QUIC Protocol : Resilience against flooding attacks and defense mechanism

Benjamin Teyssier

A Thesis

in

The Department

of

Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Information Systems Engineering) at

Concordia University

Montréal, Québec, Canada

March 2024

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Benjamin Teyssier**

Entitled: **QUIC Protocol : Resilience against flooding attacks and defense mechanism**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Suryadipta Majumdar

_____ Examiner
Dr. Abdelhak Bentaleb

_____ Supervisor
Dr. Carol Fung

Approved by _____
Dr. Chun Wang, Chair
Department of Concordia Institute for Information Systems Engineering

_____ 2024

_____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

QUIC Protocol : Resilience against flooding attacks and defense mechanism

Benjamin Teyssier

QUIC is a modern transport layer internet protocol designed to be more efficient and secure than TCP. It has gained popularity quickly in recent years and has been adopted by a number of prominent tech companies. Its efficiency comes from its handshake design. The server and the client make both the transport layer acknowledgment and the TLS agreement during the same round trip. However this process makes the packets heavy and requires more processing on the server-side than TCP. This characteristic can be used as leverage by an attacker to compromise the computing resources of its victim.

This thesis investigates the resilience of QUIC Protocol against handshake flood attacks and proposes a detection mechanism (QUICShield). I conducted comprehensive experiments to evaluate the resource consumptions of both the attacker and the target during incomplete handshake attacks, including CPU, memory, and bandwidth. We compared the results against TCP Syn Cookies under Syn flood attacks. The DDoS amplification factor was measured and analyzed based on the results. This work also proposes a detection mechanism based on a Bloom filter combined with Generalized Likelihood Ratio Cumulative Sum (GLR-CUSUM) to adapt to evolving attack patterns. It was implemented and deployed against real attacks to evaluate its efficiency. We showed that the QUIC Protocol design has a much larger DDoS amplification factor compared to the TCP, which means QUIC is more vulnerable to handshake DDoS attacks. However the mechanism proposed is accurate and efficient in terms of resources.

Acknowledgments

I would like to express my heartfelt gratitude to my supervisor, Dr. Carol Fung, for her support, invaluable advice, and insightful feedback throughout the entire thesis process. Her knowledge, dedication, and mentorship have been instrumental in this research. I am truly fortunate to have had such a supervisor, and I am deeply thankful for her contributions to my academic journey. I would like to acknowledge the thesis committee members, professors Abdelhak Bentaleb and Suryadipta Majumdar for their insightful comments that helped me considerably improve my research work. I was able to complete this thesis thanks to Mitacs and Trustcore company, that I would like to thank for their financial support. In addition, I would like to thank all members of Trustcore with whom it was a pleasure to work. In the NGNSec Laboratory, I had the pleasure to work with Himel, Rambod to whom I would like to express my gratitude for the interesting conversations and projects. I am very thankful that it allowed me to improve under different aspects. Also I want to thank all my friends Maxime, Manuel and Raphaël for their support during this work. I am grateful to my roomates, Sophie and Ryan, for this Canadian adventure and late night debates. A special thanks to my family for their unconditional encouragement and support all along my studies. Last but not least, I want to express to my girlfriend, Justine, for her unfailing support.

Contents

List of Figures viii

List of Tables x

1 Introduction 1

1.1 Overview . . . 1

1.2 Contributions . . . 2

1.3 Thesis Overview . . . 3

2 Literature review 4

2.1 Background . . . 4

2.1.1 Packet Structure . . . 5

2.1.2 Handshake . . . 6

2.1.3 SYN-Flood Attacks . . . 12

2.1.4 QUIC Handshake Flooding Attack . . . 13

2.2 Related works . . . 14

2.2.1 Bloom Filter . . . 14

2.2.2 CUSUM . . . 15

2.2.3 SYN-Flood Defense . . . 15

3 Evaluation of QUIC resilience against handshake flooding attack 17

3.1 Experimental setup . . . 17

3.2 CPU . . . 19

3.2.1 Usage over time . . . 20
3.3 Memory . . . 25
3.3.1 Usage over time . . . 26
3.3.2 Memory Consumption . . . 27
3.3.3 Memory usage and Amplification Factor . . . 28
3.4 Bandwidth . . . 29
3.5 Other implementations . . . 30
3.6 QUIC-Flooding with Early Data . . . 34
3.7 Informal analysis of the attacker . . . 35
3.8 Discussion . . . 35

4 QUICShield : Handshake flooding attack detection mechanism 37
4.1 Objectives . . . 37
4.2 QUICShield Design . . . 39
4.2.1 Storage structure . . . 39
4.2.2 Change-detection mechanism . . . 41
4.3 Experimental results . . . 43
4.3.1 Experimental setup . . . 43
4.3.2 Output under DDoS attack . . . 46
4.3.3 Selection of parameters . . . 46
4.3.4 Behavior under different attack rates . . . 49
4.3.5 Complexity analysis . . . 49

5 Conclusion 51
5.1 Work summary . . . 51
5.2 Study limitations . . . 52
5.3 Discussion and future work . . . 52

Appendix A List of Publications 55

Appendix B List of acronyms used 56

Appendix C Mathematical proofs **58**

C.1 Maximum value of W_n 58

Appendix D Measurement scripts **60**

D.1 Memory and CPU usage over time 60

D.2 Memory and CPU usage for different attack rates 61

List of Figures

Figure 2.1 QUIC and TCP Packet Structures 6
Figure 2.2 QUIC Handshake 7
Figure 2.3 QUIC Detailled Cryptographic Handshake [1] 9
Figure 2.4 SYN-Flood Attack 13
Figure 2.5 QUIC Handshake Flooding Attack 14
Figure 2.6 Standard Bloom Filter [2] 15
Figure 3.1 Experimental setup 19
Figure 3.2 CPU Usage over time in (*aioquic* and TCP Syn-cookies) 21
Figure 3.3 QUIC and TCP SYN Cookie CPU Usage (*aioquic* implementation) 23
Figure 3.4 QUIC and TCP SYN Cookie CPU Amplification Factor (*aioquic* implementation) 25
Figure 3.5 Memory usage over time (*aioquic* implementation) 27
Figure 3.6 QUIC Memory Usage and Amplification Factor (*aioquic* implementation) . 28
Figure 3.7 Bandwidth usage comparison (*aioquic* implementation) 30
Figure 3.8 CPU consumption in *quic-go* 32
Figure 3.9 Amplification factor in *quic-go* 32
Figure 3.10 CPU consumption in *picoquic* 33
Figure 3.11 Amplification factor in *picoquic* 33
Figure 3.12 Amplification factor with and without Early Data 34
Figure 4.1 QUIC Handshake Flooding Attack 39
Figure 4.2 Bloom Table incrementation 40

Figure 4.3 Probability of an attack based on cell value (with $y = 500, p = 0.01, \theta = 0.005$) 42
Figure 4.4 Outputs of the DARPA dataset with different forgetting factor α 45
Figure 4.5 Maximum value of W_n depending on α 48
Figure 4.6 Reaction of the mechanism under different attack rates 49
Figure 4.7 Comparison of CPU load for various Θ under different attack rates 50

List of Tables

Table 2.1 Cryptographic Handshake annotations 10
Table 2.2 SYN-Cookie Structure 12
Table 3.1 Implementations CPU usage comparison 36
Table 3.2 Implementations amplification factor comparison 36

Chapter 1

Introduction

1.1 Overview

Since the first major cyber attack on the Internet in 1988, cyber incidents have grown significantly. One of the most common attacks is Distributed Denial of Service (DDoS) attacks [3]. DDoS attacks can potentially overwhelm servers, rendering them unable to process legitimate requests, resulting in significant economic and reputational damage to the service provider. One specific type of DDoS attack, the QUIC-Flooding attack, has gained prominence due to the increasing popularity of the QUIC protocol [4]. The emergence of the QUIC protocol represents a significant milestone in Internet communication. It offers enhanced performance and reduced latency compared to the traditionally used TCP protocol [5]. Its capability of facilitating secure and efficient data transmission across unreliable networks has attracted much attention from both academia and industry [6]. However, as with any innovative technology, it is imperative to scrutinize its potential vulnerabilities. One such area that deserves rigorous investigation is the resilience against handshake flood attacks [7]. The handshake flood attack is a type of Denial-of-Service (DoS) attack, which poses a significant threat to network infrastructure [8]. The handshake flood attack exploits the handshake mechanism present in many communication protocols, enabling attackers to amplify their traffic and potentially trigger service outages and resource exhaustion. While the implications of handshake flood attacks have been extensively studied in protocols such as the Domain Name System (DNS) [9] and Network Time Protocol (NTP) [10], an evaluation of their impact within the QUIC

protocol remains absent in the current body of literature. Given the growing use of QUIC including many well-known tech companies such as Google and Amazon, this knowledge gap is important to bridge. In this context, an empirical evaluation of the impact of such an attack is needed as well as efficient defense mechanisms to tackle the growing number of cyber attacks. To address the critical need for a thorough security assessment of the QUIC protocol, this thesis experimentally explored QUIC's resilience against handshake flood attacks. We constructed a realistic test-bed with a QUIC server and a QUIC attacker client that generates spoofed handshaking requests. We conducted exhaustive experiments to measure the resource consumption of both the attacker and the targeted server during incomplete handshake attacks, containing CPU, memory, and bandwidth. Our results reveal that the QUIC protocol design has a significantly larger DDoS amplification factor compared to TCP Syn Cookies, indicating a higher vulnerability to handshake DDoS attacks. In contrast to previous studies [11, 12], we have identified the CPU resource of QUIC servers as the most likely bottleneck during QUIC handshake flood attacks. In order to address this issue we propose to develop an efficient change-detection-based defense mechanism. Existing research has explored various techniques to address the problem of DDoS attacks [13]. One prominent approach is using Bloom Filters for data storage and implementing the CUSUM (Cumulative Sum) algorithm for change detection [14]. These methods have been successfully applied in detecting TCP SYN-flooding attacks [15]. However, the traditional CUSUM algorithm requires prior knowledge of the probability distribution of the system's normal behavior and during an attack, which may not be accurate or available [16]. We incorporate GLR-CUSUM (Generalized Likelihood Ratio CUSUM) to increase the versatility of the mechanism. Additionally, the existing techniques can not be directly applicable to QUIC-Flooding Attacks due to the differences in the packet structure and processing requirements of the QUIC protocol.

1.2 Contributions

The major contributions of this thesis can be summarized as following:

- We investigate the QUIC protocol resource consumption during handshake flood attack

- We compare the resource consumption of QUIC protocol with TCP Syn Cookies under handshake flood attacks on real test-beds
- We measure the DDoS amplification factors of QUIC protocol and reveal a vulnerability of the QUIC protocol design.
- In order to address this vulnerability, we introduce the design of QUICShield, a novel and tailored detection mechanism designed to combat QUIC-Flooding DDoS attacks. QUICShield utilizes a modified version of the Bloom Filter for efficient data storage. It incorporates the GLR CUSUM algorithm for change detection, which overcomes the limitations of traditional CUSUM algorithms.
- This research includes emulation experiments to assess the performances against real attacks and measure the resource consumption of the algorithm.

1.3 Thesis Overview

This thesis is organized as follows:

- Chapter 2 presents the background on which this work is based, it also contains the related works from previous research.
- Chapter 3 presents the experiments led to study servers' behavior under QUIC handshake flooding attacks.
- Chapter 4 describes the proposed defense mechanism against handshake flooding attacks.

Chapter 2

Literature review

Section 2.1 provides the background explanation about QUIC Protocol and particularly its handshake mechanism. A comprehensive review of the related works on QUIC Flooding attacks and various defense mechanisms is presented in Section 2.2.

2.1 Background

QUIC is a revolutionary transport protocol initially designed by Google to enhance the performance of web applications [17]. It achieves this performance improvement by modifying the handshake process. It reduces the number of RTT (Round-trip Time) to establish a connection and allows the client and server to exchange encrypted data very early in the connection process compared to what could be done previously with TCP. It still uses TLS 1.3 [18] for the cryptographic handshake. QUIC Protocol also provides better multiplexing to avoid head-of-line blocking. Previously, with TCP you could not process a packet if there was a problem with the transmission of the previous one. The server accumulated packets until the missing one was sent again. In QUIC, the multiplexing capability removes this restriction and speeds the transmission process up. QUIC designers took into consideration the various problems caused by congestion during TCP transmissions. The congestion control mechanism in QUIC is also standardized [19]. Some variants of TCP included a decent congestion control mechanism which inspired the one in QUIC. However, differences in the nature of packets led to some mandatory modifications in the process. TCP actually

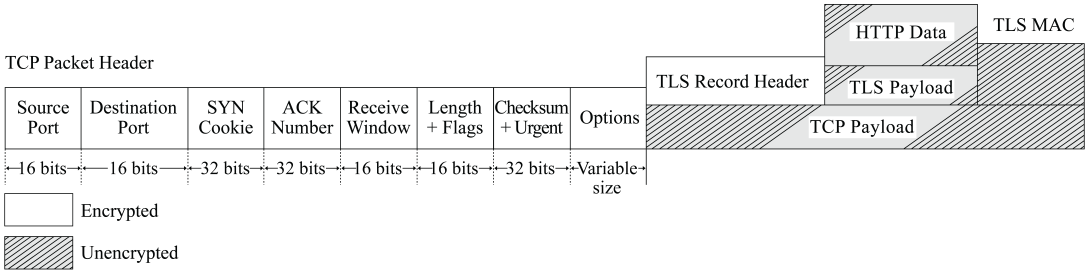
had a lot of different variants proposed by academia and industry to solve the problems in its initial design. However, the different needs and the lack of common standardization led to local compartmentalization of these variants. QUIC is trying to consider the needs, from which the TCP variants originated, and create a new Transport protocol solving performance and security issues. Most of these are solved during the handshake process which is described in Subsection 2.1.2.

