries zasielajúcich tieto dáta s využitím štruktúry HyperLogLog. Samotné rozhodovanie o najvhodnejšej mitigačnej metóde prebieha pomocou rozhodovacieho jadra. Táto množina funkcií slúži na analýzu zozbieraných štatistík, hľadanie rôznych náznakov útokov a následné priraďovanie týchto náznakov k dostupným mitigačným metódam.

Aktuálna implementácia algoritmu na dynamické prepínanie závisí na množstve prahových hodnôt, ktoré sú často volené experimentálne. Na základe zozbieraných dát sú hodnoty postupne upravované a kvalita mitigácie sa tak zlepšuje. Na dosiahnutie optimálneho stavu bude nutné ešte veľké množstvo dát, avšak v súčasnom stave je modul schopný odhaliť prebiehajúci SYN flood útok na základe prahov o počte poslaných SYNov, pomere IP adries zasielajúcich SYN a ACK segmenty, ale aj čiastočné rozpoznanie kontextu útoku na základe jeho histórie. Funkčnosť modulu je optimalizovaná pre spomínané 3 mitigačné metódy dostupné v riešení DDoS Protector, avšak návrh algoritmu počíta s akýmkoľvek množstvom funkcií, ktoré budú v budúcnosti podporované bez nutnosti jeho zmien.

Ako bolo naznačené v predošlých odsekoch, táto práca, ako aj všetky vyvíjané algoritmy sú súčasťou bezpečnostného výskumu vedeného spoločnosťou CESNET. Metóda RST Cookies je v rámci projektu už integrovaná a používaná, zatiaľ čo integrácia metódy na dynamické prepínanie je plánovaná v blízkej budúcnosti. Algoritmy popísané v tomto dokumente budú ďalej rozširované aj v budúcnosti v rámci projektu DDoS Protector, ktorý nedávno obdržal grant od Ministerstva vnitra České republiky. Časť práce zahŕňajúca teóriu a popis RST Cookies metódy bola prezentovaná na študentskej konferencii Excel@FIT 2019, kde bola ocenená odborným panelom za prínos v oblasti počítačovej bezpečnosti.

Heuristic Methods for the Mitigation of DDoS Attacks that Abuse TCP Protocol

Declaration

I hereby declare that I have authored this Bachelor's thesis independently, under the supervision of *Ing. Jan Kučera*. I have not used other than the declared sources and publications and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. According to my knowledge, the content or parts of this thesis have not been presented to any examination authority nor have been published (with the exception of faculty student's conference Excel@FIT 2019, which took place at BUT FIT on April 2019). I am aware that respective work can be considered as plagiarism and legal actions may be taken if the above statements are not true.

.....
Patrik Goldschmidt
May 16, 2019

Acknowledgements

I would like to express my gratitude towards my supervisor *Ing. Jan Kučera*, who was enormously helpful the whole time we were co-working on the DDoS Protector project. His patience with my (sometimes) silly questions and his wise answers to them have notably improved the final quality of the product and this thesis, but also helped me to gain software design and development confidence, a crucial quality of any junior programmer.

Many thanks also go to my consultant, *Ing. Martin Žádník, Ph.D.*, who had introduced me to the CESNET's security research projects back in 2017 and was the light that guided me through the dark path in my beginnings with network programming.

Contents

Chapter 1

Introduction

Transmission Control Protocol (TCP) is an integral part of the Internet protocol suite. It is a component of underlying architecture which provides functionality for services like HTTP, FTP, SMTP and many more. As the importance of TCP is fundamental for the operation of the Internet, it is often the target of various cybersecurity threats, Distributed Denial of Service (DDoS) being a popular choice. Report from Q4 2018 by Kaspersky Lab states that the most frequent target of a denial of service attacks was TCP, targeted by 66.60% of all the attacks [18]. According to [8], the number of DoS attacks will double to 14.5 million p.a. by 2022. These and many other facts should highlight the need for TCP protection and how specialized techniques are required to achieve it.

Currently used methods for SYN Flood mitigation are mostly designed to be used on the end hosts themselves. These end-host mitigation techniques like TCP SYN Cookies are often effective, but their nature is indeed not suitable in all situations. For example, a high number of segments sent by an attacker may not cause an ordinary SYN Flood DoS due to mitigation method intervention, but its execution may still cause high processor utilization of the server. This means that data from legitimate clients are processed with unacceptable delays or are not processed at all, effectively creating a DoS situation anyway.

To spare the resources of the server, many of the mitigation methods are deployed on specialized intermediary network devices. This way, potential DDoS attacks can be mitigated before reaching the server, therefore not wasting its resources on processing traffic from attackers. However, some of these methods, originally intended for end-host mitigation, are not optimal when used on intermediary devices. For this reason, the thesis aims to implement an effective native network-based mitigation method called TCP Reset Cookies. On top of that, a system to dynamically switch between several of these mitigation strategies according to discovered traffic patterns was also designed and implemented.

The project was conducted as a part of the security research for high-speed computer networks by CESNET. The implementation of the presented mitigation method, developed as a part of this thesis, is already integrated into the CESNET's anti-DDoS solution, which is actively used on its backbone network and was also recently applied to the Czech national Internet exchange point at NIX.CZ [6]. The algorithm for dynamic switching is not yet used, but its integration is planned in the near future.

Beginning of the thesis (Chapter 2) discusses theoretical functionality of the TCP protocol with the emphasis on aspects related to security. Chapter 3 analyzes, implements and evaluates the mentioned mitigation strategy TCP Reset Cookies. The algorithm for dynamic method switching is presented in Chapter 4. Summary of the achieved results and potential future improvements are discussed in Chapter 5.

Chapter 2

TCP Security Considerations

Specification RFC 793 defines TCP as a highly reliable host-to-host protocol intended for use between hosts in packet-switched computer communication networks, and in interconnected systems of such networks [21]. To create and maintain a reliable way of communication, the protocol implements several techniques for node synchronization. This chapter describes how this communication channel is established and how it affects the security of the protocol itself. The chapter also explains various types of TCP attacks as well as mitigation methods that are commonly used to reduce their impact or mitigate them completely.

2.1 Session Establishment

The establishment of a reliable communication channel is done via a process called TCP three-way-handshake. The process is started by an initiating host (client), which constructs an SYN segment and sends it to the second node (server) awaiting connection requests. As illustrated in Figure 2.1, this segment has a Synchronize (SYN) flag set and carries a value of x as its Sequence number (SEQ). Standard does not explicitly define an Acknowledgment (ACK) value, so operating systems usually set it to 0.

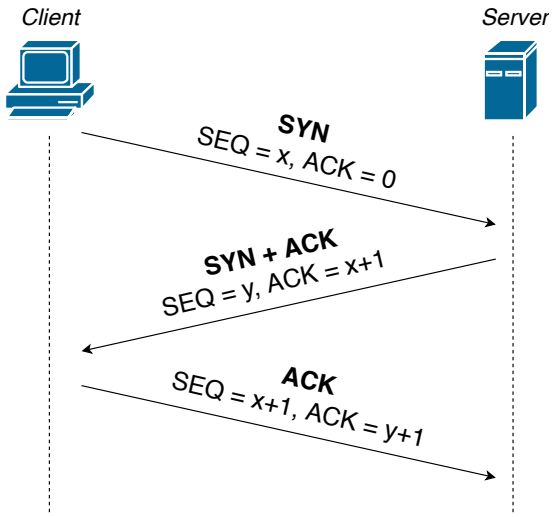


Figure 2.1: TCP three-way-handshake process.

Upon receiving an SYN from the client, the server generates its own pseudo-random number to be used as its *SEQ* while setting the *ACK* value to the Sequence number of the previously received SYN increased by one to signalize that SYN message from the client was processed. A segment with these values and SYN + ACK flags set is assembled and send as a response from the server. When the client receives this packet, and *ACK* value is exactly $x + 1$, it generates own acknowledgment segment and sends it back to the server.

After this process, the communication channel is considered established, and two endpoints are able to exchange data. The process is not used only for synchronization purposes, but also for negotiation of different transmission options like TCP window size. The three-way-handshake is fundamental for TCP operation but is also often misused by attackers in various attacks described in the following section.

2.2 Attacks on TCP

Attacks that abuse weaknesses of the TCP can be differentiated into two main categories:

- Flood attacks
- Injection attacks

Flood attacks typically target a single host or a network. Their aim is to exhaust the target's resources by flooding a large number of bogus packets. These data have to be processed by the target server, draining the CPU, memory and network resources in a fashion that regular clients cannot be served, or are served with an unacceptable delay, effectively creating a denial of service situation. On the other hand, injection attacks are based on eavesdropping the ongoing communication and injecting crafted segments into the TCP session. Injected data may contain malicious code, compromise the user's privacy [15] or reset the session [25]. This document focuses on the flood attacks, which are mostly associated with a DoS.

Since flood attacks are generally easier to perform, they became a favorite choice for attackers aiming to create a DoS situation. As mentioned back in Chapter 1, TCP was targeted by 66.60% of all the DDoS attacks in the fourth quarter of 2018, meanwhile, 58.20% of all the attacks were performed as TCP SYN Flood, the most popular variant for TCP DoS (Figure 2.2).

The following subsections will briefly describe most common TCP attacks from both categories.

2.2.1 TCP SYN Flood

TCP SYN Flood is currently one of the most widespread and most effective TCP DoS attacks. Its functionality depends on the three-way-handshake mechanism, during which a server receiving the SYN message responds with an SYN-ACK segment and waits until the ACK arrival to mark the connection as established. The rationale behind a successful DoS assumes that the victim allocates a new state for every received SYN segment and that there is a limit of such states that can be stored. These are described in RFC 793 as Transmission Control Block (TCB) data structures. TCB structures are used to store necessary state information for an individual connection. They may be implemented differently among the operating systems, but the key concept is that new memory needs to be allocated upon every new TCP connection [10].

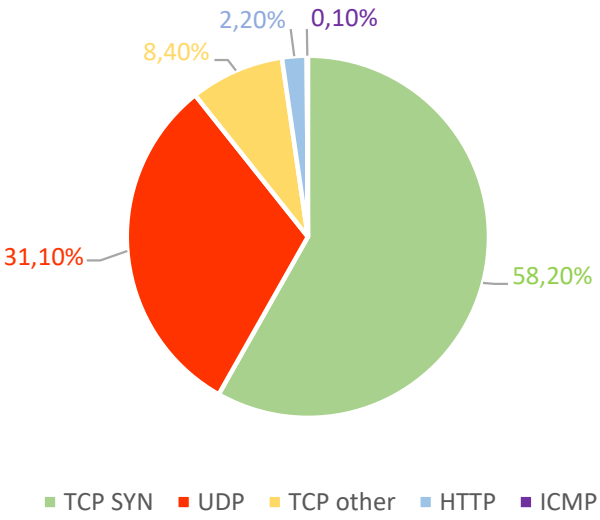


Figure 2.2: DDoS attacks distribution by type [18].

Operating system kernels normally try to protect host memory from getting exhausted by implementing a limit of contemporary TCB structures called backlog. When the backlog limit is reached, either incoming SYN segments are ignored, or uncompleted connections in the backlog are replaced. As illustrated in Figure 2.3, the primary goal of SYN flooding is to exhaust the target's backlog with half-open connections. For this purpose, spoofed IP addresses that do not generate a reply to SYN-ACKs are often used.

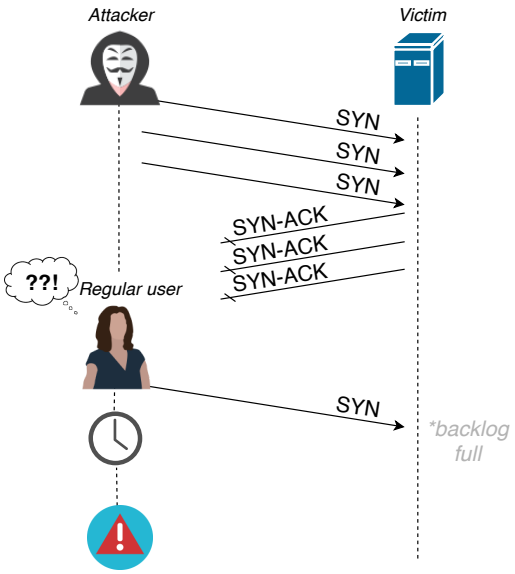


Figure 2.3: TCP SYN flood attack.

2.2.2 Spoofed Session Flood

Spoofed Session Flood (SSF), also known as Fake Session Attack and its modifications Multiple ACK SSF and Multiple SYN-ACK SSF are more sophisticated attacks able to bypass most of the standard security mechanisms. Their aim stays the same as in the TCP SYN Flood case – exhaust the target's backlog. However, security systems are typically able to filter out regular SYN flooding attacks due to the easily detectable pattern of sending many SYN segments and no other segment types at all. To make themselves harder to be revealed, Spoofed session attacks usually carry one or multiple ACK, SYN-ACK, RST and FIN segments to disguise themselves as regular TCP traffic. This way, the attacker is able to bypass defense mechanisms that rely only on monitoring incoming traffic without the use of advanced heuristics. The low SYN/ACK rate makes the attack harder to detect while the attacker is still able to create enough half-open connections for successful backlog exhaustion [23].

2.2.3 Session Attack

The most complex, but hardly detectable method is Session attack, which uses a lot of real clients generating vast amounts of legitimate traffic. For this purpose, a botnet is commonly used. At a particular time, all computers in the botnet are ordered to establish numerous TCP sessions with the victim server. These sessions are then stretched out using keepalive mechanisms and by delaying ACK responses. When a large number of bots establish several sessions each, the target server may get too busy with processing the attacker's requests. This creates unacceptable delays or may even cause that the legitimate clients are not served at all. Because the attack generates legitimate traffic from existing clients, security systems usually have no clue about the ongoing attack. Mitigation of the attack requires the usage of advanced heuristics combined with hosts reputation tables, which might be able to identify an ongoing attack originating from a botnet and mitigate it appropriately.

2.2.4 Other Flood Attacks

This category comprises attacks like SYN-ACK Flood, ACK/PSH ACK Flood, ACK fragmentation Flood, RST Flood, and FIN Flood. These are not as sophisticated as SYN Flood, because all of them work on the same trivial principle. Since neither of the listed attacks uses SYN to establish a session, exhausting a target backlog is not the goal. All methods in this subsection generate regular TCP segments, which are usually not filtered by security mechanisms. None of these generated segments are destined for an existing TCP session, but the target has to process them anyway and eventually send an RST as a response. If the attack of this type is distributed, exhaustion of the victim's processor or network resources may occur, making it irresponsive for regular clients and creating a denial of service. However, these types of floods are not as effective as previously mentioned attacks and thus are not used as commonly.

Although most of the pure TCP DoS threats were already mentioned, other application layer DoS attacks like HTTP flood may be used to achieve exhaustion of the target's resources as well. In these cases, TCP is not misused directly but is still used as a transport protocol to conduct these attacks.

2.2.5 TCP Sequence Prediction

TCP sequence prediction attack (also known as connection hijacking) is based on the assumption that the attacker is able to predict the *SEQ* number of another host during the TCP communication process. This way, the attacker may impersonate a sender and inject counterfeit packets into the session. The threat was firstly discovered in 1985 because Berkeley-derived kernels generated *SEQ* values incremented by a constant every second, and by another constant for each new connection. Thus, if an attacker established a session with a machine, he could easily estimate the *SEQ* that would be used for its subsequent session [3]. Since the attacker knew the next *SEQ* the server would send, a new connection could be established by impersonating another client and acknowledging the data send by the server without actually processing them (Figure 2.4). This technique allowed an attacker to establish a TCP session with a server while impersonating another client. To avoid a session to be reset, the impersonated host needed to get silenced. This was commonly done by DoS. After the connection was established, the attacker was still not able to see the output from the session but could execute commands as more or less any user [3].

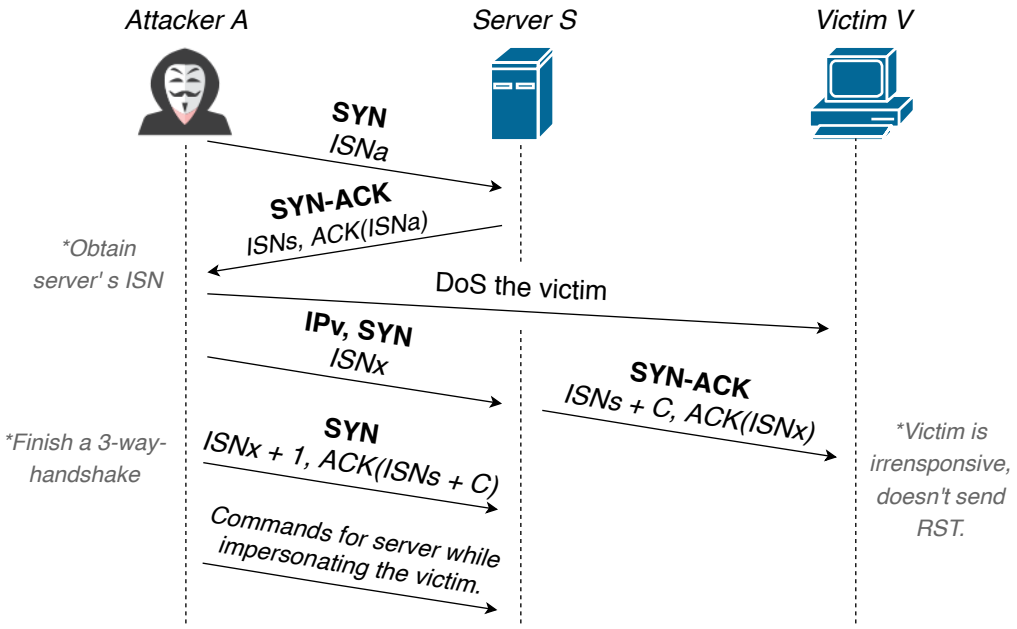


Figure 2.4: Original TCP sequence prediction attack.

Although the particular problem was addressed by changing the way of Initial Sequence Number (ISN) generation in Berkeley-derived kernels and by RFC 1948, the problem was generalized for the whole TCP stack, whose *SEQ* numbers were still relatively easy to predict. Current standard RFC 6528 specifying ISN generation addresses this problem by making the former way of sequence prediction impossible. However, an eavesdropper who can observe the initial messages for a connection can determine its sequence number state may still be able to launch *SEQ* number guessing attacks by impersonating that connection [13].

2.2.6 TCP Veto

TCP veto attack can be considered as a more advanced variant of TCP sequence prediction attack. Instead of predicting the sequence number only, an eavesdropping attacker predicts the correct payload size of the next expected message as well. A crafted segment with these values is then injected into the TCP session. Later, when the legitimate packet arrives, it is found to have the same sequence number and length as the packet already received from the attacker. This means that the legitimate packet is „vetoed“ by the previously received segment and so is silently dropped like a regular duplicate. Unlike the sequence prediction attack, the connection is never desynchronized, and the communication proceeds normally. The sender of the legitimate packet sees no evidence of the attack [14]. TCP veto gives the attacker less control over the session but makes the attack particularly resistant to detection.