2.1.1 Packet Structure

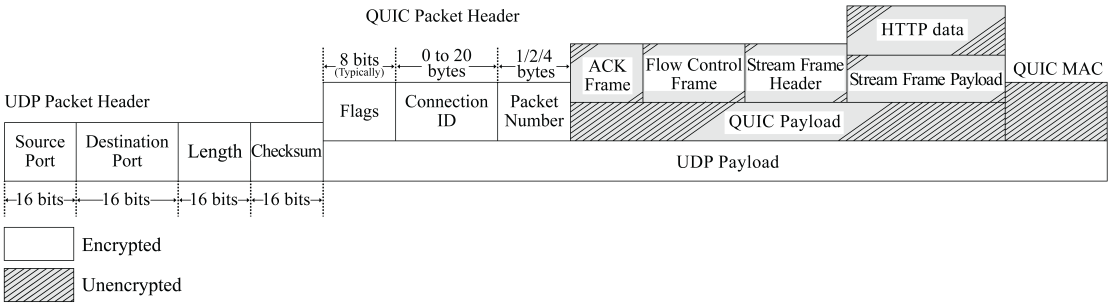
This subsection will describe the QUIC packet structure shown in Figure 2.1b and compare the main differences with the packet structure of TCP shown on Figure 2.1a. QUIC packets are carried on UDP datagrams and inherit some of UDP transport properties. For example, the packets do not need to be ordered. In TCP you cannot receive and process a packet number n if you did not receive packet number $n-1$. This is called head-of-line blocking and causes serious congestion and loss issues. QUIC combines this property and its own packet ordering mechanism to solve head-of-line blocking without affecting performance. UDP is also stateless, the server does not store anything about the client. QUIC uses this property to be stateless too during its handshake process. UDP is in fact a very basic and lightweight transport protocol on top of the IP protocol, this characteristic gave QUIC designers a lot of freedom for the architecture choices.

As shown on Figure 2.1b, the only fields in the UDP header are source and destination port, the checksum to verify the integrity of the message against physical errors causing bit flipping and the length of the packet. Note that the UDP checksum is not the only mechanism used to verify the integrity as QUIC has its own Message Authentication Code to make sure the packet was not modified. In the plaintext part of the header, few information is displayed such as connection ID and packet number. In the flags section we can find information such as the version ID. The idea is to limit the amount of information in the plaintext section to the minimum without affecting performance. However, some information can save a considerable amount of time for the server. For example if a client is sending a packet with a deprecated version, it would be a waste of time to decrypt the packet. The server can directly send a version negotiation packet. In the encrypted part, we can find different frames used for packet acknowledgment, packet ordering or multiplexing. An important difference between TCP and QUIC packets is the moment from when the packets are

encrypted. In TCP the first packet to be encrypted is ChangeCipherSpec during the TLS handshake, the TCP handshake and the first part of the TLS one are done in plaintext and the structure of Figure 2.1a is only applied afterwards.



(a) TCP SYN Cookie Packet with encryption

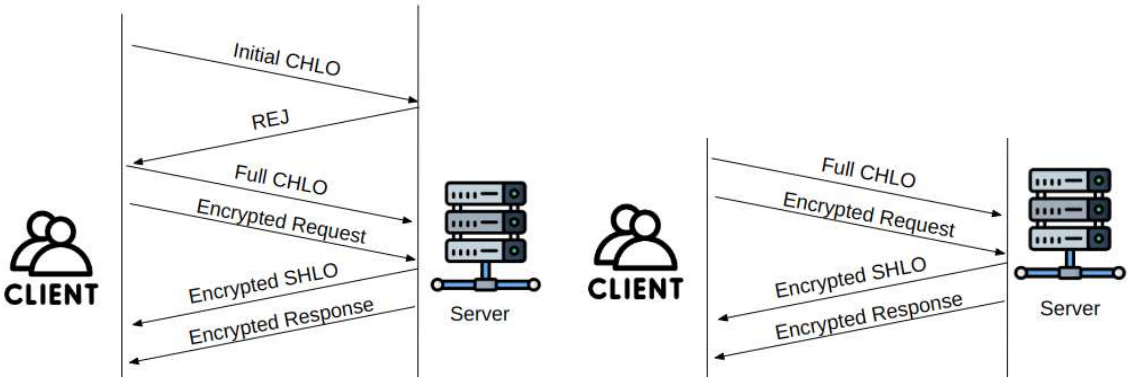


(b) QUIC Datagram or packet

Figure 2.1: QUIC and TCP Packet Structures

2.1.2 Handshake

This subsection will describe the QUIC handshake process, starting with the 1-RTT version which can be seen in Figure 2.2a and then explaining the 0-RTT feature on Figure 2.2b.



(a) 1-RTT Handshake (b) 0-RTT Handshake

Figure 2.2: QUIC Handshake

1-RTT

A particularity of QUIC compared to TCP is that clients can be authenticated before the cryptographic handshake starts to defend against vulnerabilities linked to IP Spoofing. This option called *Address validation mechanism* is usually disabled when the server receives a low number of requests to increase the performance as it removes one RTT from the connection establishment phase. Servers usually enable it when the load is increasing to make sure it only serves legitimate clients. The server uses a token mechanism very similar to what we can find in TCP SYN-Cookie. In order to compute this token, the server needs information on the client. The handshake starts with the client sending an Inchoate CHLO to the server. The server will extract some information from the header of this packet and create a token that will be used to identify the client later on. This token should be created in a way that prevents the client from creating his own token. The technique used in most of the implementations is using keyed hash functions on the client's IP address and the connection ID. The key is known by the server only which prevents the client from forging a token. Unlike TCP SYN-Cookie, the token will be reused for future connection. Therefore, the client's port is generally not used to generate the token as it probably changes between two connections. However, this method can change depending on the QUIC implementation.

The server then replies with a Retry Packet or with a NEW_TOKEN packet containing the newly created token, server's certificate and its public key. The client extracts the token from the packet and stores it. If the server wants to authorize future connection with this token, it is included in the

NEW_TOKEN frame, otherwise the Retry frame is used. In the case of long-term tokens, it will be included in the packet by the client every time he wants to establish a connection with the server. This token still has an expiration date after which the client will need to request a new one.

From there, the cryptographic handshake can start, it is based on two Diffie-Hellman key exchanges. The ephemeral keys are used to encrypt the handshake and the potential early data. The long-term keys are used to encrypt the communications once the handshake is finished. The client computes the initial keys and its long-term public key. It then sends the Full CHLO packet encrypted with the initial key. The client can also add early data encrypted with this initial key.

Once the server receives the Full CHLO, it verifies the token validity and computes the ephemeral and long-term keys. If early data was sent by the client, it is decrypted and the request is processed. Then the server replies with its long-term public key encrypted with the ephemeral key and the response to the early data request encrypted with the long term key.

The client can now compute the long-term key to encrypt and decrypt the future messages. This cryptographic process was formally verified by Zhang et al [1]. The details of the cryptographic handshake is shown on Figure 2.3. The annotations used can be found in Table 2.1.

It is important to note that the server has to perform two Diffie-Hellman key computations, a public key computation, a signature and an encryption when he receives a CHLO packet. These calculations are computation intensive and cause a load on the server side which is not present on the client side at this point of the handshake. This asymmetry can be used by an attacker to leverage a CPU amplification factor during the handshake process.

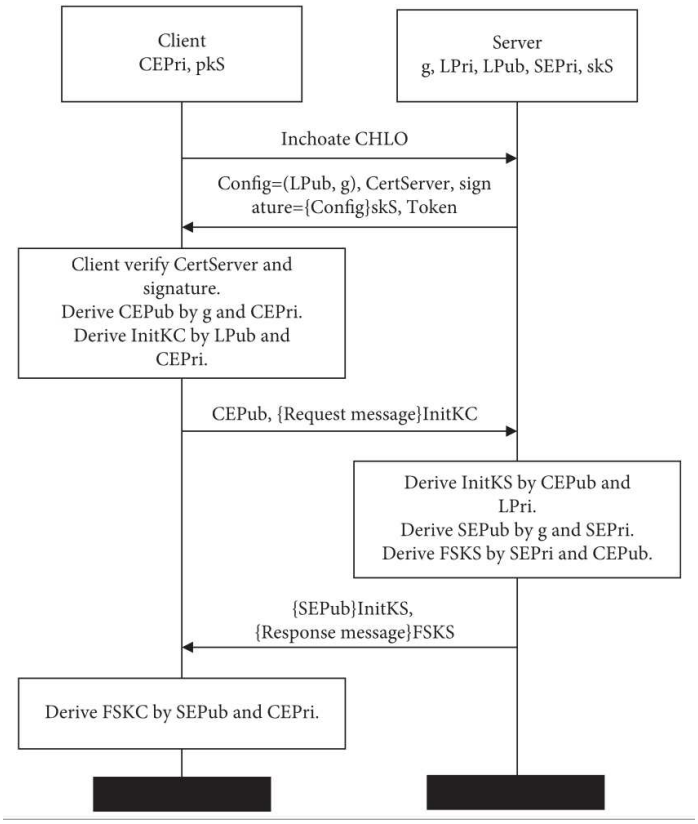


Figure 2.3: QUIC Detailled Cryptographic Handshake [1]

0-RTT

0-RTT Handshaking is a key feature of QUIC. It allows low latency exchanges between client and server by using previous keys to send encrypted messages in the early stages of the handshake. This feature provides significant performance improvement compared to what was known in TCP before.

The concept behind the 0-RTT handshake is very similar to 1-RTT. The difference resides in the fact that the client already has a token from a previous connection. In that case, he can send a new Full CHLO to the server by including this token. The client can also join a request using the initial keys computed in the previous exchanges. A new set of long-term keys will be used after the handshake. The tokens given by the server have an expiration date. If the client sends a token which is not valid anymore, the server will send a REJ packet containing a newly computed token for the client to include in his packets.

Annotation	Meaning
CEPri	Client's ephemeral Diffie–Hellman private value
LPri	Server's long-term DH private value
SEPri	Server's ephemeral Diffie–Hellman private value
LPub	Server's long-term DH public value
g	Primitive root
CEPub	Client's ephemeral Diffie–Hellman public value
SEPub	Server's ephemeral Diffie–Hellman public value
InitKC	Initial key of client
InitKS	Initial key of server
FSKC	Forward-secure key of client
FSKS	Forward-secure key of server
pkS	Public signature key of the server
skS	Private signature key of the server
{}skS	{} is signed using the private signature key of the server
{}InitKC	{} is encrypted using the initial key of client
{}InitKS	{} is encrypted using the initial key of server
{}FSKS	{} is encrypted using the forward-secure key of server

Table 2.1: Cryptographic Handshake annotations

Without address validation

However, in order to increase the performance of the exchanges a server usually decides not to validate the address of its clients unless it is under load. In that case, the client sends a Full CHLO with his chosen cipher suites, if the server supports these, he sends his certificate and all the material necessary for the handshake. The process is in fact the same as a 0-RTT handshake without Token verification.

TCP SYN COOKIE

TCP SYN-Cookies have been introduced by Bernstein [20] as a way to prevent TCP SYN-Flooding attacks. It makes the handshaking process stateless which prevents the attacker from keeping semi-open connections that would overwhelm the server's. The attackers were using spoofed IPs to maximize the number of connections without flooding their own traffic. Indeed, when the server receives the packet with a spoofed IP, it will reply to this IP which is not the one of the

attacker. Therefore, SYN-Cookies provides authentication. The challenge addressed by the mechanism is to authenticate a client without storing information of his previous connections. It also has to do so with limited resources in order to keep a decent computational intensity symmetry between the client and the server. The idea is to compute a number (a cookie) based on several pieces of information from the client's incoming connection. This number has to be hard to guess for a client so that a client cannot forge his own cookie. It also has to change so that a client cannot reuse a cookie stored from a previous connection. The 32-bits cookie is added in the header as shown in Figure 2.1a. The information chosen from the initial packet received is the source and destination IP, the source and destination port, and the Initial Sequence Number from the client (ISN_c). This information is hashed with the lightweight Siphash [21] using two secret keys. The time is also a part of the cookie to ensure that a too old connection will not be accepted anymore. A handshake should be finished by the client in the next 64 seconds following the Server's SYN-ACK response.

$$syncookie = H(s_1, s_a, s_p, d_a, d_p) + ISN_c + T \times 2^{24} + (H(s_2, s_a, s_d, d_a, d_p, T) \mod 2^{24}) + MSS_i$$

The notations are as follows :

- H : Siphash keyed hash function
- s_1 : Server's first secret key
- s_2 : Server's second secret key
- s_a : Source address of the packet
- s_p : Source port of the packet
- d_a : Destination address of the packet
- d_p : Destination port of the packet
- ISN_c : Initial Sequence Number from Client
- T : Timestamp
- MSS_i : Maximum Segment Size

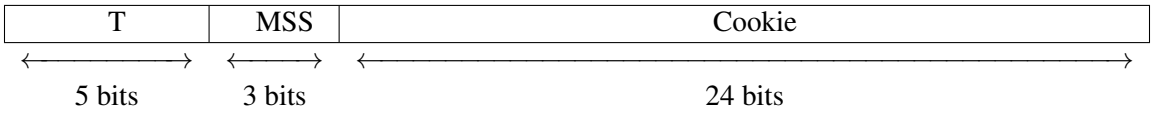


Table 2.2: SYN-Cookie Structure

The cookie is then added to the header following structure shown in Table 2.2

2.1.3 SYN-Flood Attacks

QUIC Handshake flooding attacks are very similar in terms of behavior to TCP SYN-Flooding attacks. The attacker uses the properties of the protocol which requires a low amount of resources to create the packet on the client and a higher amount on the server side. In TCP, the attacker will send a lot of SYN packets. For each SYN packet, the server will allocate a TCB (Transmission Control Block) in memory. This block will be used to remember information about the client and the session associated. By sending a large amount of packets, the attacker will force the server to allocate more memory than it can handle, creating a memory overflow. Under this condition, the server will be overloaded and will not be able to respond to the requests of legitimate clients creating a DoS. The attack scheme is shown on Figure 2.4. Most of the time, the attacker will use spoofed IPs to make the detection harder for the victim. An attacker can also leverage a botnet to reach higher attack rates.

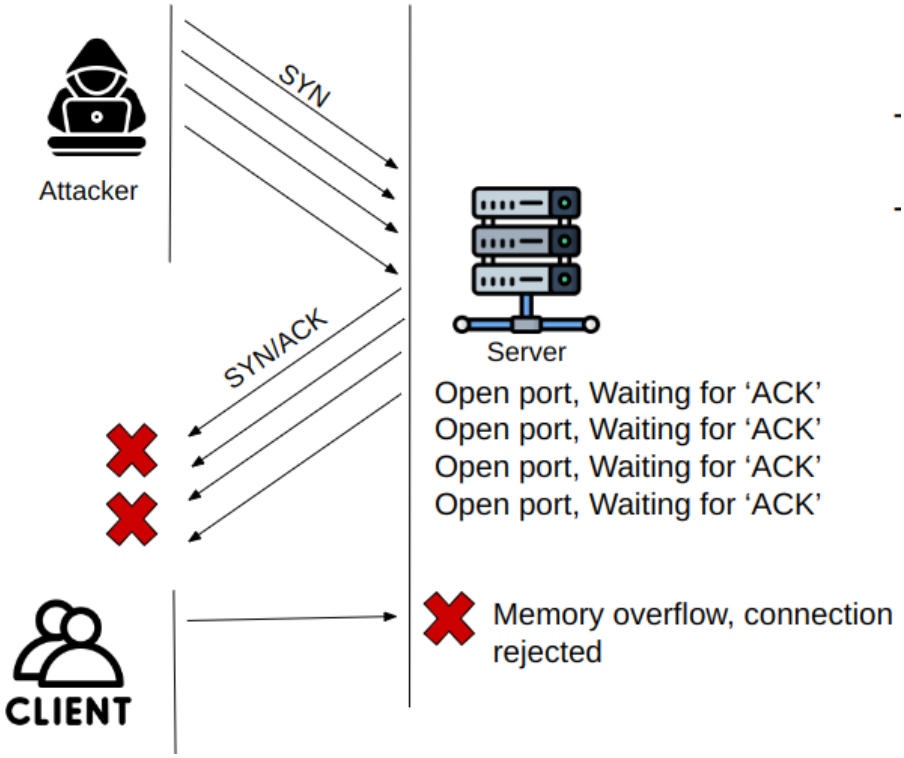


Figure 2.4: SYN-Flood Attack

2.1.4 QUIC Handshake Flooding Attack

QUIC Handshake Flooding Attack consists in sending a lot of Full CHLO packets to the server in order to overwhelm its computation resources. Indeed, due to the cryptographic design of the handshake and the asymmetry of computations, the attack can leverage a computation resource amplification factor.

First, the attacker sends an Initial CHLO (previously Inchoate in older versions) to get a token from the server. It is a mandatory step because the server will not proceed to the key computation if the token is not valid. The token verification is the first step done on the server side to avoid IP Spoofing. It saves a lot of resources because every packet without a valid token will be discarded without further processing. Once the attacker receives the token in the REJ packet, he can start sending Full CHLO at a high rate. The stateless characteristic of the protocol will force the server to compute the keys for each new CHLO as it is not possible to store those.

It is interesting to note that the designers took into consideration the problem of IP Spoofing present

in TCP SYN-Flooding. The address validation mechanism achieved with token verification prevents an attacker from using spoofed random IPs. It reduces the possibilities of an attacker as he will have to use botnets to avoid detection. Botnets are actually not using IP Spoofing, the number of bots compensate for the need of various IPs to avoid detection and to achieve high attack rates.

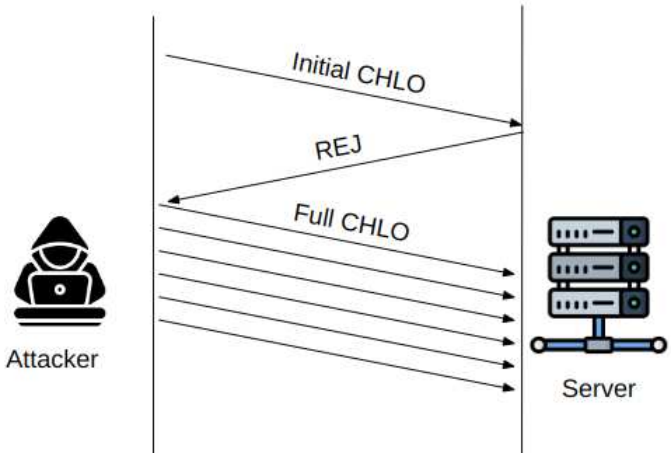


Figure 2.5: QUIC Handshake Flooding Attack

2.2 Related works

2.2.1 Bloom Filter

Bloom filters are widely used in network management and security for detection purposes [2]. It is a probabilistic data structure designed to detect if an element is a member of a set. It is space-efficient and does not allow false negatives, however the false positive rate can be high depending on the parameters chosen.

Standard Bloom Filter consists in an array of bits of size m. In the beginning, all bits are set to 0. Then, the filter takes the elements x_i of the set and hashes each of them k times. The image of the hash function is the interval $[\![0, m-1]\!]$. Each time an element is hashed into an index, the corresponding bit is set at 1. Therefore, each element is represented in the table by k bits set at 1. If a bit is already set to 1 when an element is hashed into its index, it stays at 1. Then, to test the membership of new elements y_i, they are hashed k times; if every bit corresponding to the output of the hash function is 1, the element is probably a member of the set. However if a bit is at 0,

it is certain that the element is not part of the set. False positives are caused by the overlapping of two elements in the table. Therefore, the more elements we add in the set, the higher the false positive rate. On the contrary, choosing a larger table, i.e. increasing m, reduces overlapping and false positives. Figure 2.6 shows how the mechanism works.

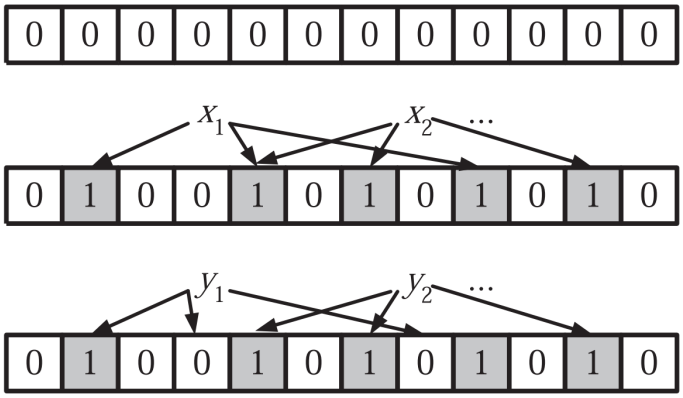


Figure 2.6: Standard Bloom Filter [2]

2.2.2 CUSUM

Cumulative sum control chart (CUSUM) is a sequential analysis technique used for change detection monitoring. This statistical method is used in many different fields such as medicine, signal processing or industrial logistics. It is based on known distributions of the monitored variable and computes the cumulative likelihood ratio. The sum of these ratios on the different steps gives the sequential aspect of the technique. *Xie et al.* work [22] provides a description of the standard CUSUM method as well as the different extensions and generalization of the technique.
GLR-CUSUM is a particularly interesting generalization when you cannot make assumptions on your data distribution. The global idea remains the same but you do not assume a known distribution over your variable. In that case, you approach the real distribution using different sets of functions. It increases the calculations necessary but makes the technique more versatile.

2.2.3 SYN-Flood Defense

SYN-Flood attacks can be defended by different methods which can be classified protocol level or external. For instance, TCP-SYN Cookie and TCP-SYN Cache are modifications of the protocol

implementation in order to limit or eliminate the threat. These methods are very effective but can present drawbacks such as performance or compatibility issues. Indeed, when modifying a protocol that constitutes the backbone of the whole internet, it is difficult to incite everyone to use it. Also, only the servers have an interest in adopting such modifications, the incentives for the clients are negligible. Therefore, external methods such as traffic filtering can be easier to implement and deploy at large scale. *Chen and Yeung* proposed a statistical method [15] based on CUSUM in order to detect high rates of TCP SYN packets. It also tackles the spoofing consideration and the impact on the system.

Chapter 3

Evaluation of QUIC resilience against handshake flooding attack

The goal of this chapter is to provide an experimental evaluation of the resource consumption under handshake flooding attack for both server and attacker. This way, the amplification factor is easily calculated by dividing the amount of resources necessary on the server side by the one on the attacker side. Three main resources are monitored : CPU usage, memory usage (RAM) and bandwidth.

This chapter is organized as follows : we start by describing the experimental setup in place for the measurements in section 3.1, then sections 3.2 to 3.4 discuss every monitored resource separately. Section 3.5 presents similar experiments for different implementations of the QUIC protocol. Section 3.6 shows how an attacker can optimize the amplification factor. The content of this chapter was published as a paper [23] in conference CNSM 2023.

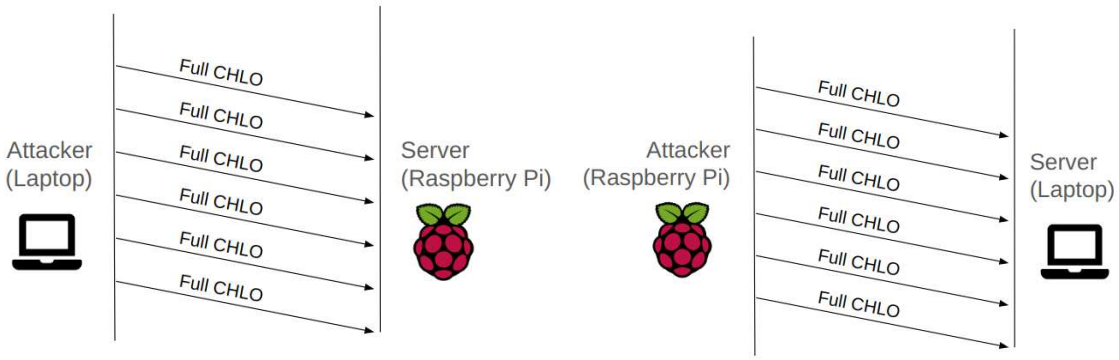
3.1 Experimental setup

The experiments were performed between two basic Linux devices, one acting as a client, the other acting as the server as shown on Figure 3.1. One device was a Raspberry PI Model 3B having 1GB of RAM available and a Quad Core 1.2GHz CPU, the other was a laptop with 16GB of RAM and Intel i7 1.8GHz CPU. The roles of server and client were reversed between every measurement

to ensure no dependence on the hardware for our results. The measurements were made ten times for each device and the average of both scenarios were taken. The 95% confidence interval was also drawn on some figures to show the variation from one measurement to the other. Following the same idea, the base resource usage of processes present on the machine should not influence the measure. To that effect, the measures were conducted on the process running the server and client to isolate the resource consumption.

The goal of these experiments is to evaluate the resilience of QUIC Protocol against handshaking flooding attacks. We compare its resource consumption under attack with another protocol. TCP being the most popular transport protocol, it is interesting to use it as a comparison. However, to ensure a fair comparison in terms of resilience and resource usage, it was necessary to deploy the stateless version : TCP SYN Cookies [24]. Then we estimated the amplification factor of both protocols to assess the lever an attacker has on its victim. The different resources monitored were the Memory Usage, CPU Usage and Bandwidth. Note that amplification factor is the ratio between the resource consumption of the server to process the packets and the needs of the attacker to craft those.