2.2.7 TCP Reset Attack

TCP Reset attack is an injection-like denial of service attack, in which the perpetrator attempts to prematurely terminate a victim's active TCP session [25]. The idea behind the attack is to inject an RST segment to the session, which causes one of the receiving ends to close the connection. For this mechanism to work, injected RST needs to be precisely crafted with specific IP addresses, exact port numbers, and a *SEQ* value. These values can be obtained via eavesdropping.

The principle is also sometimes used in network security systems to forbid a connection to the particular port or a port range. The protection against this threat requires a transport layer encryption such as IPSec VPN, so the attacker is not able to extract sequence numbers from an unencrypted TCP header.

2.2.8 SYN Port Scanning

SYN port scanning is not an attack itself, but it often precedes other types of cybersecurity incidents. The process is used during the reconnaissance phase of the attack when the attacker is trying to map the target network and reveal open ports on the individual machines. The scanning is done by sending TCP SYN segments on various ports of the target. If the system has a particular port opened, it continues to establish a connection by responding with an SYN-ACK segment. When a scanner software receives an SYN-ACK from the particular port, it marks it as opened, chooses a different one and repeats the process. However, the activity of the scanner is easily detectable and therefore many security solutions and even antivirus software are typically able block it.

2.3 Defense Against TCP Flooding Attacks

This section focuses on the mitigation of TCP flooding attacks, which are the main concern of this document. Some of the practices for injection attacks mitigation were already suggested when the concrete attacks have been presented, and they will not be furthermore discussed here. The injection attacks can generally be more devastating, but the difficulty of their execution in modern computer networks make them an unfavorable choice. On the other hand, flooding attacks are popular due to their simplicity, often insufficient countermeasures and surprisingly effective results.

Several commercial solutions and academic research projects aim to provide protection against these types of attacks, one of these research projects being *DDoS Protector* developed by CESNET. This solution utilizes a hardware-accelerated traffic filtering using FPGA technology, own firmware in conjunction with a software-based malicious traffic detection core (Figure 2.5). The product is specialized on mitigation of DNS amplification attacks, but support for TCP SYN flood attacks mitigation was also recently added. SYN floods are mitigated with the use of ACK spoofing (Subsection 2.3.6) and SYN Drop (Subsection 2.3.7) algorithms, which are a part of the software detection core. These methods fall short in certain situations, so another mitigation approach – TCP Reset Cookies was designed and implemented as a part of this thesis. The details about the approach are discussed in Chapter 3. More information about the project can be found at [5].

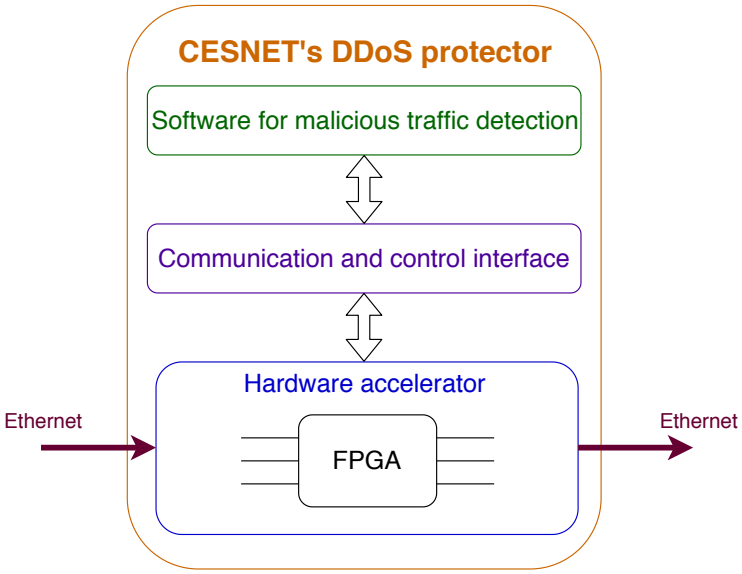


Figure 2.5: CESNET's DDoS Protector architecture.

TCP flood mitigation methods are classified according to the type of the network node they operate on. *End-host* mitigation methods, such as SYN cookies are executed straightly on the nodes that are contacted by clients. Their usage requires modification of the TCP stack of the hosts, so they are typically shipped directly with an operating system kernel or implemented as a kernel add-on. On the other hand, *network-based* mitigation methods are entirely independent of protected clients, because they run on specialized network appliances such as firewalls or IDS/IPS systems. Their functionality is typically transparent for the protected device, so no changes to the server's configuration are needed. As mentioned in Chapter 1, usage of end-host methods bring many disadvantages, because the server has to process data from attackers as well, draining its CPU and network resources. Network-based techniques eliminate this drawback, but they also tend to add extra latency to the communication. Both end-host and network-based techniques are briefly explained in the following subsections.

2.3.1 TCP SYN Cache

SYN cache is an end-host mitigation technique, which utilizes hashing to store a lightweight fingerprint of the IP address, port number and secret for every incoming TCP connection. This way, the operating system does not need to allocate the whole TCB, but only a fragment of the original memory required. A device implementing this method is, therefore, able to queue more requests, becoming harder to exhaust. In the BSD kernel from 2002, this optimization reduced the size of the per-connection data by 78% while allowing up to 15359 entries [19].

2.3.2 TCP SYN Cookies

In contrast to SYN cache, SYN cookies method does not need to store any state information at all, requiring no memory per-connection. Essential data defining the connection, alongside with a timestamp and a secret are hashed into a 32-bit value representing the *SEQ* number of the SYN-ACK segment. As depicted in Figure 2.1, the handshake is finished with an ACK message carrying the received SYN-ACK value + 1. Upon ACK receipt, the server can reconstruct original SYN parameters and successfully establish a connection. The method is exceptionally effective against SYN floods, but its nature denies SYN-ACK retransmission and restricts usage of the TCP options, such as TCP window size [4].

2.3.3 TCP Random Drop

End-host technique TCP Random drop works on a principle which replaces a random pending half-open connection when the TCB queue is full, and another SYN is received. Connection replacement is done by sending an RST segment, discarding corresponding TCB structure and allocating a new one for the incoming connection. Legitimate clients dropped with RST are expected to try to reestablish a connection again. The rationale for this approach is that by making queue large enough, a server under attack can still offer a high probability of successful connection establishment, but legitimate sessions may still be occasionally denied [22].

2.3.4 Traffic Filtering

Traffic filtering is one of the simplest ways of network-based mitigation. As described in [11], the fundamental idea is to deny all incoming traffic from IP addresses that do not match their source network prefix (packets intentionally crafted with false IP). This process allows discarding all of the traffic from forged IP addresses outside of the network prefix the generating host is currently in, but the attacker is still able to fake IP addresses from the same prefix. The method is defined as "Best current practice" and is recommended to be implemented by all Internet service providers (ISP). Despite this, specialized methods for flooding mitigation are required, because the usage of the filtering principle does absolutely nothing to protect against flooding attacks originating from valid prefixes, and one also cannot rely on the presumption that all ISPs will actually implement it.

2.3.5 SYN-ACK Spoofing

SYN-ACK spoofing is a network-based mitigation technique based on a principle of establishing a 3-way-handshake between client and the machine running the algorithm before it is actually established with the server. This principle protects the backlog of the server,

because all illegitimate SYN segments are filtered out by the algorithm, and only clients that would normally establish a session are allowed to communicate with the server. This method can be implemented in 2 ways – either by having own backlog, which is much bigger than the one provided by the server or by combining with SYN cookies to require no memory for state information at all. Either way, the client actually creates a session between itself and the SYN-ACK spoofing machine and all the data sent within the session are forwarded to the server, which has its own session with the SYN-ACK spoofer.

As it may be obvious, this process is highly ineffective due to the requirement of mapping between different *SEQ* and *ACK* values, since the server always generates its own ISN different from ISN generated by the SYN-ACK spoofer (Figure 2.6). Another drawback is the requirement on memory because tables containing all the sessions, as well as translation tables between the *SEQ*s and *ACK*s need to be maintained. The method also needs to process all TCP traffic due to mapping and session finalization requirements, disabling the ability of hardware-forwarding completely.

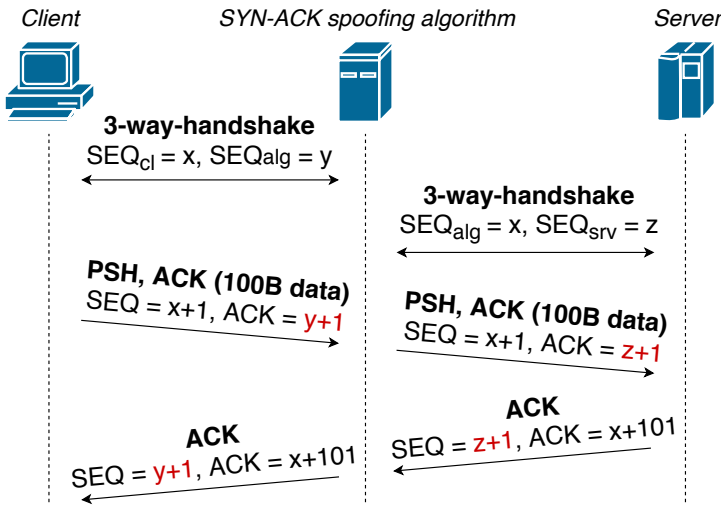
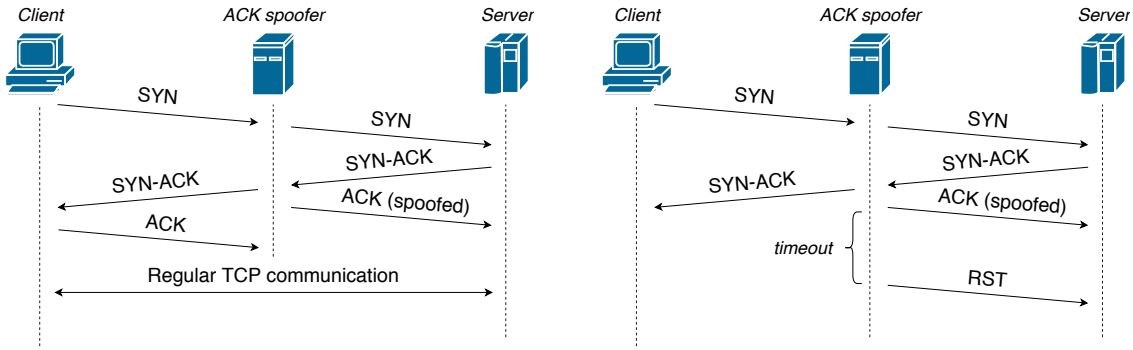


Figure 2.6: SYN-ACK spoofing algorithm simplified scheme.

Although providing adequate level security, operation of the method has high memory requirements and may add a significant delay to the TCP communication. Despite these reasons, the method is occasionally implemented in various anti-DoS solutions and even used on real networks.

2.3.6 ACK Spoofing

ACK spoofing is a method deployed on the intermediary network device, whose primary goal is to prevent the exhaustion of the protected device's backlog. The method operates by sending a spoofed ACK segment to finish every half-open session and complete the three-way handshake. This way, all of the pending connections in the backlog are completed before it may get overfilled by an attacker [9]. If the client does not generate an ACK segment within the specified timeout period, the ACK spoofing mechanism terminates the connection with an RST segment. If the expected ACK is received, the algorithm marks the connection as valid and does not interfere in the future TCP communication between the nodes (Figure 2.7).



(a) Legitimate client (b) SYN flooder

Figure 2.7: ACK spoofing functionality.

This approach protects the server's backlog, prolonging its ability to serve clients, but does not mitigate SYN flooding attacks by itself. It is important to note that a connection is established for each received SYN. This means that if an excessive number of SYN messages is received, the connections may still cause the server's memory to get exhausted. Also, each segment causes the ACK spoofer to send one spoofed ACK immediately during the connection establishment phase and one more RST segment after the timeout ticks out. Therefore, each SYN segment sent by an attacker generates two additional segments from the ACK spoofer, eventually amplifying the attack. Another drawback is that the method requires to software-process all segments with an ACK flag, disabling their ability to be hardware-forwarded. According to our traffic analysis captured on CESNET's network, pure ACKs make approximately 81% of all TCP traffic (Subsection 4.1.2), so the need of their analysis by the software has a rather significant impact on the performance.

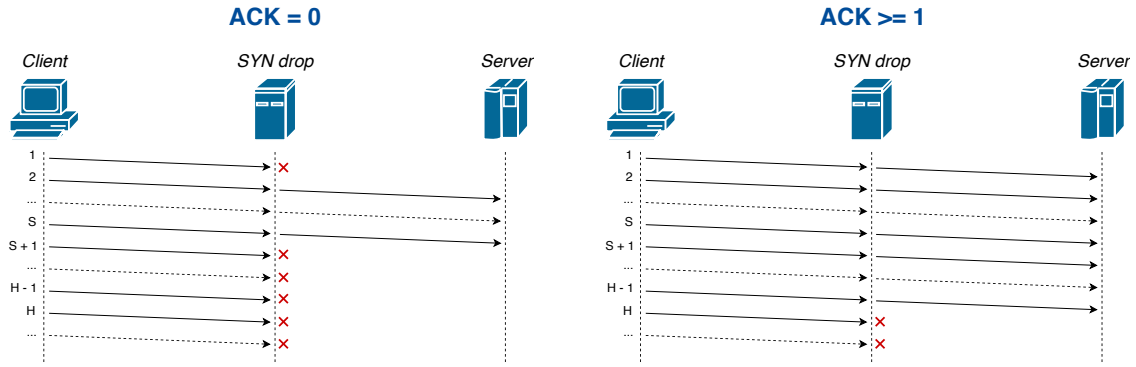
Despite all the mentioned disadvantages, the method is quite popular in IPS systems and is often used in conjunction with other mitigation methods.

2.3.7 SYN Drop

SYN Drop is a name of the proprietary method developed especially for CESNET's DDoS Protector project. Its functionality depends on soft (S) and hard (H) thresholds, which are used to limit the maximum throughput of SYN data that is allowed from a single client. The module keeps an internal table of IP addresses for all active TCP clients. Each IP address has an associated counter that represents the number of SYN segments sent by the client in the actual time window. If the number of SYNs exceeds an active threshold, all other SYN data sent by that client in the given time window are discarded (Figure 2.8). The active threshold is determined based on the number of ACK segments the particular host sends. If no ACK is sent by the host (Figure 2.8a), the soft threshold is active. Receipt of at least one ACK activates the hard threshold for current and all consecutive time windows (Figure 2.8b). On the top of traffic limiting, simple protection against SYN port scanning is also included. Its functionality is implemented by dropping the first SYN from clients with no ACKs yet sent.

The method provides decent protection against regular SYN flood attacks from spoofed addresses which do not generate an ACK reply. However, it can be easily fooled by injecting an ACK into the flood or by using more sophisticated attacks like Session Attack. The usage

is not limited for end-host nor network-based deployment, and so the method can be used in both scenarios without significant benefits and drawbacks.



(a) No ACK recieved (b) ACK recieved

Figure 2.8: SYN drop functionality.

2.3.8 TCP Anti-DoS Extensions

Alongside specialized techniques to prevent DoS mentioned in previous subsections, modifications to the protocol itself were also made. These tweaks were in the form of TCP extensions, which were supposed to provide anti-DoS functionalities even without the usage of other mitigation methods. The first extension – *TCP Cookie Transactions*, provided a cryptologically secure mechanism to guard against simple flooding attacks sent with bogus IP addresses or TCP ports. Usage of the extension avoided resource exhaustion on a server by not allocating any resources until the three-way handshake completion. Unlike SYN Cookies, the approach did not conflict with other TCP options but required support for both of the communicating hosts [24]. This restriction has proven to be crucial, because hardware vendors and software providers mostly ignored to implement it, and so the method was never popularized.

The second approach – *TCP Fast Open* replaced Cookie Transaction mechanism in 2014. The original intention of the standard was to provide a way to exchange the data between clients before establishing a 3-way-handshake, thus making the data transfers faster. However, the usage of the cookies also provided an ability to stop an attacker from trivially flooding spoofed SYN packets. On the other hand, new types of attacks specifically against TCP Fast Open may be launched. Their success may temporarily disable the mechanism, so usage with traditional SYN flood mitigation methods is still recommended [7]. Although the standard is marked as experimental, Linux and FreeBSD kernels, as well as several web browsers are already supporting it, though the method is sometimes disabled by default.

2.3.9 Current Trends in TCP DDoS Mitigation

Methods described above are generally all used for SYN flooding attacks mitigation. Defense against other types of TCP floods described in 2.2.4 requires the usage of heuristic methods with state information. The main idea behind these methods is that all legitimate TCP hosts have to establish a TCP session before sending other TCP data. The software would then block all TCP traffic except the one used to establish a TCP session in a fashion, that a certain number of ACK segments would be allowed to pass if and only if an SYN segment

from that particular IP was received within the timeout range. Ideally, this constraint would be extended by monitoring outbound traffic as well, and ACKs would then be allowed only if the server had previously responded with an SYN-ACK for the given session. This principle would beat all types of dummy TCP flooding, which does not rely on establishing a session before launching the attack. Other mechanisms such as counters and thresholds would then also be needed in cases that the flood would be conducted with established sessions. The proposed mechanism provides an undoubtedly high level of protection, but overall network performance is considerably degraded because each type of the TCP segment needs to be processed by the software and hardware-forwarding capabilities such as in CESNET's DDoS protector cannot be used.

More sophisticated DDoS attacks like spoofed session floods or session attacks are frequently able to bypass most of the techniques mentioned in previous subsections. Their mitigation has to be done with advanced methods like Deep Packet Inspection (DPI) combined with the usage of Artificial Intelligence (AI) and machine learning. DPI principle is used to analyze multiple fields of the packet headers, often up to application protocols. Its combination with AI may be able to discover traffic patterns that would not be revealed with traditional techniques. According to these patterns and possible experience of the AI, a potential attack may be triggered and particular data forming it would be dropped.

In real-world situations, both end-host and network-based solutions are frequently employed, and they generally do not interfere when used in combination [9]. Current trends in DDoS mitigation also utilize cloud technologies (e.g. Cloudflare[1]) instead of traditional IDS/IPS systems, but the mitigation principles stay mostly the same as those described in this document.

[1] https://www.cloudflare.com/

Chapter 3

TCP Reset Cookies

This chapter presents TCP Reset Cookies, a heuristic method for TCP SYN Flood mitigation. The method was designed and implemented to complement existing algorithms in CESNET's DDoS Protector mentioned in the previous chapter. This project uses proprietary high-speed FPGA networking technology and custom NDP frame headers, but all concepts mentioned in this section are not explicitly tied to any hardware and can be used in any TCP/IP network.

The first mention of TCP Reset Cookies can be traced back to 1996 according to the citation in [22]. Unfortunately, the method was never officially published, and the original proposal was only in the form of e-mail communication. The approach was never popularized because it was not compatible with Windows 95 clients [2] and the execution of the method had created unacceptable delays due to low speed in computer networks those days. Mentioned e-mail communication was probably deleted, and so only a few resources about this approach exist to this day. For the purpose of our custom implementation, the method needed to be „reinvented" by estimating the behavior of the clients according to the specification and actually testing various operating systems to confirm the expected compatibility.