The implementation chosen to run the experiments in a first place was *aioquic* [25], a Python implementation which provides a large range of API features. After finding the limitation resource, we added two other implementations in order to strengthen our results and avoid implementation biases. The additional implementations chosen were *picoquic* [26], a minimalist compliant C implementation ; and *msquic* [27] which is Microsoft's implementation of QUIC, it is written in C too. All of these are abiding by the standard RFC 9000 [17]. The attack code is a modified version of the different implementations presented above. It only contains the code to create a packet to be as lightweight as possible. The TCP SYN Cookie code is implemented in Python to have a relevant comparison with aioquic. It was written from scratch but follows the guidelines given in TCP SYN Cookies Linux implementation.



(a) First round of the experiment (b) Second round of the experiment

Figure 3.1: Experimental setup

3.2 CPU

In TCP SYN-Flooding, the attacker aimed at overwhelming the server's storage. However, due to the stateless characteristic of both QUIC and TCP SYN Cookie, the server does not store the status of each connection request. Therefore the storage need is not impacted by the attack volume. However, in order to verify the legitimacy of the incoming connection initialization requests, both QUIC and TCP servers need to do some calculations as specified in Section 2.1.2. To measure the CPU consumption of QUIC server, we increase the number of Client Hello packets from 1 packet/second to 50 packet/second and observe the CPU usage on the server side. The CPU usage over time was also estimated to monitor the behavior of the server during an attack. The CPU consumption was obtained with Linux built-in tools *ps* to measure the resources used by a specific process :

```
$ ps -p $PID -o %cpu
```

The experiment was conducted multiple times and the result plotted is the mean of all the values. It also contains the 95% confidence interval to show the variability from one row of experiment to the other.

3.2.1 Usage over time

We observed the resource consumption and the server behavior over time during handshake attacks. This experiment was conducted to observe the CPU Usage over time during attacks at different rates. An attack was started at different rates and the CPU percentage was taken every second. This experiment was performed three times with the different rates to ensure a smooth output. However, it was pretty stable and the results were very close from one round of measurement to the other. The script used for the measurement can be found in Appendix D.1.

On the QUIC server-side of Figure 3.2(b), we can observe the increase of CPU presents inertia before stabilizing. It also appears that higher rates create more inertia than lower ones. On the QUIC client-side, as shown in Figure 3.2(a), a slight decrease happens in the first seconds. Then the CPU usage is quickly stabilized to its final value which is also close to the one found previously. The inertia is caused by the queue of packets. At first the server proceeds to the verifications needed to process the packet. It checks the different values of the flags corresponding to the parameters of the connection inside the header. It also needs to verify if the address was validated. Then it starts preparing the *Server Hello* packet. During the first minutes it falls behind, it is the time for the server to allocate more resources and process the different packets. This inertia can be a double-edged characteristic for the system. It gives more time to a potential DDoS Detection system to notice the attack and to potentially prevent further harm. However, this stack filling up also means that potential clients might encounter additional delay during this period of time.

The experiment was performed under the same conditions for TCP SYN Cookie. The resource consumption is instantly the one it stabilizes to. Both on the server-side and the client-side, we can observe the same values around the beginning of the experiment as in Figure 3.2(c)(d). The CPU usage is low compared to the one obtained with QUIC.

This stability is mostly explained by the low resource consumption. It never goes above 1.5% of the total device capacity. The performance of this protocol is obviously a benefit and a key feature of the SYN Cookies. In the rest of the experiments in this thesis, we use the stabilized resource usage values to represent the usage on each attack volume.

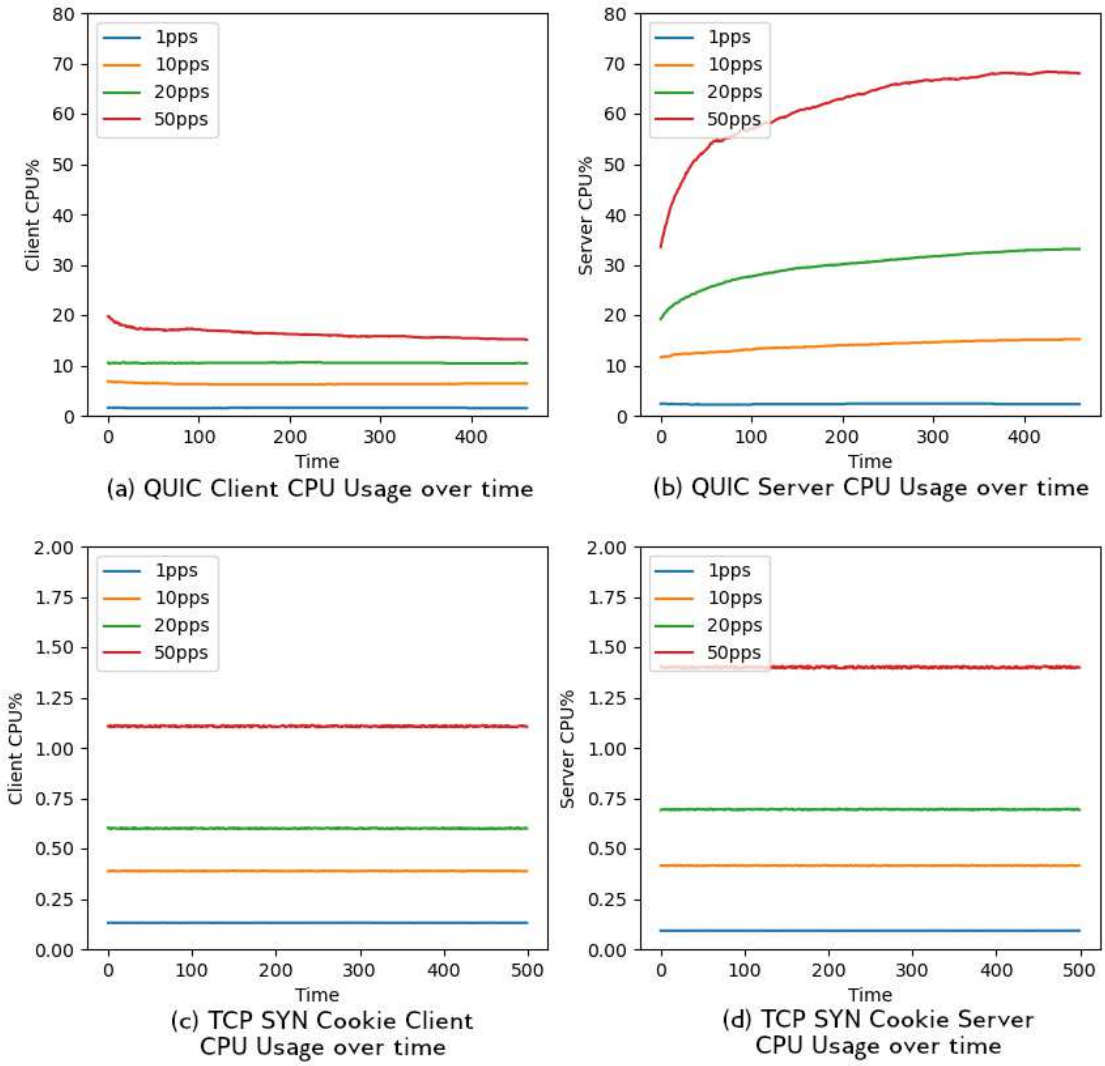


Figure 3.2: CPU Usage over time in (*aioquic* and TCP Syn-cookies)

CPU Consumption

Figure 3.3(a) shows the average and confidence interval of additional CPU usage of the QUIC server and attacker client when the attacker increases the attack volume. The x-axis corresponds to the attack rate in packets per second. We can see that under the QUIC handshake flood attack, the CPU consumption, for both the client and server, increases almost linearly with the attack volume. The server's CPU usage reaches its capacity when the attack volume is close to 50 packet/second. Note that 50 packets per seconds is not the rate limit of the DDoS attack volume since attacker still

has a lot of room to increase its attack volume. However, 50 packets was the limit of our server as it reaches 73% of CPU usage (almost 100% with the base usage). Higher values are not relevant as the overloaded server causes noise in the measures. The experiment shows that during a QUIC-Flooding attack, CPU is the bottleneck. During the QUIC handshake process, the client uses much less CPU resources than the server. The script used for the measurement can be found in Appendix D.2. The same script was then used for the different implementations.

Figure 3.3(b) was obtained by leading the same experiment as Figure 3.3(a) with the TCP SYN Cookie protocol. First of all, it appears that the CPU consumption of sending/handling the same number of TCP handshake requests is significantly lower than QUIC both for server and client. For example, the TCP server only consumes less than 1% CPU resources compared to the 30% consumed by QUIC server when attack volume is 20 packet/second. This is mainly due to the design complexity of QUIC packets and verifications compared to TCP SYN Cookies. Actually TCP SYN Cookie is very similar to usual TCP where the connection ID is just a verifiable hash instead of being a fixed number stored on the server. On the other hand, QUIC packets have a lot of fields that are used for verification purposes. The size of the packets and the verifications themselves are costful in terms of resource consumption.

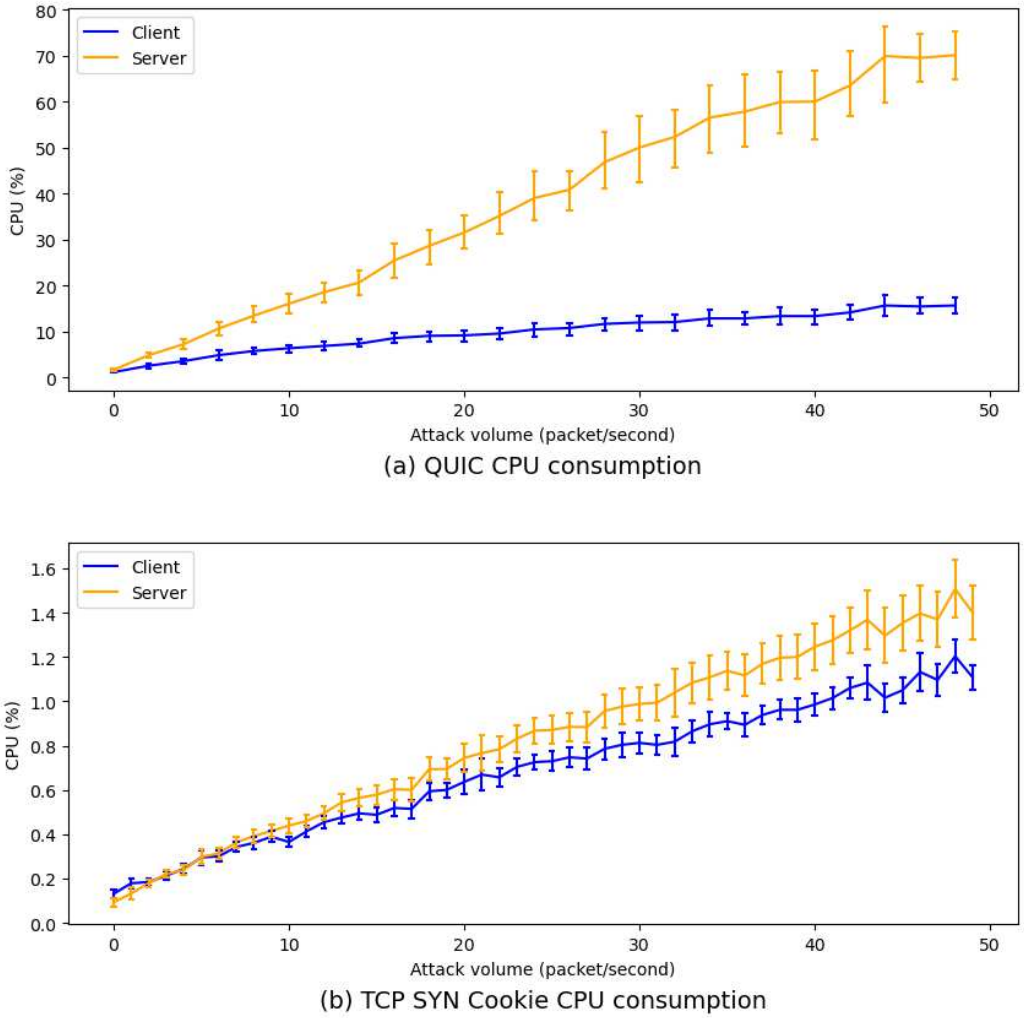


Figure 3.3: QUIC and TCP SYN Cookie CPU Usage (*aioquic* implementation)

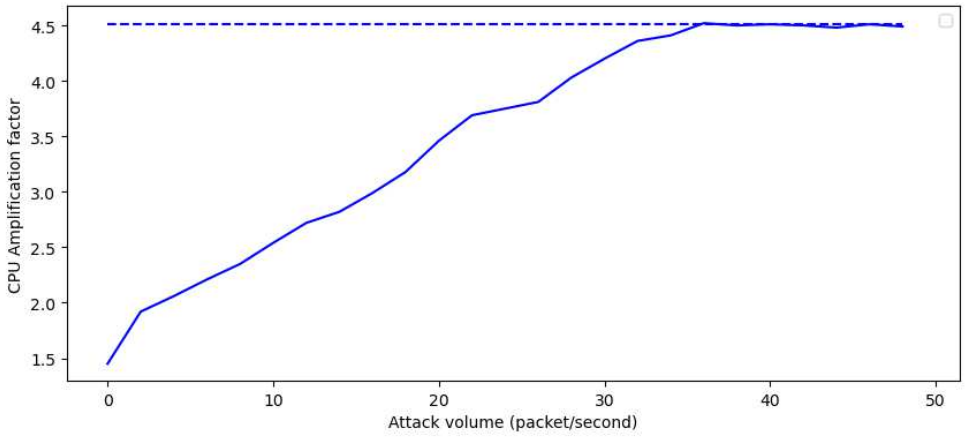
Amplification Factor

This experiment highlights the CPU amplification factor for both QUIC and TCP SYN Cookie. The attacker crafts packets in the most economical way to ensure an optimal amplification. He does not wait for the server responses and does not process those, he also does not close the handshake as it would be useless resource usage.

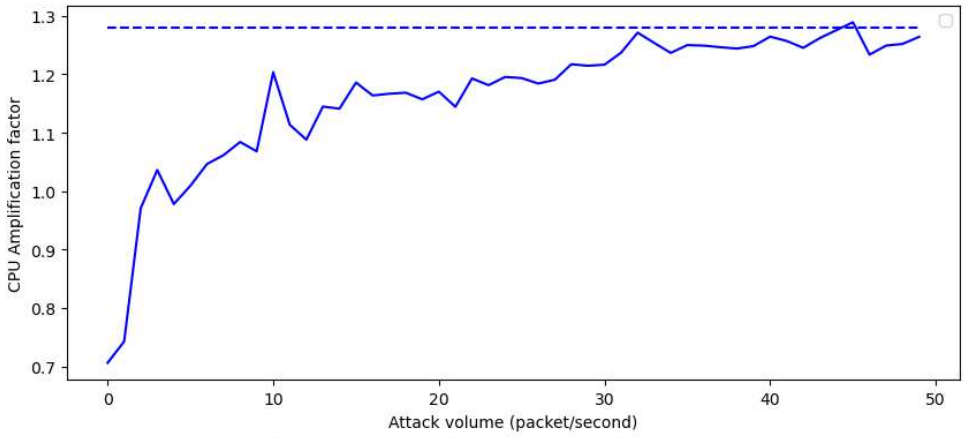
The QUIC amplification factor plotted on Figure 3.4 (a) seems to stabilize around 4.6 meaning that for every unit of computation needed by the attacker, the server will deploy 4.6 times more resources to process the packets. This high amplification factor can be explained by multiple aspects of QUIC design. First the server has to validate the parameters provided by the client. If not, the server

might send a Retry Packet. Then the server has to craft the Handshake packet depending on the information contained in the header of the client's one. An amplification factor over 4 is potentially a serious vulnerability. With the adequate infrastructure, an attacker can take down systems with more than 4 times its computing power, reducing drastically the complexity and the cost of an attack. The amplification factor of TCP SYN Cookie presented in Figure 3.4(b) is notably lower, it stabilizes around 1.28 meaning the server has to do 28% more work than the client to process the packet. It is mainly due to the design of the protocol. During the handshake, the server only has to compute a hash based on information from the incoming packet and the time when it received it. This hash will be the cookie sent back to the client to identify the connection without storing any connection ID. Siphash [21] is the pseudorandom function used to hash this information. It was designed specifically for TCP SYN Cookie with performance in mind to make sure the server is able to process the incoming packets without overloading the computation resources.

This low amplification reduces the lever an attacker has on its victim. It means an attacker must have at least 80% of its victim resources to ensure the success of its attack.



(a) QUIC CPU Amplification factor



(b) TCP SYN Cookie CPU Amplification factor

Figure 3.4: QUIC and TCP SYN Cookie CPU Amplification Factor (*aioquic* implementation)

3.3 Memory

The second resource we measure is the memory usage. During this experiment, we apply the same conditions as in the measure of CPU Amplification and CPU usage over time. The *ps* tool used to get the CPU also allows monitoring memory so the same protocols were followed to optimize relevance of the results and enhance comparability.

```
$ ps -p $PID -o %mem
```

3.3.1 Usage over time

Even if it was established that memory would not be the bottleneck when the system reaches stability, it is important to study memory over time to make sure there is no behavior that could be a vulnerability for the server. For instance, an important memory peak caused by the attacker in a short amount of time before reaching stability could be harmful to the system.

We can observe on Figure 3.5 that the memory follows the behavior of the CPU for the QUIC protocol. There is inertia in the first seconds but the memory usage increases before reaching the stabilized value. There is no resource consumption peak at the beginning of the attack. TCP SYN Cookie was also plotted on this graph as a comparison but memory usage is so low that the measures were not accurate on the client-side and on the server-side for rates under 50pps. The highest usage reached is 0.06% of the device total capacity. The small jumps in the graphs can be explained by the very low usage and the accuracy of the tool when the memory usage is below 1%.

These results confirm the fact that memory is less likely to be the bottleneck under handshake flood attacks. It also highlights how low the memory usage for TCP SYN Cookie is.

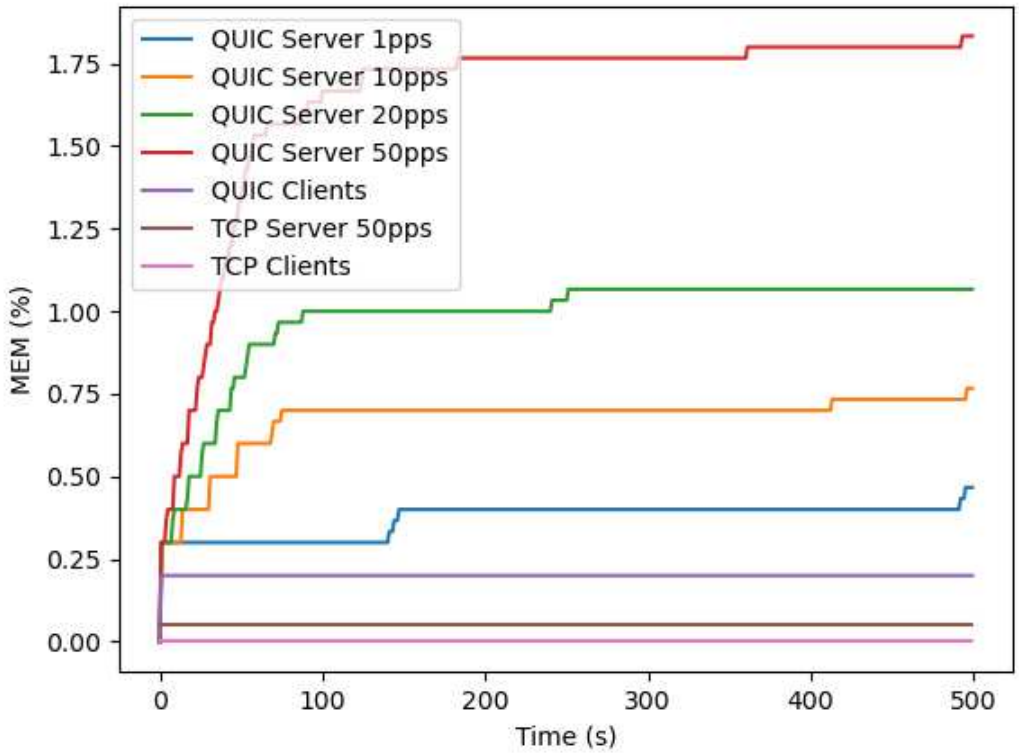


Figure 3.5: Memory usage over time (*aioquic* implementation)

3.3.2 Memory Consumption

As shown in Figure 3.6, on the QUIC client-side, memory usage remains stable when the attack rate increases. The confidence interval is very tight meaning the results obtained during the different rounds of the experiment provided very similar results. The value is low around 0.1% of the device total capacity. On the server-side, there is an increase in memory usage along with the attack rate. The confidence interval is also close to the curve meaning the impact of this type of attack on resources is relatively predictable. The highest value reached is below 2% which is low compared to the CPU usage for a similar attack rate. Compared to CPU usage, the memory usage of QUIC server is much lower and CPU is much more likely to be exhausted first under handshake flooding attack.

3.3.3 Memory usage and Amplification Factor

As shown in Figure 3.6, the memory amplification factor increases along with the attack rate following the drift of the server memory usage reaching 8.6. It is worth noting that the amplification factor may keep on increasing with the attack rate. However, since the CPU resources on the QUIC server has already reached its capacity, a larger attack rate is not meaningful.

We can see that the server memory usage increases linearly with the number of packets because of the asynchronous functions used to handle multiple packets and connections at a time. However once the packets are processed the memory usage falls down to 0 as everything is dumped. Even if the amplification reaches high values, these metrics are not very relevant considering the fact that CPU is a bottleneck before memory could be an issue. In addition to that, it is easy to add memory to an infrastructure but adding CPU is more difficult and certainly more costful.

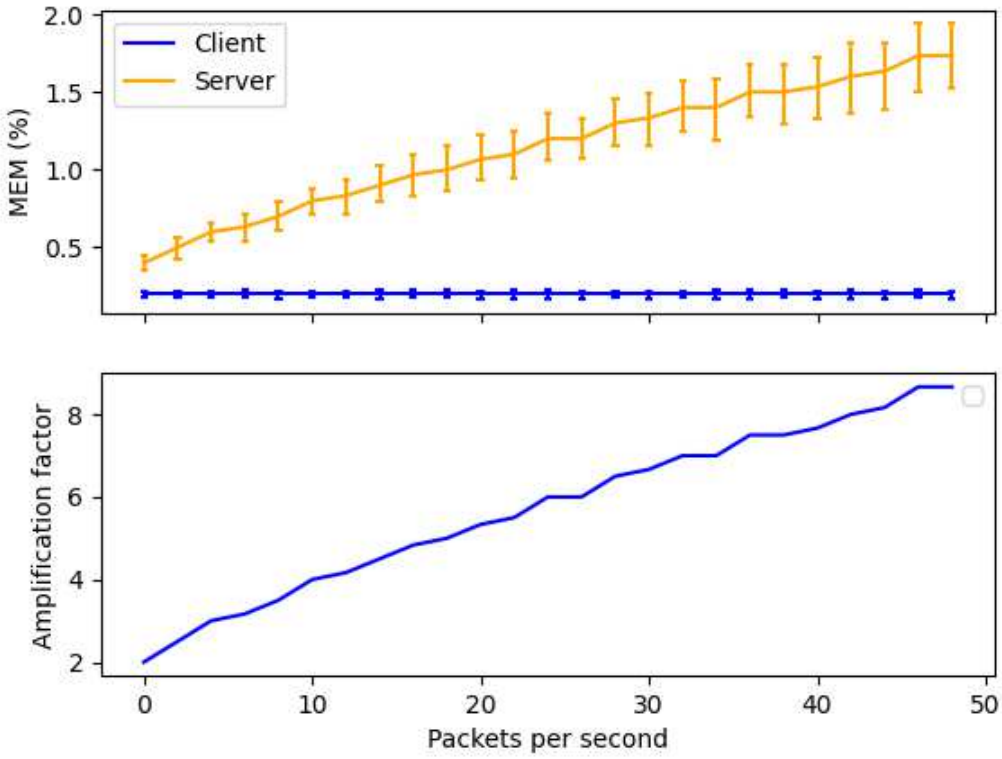


Figure 3.6: QUIC Memory Usage and Amplification Factor (*aioquic* implementation)

3.4 Bandwidth

TCP SYN Flooding could be a threat for the stateful TCP servers in terms of memory because it stores the Sequence Number to verify connections. However, it can also be the target of attacks on the bandwidth. This experiment was conducted to monitor the bandwidth usage under flooding attacks with QUIC and TCP SYN Cookie. As shown in Figure 3.7, for both protocols, the incoming and outgoing traffic are close. This is mainly due to the design of both protocols to ensure a low bandwidth amplification factor with symmetrical sized packets.

Following the previous experiment protocols, we isolated the bandwidth of the process from the base usage. We can observe a more significant bandwidth consumption for QUIC than for TCP SYN Cookies. This can be explained by the numerous fields in a QUIC packet. As explained in Section III, the QUIC Handshake contains the TLS one. The key exchange is done during the 1-RTT handshake with the server certificate embedded in the *Server Hello* message.

The symmetrical aspect of these designs ensure a bandwidth amplification factor of 1. The client sends a packet with some padding and the server has to send the exact same amount of data by padding the packet as much as necessary. This feature slightly affects performance but significantly reduces the vulnerabilities regarding bandwidth.