Sections at the beginning explain the theoretical foundations of the strategy, as well as its design and implementation aspects. Latter sections summarize the achieved results, compare the method with its adversaries and discuss its usability in real networks.

3.1 Theoretical Background

TCP Reset Cookies functionality is based on the three-way handshake mechanism and relies on the client's behavior as defined in the RFC 793. The main idea is to establish a security association with clients before allowing their connection requests. This is achieved by intentionally crafting invalid SYN-ACK responses to SYN data received from a client. When an invalid SYN-ACK is received, the RFC 793, section 3.4 [21] defines the behavior as follows:

> *If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), a reset is sent.*

To distinguish that the RST segment is associated with the receipt of invalid SYN-ACK, RFC 793, section 3.4 [21] also defines requirements on the sent RST:

If the incoming segment has an ACK field, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment.

According to these preconditions, the algorithm is able to distinguish a legitimate client from an attacker, supposing that the the client will send an RST reply with the expected *SEQ* value, whereas an attacker will not. When an RST with the correct *SEQ* field is received, a security association is established by whitelisting the client's IP address. SYN traffic originating from whitelisted IP addresses is forwarded to its desired destination without further tampering (Figure 3.1).

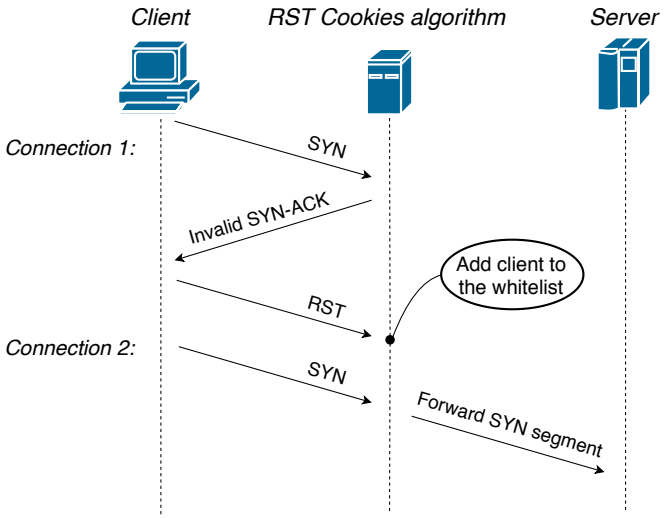


Figure 3.1: RST Cookies functionality.

3.2 Method design

As outlined at the beginning of this chapter, the developed method is supposed to complement existing algorithms in CESNET's DDoS Protector solution. Section 2.3 stated that these algorithms, already provided by the solution, are SYN Drop and ACK Spoofing. As described in their respective subsections, both of the approaches have their pros and cons. The SYN Drop algorithm provides adequate protection against regular SYN flooding attacks but falls short against attacks carrying spoofed ACK segments. ACK Spoofing protects the backlog during the attack with low packet rate but is easily overwhelmed on high packet rates and eventually even amplifies the attack in those cases. As may be seen, the currently deployed methods are not suitable in certain scenarios and a solution to cover other variants of SYN floods is required.

For this purpose, RST Cookies – a network-based mitigation method able to handle both regular and more sophisticated attacks is designed. Simple SYN floods are dropped by default because SYN sending IP addresses will never be added to the whitelist since they would not pass the security association phase of the algorithm. This mechanism is especially effective against attacks from spoofed IP addresses, which can not generate a valid RST reply. Spoofed Session Floods meet the same fate, since generating random

ACK, RST, and other segments can not fool the security mechanism because the specific value in the *SEQ* field of the RST is expected. The only way for an attacker to bypass the association phase is to monitor the traffic and inject an RST segment with the desired *SEQ* to the session. Another way is to use legitimate clients with the implemented TCP stack. Either way, the attacker can not use spoofed IP addresses, because the only way how to get to the whitelist is by responding to received invalid SYN-ACK with a valid RST. Proposal for this method presents an undoubtedly higher level of security than the currently used techniques.

The following subsections will shortly describe individual design concerns of the developed algorithm.

3.2.1 RST Cookies as a Module

Since the implementation is supposed to be part of the way more complex security solution, the initial design has to be adjusted to meet the specific needs. The algorithm will not be used permanently, but its caller will typically switch between different mitigation methods. For this reason, the algorithm will not be implemented as a standalone application, but as a module, which needs to be easily activated, disabled or removed on demand. This requirement also implies that the module will not capture the TCP segments itself, but will process already-parsed data received directly from the caller. The module will also not forward nor drop segments, but will only suggest how the packet should be handled via the return value of its functions.

The initial requirements for the module are rather straightforward. At first, the module needs to be initialized and be ready to process SYN and RST segments, while being able to generate invalid SYN-ACKs with secure *ACK* values. Whitelist needs to provide a way of aging, so older records are considered invalid and are automatically removed from the list. Ability to clear the module to its initial state after initialization needs to be supported as well. Finally, the module is required to have proper memory management. This decomposition leads us to create an initial draft of the module as shown in Figure 3.2.

The CESNET's DDoS Protector natively runs on the Scientific Linux, which is based on Red Hat Enterprise Linux core. For this reason, the module will prioritize to provide support for this particular operating system, although compatibility with other Linux systems should be achieved as well.

3.2.2 Module Initialization and Finalization

As mentioned in Section 3.1, the RST Cookies algorithm requires a whitelist to keep a record of the clients that have already passed the security association phase. Accordingly to this precondition, module requires an initialization phase, during which these data structures will be allocated and other internal variables set. The initialization phase is corresponding with the constructor in the object-oriented (OO) design. Due to internal politics of the DDoS protector, procedural design needs to be applied for this project. However, we will try to emulate OO design to be able to use its principles like abstraction and encapsulation. This approach will produce cleaner code while maintaining an easy way to rewrite the module into OO design when desired.

To sum it up, the module needs to provide a simple interface allowing the user to initialize and finalize the module. Initialization will allocate data structures posing as whitelist and set all the switches controlling the behavior of the module. The values used to set up the module were initially passed as function parameters, but they were eventually transformed

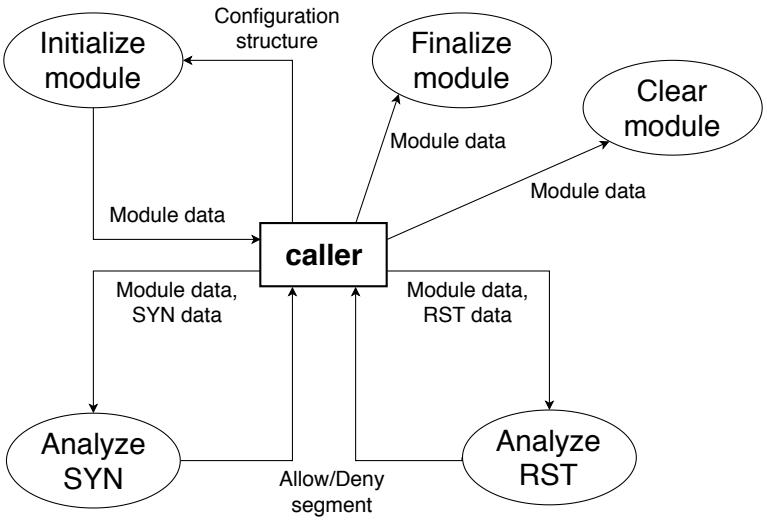


Figure 3.2: RST Cookies module initial design.

into the configuration structure as their number increased by the module adjustments. The module configuration parameters are furthermore explained in Subsection 3.2.8.

3.2.3 Whitelist

Previous sections have mentioned the need for a whitelist structure, which will be able to store IP addresses of the hosts that have already passed the security association phase. For this purpose, two data structures – *Hash table* and *Bloom filter* may be considered.

Hash table is a data structure implementing an associative array abstract data type, providing a mapping of keys to values. The mapping is done by a hash function, which is computed for the key and as a result, a concrete block of memory representing the value is returned. For our purpose, the IP address of the packet could be used as a key. This way, IP addresses contained in the hash table would be considered already-associated, and IP addresses not contained in the hash table would be considered unassociated. Usage of hash tables provides excellent performance, the certainty of the result and most importantly, the ability to store additional data for each key. However, this functionality comes at the cost of high memory requirements, which become unbearable on tens and hundreds of millions of entries.

Bloom filter is a space-efficient probabilistic data structure, which provides an ability to test whether a given element is a member of the set. As a result of its probabilistic property, false positives may happen. The filter is thus able to identify whether the element is possibly in the set, or definitely not in the set [20]. Due to leveraging a property of uncertainty, the structure can store its results with exceptionally low memory requirements (only 114 MiB for 100 million entries with the 0.01 probability of false positives [17]). However, the principle of mapping a key to a value is not possible, so no additional data could be stored for each IP address. Additionally, elements added to the structure cannot be removed. The operation of adding and querying an element is also much slower because multiple hash functions need to be computed.

Although usage of both data structures would be possible, for the purpose of this project and the needs of the DDoS Protector, a hash table was chosen. The main reason for it is

its ability to store data related to the given element, potentially allowing advanced security tweaks, entries aging and various optimizations discussed later in this chapter.

Despite the fact that one hash table could be used for both IPv4 and IPv6 hosts, a disproportion between the usage of these types of addresses on the Internet still exists. For this reason, we decided to use a specialized whitelist for each address type. This way, the user can specify the size of the hash table for each of the address types separately. This feature allows memory requirements optimizations and overall better performance of the module.

The concrete hash table implementation[1] was provided by CESNET, which maintains a specialized fast hash table optimized for the usage with IP addresses. This hash table is designed with constraint that number of its columns must be a power of two. More importantly, another specific property is a limited row capacity, and so inserting an element to the full row causes the oldest accessed to be replaced. These features extend the regular table functionality by a proper memory management and entries aging mechanism, which is especially significant for the RST Cookies cause.

3.2.4 SYN Processing

For the method to function correctly, a caller must ensure that all ingress SYN and RST segments originating from outside of the protected network are processed. When such SYN is received, the RST Cookies algorithm must determine whether it is from a new client or a client that is already associated. For this purpose, a source IP address is chosen as a key in the whitelist to search for. Each entry contains a nanosecond timestamp specifying when the association has been created (t_a), allowing entries to age. So, upon an SYN segment arrival (t_s), the algorithm has to check whether the IP address of the source is contained in the whitelist and its entry timestamp does not exceed the maximum specified age time (t_m), thus validating the following condition:

$$t_s - t_a < t_m$$

If the preceding condition is met, the SYN segment is forwarded into its desired destination. Otherwise, an invalid SYN-ACK is assembled and sent as a response to the processed SYN as defined Section 3.2.

If regular data structures were used, manual removal of the expired entries would be required. However, our hash table automatically replaces the oldest entry when insertion to the full row is made. As outlined in the Subsection 3.2.3, this process ensures proper aging mechanism, because oldest entries are systematically removed by default. Another assurance is the automatic memory management since the size of the table never exceeds its initial size at the time of the initialization no matter how many IP addresses are added to it.

The system of client validation described previously may stop regular and most of the sophisticated SYN floods, but may fall short in certain situation such as when an attacker manages to bypass the association phase. For this reason, an algorithm enhancement furthermore discussed in Subsection 3.2.7 is proposed.

[1] https://github.com/CESNET/Nemea-Framework/tree/master/common/fast_hash_table

3.2.5 RST Processing

When an RST segment is being processed, the module has to decide whether the message is a part of its mechanism or belongs to the regular TCP traffic. This is achieved by looking at the *SEQ* value of the analyzed RST. If its *SEQ* is equal to the *ACK* sent in the previously generated invalid SYN-ACK, the RST is a response to it. In this case, the processed RST segment is dropped and client IP address added to the whitelist. Otherwise, the segment is not part of the RST Cookies algorithm, hence gets forwarded to its desired destination.

3.2.6 Invalid ACK Generation and Validation

As mentioned in previous subsections, the algorithm needs to generate an invalid SYN-ACK segment and then match a corresponding RST to it. Generally, invalid SYN-ACK is crafted by violating the three-way handshake process through setting a segment's *ACK* value differently from $SEQ + 1$ of the SYN it is responding to. Responding with random numbers is possible, but would not allow the process of client verification.

The initial design allowing the incoming RST validation used a constant value, that was placed in each SYN-ACK response and then checked for the match in the RST. This approach was functional, but its security properties were insufficient. A smart attacker that is able to monitor the traffic could easily inject an RST segment with the given constant to trick the security mechanism. To tackle this issue, a system for dynamic ACK generation and validation is proposed.

The main purpose of the *Dynamic ACK Generator* is to provide a secure way to generate invalid *ACK* values for SYN-ACK messages and to validate *SEQ* numbers in received RST segments. The main concerns of the generator are CPU requirements and security of the generated results. Based on these factors, two generator policies – *Random windowed mode* and *Hash mode* have been designed. Each of the methods focuses on one of these factors meanwhile weakening the other one. This subsection will furthermore discuss each of these policies in detail.

Random Windowed Mode

The fundamental idea behind this policy is to generate random numbers periodically and to assign *ACK* values from the particular time window to SYN-ACK segments according to the time of their generation. When an RST segment is being processed, the algorithm iterates over the structure of these lastly generated values and searches for a match between the generated *ACK*s and the *SEQ* read from the RST segment. The number of iterated elements depends on the *ACK* generation period and the validity of the generated values. When configured sensibly, this method is faster and allows better message throughput than its counterpart.

The functionality of this principle depends on a fixed-sized array, having the minimum of $N = \lceil V/T \rceil$ elements. Value V denotes the validity of the generated *ACK* values while T represents the new *ACK* generation period. This structure is used as a ring buffer (Figure 3.3), which stores lastly generated values and is iterated when a value validation is needed. When an invalid SYN-ACK is requested, the algorithm needs to look on a timestamp of the lastly generated value and compare this timestamp with the current time. If their difference exceeds the specified *ACK* generation period, lastly generated value buffer index is incremented (from the window with time t to the window $t + T$), and a new value is generated on its position. The algorithm then takes a value from the buffer element

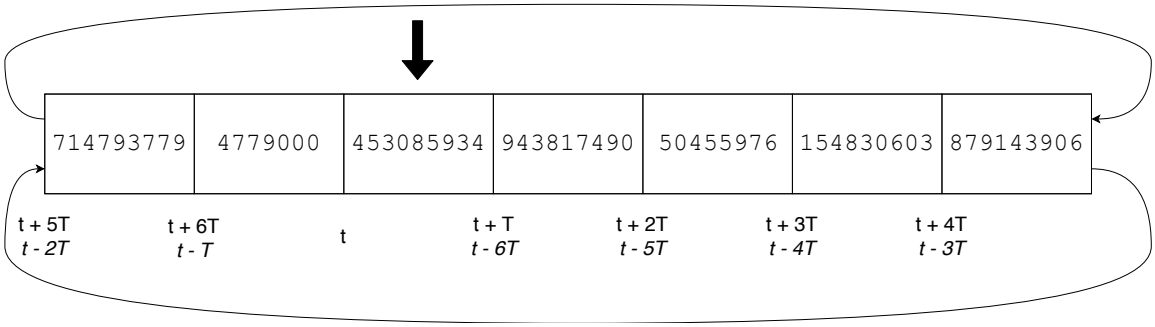


Figure 3.3: Invalid ACK generator – Windowed mode principle.

currently being pointed on and returns it as a generated *ACK* value. Validation is done by iterating these time windows in reverse order, starting in the time t and proceeding up to the time $t - N \cdot T$, where N is a minimum size of the array calculated previously.

This policy provides a relatively good performance, which is mostly affected by the speed of the random number generation and the iteration of the ring buffer. Sensible size of the buffer is up to 10 elements while respecting the minimum recommended *ACK* validity of at least several seconds. Lower values of the *ACK* generation period may provide better security, but its effect can be neutralized by the long validity of *ACK* values. The main drawback of this approach is apparent – the same *ACK* value is used for all invalidly acknowledged SYN segments in the given time window. This practice allows the attacker analyzing the traffic to extract the invalid *ACK* value and place it as a *SEQ* of the crafted RST, which will be injected onto the network to fool the security mechanism and whitelist the attacker's IP address. However, this scenario is not very likely to happen, but the security concern still exists, so the hashing method has been developed to provide a higher level of security.

Hash Mode

The second approach is somewhat inspired by the SYN Cookies principle. As illustrated in Figure 3.4, a unique hash is computed for every connection according to its parameters. Segment source IP, a 32-bit secret, TCP source port, destination port, and a 32-bit timestamp are hashed into a 128-bit string. The first 32-bits are taken, and 12 least significant are replaced with a modulo of the shifted timestamp with 4-second precision. This technique provides a reasonable trade-off between security and performance because the attacker would have to guess 2^{20} possibilities from the hash alongside four different timestamps. Four-second precision was chosen because it would take $2^{2^{12}}\text{s} \sim 194$ days to perform a replay attack due to the timestamp repetition. Though not yet implemented, this duration could be prolonged by a pool of secrets, which would prolong the possibility of a replay to the previously calculated value multiplied by their number. To verify a received RST, the algorithm reconstructs the timestamp by deriving its value before modulo application, shifting it back to 1-second precision, and computing the hash function for every possible second in the given time window, because perfect precision was lost due to the previous shift. If the reconstructed timestamp is within the timeout range and first 22-bits of the computed hash match the first 22-bits of the *SEQ* in analyzed RST, the client is considered legitimate.

This method provides undoubtedly stronger security since an unique *ACK* is generated per every connection instead of the one value for all segments in the given time window. On the other hand, the CPU is utilized significantly more because hash functions need to be calculated for every processed segment. For this reason, latency may be slightly increased, and packet throughput could also be fairly reduced. Comparison of both *ACK* generation/validation policies is furthermore discussed in the results section.

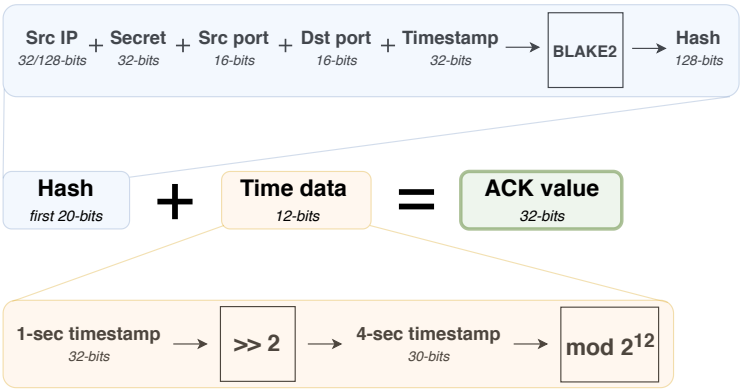


Figure 3.4: Invalid ACK generator – Hash mode.

ACK Generator as a Submodule

To provide an adequate level of flexibility and abstraction, the ACK generation algorithm was designed as an independent module. It will be used for the internal calls of the RST Cookies algorithm, providing the functionality of invalid *ACK*s generation and validation.

Adaptive way of generation is achieved by an ability to switch between different policies during the program execution. For this reason, both of the generation algorithms need to be used to determine the validity of the analyzed value. For the optimization purposes, each of the generators contains an internal nanosecond-precision timestamp that indicates the time when the generator was lastly used. This way, evaluating a single condition tells the algorithm whether the particular generation method has to be used for verification.

As it emerged from the previous paragraphs, the functionality of the ACK generator requires state information to be kept. Each of the generators contains own internal data (Figure 3.5), while the main generator structure wraps these two generators and provides a switch defining which of the generator policies is currently active.

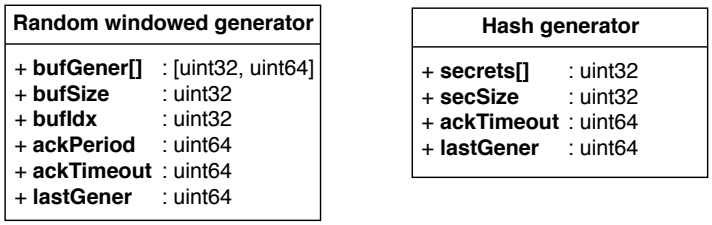


Figure 3.5: ACK generators internal structure.

The random windowed generator requires a mentioned ring buffer of lastly generated values (`bufGener[]`). For the optimization purposes, each generated value is internally

stored as a pair of a 32-bit value and a 64-bit timestamp denoting its generation time. This way, the algorithm does not need to iterate whole ring buffer but stops at the first value with expired timestamp. Index to the buffer (`bufIdx`) defines the item corresponding to the current time window. The generator also requires to know the generation period (`ackPeriod`) and validity of the generated values (`ackTimeout`). The last item (`lastGener`) represents a timestamp of the last generator usage. The hash generator contains a buffer of secrets (`secrets[]`) that are used to calculate a hash. Other elements have the same meaning as in the random windowed generator.

The interface of the submodule aims to provide a simplistic way to achieve all the requirements from the ACK generator. First of all, functions to initialize, finalize and clear the module are needed. Initialization constructs both types of the generators and sets them to the initial state, clearing sets the module to the state after initialization, and finalization destroys the generators and deallocates the memory. The requirement to switch between different generation policies is addressed by another separate routine, which takes a generator instance and the policy to be switched to. Function to generate an *ACK* value accepts the ACK generator instance and returns the generated value according to the active policy as described in the paragraphs above. Function for validation processes two conditions, each checking timestamp of lastly generated value from the particular generator and calling the corresponding generator routine if the timestamp does not exceed the timeout. Since the hashing variant requires to obtain IP addresses and port numbers, IP and TCP headers are also passed to these methods.

3.2.7 Enhancing the Security – SYN Limiting

The original proposal of the algorithm considers a client to be trustworthy after the security association phase is completed. Although this approach may stop most of the attacks, a situation when an attacker successfully bypasses the mentioned association phase may occur. Recall that when an RST with the expected value is received, the IP address of the source is added to the whitelist. From this moment, the particular host is able to freely send any number of SYN segments which will not be intervened by the TCP Reset Cookies algorithm. This behavior may be abused by attackers, who may be smart enough to utilize a regular TCP stack at the start of the attack or somehow inject an RST segment with the desired *SEQ* value.

Our proposal tries to address this problem addresses by enhancing the regular algorithm functionality. This is achieved by adding a counter and timestamp to the hash table data alongside the existing association timestamp. The counter is used for counting SYN segments from the associated clients, and the timestamp denotes the start of a 1-second time window. By using these two extra variables, the algorithm can limit the number of SYN segments sent by already-associated clients. This approach might stop even more sophisticated attacks that successfully pass through the security association phase. When combined with a blacklist, the ability to detect these smart attackers and deny their traffic completely is available.

The enhanced variant of SYN processing with SYN limit feature is depicted in Algorithm 1. The mechanism firstly looks for the data related to the Source IP address (line 1). If the entry exists, the check for an entry validity is performed (line 5). When the SYN counter is enabled and a 1-second time window is already started, the SYN limit is checked if it has not been reached. When that is the case, the algorithm proceeds accordingly (line 11 - 18). If the time window is not in progress, the counter is set to 0 and a new

Algorithm 1: RST cookies – SYN processing.

```
1  entry ← Source IP data from association table;
2  if entry == NIL then
3  |   send invalid SYN-ACK;
4  |   Drop packet and exit;
5  else if t_s − t_a < t_m then
6  |   Delete src IP from association table;
7  |   Send invalid SYN-ACK;
8  |   Drop packet and exit;
9  end
10 if SYN limiting enabled then
11 |   if t_s − t_entry.window_start < 1s then
12 |   |   if entry.syn_cnt ≥ SYN limit then
13 |   |   |   if Blacklist enabled then
14 |   |   |   |   Add IP to blacklist;
15 |   |   |   end
16 |   |   |   Delete src IP from association table;
17 |   |   |   Drop packet and exit;
18 |   |   end
19 |   else
20 |   |   t_entry.window_start = t_s;
21 |   |   entry.syn_cnt ← 0;
22 |   end
23 |   entry.syn_cnt ← entry.syn_cnt + 1
24 end
25 Allow packet and exit;
```

time window is started (lines 19 - 22). Segment that has not been dropped yet has its SYN counter incremented, and gets forwarded (lines 23 - 25).

3.2.8 Putting It All Together

With most of the major design concerns already discussed, a final version of the RST Cookies internal structure may be revealed (Figure 3.6). This data will be hidden from the external access and are supposed to be merely visible by functions of the module itself and no other entity. The module utilizes two whitelist tables for IPv4 and IPv6, comprises a dynamic *ACK* generation submodule and stores user-entered arguments of maximum whitelist entries age time, SYN limiting status, and the actual SYN limit. These settings are specified by the user in the initialization function via a configuration structure, which mostly copies the content of the internal module's structure with a difference, that the user specifies the size of whitelist tables and includes configuration structure for the Dynamic ACK generator.

Because the module makes use of the specialized hash table discussed in Section 3.2.3, only four entries per row may be stored. On account of this behavior, a situation when a legitimate client is removed from the table before its age time expires may occur. This event happens when the chosen size of the hash table is not respecting the properties of a protected network. To tackle this issue, the module offers a statistics logging mechanism that may help to detect this situation and adjust the hash table size appropriately. These statistics are stored in the internal structure and can be obtained via the corresponding function.

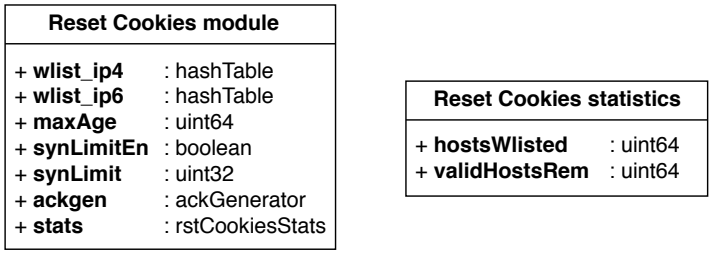


Figure 3.6: RST Cookies module internal structure.

Alongside the features discussed in previous subsections, the module also needs to provide an interface to manipulate *ACK* generator by using RST Cookies functions. Since the generator is used internally in the module, its behavior cannot be changed by direct calls, and so the RST Cookies interface has to provide wrapper functions to control generator behavior like setting the current generation policy.

3.2.9 Module Wrapper for Testing Purposes

For the purpose of verifying functionality and performance of the module, it needed to be used as a part of a more robust program. Since the module interface and features were adapted for the needs of DDoS Protector, it heavily depends on its caller. For this reason, it cannot be used as a standalone program by itself, but a specialized application providing environment the module requires needs to be used.

One of the possible solutions is to integrate the module straightly into the DDoS Protector and test it like that. However, this approach would not allow detailed debugging and would make executing the performance tests harder. Also, the functionality of the method was questionable from the beginning, so we needed to ensure compatibility with various operating systems (Subsection 3.4.1) before developing the module further. This approach led to design and development of a small program that represented a wrapper utilizing the RST Cookies module while providing DDoS Protector-like interface and features to make the module functional. A sequential evolution of this small application created a standalone program, that may be used on the intermediary device to provide RST Cookies functionality by itself.

The main concern of the wrapper is to provide or simulate an interface the module depends on while being able to utilize the functionality of the module itself. This means that the wrapper needs to read all the incoming data from the given interface and pass all SYN and RST segments to the wrapped module. Other data the RST Cookies does not process should be forwarded to their destination. The wrapper then needs to respect decisions made by the module, so SYN and RST segments are forwarded or dropped as the RST Cookies algorithm suggests. One the of most important responsibilities of the wrapper is to simulate the DDoS Protector's system for queuing and sending packets. Using this system, modules willing to send a segment need to "ask" for the memory to load the segment into. After this process, the data are automatically sent. Modules using this approach also do not need to include data link (L2) headers, which are automatically prepended by the system when the packet is sent. This approach allows greater flexibility of the modules because they do not need to rely on low-level layers for their functionality.

3.3 Method Implementation

As mentioned back in 3.2.2, the DDoS Protector policy forbids the usage of OO design, because of the required implementation in *C*, a procedural language. The chosen language disables various OO functionalities but allows us to write remarkably fast programs, which is indeed the primary concern of the software that processes and filters real-time data.

The development of the module and the wrapper was performed using evolutionary prototyping development methodologies. Following these principles, the initial program confirming theoretical assumptions of the method's functionality was rebuilt into several prototypes as new features were added. Extending and verifying the last prototype formed the final product described in this document.

The following subsections will provide a high-level overview of the module implementation, describe interesting facts and minor deviations from the initial design related to the method implementation.

3.3.1 Memory Management

All the required memory for the correct module functionality is allocated using a `malloc()` call during the phase of the module initialization and deallocated with a `free()` call when the module is finalized. The module would normally also allocate a buffer that would serve as a place to assemble invalid SYN-ACKs to. However, as a module for the DDoS Protector, a different approach needs to be taken. The Protector's API provides a function `packet_queue_get_data()`, which returns an allocated buffer of a requested size to its caller and automatically sends the data after the buffer is filled. The algorithm is thus not supposed to allocate its own buffers, but rather to request a memory buffer from the main application when an invalid SYN-ACK needs to be sent.

This mechanism is employed due to optimization purposes, because the core of DDoS Protector would need to copy the module's buffer contents into the interface buffers, draining resources unnecessarily. While the system packet queuing is being used, the module straightly obtains desired memory it can write into, requiring no extra buffer copies.

3.3.2 Invalid SYN-ACKs Assembling

Invalid SYN-ACK segments are sent for every analyzed SYN whose IP address is not contained in the whitelist. The routine for SYN analyzing is thus required to find out the IP address family of the analyzed packets to choose an appropriate whitelist to search source IP address in. Since IP headers of both families are different, two internal procedures – `rst_cookies_respond_ip4()` and `rst_cookies_respond_ip6()` were implemented. Both of them ask for the memory as described in the previous subsection and fill the buffer with the appropriate information forming a TCP SYN-ACK segment. Both of these functions accept the same parameters, so the SYN analyzer chooses the appropriate one the same way as in the case of whitelists.

Forming the SYN-ACK

SYN-ACK segment is assembled with the use of system in-built networking header structures. Each structure represents a concrete protocol header used at a specific layer of the OSI model. These structures are then stacked onto themselves to form a valid TCP segment. Normally, each of the layers from L2 up to L4 would need to be included, but in our

case, only IP (L3) and TCP (L4) headers are required. Since the module uses a specialized function to obtain a buffer to fill the data into, Ethernet (L2) and proprietary NDP headers are automatically prepended by the internal mechanisms of the DDoS Protector before the packet is sent. This way, the Protector's modules are isolated from handling low-level data and thus provide better flexibility and cleaner code.

After the SYN-ACK segment is formed, its fields need to be filled with the IP and TCP data to form a legitimate network message. For this purpose, source IP address, destination IP address and TCP port numbers are taken directly from the received SYN while swapping their source destination fields. To provide a desired functionality, *ACK* values are generated with the `rstcks_ackgen_getack()` call, which returns a value from the *ACK* generator based on the currently active policy. *SEQ* values of the SYN-ACKs play no role in the RST Cookies mechanism, but they are also generated pseudorandomly to make produced segments look like legitimate traffic. Other fields for IPv4 and IPv6 header are filled according to RFC 791 and RFC 2460 standards. TCP header fields comply to RFC 793.

Checksum calculation

An integral part of the SYN-ACK assembling process is an IP and TCP headers checksum computation. The segments without valid checksums may be dropped by intermediary devices or ignored at their destination. The computation process is typically handled by operating systems when programming with networking API using sockets, but since the module is assembling whole segments from scratch, a checksum computation needs to be executed manually. The required way of checksum computation is defined in each RFC separately, but each of them follows the algorithm initially described in RFC 791 [1]:

> *The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.*

As this definition is not so straightforward, several Internet sources needed to be consulted to understand and implement the required algorithm at last. Actually, all the checksum function has to do is to compute a sum of 2-byte blocks, add the left-over byte in odd data sizes and fold a possible 32-bit sum to a 16-bit checksum by taking the 16 least significant bits and adding them to bits on the position 16 to 31. Finally, a bitwise negation of the computed value is performed and the result is returned. This process is the same for both IPv4 and TCP headers. The header for IPv6 does not contain a checksum field, so the calculation process is not performed.

A specialized calculation needs to be performed for TCP header, which requires to execute a checksum algorithm for both TCP header and pseudo-header composed of source and destination IP addresses, length of the TCP segment and a protocol identifier. Structure of this pseudo-header is described in RFC 793, section 3.1 [21] for IPv4 and in RFC 2560, section 8.1 [16].

Algorithms for checksum calculation found on the Internet are typically implemented in a generic way, allowing computation of the algorithm for any data they receive without other functionalities. This approach is, indeed the variant with the biggest flexibility, but does not provide a performance our module requires. Usage of the generic approach would require allocating the dynamic memory, copying the header contents to it, appending and filling a pseudo-header and then passing it to the computation method. Even if the buffer

would be already preallocated, the process of copying the header for every processed packet is rather ineffective. Instead, our functions for checksum computation – `chksum_calc_ip()` and `chksum_calc_tcp()` specialize on calculating the checksum for the respective type of the header, being able to create and calculate checksum values of pseudo-headers on the stack, avoiding unnecessary memory allocation and data copying. These functions provide better performance at the cost of less flexibility since each of them is explicitly tied to the particular header and cannot be used for any other one.

The problem with this specific approach was actually experienced during the compilation with the last version of GCC – `gcc 8.2.1` using the `-O3` optimization flag. An error in the compiler caused a part of the created pseudo-header structure assignments to be ignored (the first 5 lines in Figure 3.7a). Because of this, all the necessary data in pseudo-header were not included, and so the checksum was computed improperly. Compilation with lesser optimizations using flag `-O2` did not trigger the error, and the checksum was computed validly. However, DDoS protector required the usage of the `-O3` flag, so a hack to disable particular optimization which caused the problem was needed. The issue was resolved by replacing a direct structure initialization with two `memcpy()` calls (first half of Figure 3.7b), which apparently do not trigger the optimization, and so the checksum calculation returns a correct result.

```
/* Prepare the pseudo-header structure. */
tcp_pheader6_t pseudohdr = {
   .len = htonl(tcpdata_len),
   .next_hdr = htonl(IPPROTO_TCP),
};

memcpy(pseudohdr.ip6_src, &(((struct ip6_hdr *)
   ipdata)->ip6_src), IP6_ALEN);
memcpy(pseudohdr.ip6_dst, &(((struct ip6_hdr *)
   ipdata)->ip6_dst), IP6_ALEN);



/* Process the pseudo-header. */
uint16_t *pdata_ptr = (uint16_t *)(&pseudohdr6);
for (unsigned int i = 0;
     i < sizeof(tcp_pheader6_t); i += 2) {
   sum += *pdata_ptr++;
}
```

(a) Original code

```
/* Prepare the pseudo-header structure. */
unsigned int proto_tcp = htonl(IPPROTO_TCP);
unsigned int payload_len = htonl(tcpdata_len);
tcp_pheader6_t pseudohdr6;

memcpy(&pseudohdr6.next_hdr, &proto_tcp,
   sizeof(unsigned int));
memcpy(&pseudohdr6.len, &payload_len,
   sizeof(unsigned int));
memcpy(pseudohdr6.ip6_src, &(((struct ip6_hdr *)
   ipdata)->ip6_src), IP6_ALEN);
memcpy(pseudohdr6.ip6_dst, &(((struct ip6_hdr *)
   ipdata)->ip6_dst), IP6_ALEN);

/* Process the pseudo-header. */
uint16_t *pdata_ptr = (uint16_t *)(&pseudohdr6);
for (unsigned int i = 0;
      i < sizeof(tcp_pheader6_t); i += 2) {
   sum += *pdata_ptr++;
}
```

(b) Modified code

Figure 3.7: Pseudo-header checksum computation GCC optimization bug fix.

3.3.3 Using Networking Header Structures

As mentioned in Section 3.3.2, invalid SYN-ACK segments are formed using operating system in-built networking header structures. These structures are also used for packet parsing – e.g., obtaining an IP address of the sender. In Linux operating systems, two variants of networking headers exist. The original *netinet* headers were historically included from the first versions of BSD and *nix operating systems. Linux subsequently added own *linux* networking headers to its kernel, and consequently, both header types are currently available to be used in Linux programs. They mostly consist of the same content, providing identical structures and constants.

Although both header types can be used, POSIX standard recommends the usage of *netinet* networking headers for application programs, while reserving the *linux* headers for internal usage within the kernel. However, a problem related to these different types of headers emerges because they historically implemented own structures with different names and structure members. This phenomenon was not a problem in the past because each header file declared only its corresponding structure, and so the application developer specified which type of structure was desired by including a particular header file. However, this was changed and *netinet* headers now include both its original and Linux structures in modern Linux kernels. Merging the different headers declarations into one worked fine for most structures, but combining both TCP and both UDP headers created a dubious situation.

As stated, structures in both header files define the same networking headers, but under a different name, so they did not come into the conflict when a *linux* to *netinet* headers merging was made. However, TCP and UDP headers were historically defined under the same name (`struct tcphdr`, and `struct udphdr` respectively) but names of their members were different. This fact indeed created a conflict and a problematic situation lasting up to this day. When a programmer includes the `netinet/tcp.h` header file and uses one of the available TCP structures, a mechanism able to determine which of the headers was used is needed during the compilation. And this particular behavior is the cause of many problems related to TCP/UDP Linux network programming.

A typical problem arises when the program tries to utilize former *netinet* TCP/UDP headers. Most Linux systems prioritize native *linux* headers by default, and so the error '`struct tcphdr has no member named ...`' is issued during the compilation phase. However, the tricky part is that some systems still prioritize former *netinet* structures, hence the compilation proceeds without errors on a few systems. The best cross-system compatible solution we discovered so far is to include *netinet* header files and use Linux structures (ending with "`hdr`"). Alternatively, *netinet* structures can be chosen, but macros `-D_BSD_SOURCE`, `-D__BSD_SOURCE`, and `-D__FAVOR_BSD` have to be used during compilation to tell the system to prioritize original *netinet* structures.