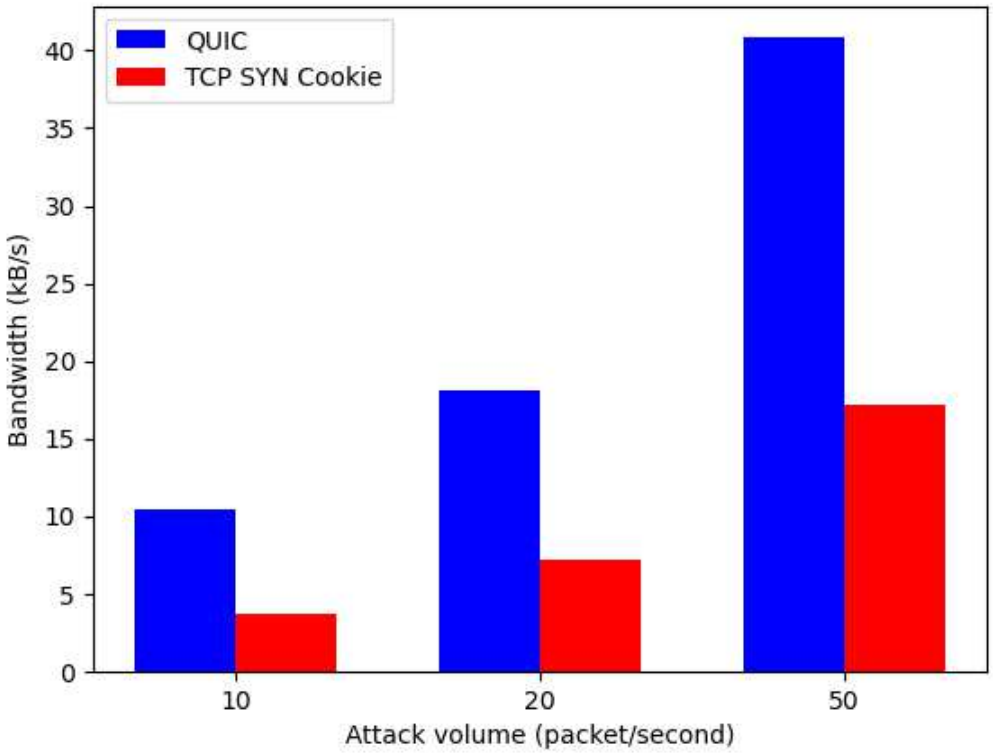


Figure 3.7: Bandwidth usage comparison (*aioquic* implementation)

3.5 Other implementations

In order to solidate the results, we conducted the same experiment on multiple implementations. It was to understand whether the problem is indeed coming from the QUIC design or it was from the implementation we were using in the first experiment.

The first implementation used was *aioquic*, it is written in Python, therefore the code execution is not very efficient. However, the cryptographic computations are done by a C library called in the code to make the encryption and decryption processes faster. The cost of interpreting the code in Python is higher, which explains the higher CPU usage for both Client and Server at a relatively low attack rate.

The second implementation was conducted on *quic-go* [28], written in Golang. It is lower-level

and more efficient than Python. In opposition to Python, it is compiled and not interpreted for higher performances. The experiment process was the same as in Section 3.2.1. The laptop and the Raspberry were interchanged between each row of the experiment and the attack rate was varying from 1 packet per second to the maximum amount until the server CPU reaches 100%. The increase in CPU usage is approximately linear with the attack rate as shown on Figure 3.8.
As we can see on Figure 3.9, the attack volume is higher, it is due to the efficiency of the implementation. The amplification factor obtained is lower ($\sim$2.9) but still significant enough to be a vulnerability. This difference is mainly explained by two factors. First, the *quic-go* implementation presents a more performant architecture. But also, as *Go* is a compiled language, there is no interpretation cost like the one in Python.

The last implementation chosen was *picoquic* [26]. Written in C, this implementation is widely used on the web. The C language is more efficient than *Go* and provides better performance. The language itself is very efficient but the well designed code architecture also makes this implementation faster. It made it possible for our experiment to reach high attack rates such as 10000 packets per second before reaching too high CPU as we can see on Figure 3.10.
The results in terms of amplification factor are centered around 2.2, see Figure 3.11.

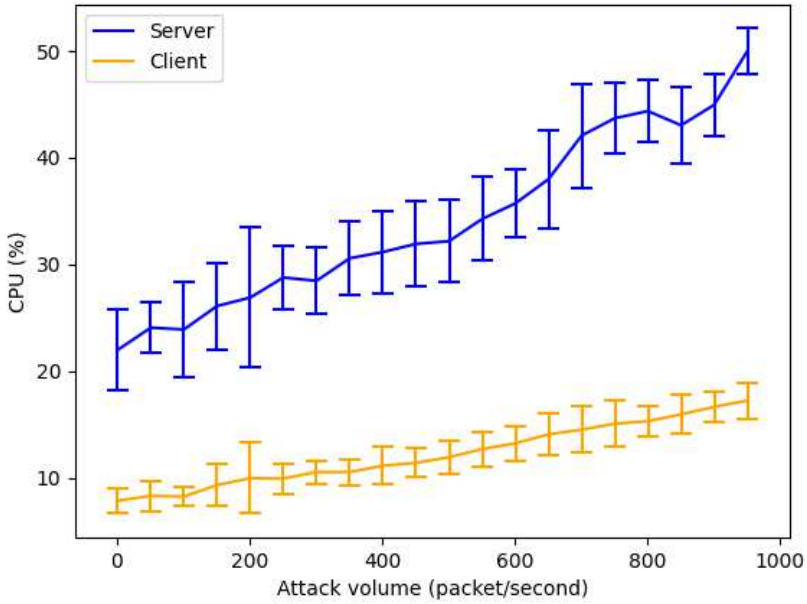


Figure 3.8: CPU consumption in *quic-go*

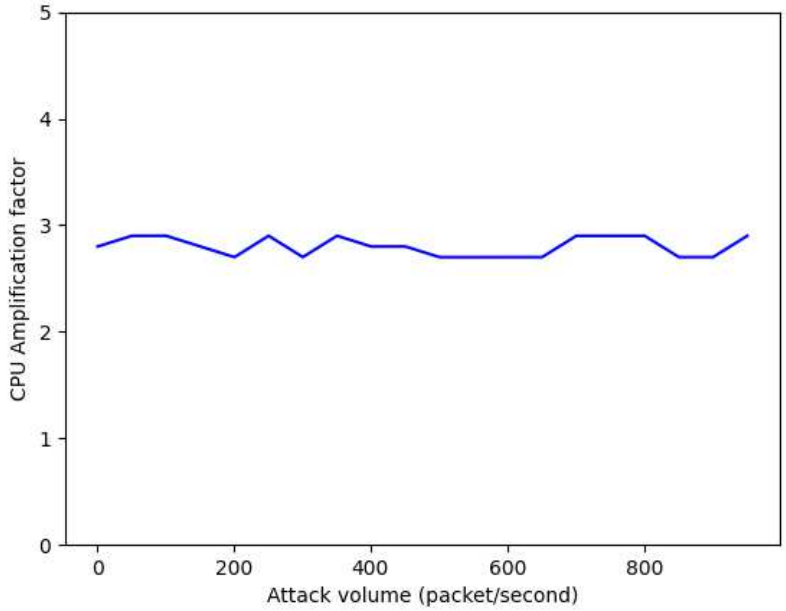


Figure 3.9: Amplification factor in *quic-go*

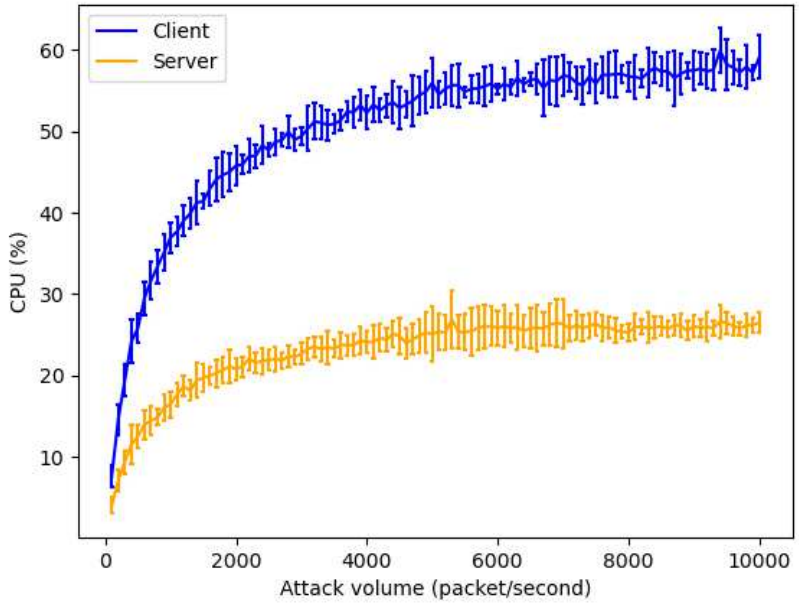


Figure 3.10: CPU consumption in *picoquic*

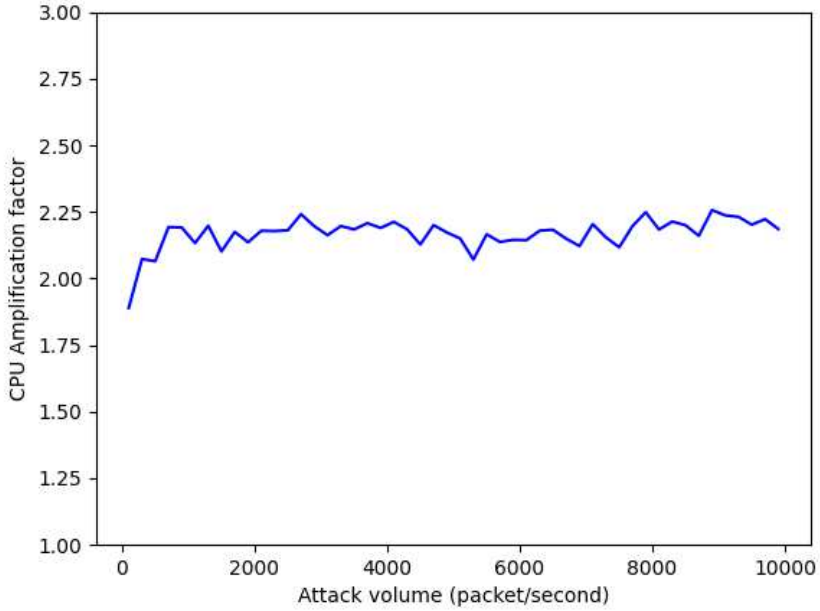


Figure 3.11: Amplification factor in *picoquic*

3.6 QUIC-Flooding with Early Data

Due to the 0-RTT feature of QUIC, an attacker may be able to use the Early Data packets to increase the load on the server side. For example, the attacker can paste some random bytes in the content without any encryption, and the server must decrypt the early data, which is more resource intensive. We designed this experiment to examin how much the random early data effects the amplification factor. This experiment was done with the *aioquic* implementation on a laptop, and it was repeated 20 times. The attacker added random bytes in the early data packet to approximately fill it to the maximum size of 1500 bytes. Figure 3.12 shows the result of this experiment. When the attacker adds an Early Data packet, the average amplification factor is 4.9, with the 95% confidence interval. In comparison, it is 4.0 for the basic attack. This increase is mainly due to the decryption cost of the packet. The server also returns an error as he could not process the content of the payload. As a conclusion, the attacker is able to further increase the CPU amplification of the attack by 22.5% through random early data injection.

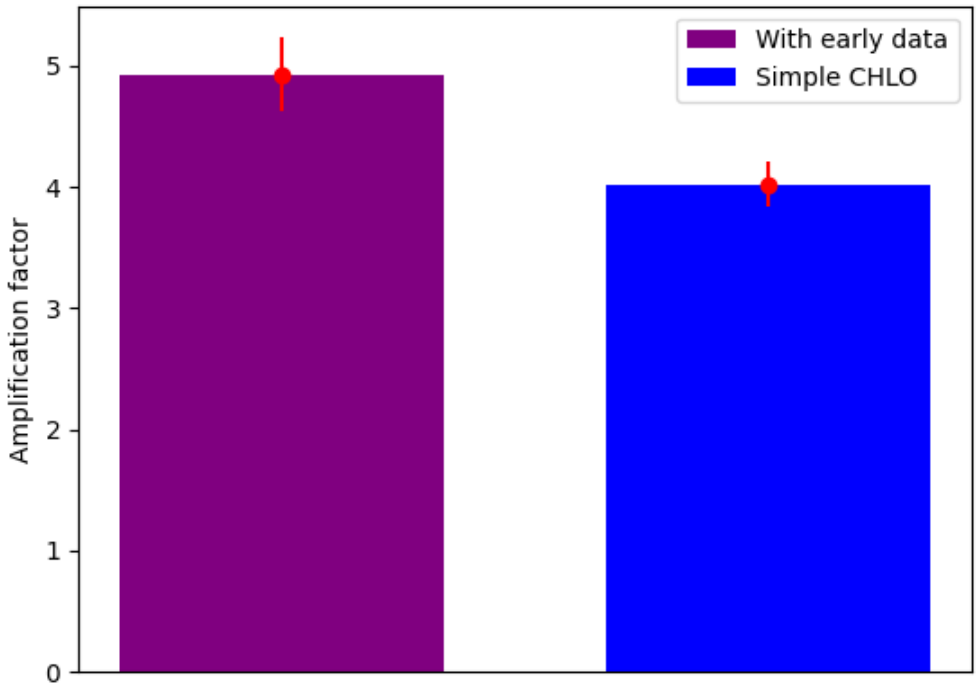


Figure 3.12: Amplification factor with and without Early Data

3.7 Informal analysis of the attacker

The attacker aims at overwhelming the server's resources. From the results of Section 3.2, we saw that the CPU is the bottleneck during this attack. Therefore an attacker will try to increase the CPU as much as possible for the server. A way to do that is to increase the amount of decryption operations needed. An attacker will then use the early data feature of QUIC to add some decryption during the handshake phase.