Initial versions of the program used original *netinet* structures with the mentioned macros included. This version of the module is also submitted as a practical part of the thesis on the attached CD/DVD medium. However, a newer policy of the DDoS Protector suggests the usage of *linux* headers, so the module was ported to these types of header structures and integrated to the solution in that particular state.

3.3.4 Invalid ACK Generation

According to the initial design, the *ACK* generator was implemented as a standalone module, providing a standardized way to be initialized, finalized, and cleared. Alongside these functions, two calls – `rstcks_ackgen_getack()` and `rstcks_ackgen_validate()`, providing the main functionality of the module, are available.

ACK generation is controlled by the switch, which determines the particular generation type based on the internal state of the module. *ACK* validation is done by checking last generation timestamps of both generators and calling their validation routine if the timestamp is within the *ACK* timeout range. The analyzed *ACK* is thus considered valid, if at least one generator acknowledges that the given value was generated by it within the *ACK* timeout range. This subsection will furthermore describe implementation specifics of each of the generator types.

Security Considerations

In order to minimize the chances of generated *ACK* values to be estimated, the *ACK* generator module utilizes the *libsodium*[2] cryptographic library. Its API provides a generation of cryptographically secure random numbers, which are used as *ACK* values while the random windowed generation policy is active. Random numbers are also used as secrets for hash generation policy, which combines session parameters, a timestamp, and a 32-bit secret to generate a hash used as an *ACK* value. To make this way of the generation even more secure, hashes are computed with the cryptographic one-way hash function *BLAKE2b*, also provided by *libsodium*. With all these security measures, a possible attacker is not able to estimate the generated *ACK* sequence, as well as decipher the values used to create a hash, making both policies resistant against various cryptoanalytic attacks.

Random Windowed Generator

Generation of random *ACK*s with windowed policy is rather straightforward. Algorithm checks whether the current timestamp subtracted by lastly generated value's timestamp is less than the *ACK* generation period. If this condition is true, new value is not generated and lastly generated value is returned. Otherwise, the `randombytes_random()` call to *libsodium* is used to obtain a block of 32-bit random data. This value forms a pair with the current time timestamp and is inserted into the internal data structure, as well as returned as a newly generated value.

```
1  bool rstcks_randgen_check(uint32_t ackval, uint64_t timestamp, const rstcks_randgen_t *randgen) {
2      unsigned idx = randgen->q_end; /* Index for iteration purposes. */
3
4      /* Iterate through the buffer to find match and check timestamps for validity. */
5      while ((idx + 1) % randgen->q_size != randgen->q_start) {
6          if (timestamp - randgen->data[idx].tstamp < randgen->ack_timeout) {
7              /* Entry in the queue has its timestamp still valid. */
8              if (ackval == randgen->data[idx].ackval) {
9                  return true;
10             }
11
12             /* If match was not found on the current index - proceed to the next. */
13             idx = (idx != 0) ? idx - 1 : randgen->q_size - 1;
14         } else {
15             /* Entry is not valid -> all other the until end are not as well. */
16             break;
17         }
18     }
19
20     return false;
21 }
```

Figure 3.8: Random windowed generator value validation.

The process of value validation is depicted in Figure 3.8. For various optimization purposes, the internal data structure is implemented as a fake queue, which is initialized as already full and contains one extra element that is used as a sentinel value. As can be seen, the iteration process starts on the last added element defined by the generator structure member `q_end`. The data structure is traversed backward up to the second elements before the latest generated one (line 5). The first element before the latest is

[2] https://libsodium.gitbook.io/

the sentinel value, which is not used in this function, and so is skipped. Line 6 evaluates the condition of the element timestamp validity. If the timestamp on the current index is invalid, all other elements will have their timestamp only older, so there is no reason to iterate further. For this reason, the algorithm signalizes that the *ACK* match was not found by returning `false`. If the timestamp is valid, the algorithm compares its value with the analyzed one and returns true if they match (line 8 - 10). Since the data structure is iterated as a ring buffer, simple decrementation to the iterator variable cannot be made, but we have to move to the end of the queue if the current index is 0.

The described mechanism and other adjustments provide a relatively fast way to work with the internal data structures, providing adequate security and decent performance.

Hash Generator

Instead of returning the same value for multiple requests in a certain time window, the *ACK* hashing method returns a unique value for every processed connection. This is achieved by filling the buffer with IP addresses, port numbers, current timestamp and a 32-bit secret. The prepared buffer is passed into the `crypto_generichash()` function, which returns a 128-bits long cryptographic hash computed from the given data. This process is demonstrated on the code snippet included in Figure 3.9. The cryptographic hash function takes a pointer to the buffer `data`, specified by its size `datalen`, and saves 128-bit hash string into the result buffer named `hash`. The 12 least significant bits are then taken and replaced with a shifted timestamp with a 4-second precision modulo 2^{12} to make sure that the time data will fit into 12 bits.

```
crypto_generichash(hash, 16, data, datalen, NULL, 0);

uint32_t ack_result = (hash & 0xFFFFF000 | ((timestamp >> 2) % (1 << 12));
```

Figure 3.9: Hash generator value generation.

As in the case with the random windowed generation, the validation of the results is actually more complicated than generating them. During the hash mode value validation, the algorithm needs to reconstruct the original timestamp used to compute the hash with. The process of value validation is demonstrated in Figure 3.10. The function firstly computes a modulo of the shifted timestamp and extracts the *ACK* value to be analyzed. A difference between the calculated and extracted moduled timestamps is computed. Each point in its result represents a 4-second block. Since original timestamp has 32-bits and a modulo version in *ACK* only 12, an overflow may occur. For this reason, line 10 performs a check and a potential fix if such situation occurs. Line 11 uses the computed delta to determine the start of the 4-second time window that was used in the computation of the extracted hash value. The algorithm then needs to try all different timestamps from the particular time window to determine whether an *ACK* match occurs. The hash function computation for all timestamps in a given window would be ineffective, so the condition on line 15 is firstly evaluated for every timestamp before the cryptographic hashing occurs. If the currently processed timestamp exceeds the *ACK* validity timeout, the computation is interrupted and `false` is returned. If the timestamp is valid, the hashing occurs, and the result is compared to the analyzed *ACK* to determine a match.

Optimizations with a timestamp explained previously improved the overall module performance significantly, but the need for cryptographic hash calculation still negatively im-

```
1  bool rstcks_hashgen_check(uint32_t timestamp, const void *ip_hdr,
2                            const void *tcp_hdr, const rstcks_hashgen_t *hashgen) {
3     /* Calculate current timestamp as it would be in a part of the ACK. */
4     uint32_t tstamp_mod = (timestamp >> 2) % (1 << 12);
5     /* Obtain analyzed ACK from the header and calculate difference between timestamps. */
6     uint32_t their_ack = ntohl(((struct tcphdr *) tcp_hdr)->th_seq);
7     int time_delta = tstamp_mod - (their_ack & 0xFFF);
8
9     /* Correct the possible overflow and calculate beginning of the hashed timestamp. */
10    if (time_delta < 0) { time_delta += 1 << 12; }
11    uint32_t tstamp_hashed = ((timestamp >> 2) - time_delta) << 2;
12
13    for (unsigned i = 0; i < (1 << 2); i++) {
14       /* Check if the calculated timestamp is within acceptable time window. */
15       if ((timestamp - (tstamp_hashed + i)) * 1000000000ULL <= hashgen->ack_timeout) {
16          uint32_t our_ack = rstcks_hashgen_generate(tstamp_hashed + i, ip_hdr, tcp_hdr, hashgen);
17
18          if (our_ack == their_ack) { return true; }
19       } else {
20          break;
21       }
22    }
23
24    return false;
25 }
```

Figure 3.10: Hash generator value generation.

pacts the speed of the method, as well as its data processing abilities. The policy provides undoubtedly stronger protection, but its other drawbacks prove improper in certain situations. For this reason, a caller using the module has to determine, whether the security of the application is the primary concern or a trade-off between security and performance is acceptable.

Note: Code snippets included in this section were partially modified for better readability and fewer space requirements. The code in the source files respects best coding style practices, like using macros for timestamp shifting, masking, etc., instead of magic numbers shown here.

3.3.5 RST Cookies as a Standalone Program

The wrapper providing a standalone RST Cookies functionality was initially developed for regular Ethernet networks by employing the *libpcap*[3] library. In order to test in the CESNET's 100 Gbps environment, the program was also ported into the proprietary NDP frame headers. The simulation of the DDoS Protector's environment was achieved by adapting several header and source files with macros, return values, and fast hash table from the DDoS Protector git repository.

The execution of the wrapper program starts by binding to the specified interface, initializing the module and entering an infinite packet reading loop. Each received packet is processed in a packet handler function, which sets the action flag to *forward* at its start. If the IP address of the parsed packet is not contained on the blacklist and its content is classified to be either SYN or RST TCP segment, it is passed to a respective RST Cookies function. Its return value replaces the previously set flag so that the RST Cookies

[3] https://www.tcpdump.org/

algorithm ultimately defines the fate of its processed segments. At the end of the packet handler function, a subroutine to process the packet according to its action flag is called. Its execution causes the packet to be either forwarded or dropped. If the discarded packet has an SYN flag, the wrapper knows that the wrapped module has created an invalid SYN-ACK and filled the buffer with a `packet_queue_get_data()` call. The filled buffer is sent, and thus processing of a single packet ends and the wrapper is ready to process another one. The infinite loop is terminated with the utilization of signal handling. When the wrapper receives `Ctrl^C` sequence, the packet reading process is interrupted, resources properly deallocated and wrapper exits with the exit code 0.

To provide or simulate the functionality of an intermediary device, the L2 header needs to be changed so that MAC addresses correspond to the predefined ones – the MAC of the intermediary device running RST Cookies as a source, and the MAC of the server or other network device with a path to the server as a destination. Simulation of Protector's packet queuing is done by allocating a buffer of static size and preparing Ethernet header by filling it with desired MAC addresses. Each call to the `packet_queue_get_data()` function then returns a pointer to the buffer right after the end of the pre-filled Ethernet header, just as the DDoS Protector does.

Alongside providing the interface and features the wrapped module needs, the wrapper also supports various informational and debugging outputs. As a consequence of the encapsulation, structures forming the module are hidden within its source files, hence the caller cannot access them. This is okay in most cases, but for the purpose of debugging, one needs to see the internal contents of the module, ideally without modification of its source files. This behavior was achieved by a hack, which redefined the module's internal structure under a different name, but with the same fields. This way, the internal structure of the module in the form of a void pointer is cast to a redefined structure pointer, and so the access to private module data is available. In addition to printing the internal state of the module, the wrapper is also able to provide a processed SYN/RST status reporting system (Figure 3.11). When the SYN or RST segment is processed, the message in the form of "`Source IP -> Destination IP : Action.`" is printed to the standard output.

```
147.229.182.8 -> 147.229.12.222 : SYN dropped.
147.229.182.8 -> 147.229.12.222 : RST dropped.
147.229.182.8 -> 147.229.12.222 : SYN forwarded.
2001:67c:220:0c:cf:fd29:aa9b:d96 -> 2a00:1e50:4017:80d::17ab : SYN dropped.
147.229.182.8 -> 147.229.12.222 : SYN forwarded.
147.229.182.8 -> 147.229.12.222 : SYN forwarded.
```

Figure 3.11: RST Cookies wrapper – SYN/RST status reporting.

Debugging outputs are controlled during the compilation phase. User aiming to receive debugging information needs to compile with `-DDEBUG` flag. Verbosity is controlled via macros `CONFIG_PRINT` (shows module internal data) and `STATS_PRINT` (periodically prints module statistics). These and other settings controlling the wrapper functionality are contained within the `rst_cookies_test.h` file.

Although a considerable part of the wrapper code is the same for both PCAP and NDP variants, they are implemented in separate source files. This is done mainly due to the readability of the code, which contains a large amount of `#ifdef ... #endif` compilation conditions already due to debugging outputs. Porting the wrapper from PCAP to NDP was not problematic at all since both APIs provide quite a similar way of network interface

handling and packet processing. The most significant difference between the two approaches is a style of how the received are data obtained. PCAP API provides a comfortable way by calling `pcap_loop()`, which blocks the application until the packet is received and calls a specified packet handler function when such an event occurs. On the other hand, the approach using the NDP API is not as straightforward. A packet is obtained via the `ndp_rx_burst_get()` call, which does not block but returns `NULL` when there are no data to be read. This behavior creates a problem because when the function is executed in a loop, it keeps repeating itself as fast as possible, consuming all the available processing power. The issue is solved with the `nanosleep()` call, used to put a thread into sleep before polling the hardware packet queue again. This approach saves CPU resources while having little to no impact on the packet processing speed.

The application created by wrapping the module represents a standalone software that provides the RST Cookies functionality for intermediate devices running the Linux operating system. The current state of the application accepts only one parameter – network device to listen and send traffic on. All other features are controlled with macros defined in the wrapper header file. These values could be easily parametrized, and so a flexible and scalable software solution could be created. However, the purpose of the wrapper is to provide a way to verify and debug the module, flexibility not being one of the primary concerns. The wrapper is currently able to simulate only intermediary device mode, but with little tweaks to both the wrapper and the host OS, the usage as a host-based network mitigation method could be supported as well. The functionality of this software solution as a whole, alongside several tests and experiments, is described in Section 3.4.

3.4 Results and Closing Remarks

The RST Cookies module undoubtedly provides a high level of security able to stop most of the SYN flooding attacks found on the computer networks nowadays. However, utilization of the session reset, hashing, and software packet processing may increase a delay in TCP communication or significantly decrease traffic throughput. This section will cover the validation of the module functionality as well as describe various tests used to reveal its impact on the overall network performance. For this purpose, the wrapper able to provide a standalone RST Cookies functionality, as described in Subsection 3.3.5, was used.

3.4.1 Compatibility

As one of the first steps of the RST Cookies method analysis was to ensure its compatibility with modern operating systems. A compatible OS is a system with a properly implemented TCP stack respecting the standard. More precisely, it always responds with an RST segment when an invalid SYN-ACK is received. Our tests with a prototype have proved, that all tested systems, namely *Windows XP*, *Windows* 7, *Windows* 8*(.*1*)*, *Windows* 10, *Linux kernels* 3, *Linux kernels* 4 (including *Android*), *FreeBSD* 11, *Apple iOS* 12, and *macOS* 10.14 are all compatible with the RST Cookies technique.

The testing was performed using a peer-to-peer network established between Fedora 28 and a virtual machine with a tested operating system (Figure 3.12). The Fedora host was running the RST Cookies application while having a port 80 opened. In the cases when the PCAP wrapper is used, the program is bound to the particular network interface in promiscuous mode, so all the received packets are forwarded to it before they are processed by the kernel. Passing the packet into the kernel would mean, that it could interfere with the

TCP communication, which is undesirable for our tests. To disable this behavior, a rule to drop all inbound TCP segments for the specified port during the `PREROUTING` stage needs to be applied. This rule is normally added to `iptables` or `firewalld` based on the host OS. When the NDP API is used, received packets are not forwarded to the kernel, so no additional settings are necessary.

Since the wrapper was implemented to simulate network-based mitigation, forwarding of allowed TCP SYNs and TCP RSTs to the other system processes was not achieved. For this reason, a web server on port 80 was not actually launched. The main goal was to make the tested operating system try to establish a TCP session, send an invalid SYN-ACK with the RST Cookies and examine how the client will react.

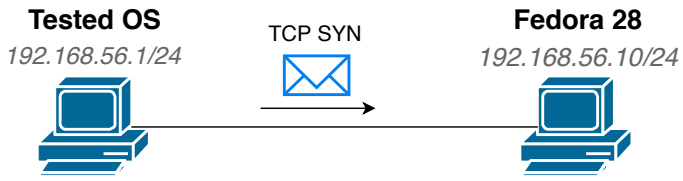


Figure 3.12: RST Cookies compatibility testing topology.

After entering the IP address of the Fedora machine to the second node's web-browser, it started by establishing a TCP session with a three-way-handshake process. As can be seen in Figure 3.13, the RST Cookies mechanism responded with an invalid SYN-ACK carrying *ACK* value of 0. Its receipt caused the tested OS to respond with an RST segment (data with the red background) which carried the same *SEQ* equal to 0. At this moment, the algorithm has processed the RST message and added the client's IP address to the whitelist. After this process, the mechanism did not respond with invalid SYN-ACKs to other TCP SYNs sent by the particular client IP address again. Since the web server is not launched, the client receives no response but keeps trying to establish a session without the intervention of the RST Cookies algorithm. This specific test was performed with all mentioned operating systems, which all behaved similarly. Based on these findings, we can conclude that all tested operating systems have their TCP stack implemented correctly and thus are compatible with the RST Cookies technique.

Source	Destination	Protocol	Length	Info
192.168.56.1	192.168.56.10	TCP	66	57341 → 80 [SYN] Seq=4278186261 Win=64240 Len=0 MSS=1460 WS=256
192.168.56.10	192.168.56.1	TCP	54	[TCP ACKed unseen segment] 80 → 57341 [SYN, ACK] Seq=999 Ack=0
192.168.56.1	192.168.56.10	TCP	60	57341 → 80 [RST] Seq=0 Win=0 Len=0
192.168.56.1	192.168.56.10	TCP	66	57342 → 80 [SYN] Seq=2560683974 Win=64240 Len=0 MSS=1460 WS=256
192.168.56.1	192.168.56.10	TCP	66	57341 → 80 [SYN] Seq=4278186261 Win=64240 Len=0 MSS=1460 WS=256
192.168.56.1	192.168.56.10	TCP	66	[TCP Retransmission] 57342 → 80 [SYN] Seq=2560683974 Win=64240
0a:00:27:00:00:09	PcsCompu_e1:5f:3b	ARP	60	Who has 192.168.56.10? Tell 192.168.56.1
PcsCompu_e1:5f:3b	0a:00:27:00:00:09	ARP	42	192.168.56.10 is at 08:00:27:e1:5f:3b
192.168.56.1	192.168.56.10	TCP	66	[TCP Retransmission] 57341 → 80 [SYN] Seq=4278186261 Win=64240
192.168.56.1	192.168.56.10	TCP	66	[TCP Retransmission] 57342 → 80 [SYN] Seq=2560683974 Win=64240

Figure 3.13: RST Cookies compatibility testing packet capture.

3.4.2 Reset Cookies in Practice

When the RST Cookies technique is deployed on a real network, different approaches to look on and evaluate the method can be taken. The algorithm behaves differently for clients and a protected network, but intermediary devices running the method need to be taken

into account as well. This subsection will analyze the behavior of the method from various perspectives, discussing the important aspects related to each of them in detail.

Protected Network's Perspective

When deployed on an intermediary device, the algorithm behaves transparently for all protected systems. Method blocks all received SYN segments until a security association is established. After this process, the TCP communication is not intervened anymore. As a result of this behavior, the protected server does not know about the client's intention to establish a session and thus does not need to allocate any state information. This way, all the devices in the protected network are not vulnerable to most of the SYN flooding attacks, while not requiring their configuration to be changed.