With slight modifications, some mechanisms which were designed to detect SYN-Flooding attacks might be triggered by this type of attack. In order to prevent that, the attacker can send invalid FIN packets to the server. However, the attacker cannot forge valid FIN packets without the encryption keys calculated using the information contained in the SHLO.

An attacker could also want to use the first phase of the handshake when the token is computed as

there is no address validation yet. It would allow him to use spoofed IPs and potentially increase the attack volume. However this first phase was designed to be very efficient, it should not cost an unreasonable amount of resources because the client is not authenticated. Therefore the calculations are very similar to the ones in TCP SYN Cookies. The amplification is not high enough to be interesting for an attacker to leverage.

3.8 Discussion

The experiments showed significant differences from one implementation to the other. First the maximum attack rate is higher for implementations using a less efficient language. As expected, *aioquic* using Python, can process considerably less requests on the server side than the implementations using lower level languages. *Picoquic* on the other hand is the most efficient implementation. The raw CPU usage for an attack rate of 50 packets per second can be seen in Table 3.1.

Implementation	Server CPU Usage (R=20)	Server CPU Usage (R=50)
aioquic	31.5%	69.9%
quic-go	21.9%	24.1%
picoquic	2.1%	3.8%

Table 3.1: Implementations CPU usage comparison

In addition to the number of requests the server is able to process, the amplification factor is also impacted by the efficiency of the implementation. As the server will have more operations to do than the client, the efficiency of those is vital. In *picoquic*, the cryptographic computations for the Diffie-Hellman keys are done in C inside the code directly, this very efficient way to proceed allows the server to contain the amplification factor at 2.25. In *quic-go* the cryptographic operations are also done in *Go*, the impact on the amplification factor is significant as it reaches 2.9. But for *aioquic*, as the Python language is not powerful enough for cryptographic computations, the implementation calls an external C library to do those. The calling of the external library and the communication time between it and the main code also explains the performance issues. The results can be seen

in Table 3.2. The different amplification factor curves can be explained by the base usage ratio. In aioquic and picoquic the base usage amplification factor is low. Therefore when the attack rate is low and the base usage is predominant, the amplification factor is lower. When the attack rate increases, the base usage becomes less important than the attack usage and the amplification factor converges to the one of the attack itself. In quic-go, base usage ratio is close to the amplification factor of the attack explaining the approximately constant amplification factor on figure 3.9.

Implementation	Max amplification factor	Corresponding attack rate (packets per second)	Language
aioquic	4.5	50	Python
quic-go	2.9	800	Golang
picoquic	2.25	8500	C

Table 3.2: Implementations amplification factor comparison

Chapter 4

QUICShield : Handshake flooding attack detection mechanism

The goal of this chapter is to design, implement and test a detection mechanism against Handshake flooding attacks. It is organized as follows : Section 4.1 presents the challenges our mechanism has to address and briefly describes the solutions chosen, Section 4.2 details the design of the different parts of the mechanism including storage and change-detection. The content of this chapter was published as a paper in VCC Conference 2023 [29].

4.1 Objectives

The aim of this work is to develop an efficient change-detection based defense mechanism. Existing research has explored various techniques to address the problem of DDoS attacks [3]. Many machine-learning based methods are designed but they are usually computation intensive and require a lot of training data in order to be accurate. However, QUIC is still recent and it is too soon to have relevant datasets yet. That is why this work is a statistical approach, it is more lightweight in terms of computation and storage. One prominent approach is using Bloom Filters for data storage and implementing the CUSUM (Cumulative Sum) algorithm for change detection [4]. These methods have been successfully applied in detecting TCP SYN-flooding attacks. However, the traditional CUSUM algorithm requires prior knowledge of the probability distributions of the system's

normal behavior and the one during an attack, which may not be accurate or available [30]. Additionally, the existing techniques can not be directly applicable to QUIC-Flooding Attacks due to the differences in the packet structure and processing requirements of the QUIC protocol.

To overcome the limitations of existing methods, we propose a mechanism incorporating a modified Bloom Filter for efficient data storage and the Generalized Likelihood Ratio (GLR) CUSUM algorithm for change detection [31]. Our method does not need any pre-existing information about probability distributions since the GLR-CUSUM algorithm can adjust itself to the data it processes by employing a set of exponential functions [32]. This special feature makes our QUICShield approach more versatile and resilient in identifying QUIC-Flooding Attacks in different situations where IP addresses are spoofed.

The attacker can use CHLO and FIN (Finish) packets as they are both encrypted with a publicly available key. However, to be valid, the FIN packet has to consider the Encrypted Extensions and the certificate exchanged in the SHLO packet. Therefore, if the attacker sends randomly crafted FIN packets, they will be seen as invalid for the server. However, even if the server rejects those packets, the cryptographic calculations still have to be done. To be accurate, in QUIC there is not a specific packet design for FIN, it is added as a flag into the first data packet. However, the concept is the same and the mechanism works in the same way.
In this attack model, the attacker sends a lot of CHLO and invalid FIN packets to the server. He can also send valid ones and start legitimate connections at any moment to mislead some defense mechanisms. From the attacker's perspective, the main difference with an SYN-Flooding attack is the fact that an invalid FIN actually increases slightly the load on the server's side. FIN packets are sent mainly in order to avoid detection mechanisms. The attacker's behavior is shown on Figure 4.1

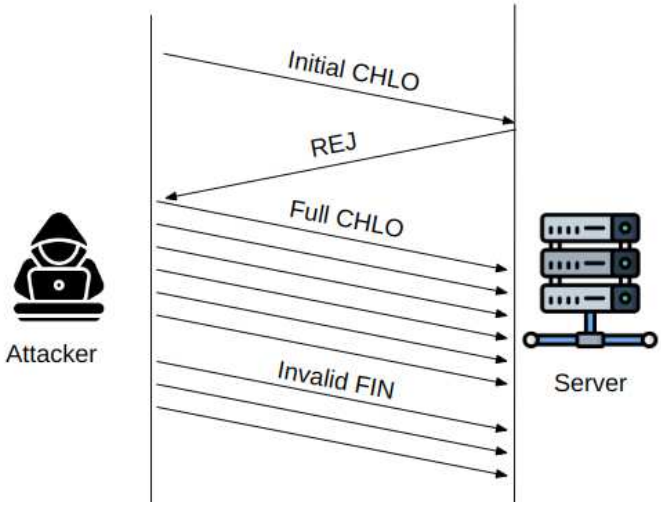


Figure 4.1: QUIC Handshake Flooding Attack

4.2 QUICShield Design

4.2.1 Storage structure

In order to detect DDoS attacks, some connection data has to be stored. It is important to do so efficiently, which is why we opted for a modified Bloom Filter. The output of the filter is a vector v of length m. It will be referred to as Bloom Table later in this paper. For each incoming packet, the source IP address of the packet is hashed into a number corresponding to the index of the Bloom table.

$$h: \begin{cases} [\![0; 2^{32} - 1]\!] \to [\![0; m - 1]\!], \\ a \mapsto h(a) \end{cases}$$

However, the regular Bloom Filter is used to detect if an element is a part of a set, i.e. if the IP address is already present in the table. The version we use hashes the packet's source IP a into $h(a)$ and increments v at the index $h(a)$. Therefore the cell's value is not zero or one as in the usual one but an integer corresponding to the number of times an IP was hashed into this index. Due to the structure, some addresses are hashed into the same index; however, an address always increments the same cell.

We distinguish between initial and final packets, namely CHLO and FIN packets, for QUIC. CHLO

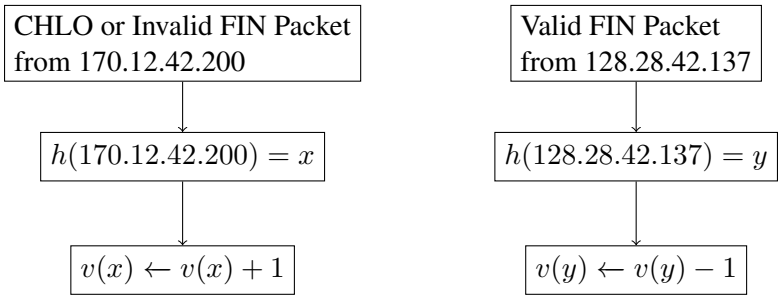


Figure 4.2: Bloom Table incrementation

packets increase the value of the cells, whereas valid FIN packets decrease them. During a QUIC-Flooding attack (similar to SYN-flooding for TCP), an attacker sends numerous CHLO packets without finalizing any handshake, causing the counters to increase. However, legitimate incomplete handshakes exist, which is why we need to set a maximum value for each cell and reset the index if it exceeds this limit. We also need to be cautious about crafted FIN packets. In TCP SYN-flooding, the attacker overwhelms the server's storage with sequence numbers, so he has no interest in sending fake ACK packets as it would reduce the amplification factor of the attack for no gain. However, in QUIC, the threat is inherent in the computations used to process and validate the packets, and an attacker can overwhelm the server's CPU or RAM. The server performs cryptographic computations on every packet to verify its validity. An attacker could therefore send CHLO packets and fake FIN packets to double the server's workload and appear legitimate in the Bloom Table. To avoid this issue, instead of decrementing the counter for invalid FIN packets, it is incremented.

By using this structure, it is possible to mitigate an attack. If an attack is detected on one cell of the Bloom Table, the defense would be to drop the packets coming from IPs that are hashed into the index of this cell. Therefore it is important to choose m large enough to avoid too many collisions as it would block a proportion $\frac{1}{m}$ of clients.

To analyze this data structure's space complexity, we assume it is implemented in C (which is the case in our experiment). Every element of v is an int encoded on 4 bytes, so the space needed for the table is 2^{m+2} bytes.

A forgetting factor α is introduced to make sure the table does not flood the memory of the system and the past attacks have no incidence on the detection system. Every second, each cell of

the table is flushed by a certain amount.

$$\forall k \in [\![0; m-1]\!], v_t(k) = v_{t-1}(k) \times \alpha$$

If α is high, the cells will keep most of their values. On the contrary, a small α will flush the table faster. In addition to the impact on storage, these parameters affect the detection rates.

4.2.2 Change-detection mechanism

The change point detection mechanism [33] aims to discern the change in a variable's distribution over time. *Wei Chen* and *Dit-Yan Yeung* [34] introduced the proof of concept for TCP SYN-Flooding Attacks. However, they made the assumption that the probability distributions of the variables in usual behavior and during an attack were known. This is sufficient for SYN-Flooding Attacks because the packets crafted by the attackers are all the same and impact the Bloom Table similarly. This assumption requires a certain amount of attack data to ensure the accuracy of the probability distribution chosen. However large datasets of SYN-Flooding attacks allow the mechanism to use an accurate distribution by knowing the behavior of the attackers.