Client's Perspective

From the perspective of a client, the first attempt to establish a session always fails. As demonstrated in Subsection 3.4.1, this is not a problem in modern operating systems which send an RST segment and try to reestablish the session. However, the duration of the reestablishment process is dependent on the host OS. Our tests have measured this time to be roughly 250 ms on Apple systems, whereas Linux and Windows kernels tried to reestablish the session after approximately 1 second. According to these findings, we can conclude that the first connection through RST Cookies is delayed by up to 1 second, but all consecutive connections experience no significant delay (Figure 3.14, last two columns).

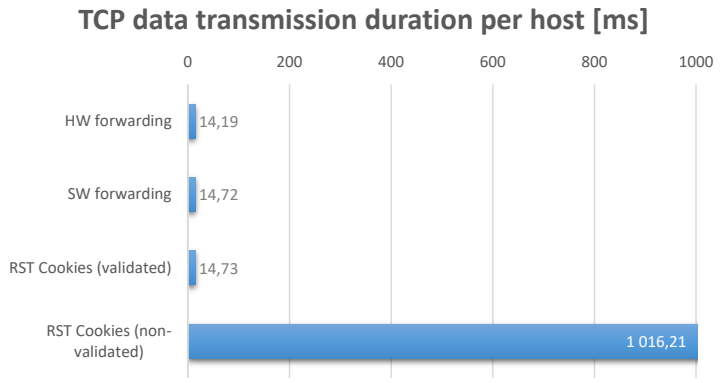


Figure 3.14: Transaction time performance comparison – Scientific Linux 7.4.

Another interesting result obtained from the analysis is a difference between the hardware and software processing speed. As may be seen, the HW forwarding value of 14.19ms is lower than SW forwarding, but the difference is not so drastic. This result convinces us that the hardware processing is still a preferred way, but the end user will most probably not notice the change since the difference is lower than 1ms. However, one needs to keep in mind that hardware is able to process several times more data, providing much favorable packet throughput when necessary.

Though measured delay of the first connection caused by the algorithm seems horrific, it is important to realize that the method should only be active during the ongoing attack. For example, the DDoS Protector can detect abnormal traffic and turn on the mitigation mechanisms when necessary. Therefore, no delays are caused during regular operation, and

when the method is active, a 1-second delay is definitely an acceptable trade-off for service availability during the attack.

Intermediary Device's Perspective

When considering an intermediary device running RST cookies, the most significant factors are memory requirements and packet throughput limitations. The usage of hash tables as whitelists plays a notable role in both of them. Every entry in the whitelist requires 8B of data for a timestamp and an additional 12B if SYN limiting is enabled. Considering 2^{20} whitelist rows, 4 clients per row, we obtain 4.2M client entries taking up to 80MB of memory. This result could be considered as a reasonable outcome. In the context of packet throughput, the most important metrics is the number of processed hash functions. Our implementation contains at least one hashing per TCP segment. When the hashed ACK mechanism is used, two hashes per SYN and up to five hashes per RST are needed.

R:S ratio	**Method throughput (Mfps)**		
	SW forwarded	RCks (window)	RCks (hash)
0	17.97	7.30	2.02
0.1	17.33	7.36	2.12
0.2	17.07	7.37	2.24
0.3	16.64	7.40	2.42
0.4	16.52	7.61	2.61
0.5	16.56	7.61	2.77
0.6	16.49	7.62	2.91
0.7	16.54	7.76	3.03
0.8	16.35	7.85	3.22
0.9	16.40	7.87	3.34
1.0	16.48	7.90	3.47

Table 3.1: Million of frames per second per thread throughput comparison. Based on RST : SYN segments traffic ratio. 1Mfps $\sim$ 680 Mbps.

Table 3.1 illustrates the RST Cookies algorithm frame processing ability during a simulated attack. In this case, *SEQ* values in RST segments were randomized. When the timestamp of the received *SEQ* does not fit into the specified time window, no hashing occurs, and the segment is straightly forwarded. That is why the actual module throughput was increasing as the ratio between RST and SYN segments was growing. In the real network, legitimate clients would send RSTs fitting into the time window, so at least one extra hashing would need to occur, and the actual throughput would decrease.

3.4.3 Limitations and Drawbacks

In addition to considerations discussed in the previous subsection, other aspects resulting from the method itself or our specific implementation should be examined. The following paragraphs present other factors that not are as significant as those mentioned previously, but need to be consulted for the sake of completeness.

Simulating real traffic

As mentioned previously, an attacker who utilizes real TCP stack or is somehow able to inject an expected RST segment into the session is granted the right to establish TCP connections. Of course, that this right is allowed only until the whitelist entry timeout ticks out (commonly dozens of seconds up to several minutes), but this gives the attacker enough time to perform an SYN flood anyway. Also, when the attacker was able to bypass the mechanism once, he probably will not have any problems in bypassing it again.

Our implementation provides partial protection against this phenomenon because each unique IP address is given a counter of how many SYN segments can it sent each second. This approach does not deny the attacker to flood SYN segments, but can considerably limit the amount of data he is able to send. Because RST Cookies requires clients to pass the security phase by responding, spoofed IP addresses cannot be used to perform the attack. For this reason, the attacker would require a large number of real computers (botnet) to successfully perform an SYN flood when the SYN counters are set sensibly. SYN flooding clients typically try to send as many SYNs as possible in most cases. If the RST Cookies with the SYN counter is used in conjunction with a blacklist, IP addresses of the SYN flooding computers can be temporarily cut off, thus mitigating the attack entirely.

Acknowledgment number match

Since the algorithm generates random SYN-ACK *ACK* values, a situation when the supposedly invalid SYN-ACK segment is accidentally valid may happen. This incident happens when the generated *ACK* value is exactly equal to the $SEQ + 1$ of the acknowledged SYN. In this case, the sender of the SYN will not generate RST but will try to finish session establishment by sending an ACK segment. This message will not be blocked by RST Cookies, and so it will be forwarded to the server that will generate RST according to RFC 793, section 3.4. [21]:

> *If the connection does not exist (CLOSED) then a reset is sent in response to any incoming segment except another reset.*

When the client receives an RST, it will try to reestablish the session with a new SYN having different *SEQ* number. Because the host is still not contained on the whitelist, the RST Cookies algorithm will generate another SYN-ACK with supposedly invalid *ACK*, and the process will continue as usual (Figure 3.15). This situation will cause one more connection reset, but the host will be eventually added to the whitelist, and its next attempts to establish a session will be successful. The probability of this phenomenon is $1/2^{32}$ while having no significant impact on the regular TCP operation.

The explained problem can be addressed by adding one more condition when generating the *ACK*s, but this would require extra processing to both SYN analyzer and RST analyzer routines. Because the probability of the event is low and there are no significant consequences when it happens, we decided not to implement this additional check to achieve as high segment processing rate as possible.

Whitelist Entry Deletion Before Expiration

As outlined back in Subsection 3.2.8, the proprietary implementation of the hash table used by the module allows only 4 entries per row. Because of this, the hash table entries are not getting chained, but the oldest one is replaced when the particular row is full, and

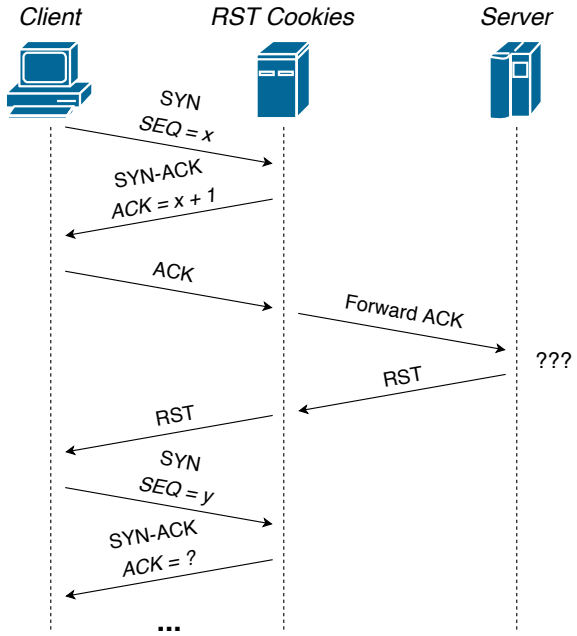


Figure 3.15: RST Cookies – accidental ACK match situation.

a new entry is being added. This behavior provides many useful features like entries aging and proper memory management but may cause an undesired behavior when the algorithm has to process more unique IP addresses than its available whitelist capacity. In these scenarios, legitimate clients that already passed security association may be removed from the whitelist sooner than defined timeout, requiring them to perform the association again. The process of resetting the session takes some time, therefore slowing the entire communication. In extreme cases, records may get replaced so often that effect of whitelist is almost nullified and clients have their SYN connections reset so often, that unacceptable delays or overall inability to establish a session may occur.

This behavior is, indeed not typical, though possible in situations when the chosen size of the hash table does not respect the properties of a protected network. Our implementation offers a statistics logging that provides information about the number of hosts that were whitelisted and the number of them that were removed before their supposed expiration time. These data may help to detect the discussed situation and help to adjust the hash table size appropriately, preventing the phenomenon from happening in the future.

3.5 Summary and Conclusions

This chapter has presented a network-based mitigation method TCP RST Cookies. The motivation behind its design and implementation was to provide an alternative way to mitigate TCP SYN Flood attacks in CESNET's DDoS protector. The method has proven to be especially effective against regular SYN floods from the spoofed IP addresses. Its capabilities were expanded with our custom security extensions like Dynamic *ACK* Generator, SYN Counter and blacklisting mechanisms. This enhanced version of the module is able to repel even most of the more sophisticated SYN floods used by the attackers nowadays.

Despite all the advantages the technique provides, it is not suitable in all cases. It is mainly due to the effect of directly impacting the clients with extra 1-second delay upon a new connection establishment. Other considerations include significant throughput decrease and relatively high memory requirements. According to these properties, the algorithm should be used alongside other mitigation methods with different attributes. The methods with a lesser impact on the network should be deployed against regular SYN floods, while RST Cookies should be used to defend against more sophisticated attacks that cannot be mitigated using other strategies.

The current approach towards the mitigation in DDoS Protector is based on rules. These specify which mitigation method should be used for the particular protected prefix. Defined mitigation techniques are used when the ongoing attack is present, but the implemented system does not allow the methods to switch, even if they are used ineffectively or their mechanisms are not able to mitigate the attack. This inflexible approach wastes resources of the intermediary device and requires a manual intervention of the administrator to change the mitigation strategy when required. For this reason, an SYN Flood Dynamic Mitigation Method Management system discussed in Chapter 4 is proposed.

Chapter 4

Dynamic Mitigation Method Management

SYN Flood Dynamic Mitigation Method Management mechanism proposed in this chapter is aimed to provide a flexible way to switch between different SYN Flood mitigation methods. The module was designed and implemented especially for the needs of CESNET's DDoS Protector, which currently supports three mitigation methods – *SYN Drop*, *ACK Spoofing*, and *RST Cookies*. However, the design of the module was aimed to be as general as possible, providing support for any number of mitigation methods and any environment it may be deployed in.

The main motivation behind the development of the technique was the need for DDoS Protector's adaptability to the different types of SYN Flood attacks. Currently used approach is based on rules, which specify mitigation settings for the particular protected prefix. One of these settings is a specific mitigation method, which is activated during the ongoing attack. Using this system, the same method is used every time the attack is triggered. Also, when the mitigation method needs to be switched, a rule has to be manually changed. This inflexible approach cannot effectively utilize system resources while its mitigation capabilities may be weakened as well. For this reason, a dynamic approach able to detect properties of the traffic and choose an appropriate mitigation algorithm is needed.

The beginning of the chapter provides an overview of the design and implementation concerns related to the developed dynamic management mechanism. Subsequent sections evaluate the approach, consider its usability on real networks, and suggest possible future enhancements.

4.1 Theoretical Concepts

This section describes important theoretical concepts that needed to be taken into account during the design and implementation phases. Since the design is closely tied to the theory in this case, the section may contain concepts the reader may find more suitable to be included in the design section and vice versa. The following subsections will take a look on the evaluation process of the used mitigation methods and TCP traffic and will explain a *HyperLogLog* probabilistic data structure, which was a suitable choice to be used by the mechanism.

4.1.1 Mitigation Method Evaluation

As outlined in Chapter 2, many different types of SYN flooding attacks exist, and their mitigation methods have different pros and cons. The purpose of the Dynamic Method Management mechanism is to differentiate these unique characteristics and choose the appropriate mitigation method to deflect an ongoing attack. For this reason, a generalization of different algorithm properties and the creation of the evaluation system for them was required.

This task proved to be rather problematic because common patterns in fundamentally different algorithms needed to be distinguished. For this purpose, we used inductive techniques to describe three currently available mitigation methods in detail and then tried to generalize their properties. According to the information from Section 2.3, a summary Table 4.1 was created.

	SYN Drop	**RST Cookies**	**ACK Spoofing**
Advantages	• Processes only 1 segment type • Low memory requirements per host • Effectively cuts high-rate SYN senders	• Repels more sophisticated SYN floods • Comprises SYN Drop functionality	• Minimum extra latency
Drawbacks	• Ineffective against large number of spoofed IPs with low footprint	• Higher memory requirements • Significant throughput decrease • First session establishment time	• Does not mitigate the attack itself • Amplifies high-rate attacks • Cannot identify high-rate SYN senders
Processes	• ingress SYNs	• ingress SYNs • ingress RSTs	• ingress SYNs • ingress ACKs • egress SYN-ACKs

Table 4.1: Available methods analysis in DDoS Protector.

By examining these data, the induction process could be started. Every mitigation method is defined by a set of ingress and egress TCP segment types it processes. The performance of the method is also defined by its memory and CPU requirements, where the number of hash functions plays the most significant role. These functions should be furthermore divided into regular hashes (hash table access) and cryptographic hashes (RST Cookies), which both have different CPU requirements. All of the mitigation methods need to retain state information, usage of hash tables being a traditional choice. For this reason, the algorithm should keep a track about how these hash tables are filled and act if there is a chance they may get overfilled. Other specific aspects of methods such as whether it creates an SYN retransmit, causes a session reset or generates traffic need to be reckoned with as well.

The information described in this subsection were the essential building blocks of the mitigation method evaluation process described in 4.2.2.

4.1.2 Traffic Evaluation

As mentioned in the previous subsection, different mitigation algorithms process different types of TCP segments. Data that are not processed by the software can be hardware-forwarded. This process is much faster, allowing better packet throughput and lower TCP communication delays. For this reason, methods that process fewer packets provide generally better performance.

When the optimal mitigation method is being chosen, its performance plays a crucial part. Therefore, choosing an ideal method for the given situation requires information about how the processed types of traffic actually impact the overall network performance. However, no such data are publicly available, so own traffic analysis result had to be conducted at first.

In particular, the aim of this research was to determine the proportions of TCP segment types used on regular networks. Since this thesis is a part of the CESNET's security research project, we managed to obtain captured data from the communication link between CESNET and ACONET[1], two national research and academic networks. The analyzed file had 367GB, containing over 500M packets. Its contents represented common network traffic captured on 14th November 2018. After filtering out the TCP communication only, a sample of 450M segments was obtained. These data were analyzed further with the `tcpdump` Linux utility. This tool was used to strip and count the segments with various combinations of TCP flags.

Table 4.2 shows the results of the performed analysis, displaying most common TCP flags and their combinations that may be interesting for the module performance evaluation. According to the results, SYN segments, analyzed by all modules take only 0.89% of all the traffic, making potential performance degradations caused by SYN analyzer modules rather insignificant. On the other hand, ACK segments and their combination are the most prevalent type of TCP network traffic. Other non-standard combinations or individual flags with an irrelevant number of entries (FIN, URG, PSH) are listed under others, which make up 3.91% of total analyzed traffic.

TCP flags	**Segment count**	**Ratio [%]**
all	450 649 793	100.00
ACK	365 496 097	81.10
PSH + ACK	56 582 935	12.56
FIN + ACK	4 388 605	0.97
SYN	4 024 870	0.89
SYN + ACK	1 833 904	0.41
RST	601 030	0.13
ECE + CWR	97 878	0.02
others	17 624 474	3.91

Table 4.2: CESNET ⟷ ACONET link traffic analysis.

[1] https://www.aco.net/

Traffic analysis results presented in this section provided an overview of the ratios of different TCP segments types used in modern computer networks. Although these results come from only one dataset and may partially vary depending on the network properties, it can be assumed that segment ratios captured in other networks would not be drastically different. Analysis of this data indicates that software-processing SYN segments and alternatively RSTs, does not have a high impact on the overall TCP communication. However, ACK segments and their combination represent the majority of all TCP communication, so algorithms analyzing them may cause a significant traffic performance decrease.

4.1.3 Probabilistic Data Structures Utilization

Probabilistic data structures, more precisely *LogLog* and its enhanced variant *HyperLogLog* were chosen as suitable alternatives for counting the number of unique IP addresses processed by the Dynamic Method Management algorithm. These considerations are described later in the chapter. The following subsection focuses on the description of the HyperLogLog algorithm, its functionality, and the consequences of its utilization in method manager software.

HyperLogLog is an algorithm able to approximate the number of distinct elements (cardinality) in the multiset, providing a solution for a count-distinct problem [12]. A standard way to calculate exact cardinality requires an unacceptable amount of memory for large data sets. For this reason, probabilistic cardinality estimators, such as HyperLogLog can estimate this value demanding significantly lower memory. For example, cardinalities of 10^9 elements or lesser can be calculated using 1.5 kB memory with only 2% error [12].

The fundamental idea behind the algorithm is based on the observation that the cardinality of the multiset of uniformly distributed values can be estimated by calculating the maximum number of leading zeros of each number in a set. Simulation of the uniform distribution is achieved by hashing each element and logging its result to one of the multiset subsets (buckets). Estimate of distinct elements is then calculated as 2^N, where N is a harmonic mean of the maximum values of observed leading zeros of each subset [12].

For our purpose, a multiset can be considered a number of source addresses, and the goal is to determine the number of unique hosts. These statistics may then be used to trigger various mitigation techniques as described in 4.2.4 or simply used for statistics logging. However, a caller utilizing the algorithm needs to realize that obtained results are not precise, but rather within the range of the standard error based on the algorithm properties. The standard error is defined by the number of subsets used to store counting information. Thus, more available memory for the algorithm allows the creation of more buckets, resulting in lower error.

The use case of Dynamic Mitigation Management algorithm does not require extremely low error rates. However, errors above 10% may produce undesired results by activating triggers too early or too late, effectively weakening the mitigation abilities. For this reason, the caller should always consider properties of the protected network (number of possible unique IP addresses, etc.) and choose the size of the HyperLogLog structure appropriately. The idea is to obtain the best memory to standard error ratio as possible, but also to keep the error rate low, ideally not exceeding 5% – 10%.

4.2 Mechanism Design

Similarly to the RST Cookies, Dynamic Method Management algorithm is also supposed to be a part of the CESNET's DDoS Protector in the future. For this reason, the same constraints as described in Section 3.2 need to be respected. The mechanism has to be designed without an object-oriented approach, providing flexibility and expected behavior of the module.

The primary concern of the algorithm is to provide a way of dynamic mitigation method switching based on the current mitigation effectiveness, statistics and system resources. Considering these requirements, the design process may be started. The following subsection discusses the design aspects related to integration of the module to the main DDoS Protector application. Subsequently, the 3 main logical parts of the module – fitness, traffic analysis and decision-making cores are described in detail.

4.2.1 Dynamic Method Manager as a Module

Like all algorithms that aim to extend DDoS Protector's core, the Dynamic Method Management technique needs to be designed modularly, providing an ability to be initialized, finalized and cleared. After the initialization process is completed, the caller has to inform the module of the available mitigation algorithms. After all desired mitigation methods are registered, the module may receive TCP segments in order to generate statistics of the current TCP traffic. The mitigation method is then switched according to these statistics and discovered traffic patterns.

The required functionality may be achieved by the following two approaches:

- Tightly integrated module (Figure 4.1)
- Loosely integrated module (Figure 4.2)

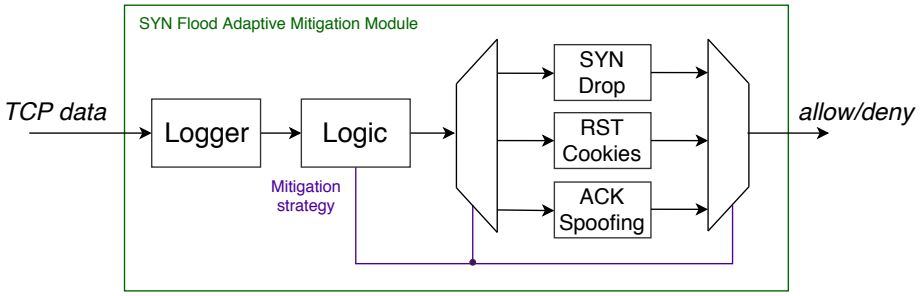


Figure 4.1: SYN Flood Adaptive Mitigation module (tight integration).

As shown in Figure 4.1, the tightly integrated version of the module would comprise all existing mitigation modules and be solely responsible for their management. This principle would allow simple usage since the Protector would only pass the TCP data into the function and straightly receive an answer if the packet should be forwarded or dropped. The mentioned approach is easier to use and provides better performance. On the other hand, its low flexibility, the difficulty of implementation and integration make it a quite unfavorable choice.

Loosely integrated version (Figure 4.2) is designed as an autonomous module, which processes TCP traffic, logs it and waits until a request to determine the best available option

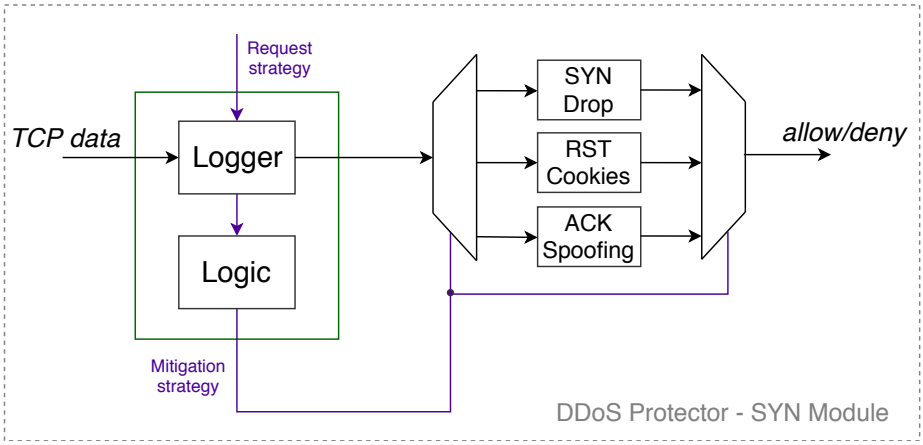


Figure 4.2: SYN Flood Dynamic Management

is issued. This request causes the algorithm to fetch logged information and search for traffic patterns. Found patterns are processed by various predefined rules and thresholds, and an optimal mitigation method is suggested. Still, it is important to realize that the module does not actually make a mitigation method switch, but rather only suggests which method should be used. This suggestion may be accepted or ignored by the caller. The Dynamic Method Management module is thus designed independently of all the mitigation methods. In contrast to tight integration, the mechanism does not manage the mitigation methods itself, but their management is purely dependent on the caller. This approach provides undoubtedly better flexibility, easier integration, and overall cleaner design. However, more responsibilities are left to the DDoS Protector, making the module harder to use. The nature of the approach also provides slightly lower performance, because data are processed by more routines and more function calls are required.

After considering all the pros and cons of both mechanisms, the loose integration variant was chosen as a more suitable approach with respect to future works and improvements of the module and DDoS Protector solution. This means that a loosely integrated module will be easier to maintain, extend and debug. Also, its presence will not require the existing code of the Protector to be changed that rapidly as in the case of tight integration.

4.2.2 The Fitness Core

As mentioned at the beginning of this section, the module is composed of three main logical parts. This subsection describes the fitness core, a set of procedures aimed to evaluate and store the information about available mitigation methods. Its name comes from the purpose of calculating a rating of the method, thus determining how "fit" the method is. These data are then used in decision-making core (Subsection 4.2.4) during the process of optimal mitigation method estimation. This subsection will describe how the mitigation methods are handled and evaluated.

Mitigation Methods Management

According to the loose integration design, the Dynamic Method Manager is not dependent on any mitigation modules. In regard to this system, the methods that can be used during the algorithm decision-making phase need to be specified. Module management is done

via registration, update and unregistration routines. The registration function informs the algorithm of the method's availability. The registration is done by specifying technique ID and its properties (Figure 4.3). Data of the mitigation method property structure correspond to the information obtained via induction in Subsection 4.1.1. These data are crucial for the method evaluation process described further in this subsection.

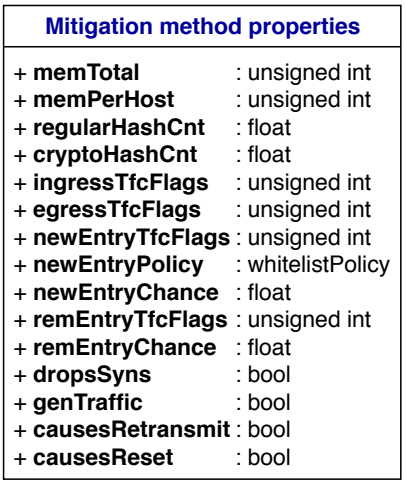


Figure 4.3: Mitigation method property structure contents.

Meaning of some fields was already explained in the regarding theoretical section. The number of hash functions `regularHashCnt` and `cryptoHashCnt` is proposed to be stored as a floating number. This is because some data processing functions may require several hashes, while others may not require hashing at all. For this reason, the user defining the properties of the mechanism is supposed to estimate the average number of hashes per processed segment. Fields containing `TfcFlags` specify types of TCP segments that cause certain events. *Ingress* and *egress* define TCP segment types that the algorithm analyzes on input or output from the protected network. Entries `newEntryTfcFlags` and `remEntryTfcFlags` specify which TCP segment types cause new entry to be added or removed from the internal data structures. These are complemented with `newEntryChance` and `remEntryChance` specifying the chance of the insertion or removal event. Some mitigation methods may fill their internal structures per every received segment, some per every unique IP address and some may not use whitelisting principles at all. For this purpose, the `newEntryPolicy` field specifying this behavior exists. Other fields determine if the method generates traffic, drops SYNs, etc.

Registration of the method is done by filling the previously described data structure and calling a corresponding function. When the properties of the method change, an update routine may be used. If the mitigation method is no longer desired to be used, the module should provide a way to unregister it, so it is no longer available during the decision-making algorithm execution.

Mitigation Method Evaluation

The mitigation method evaluation process is triggered every time a mitigation module is registered or updated. The purpose of the evaluation is to obtain an approximate quality of the method's performance. These data are then used to choose the most suitable mit-

igation method based on the analyzed traffic. The overall rating of the function consists of three components:

- Entries growth indicator
- Throughput limitation indicator
- Latency limitation indicator

Each of the indicators is impacted by its corresponding method properties. In some cases, a single attribute may impact more ratings, e.g., the number of calculated hashes impacts the throughput the most, but it may cause small additional latencies as well.

The *entries growth* indicator represents how the mitigation method's internal structures (typically whitelists) are likely to get filled. This value defines the ability of the method to process a certain amount of data before its whitelists capacity is reached. This rating is mostly influenced by `newEntry*` and `remEntry*` fields.

The *throughput limitation* indicator defines how the utilization of the method impacts the packet throughput of the overall system. Its value is determined by the number of hashes computed per segment, processed TCP data types (`ingressTfcFlags` and `egressTfcFlags`) and traffic generation predicate.

The *latency limitation* indicator reflects how the utilization of the method impacts endhosts, whose only concern is how fast the communication through the active mitigation method will be. Computing an excessive amount of hash functions may impact this rating, but `causesReset`, `causesRetransmit` and `generatesTraffic` predicates have a more significant impact on this indicator.

The indicators are then combined to form a final rating of the method. The process of combining is done by a weighted sum, each of the indicators having a different weight. We decided that the latency indicator, representing a direct impact on the clients, will have the biggest weight of 2.5. Another important indicator is throughput with the weight equal to 1.5. Entries growth indicator is assigned the weight of 1.0. These values were set experimentally and are probable to be changed in the future.

According to the current design, the *lowest* calculated rating represents the mitigation method with the best performance, allowing the highest traffic throughput and lowest delays. As mentioned previously, this value is used to determine the performance of the method, rather than the actual mitigation strength.

The previous paragraphs have mentioned the calculation process, but haven't discussed what is actually calculated and where do the values come from. All of the variables used in indicators and ratings computations were given values used as weights. Thus, the indicator computation consists of multiplying values from the method property structure with their corresponding weights and then creating a sum of these values to form an indicator. Most of the weights are currently experimental, but the data defining TCP segments types ratios were taken directly from the traffic analysis research included in Subsection 4.1.2.

4.2.3 The Traffic Analysis Core

The traffic analysis core (logger) is a set of routines and data structures used for traffic logging, sampling and statistics gathering. This information is utilized for attack pattern searching and optimal mitigation method estimation. The module accepts a TCP segment, furthermore parsing it and extracting data like IP addresses and TCP flags. TCP flags

are used to count the occurrence of different data types on the network using SYN, ACK and RST counters. Other interesting entries include the numbers of unique IP addresses sending these types of segments. As outlined in Subsection 4.1.3, usage of the standard data structures would be ineffective. Therefore, three HyperLogLog instances to track the number of unique IP addresses of TCP SYN, ACK and RST segments, are also employed.

However, some information that may be helpful for the decision-making process cannot be obtained directly by the module itself. These comprise the number of allowed and denied SYNs and whether the attack is currently ongoing. Information about the active mitigation approach needs to be supplied as well, because the module only suggests the particular method, but receives no acknowledgment from the caller whether it was actually applied. According to these facts, the caller should specify this information manually. This process provides a necessary context that could be used by the optimal mitigation method estimator. By putting these things together, a statistics structure shown in Figure 4.4 is obtained.

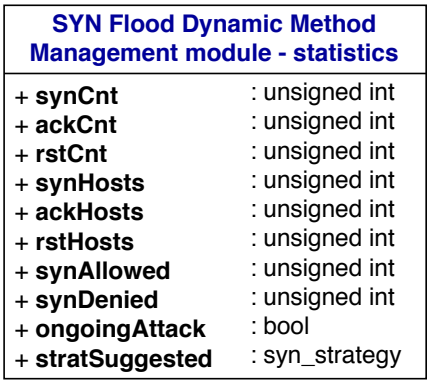


Figure 4.4: Dynamic Module Management – Statistics structure.

An important concept of the statistics logging and module functionality is time windowing. A time window is a period between the start time and the end time. For the purpose of this module, windows are used to divide time into periodic blocks and save collected statistics in them. Statistics are thus not collected per whole module lifespan, but rather per each window. This approach allows to view changes in time, providing various traffic patterns analysis options, creation of traffic logs, charts, etc.

Because the module does not provide a mechanism to track the time by itself, the responsibility of keeping time windows synchronized is left to the caller. The calling program is therefore supposed to periodically invoke a particular function of the module, which creates a new time window and rotates the logs. Periodical time windowing is thus crucial to preserve the correct module functionality. Small deviations from the period are acceptable, but longer may cause inefficient method suggestions or even overall inability to provide an appropriate mitigation strategy. The situation, of time windows desynchronization, may occur, and so a function to invalidate all logs in the history is also provided.

4.2.4 The Decision-making Core

As indicated in the previous subsections, the decision-making core is responsible for matching discovered traffic patterns to suitable mitigation methods. This process is done by look-

ing at the statistics provided by the traffic analyzer core and choosing a mitigation technique according to its properties and ratings obtained from the fitness core. The decision-making process is launched by a direct call from the user. It is important to note that the decision-making process works primarily with the current time window, so the request for mitigation method should be issued just before the time window ends. Statistics in the previous time windows (history) is taken into the account as well, but when the current time window contains no data, the algorithm may struggle to provide a reasonable outcome.

The process of estimating the most suitable mitigation method is composed of 2 independent phases. The first phase – *Action determination* is used to find out how the module should react based on the traffic statistics and current mitigation state. Actions returned by this phase (`NONE`, `MITIGATE`, `LOWER_MITIGATION`) determine the action for the second phase – *Strategy determination*. This phase returns a particular mitigation strategy based on the received action while respecting the current state of the module and traffic.

Both phases are controlled by a set of thresholds, which trigger a respective event when exceeded. Each threshold represents a certain traffic pattern. The current threshold values (patterns) and the weights of individual statistic entries are often set experimentally, but are expected to be changed as more data about the attacks will be collected in the future. The threshold triggers may be too sensitive in some cases, which may lead that the method switch is suggested unacceptably often. To tackle this issue, the module defines four switch policies that control how significant the pattern needs to be before the threshold corresponding to is triggered. One of the switch policies is even able to disable the method switching mechanism completely, that the call to determine strategy will always return the same mitigation method.

As outlined in the traffic analysis core subsection (4.2.3), some statistics cannot be obtained by the module but have to be specified manually by the user. Processing of some thresholds often requires these statistics to be present, so the user is advised to provide them right before the request for the mitigation method is issued. On the other hand, other thresholds are based on history, so they can never be triggered if statistics from previous time windows are not present. The mechanisms of the algorithm try to predict these situations and try to utilize types of thresholds that are available based on the data currently available. For this reason, the algorithm may provide reasonable suggestions even if the user did not specify statistics manually. However, relying on this system and intentionally omitting the manually inputted statistics may significantly decrease the mitigation capabilities of the module.

Event type	**History needed**	**Stats needed**
SYN/ACK unique host ratio > SYN/ACK ratio threshold	false	false
Ongoing attack (manual)	false	true
SYN allowed > SYN threshold	false	true
SYN allowed > SYN threshold (lowered) *AND* weighted history SYN sum > SYN threshold (history modifier)	true	true

Table 4.3: Thresholds triggering mitigation example.

For the illustration purposes, few of the currently active thresholds used to determine next action are included in Table 4.3. As can be seen, the user is able to specify that the

ongoing attack is in process manually, and thus the algorithm will provide a mitigation method without further pattern searching.

The strategy determination phase aims to provide the most efficient mitigation method with the lowest impact on performance. For this reason, the registered mitigation methods are ordered by the rating determined by the fitness core. At the beginning of the attack, the algorithm tries to return the method with the lowest impact on performance. If this method proves to be unable to mitigate the attack in future time windows, the algorithm searches for traffic patterns and tries to suggest the method with possible better mitigation capabilities, but also higher requirements on the performance. This way, the algorithm tries to gradually increase the mitigation capabilities for the higher impact on the performance. However, method switching is not always gradual. The algorithm tries to predict the future effectiveness of the method by looking at its properties and evaluating them with current patterns found in the traffic. If the utilization of the technique would not be sufficient to mitigate the ongoing attack, the position of the strategy is skipped and the process of determining future effectiveness is applied to another method in the list.

Using the mentioned approaches, the decision-making core is able to recognize different traffic patterns according to thresholds. Triggering a threshold value determines an action which should be taken. If no threshold is reached or an insufficient amount of data is collected, a `NO_ACTION` is issued. This special value tells the algorithm to suggest the exact same mitigation method that is currently active. Other actions determine if the algorithm needs to suggest a method to mitigate, lower the currently used mitigation method or inform that no mitigation method is necessary.

4.2.5 Module Wrapper

As in the case of RST Cookies (Subsection 3.2.9), a wrapper software needed to be designed to test and debug the Dynamic Method Management module as a standalone program. The design of the wrapper is mostly the same as in the mentioned subsection. The wrapper needs to listen to all network traffic, filter TCP segments and pass them to the module. However, before entering an infinite packet-reading loop, the wrapper has to register the available SYN Flood mitigation methods. Since DDoS Protector currently supports 3 methods – *SYN Drop*, *RST Cookies* and *ACK Spoofing*, properties from all of them were gathered and these 3 methods are used in the wrapper for testing purposes.

After the methods are registered, the wrapper also needs to provide a way to periodically call the module's function to mark the beginning of new time windows. From time to time, the wrapper will also ask for the mitigation method suggestion. Since the wrapper will not comprise mitigation modules itself, the result of this suggestion will be written to standard output and the wrapper will simulate that the switch truly happened and the suggested mitigation method is now being used. Since the mitigation module is not actually used, statistics of the number of allowed and discarded SYN segments need to be set manually or randomized.

The usage of this wrapper should be able to provide enough information to test the module and eventually even tweak threshold values when needed.

4.3 Implementation

Similarly to RST Cookies (3.3), the used implementation language was *C*. OO functionalities could not be applied, but the module simulated encapsulation principles, hiding the

internal structure into the source file and letting the caller work with `void` pointers instead. Based on the good quality of the initial design, only minimum number of adjustments were need to be made, thus developing the module in almost waterfall model principles. This section describes implementation considerations related to each of the cores and the wrapper.

4.3.1 The Fitness Core

The module management and evaluation is provided by the fitness core, which is represented by the `synf_dmgmt_register()`, `synf_dmgmt_update()` and `synf_dmgmt_unregister()` calls. Functionality of these functions was mostly described in back in the core design phase (Section 4.2.2).

Since the mitigation methods are ordered by their rating, from lowest to highest, an insertion or a move needs to be made to reorder the methods when necessary. This could be addressed by the array of pointers, which would provide a fast and flexible way or rearranging any number of mitigation methods. However, the quantity of registered methods is typically not high and reordering them is not a standard operation, so we decided to do reordering directly with the memory storing method properties, so no extra pointer array needs to be maintained. The reordering is thus done via `memmove()` call because of the moved methods represented by overlapping memory blocks in most cases.

4.3.2 The Traffic Analysis Core

Traffic statistics creation and storing is handled by the traffic analysis core, which provides five external functions:

- `synf_dmgmt_data_process()`
- `synf_dmgmt_start_new_window()`
- `synf_dmgmt_set_syn_stats()`
- `synf_dmgmt_get_stats()`
- `synf_dmgmt_clear_history()`

The first function is used to process (log) the data. The function takes a pointer to IP and TCP headers and updates the statistic counters described in Figure 4.4. Each processed segment updates its respective counter type (`synCnt`, `ackCnt` or `rstCnt`) and is then passed to the HyperLogLog (HLL) module corresponding to its segment type. The unique IP counters are not updated straightly but at the end of the time window during which the values from HLL structures are fetched. The concrete HLL implementation was obtained from the Github repository of the user *avz*[2].

A new time window is started by `synf_dmgmt_start_new_window()` call. This function causes the HyperLogLog data to be evaluated and stored into the current time window statistics structure. HLL structures are then cleared, preparing it to count new time window and logs are shifted.

Function `synf_dmgmt_set_syn_stats()` sets the SYN mitigation statistics (`synAllowed`), `synDenied` and `ongoingAttack`). Since these stats cannot be obtained by the module, the

[2] https://github.com/avz/hll

caller needs to set them manually, ideally just before the time window ends. These data provides a valuable information for the decision-making process.

Stats from the current time window or a window in the history can be obtained with `synf_dmgmt_get_stats()` call. This function accepts a parameter specifying which time window should be returned from the current time window back to the history. For example, 0 represents a current window, 1 a window before the current window, etc. The function returns a pointer to the desired statistics structure.

The `synf_dmgmt_clear_history()` call is typically used when the time windows get desynchronized for some reason, and usage of the history would cause the method to provide incorrect suggestions.

4.3.3 The Decision-making Core

Functions related to the decision-making core include following:

- `synf_dmgmt_determine_strategy()`
- `synf_dmgmt_set_current_strategy()`
- `synf_dmgmt_set_switch_policy()`
- `synf_dmgmt_get_switch_policy()`

According to the design of the module discussed earlier, the algorithm does not know whether the suggestions it gives are actually taken into account or not. For this reason, the caller has to inform the module about currently active mitigation strategy using `synf_dmgmt_set_current_strategy()`. This allows the decision-making algorithm to determine the effectiveness of the currently used mitigation approach and decide accordingly.

Switch policy discussed in the design section defines the value of different thresholds that are used to trigger various events in the internal mechanism logic. Values of the switch policies – `SWITCH_ALWAYS`, `SWITCH_SMARTLY`, `SWITCH_SPARINGLY`, and `SWITCH_NONE` are defined. "always switching" mode provides the lowest thresholds that are easier to trigger, while "none switching defines triggers that are impossible to trigger. The logic behind this approach is that when no threshold is reached, no action is triggered and thus the module does return currently used mitigation method set by `set_current_strategy()` call. This system provides better performance because frequent method switching takes memory resources on allocation/deallocation calls at the host lesser flexibility provided by the module.

All functions presented so far were supportive routines aiming to provide the necessary environment for the main routine providing the functionality the whole module needed to – `synf_dmgmt_determine_strategy()`. At the beginning, the functions checks if module switch policy is not set to none. If that is true, the function straightly returns the currently used mitigation method. Otherwise it firstly determines an action to be taken and then chooses an appropriate mitigation method according to it.

The action is chosen by processing a number of `IF` conditions representing various thresholds. Triggering a threshold is thus execution of the condition body if its evaluation is true. The mitigation strategy is chosen from the internal structure of the ordered registered strategies in a way that the mechanism starts at the method with the lowest rating (index 0) and evaluates the method properties according to the available traffic statistics. If the properties of the strategy on the current index indicate that it may be able to mitigate

the ongoing attack, the ID of the strategy is returned. Otherwise, the internal structure index is incremented and the process is repeated. This way, the algorithm ensures that the method with the best performance is always active.

4.3.4 Wrapper

The implementation of the wrapper was very similar as in case of the RST Cookies. The wrapper needed to provide an environment simulating the DDoS Protector. In this case, no special functions were needed, only the mitigation method identifiers were included in the separate header file. The main wrapper function initialized and bound the interface, either using PCAP or NDP API, entered an infinite packet loop and waited for packets. Instead of the RST Cookies, the received data were parsed using Linux networking headers included from netinet header files. This change was made due to internal politics of the DDoS Protector, which forced the usage of Linux headers. Keeping the time windows synchronized is done by `alarm()` call.

Similarly to RST Cookies wrapper, the Dynamic Method Management wrapper also offers a way of debugging with the multiple levels of verbosity. These can be changed in the corresponding header file, which contains other settings like SYN attack threshold or time window duration. These values may be changed for the purpose of experimentations.

4.4 Testing

This section describes the testing process that was taken to verify the functionality of the module and determine its mitigation capabilities. The following subsections describe the phase of the environment and test preparation, as well as the process of various tests that the Dynamic Mitigation Management module had taken.

Environment and Tests Preparation

The testing of the module was done with the prototype described in previous sections. NDP variant of the wrapper was used to test the behavior under CESNET's NDP environment. This way, a specialized software (Spirent TestCenter[3]) to generate forged packets at the rate 100 Gbps could be used. This environment allowed the simulation on real network packet rates with the advanced options of packet analysis.

For the sake of simplicity, the wrapper was set to generate new time window every 10 seconds and the SYN Flood attack threshold was set to 100 000. Thus, when the 100 000 TCP SYN segments were processed in a 10-second time window, a Dynamic Method Management module would detect an ongoing attack and provide appropriate mitigation. Note that an active attack can be set also manually by the caller. For the purpose of dynamic method switching, three mitigation algorithms available in the DDoS Protector were registered for the module to use. Properties of these used algorithm that were used during the testing are shown in Figure 4.5. All mitigation methods were simulated to have 2^{20} whitelist rows, supporting up to 4.2M clients. Float values (hash counts, new/remove entry chance) in all three cases were calculated using weighted sums according to the ratios of the data as mentioned in Table 3.1. However, these value are experimental and are likely to be changed when the module will be integrated to the real DDoS Protector solution.

[3] https://www.spirent.com/products/testcenter

SYN Drop properties	
+ **memTotal**	: 20 971 520
+ **memPerHost**	: 5
+ **regularHashCnt**	: 1.0
+ **cryptoHashCnt**	: 0.0
+ **ingressTfcFlags**	: SYN
+ **egressTfcFlags**	: NONE
+ **newEntryTfcFlags**	: SYN
+ **newEntryPolicy**	: PER_CLIENT
+ **newEntryChance**	: 0.1
+ **remEntryTfcFlags**	: NONE
+ **remEntryChance**	: 0.0
+ **dropsSyns**	: true
+ **genTraffic**	: false
+ **causesRetransmit**	: true
+ **causesReset**	: false

RST Cookies properties	
+ **memTotal**	: 83 886 080
+ **memPerHost**	: 20
+ **regularHashCnt**	: 1.18
+ **cryptoHashCnt**	: 0.0357
+ **ingressTfcFlags**	: SYN, RST
+ **egressTfcFlags**	: NONE
+ **newEntryTfcFlags**	: RST
+ **newEntryPolicy**	: PER_CLIENT
+ **newEntryChance**	: 0.9
+ **remEntryTfcFlags**	: NONE
+ **remEntryChance**	: 0.0
+ **dropsSyns**	: true
+ **genTraffic**	: true
+ **causesRetransmit**	: true
+ **causesReset**	: true

ACK Spoofing properties	
+ **memTotal**	: 4 194 304
+ **memPerHost**	: 4
+ **regularHashCnt**	: 1.0
+ **cryptoHashCnt**	: 0.0
+ **ingressTfcFlags**	: SYN, ACK
+ **egressTfcFlags**	: SYNACK
+ **newEntryTfcFlags**	: RST
+ **newEntryPolicy**	: PER_CONN
+ **newEntryChance**	: 1.0
+ **remEntryTfcFlags**	: ACK
+ **remEntryChance**	: 0.025
+ **dropsSyns**	: false
+ **genTraffic**	: true
+ **causesRetransmit**	: false
+ **causesReset**	: false

Figure 4.5: Dynamic Method Management testing – methods properties.

4.4.1 Method evaluation results

After the properties of the methods have been set, the method evaluation process could be started. Since we want to analyze how the methods are actually evaluated, we wanted to keep an eye on the internal structure storing mitigation methods. More importantly, we wanted to know the rating of these methods and their position in the list. For this purpose, the wrapper was compiled and run `DEBUG` and `INTERNAL_PRINT` macros active. After running the program, the result as shown in Figure 4.6.

```
Registered methods order:
- 0. --> 1 Rating: 127.013626
- 1. --> 3 Rating: 149.100006
- 2. --> 2 Rating: 491.927094
```

Figure 4.6: RST Cookies wrapper – SYN/RST status reporting.

The method with the number 1 represents an SYN Drop module, which was evaluated as the best because it processes only SYN segments and does not have that drastic impact on the traffic. The first index was taken by the RST Cookies method. The third, but the shocking result was ACK spoofing, which was rated very high due to our previous traffic analysis discovering that 81% of the TCP traffic consists of ACK segments. This indicator played a huge part in the overall method rating.

4.4.2 Method suggestion results

When all the mitigation methods were ready, the process of generating the TCP traffic and monitoring output of the module could be started. Firstly, we generated traffic of 5 000 SYN segments per second (half of the threshold). Since the specified SYN threshold has not been reached, a request for the mitigation method suggestion returned `NO_STRATEGY`, signalizing that no mitigation is needed. When the ongoing attack was manually specified, the method returned `SYN_DROP` method as the best match, because it is stored at the first index of the internal mitigation methods list.

Without providing additional statistics about the mitigation efficiency, the method keeps returning `SYN_DROP` value because it detects no required change. When the manual ongoing attack switch is removed, the function starts to suggest to not mitigate, but not suddenly.

This happens because there is a mechanism that keeps track of the past mitigations and so if the mitigation was active past N time windows, it will be activated again in this version. However, this condition cannot be used on its own, because mitigations in the past would always trigger mitigations in the current time window, and so it may get "looped" and would return a mitigation method infinitely, even if no other patterns would be present. Because of this, the past mitigation threshold is also combined with various others (like SYN counter, unique IP count, etc.).

Considering an active attack cause by triggering SYN threshold without further information, the module would always suggest `SYN_DROP` to be used. However, if the caller specifies efficiency of the mitigation by specifying a high number of allowed SYN segments above the threshold while keeping the dropped SYN segments value low, the algorithm will consider the currently used mitigation method to be ineffective and will try to suggest another in the list. In our case, the next method is *RST Cookies*, which would satisfy the requirements to be chosen and so the next call to determine strategy would return it. Following this pattern, if the caller specifies that the RST Cookies is ineffective, a mechanism would try to employ other mitigation methods than currently active. SYN Drop was marked as ineffective in the past, so the algorithm would temporarily skip it and proceed on the third position of the internal structure, where *ACK Spoofing resides*. However, this method would not pass satisfy the checks and so would not be chosen. More precisely, when the SYN threshold is exceeded and the tested method does not drop segments, it is never chosen because it can not reduce the number of segments on the already congested link.

According to this phenomenon, the *ACK Spoofing* method would always never be chosen, because most of the SYN attacks tend to exceed the threshold. This behavior is caused by bad rating estimation of the mitigation method, which was placed too high in the method hierarchy (according to the ACK segments weight). For this reason, the chosen experimental weights are probably set a bit inappropriately. On the other hand, as mentioned in Subsection 2.3.6, the ACK spoofing is not that effective anyway, so the algorithm may just actually work perfectly.

4.5 Mechanism Conclusion and Closing Remarks

This chapter has presented a mechanism able to dynamically switch between different SYN Flooding mitigation methods. Switching is based on various aspects like traffic, system resources, and mitigation efficiency. The method can be currently classified more as a theoretical concept rather than a usable module. However, many aspects presented in this chapter will definitely be utilized when dynamic manager version for the DDoS Protector will be created. Although many modifiers and thresholds are marked as experimental, the module was already able to provide reasonable mitigation method suggestions as described in 4.4.2. Nevertheless, the presented technique needs to be developed and tuned out a little more to provide relevant information when used on the real network.

Chapter 5

Conclusion

This thesis has provided an overview of the most common attacks on the Transmission Control Protocol. All were analyzed quite in detail, but the special focus was put on one particular type – the *TCP SYN Flood*. This attack is currently the most popular performed DoS/DDoS attack type, posing as a significant threat to modern computer networks. Many methods used for its mitigation are either ineffective against more sophisticated variants or cannot be effectively deployed on intermediary devices. For this reason, the thesis aimed to design and implement a mitigation method able to deflect advanced SYN floods, while being efficient when used as a part of the intermediary network mitigation device.

One of the methods with these parameters is *TCP Reset Cookies*, a specialized network-based SYN Flood mitigation technique. This method provides an efficient way to block all SYN flooding attacks from spoofed IP addresses. Regular attacks from legitimate hosts are blocked as well, but other methods are often more suitable when dealing with this variant of the attack. The main advantage of the RST Cookies is the ability to mitigate more sophisticated SYN floods that are typically able to bypass other defense mechanisms. This is achieved by establishing a security association with the client before forwarding its SYN data. This mechanism stops all attacks that rely on dummy segment flooding, however it may be fooled by employing or simulating a legitimate TCP stack. This vulnerability is addressed by enhancing the method with SYN counters and blacklisting mechanism. However, utilization of this method causes an approximate 1-second delay for the first connection and significantly limits segment throughput due to its higher CPU requirements.

Unfortunately, this method is not suitable for all attack vectors due to the performance degradation it causes. Because of this, the Dynamic Method Management algorithm was developed to provide a way to choose the optimal mitigation method according to the current traffic and other factors like mitigation efficiency. The method consists of three separate parts – the Fitness, Traffic analysis, and Decision-making cores. The mechanism evaluates available mitigation functions, analyzes traffic and statistics, and chooses the most suitable method. This approach aims to provide an automatic method switching technique, which should be able to respond to the dynamic environment of modern SYN Flood attacks.

Mentioned algorithms were developed as a part of the CESNET's DDoS Protector security research project. The RST Cookies method is already integrated and used, whereas the method management module is planned to be integrated in the near future. These algorithms will be further developed as a part of the DDoS Protector, which recently received a grant from the Ministry of the Interior of the Czech Republic. Part of the thesis comprising theory and the RST Cookies algorithm was presented at the student's conference Excel@FIT 2019, where it was awarded for a contribution in the computer security field.

Bibliography

[1] RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification. RFC 791. September 1981.

[2] T/TCP: SYN and RST Cookies. Archive for Linux Kernel Mailing List. April 1998. (online). Retrieved on 12.03.2019.
Retrieved from: https://lists.gt.net/linux/kernel/12829

[3] Bellovin, S.: Defending Against Sequence Number Attacks. RFC 1948 (Informational). May 1996. obsoleted by RFC6528.

[4] Bernstein, D., J.; Schenk, E.: SYN Cookies proposal. September 1996. (online). Retrieved on 12.03.2019.
Retrieved from: http://cr.yp.to/syncookies/archive

[5] CESNET: DDoS Protector. (online). Retrieved on 04.05.2019.
Retrieved from: https://www.liberouter.org/technologies/ddos-protector/

[6] CESNET: Národní propojovací uzel NIX.CZ bude testovat DDoS ochranu vyvinutou ve sdružení CESNET. Press release (czech). March 2019. (online). Retrieved on: 02.05.2019.
Retrieved from: https://www.cesnet.cz/sdruzeni/zpravy/tiskove-zpravy/narodni-propojovaci-uzel-nix-cz-bude-testovat-ddos-ochranu-vyvinutou-ve-sdruzeni-cesnet/

[7] Cheng, Y.; Chu, J.; Radhakrishnan, S.; et al.: TCP Fast Open. RFC 7413 (Experimental). December 2014.

[8] Cisco: Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper. Technical report. 70 West Tasman Dr., San Jose, CA 95134 USA. January 2017. updated on February 27, 2019.

[9] Eddy, W., M.: Defenses Against TCP SYN Flooding Attacks. In *The Internet Protocol Journal*, vol. 9. Cisco Systems Inc.. December 2006.

[10] Eddy, W., M.: TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational). August 2007.

[11] Fergusson, P.; Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827 (Best Current Practice). May 2000.

[12] Flajolet, P.; Fusy, E.; Gandouet, O.; et al.: Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *In AOFA '07: Proceedings of the 2007 International Conference on Analysis of Algorithms*. 2007.

[13] Gont, F.; Bellovin, S.: Defending against Sequence Number Attacks. RFC 6528 (Standards Track). February 2012.

[14] Hagen, J. T.; Mullins, B. E.: TCP veto: A novel network attack and its Application to SCADA protocols. In *ISGT*. IEEE. February 2013. ISBN 978-1-4673-4894-2. pp. 1–6.

[15] Harris, B.; Hunt, R.: TCP/IP security threats and attack methods. *Computer Communications*. vol. 22. June 1999: pp. 885–897.

[16] Hinden, B.; Deering, D. S. E.: Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Standards Track). December 1998.

[17] Hurst, T.: Bloom Filter Calculator. (online). Retrieved on 05.05.2019. Retrieved from: https://hur.st/bloomfilter/

[18] Kupreev; Badovskaya; Gutnikov: DDoS attacks in Q4 2018. Technical report. Kaspersky Lab. February 2019. (online). Retrieved on: 01.05.2019. Retrieved from: https://securelist.com/ddos-attacks-in-q4-2018/89565

[19] Lemon, J.: Resisting SYN Flood DoS Attacks with a SYN Cache. In *Proceedings of the BSD Conference 2002 on BSD Conference*. BSDC'02. Berkeley, CA, USA: USENIX Association. January 2002. pp. 10–10.

[20] Lu, Y.; Prabhakar, B.; Bonomi, F.: Bloom Filters: Design Innovations and Novel Applications. In *43rd Annual Allerton Conference on Communication, Control and Computing 2005*, vol. 2. PublisherUniversity of Illinois at Urbana-Champaign, Coordinated Science Laboratory and Department of Computer and Electrical Engineering. January 2005. pp. 1006–1015.

[21] Postel, J.: Transmission Control Protocol. RFC 793 (Standard). September 1981. updated by RFCs 1122, 3168.

[22] Ricciulli, L.; Lincoln, P.; Kakkar, P.: TCP SYN Flooding Defense. In *In Comm. Net. and Dist. Systems Modeling and Simulation Conf. (CNDS' 99)*. Computer Science Laboratory SRI International. January 1999.

[23] Riorey: Taxonomy of DDoS Attacks. (online). Retrieved on 11.03.2019. Retrieved from: http://www.riorey.com/types-of-ddos-attacks

[24] Simpson, W., A.: TCP Cookie Transactions (TCPCT). RFC 6013 (Experimental). January 2011.

[25] Watson, A., Paul: Slipping in the Window: TCP Reset Attacks. April 2004. Retrieved from: https://www.researchgate.net/publication/240246042_Slipping_in_the_Window_TCP_Reset_Attacks