Suppose $X_k, k \in [0; m-1]$ the sequence of independent random variables corresponding to the value in each cell of the Bloom Table. Independence is ensured by the properties of the hash function used to fill it. Define f_0 and f_θ, respectively the probability density function of the cells before and after the Change-Point, i.e. in the usual behavior of the system and during an attack. As the exact probability distribution during an attack is not assumed to ensure more flexibility for new attacks, it is approached by a one-parameter exponential family of functions in Θ.

$$\forall x \in \mathbb{N}, f_\theta(x) = 1 - e^{-\theta x}$$

The range of value for θ is chosen manually in line with the computing power of the machine the algorithm runs on. The granularity between two values of theta also has to be selected carefully to keep a good accuracy without doing useless calculations. If θ is chosen too small, it will increase the false negative rate. On the contrary, if it is too high, the rate of false positives is going to be

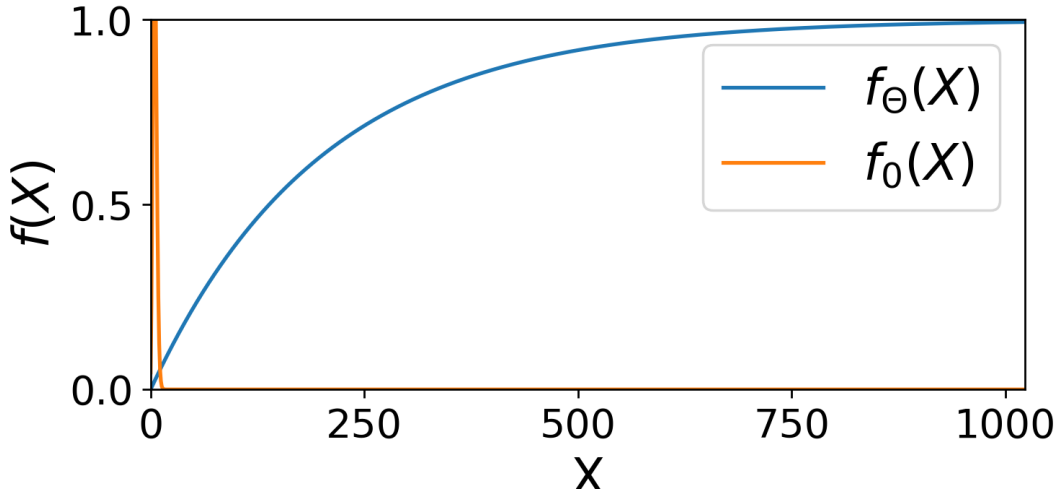


Figure 4.3: Probability of an attack based on cell value (with $y = 500, p = 0.01, \theta = 0.005$)

elevated. Note that $\Theta \subset \mathbb{R}_+$, in order to ensure $\forall x \in \mathbb{N}, f_\theta(x) \in [0; 1]$. Under normal behavior, the value of each cell is modeled by a binomial law of parameters y and p where y is the number of packets hashed into the address of the cell and p the legitimate incomplete handshake rate, i.e. legitimate incomplete handshakes over the total number of handshakes. Therefore at each step :

Let $p \in]0; 1[$,

$$f_0(x) = \binom{y}{x} p^x (1-p)^{y-x}$$

The probabilities are shown on figure 4.3. When the counter in the cell is low, the system is likely under normal behavior. The reason the counter is not at 0 exactly can be because of failed or ongoing handshakes.

$$l(X) = \sup_{\theta \in \Theta}(\log(f_\theta(X)/f_0(X)))$$

is known in CUSUM as log-likelihood ratio. While the behavior of the system is normal, X likely follows f_0 so $f_0(X) > f_\theta(X)$ and $l(X) < 0$.

$$S_n = \sum_{k=0}^{m-1} l(X_k)$$

If the probability density functions are accurate, S_n has a negative drift under usual behavior and a positive one during an attack. n represents the step of calculation. If the time between two values of

n is small, the system will be more reactive but it will compute more often and therefore use more resources. On the contrary, if the time between two steps is larger, the system will be lighter but less reactive. CUSUM detection compares the difference between S_n and its minimum to a threshold. It can be summarized as :

$$W_n = S_n - min(S_n)$$

A characteristic of an efficient DDoS detection mechanism is the time complexity. If it requires too many calculations, it risks overwhelming a server which is already under load because of the attack. The interesting property of this mechanism is that it can be computed recursively. Using GLR-CUSUM [33], the objective is to approach the real distribution during an attack by looking for a supremum of W_n with $\theta \in \Theta$. However, this method which cannot be computed in recursive time, is not desirable for our use. The window-limited approach consists in considering only a relevant part of the function's family set by choosing bounds for the parameter. Once the bounds are determined, the granularity can be chosen. This method reduces significantly $\#\Theta$ and therefore the time needed for the calculations.

Algorithm 1 shows the calculations done at each timestep of the process.

4.3 Experimental results

We evaluated QUICShield on our implementation, it was written in C. In this section, we will present the experimental setup and the evaluation results.

4.3.1 Experimental setup

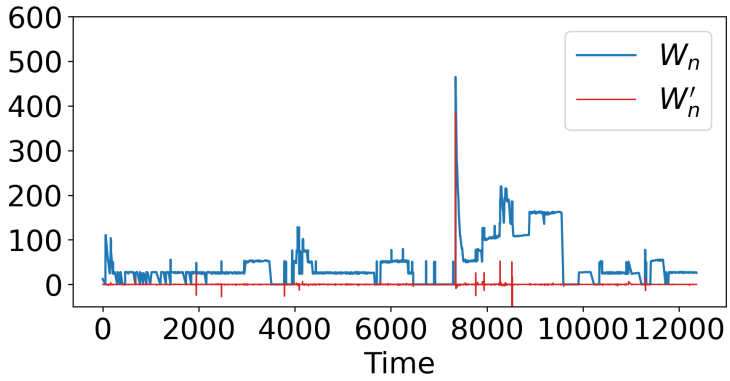
To evaluate the effectiveness of our proposed mechanism, we implemented a QUIC server and a QUIC client on our lab machines. Our QUIC server implementation is based on AIOQUIC [25]. We modified their QUIC client by adding the same DDoS attack function used in chapter 3. We implemented QUICshield using the C language. A Python script is used to filter the incoming packets and send the relevant packets (CHLO or FIN) to QUICShield. This method allows us to emulate a real attack and test our algorithm in situations close to real-life deployment. In order to evaluate the complexity of our solution, we run the QUIC server and QUICShield on a RaspBerry

Algorithm 1 QUICShield process

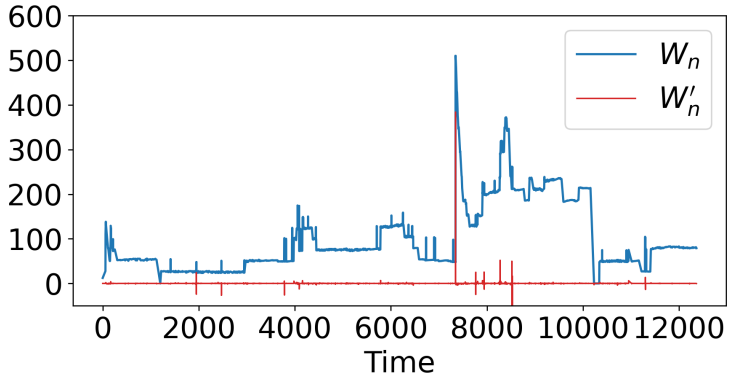
$v \leftarrow Array\ of\ integers\ of\ size\ m$
$n \leftarrow 0$
$S_n \leftarrow 0$
$W_n \leftarrow 0$
while System is active **do**
  for each $packet$ received since last timestep **do**
    $i \leftarrow h(packet.address)$
    if $packet.type$ is $CHLO$ or invalid FIN **then**
      $v[i] \leftarrow v[i] + 1$
    end if
    if $packet.type$ is valid FIN **then**
      $v[i] \leftarrow v[i] - 1$
    end if
  end for
  for i in $[\![0; m-1]\!]$ **do**
    $l \leftarrow \sup_{\theta \in \Theta}(\log(f_\theta(v[i])/f_0(v[i])))$
    $S_n \leftarrow S_n + l$
  end for
  $W_n \leftarrow S_n - min(S_n)$
  for i in $[\![0; m-1]\!]$ **do**
    $v[i] \leftarrow v[i] \times \alpha$
  end for
  $n \leftarrow n + 1$
end while

PI machine and the QUIC attack client on a laptop.

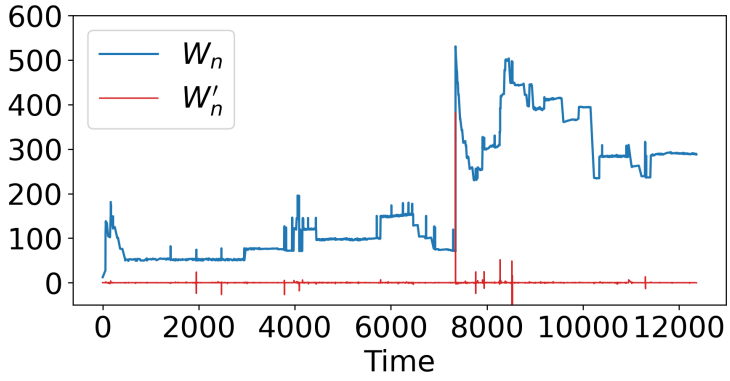
We also use a DARPA dataset containing SYN-Flood attacks that we modified to convert it to QUIC-Flood attacks in order to see how the system reacts to larger scale attacks.



(a) $\alpha = 0.98$



(b) $\alpha = 0.99$



(c) $\alpha = 0.995$

Figure 4.4: Outputs of the DARPA dataset with different forgetting factor α

4.3.2 Output under DDoS attack

We evaluate the effectiveness of QUICShield on DARPA DDoS data. Fig 4.4 shows W_n and its derivative over time of the DARPA dataset under different α values. In this data set, an attack is occurring at $t = 7350s$. A significant rise happens to W_n at this time. It is highlighted by the bump on W'_n. A threshold should be chosen to detect unreasonable high values of W_n. The forgetting factor α impacts the time necessary for the system to return to normal after an attack. We can observe that with high values of α, W_n is decreasing slower. However, too low values of α will make the system insensitive to attacks as the maximum value of W_n might never exceed the detection threshold chosen.

4.3.3 Selection of parameters

The maximum value of W_n is a function of α and the attack rate R. R is the number of attack packets per second. If we model the value inside the cell as this following sequence :
Let $\alpha \in]0;1[, R \in \mathbb{N}$

$$u_0 = 0 \; and \; \forall n \in \mathbb{N}, u_{n+1} = (u_n + R) \times \alpha$$

Let $n \in \mathbb{N}$, this sequence being arithmetico-geometric, we obtain :

$$u_n = \alpha^n \times (u_0 - \frac{\alpha R}{1-\alpha}) + \frac{\alpha R}{1-\alpha}$$

$$\Leftrightarrow u_n = \frac{\alpha R}{1-\alpha} \times (1 - \alpha^n)$$

Therefore:

$$u_{n+1} - u_n = \frac{\alpha R}{1-\alpha} \times (1 - \alpha^{n+1}) - \frac{\alpha R}{1-\alpha} \times (1 - \alpha^n)$$

$$\Leftrightarrow u_{n+1} - u_n = \frac{\alpha R}{1-\alpha} \times (\alpha^n - \alpha^{n+1})$$

As $\alpha < 1$, $\alpha^n < \alpha^{n+1}$ and $\frac{\alpha R}{1-\alpha} > 0$. Therefore :

$$u_{n+1} - u_n > 0$$

Hence (u_n) is increasing and :

$$\sup_{n\in\mathbb{N}} u_n = \lim_{n\to+\infty} u_n = \frac{\alpha R}{1-\alpha}$$

We can now compute the probability functions for this value :

$$\forall\theta \in \Theta, f_\theta = 1 - e^{-\theta\times\frac{\alpha R}{1-\alpha}}$$

In the worst case scenario, during an attack, the server only receives attack packets at a rate R and $y = x$. Two cases might happen; the attacker can use his own IP to launch the attack or he can use botnet or spoofed IPs to avoid detection. Note that if the address validation mechanism is enabled on the server side, the attacker will not be able to spoof IP addresses and will have to use bots.

If he is using his own IP, the same cell is incremented for each packet received and the other cells stay at 0, therefore :

Let $\alpha \in]0;1[, R \in \mathbb{N}, p \in]0;1[$

$$f_0 = \binom{\frac{\alpha R}{1-\alpha}}{\frac{\alpha R}{1-\alpha}} \times p^{\frac{\alpha R}{1-\alpha}} \times (1-p)^{\frac{\alpha R}{1-\alpha}-\frac{\alpha R}{1-\alpha}}$$

$$\Leftrightarrow f_0 = p^{\frac{\alpha R}{1-\alpha}}$$

Therefore, we get the maximum value possible for W_n:

$$W_{max\_unspoofed}(\alpha, R) = \log \frac{1 - e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}}{p^{\frac{\alpha\times R}{1-\alpha}}}$$

However, if he is using spoofed IPs or botnets, by the properties of the hash functions, the cell index incremented for each packet will follow a uniform distribution. The value inside each of the cell is then $\frac{\alpha R}{m(1-\alpha)}$

Therefore, for each cell we get :

$$f_0 = \binom{\frac{\alpha R}{m(1-\alpha)}}{\frac{\alpha R}{m(1-\alpha)}} \times p^{\frac{\alpha R}{m(1-\alpha)}} \times (1-p)^{\frac{\alpha R}{m(1-\alpha)} - \frac{\alpha R}{m(1-\alpha)}}$$

$$\Leftrightarrow f_0 = p^{\frac{\alpha R}{m(1-\alpha)}}$$

Then :

$$W_{max}(\alpha, R) = \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}}$$

As :

$$m \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}}}{p^{\frac{\alpha \times R}{m(1-\alpha)}}} \leq \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}}$$

The proof can be found in Appendix C.1. The highest anomaly score is obtained when the attacker is using unspoofed IP. It can also behave like this if the attacker is using a botnet and spoofing all the source IPs to the same value.

To keep a low false positive rate from normal unfinished connections, we should choose a higher R value (i.e. the maximum the system can handle without deterioration). Fig 4.5 represents the maximum value of W_n depending on α for different attack rates. Regarding the α value, we should choose the largest α (most sensitive one) so that W_n does not risk memory overflow under maximum attack the system is going to face. For the threshold, we should set it at the value of W_n under the minimum attack rate we want to detect.

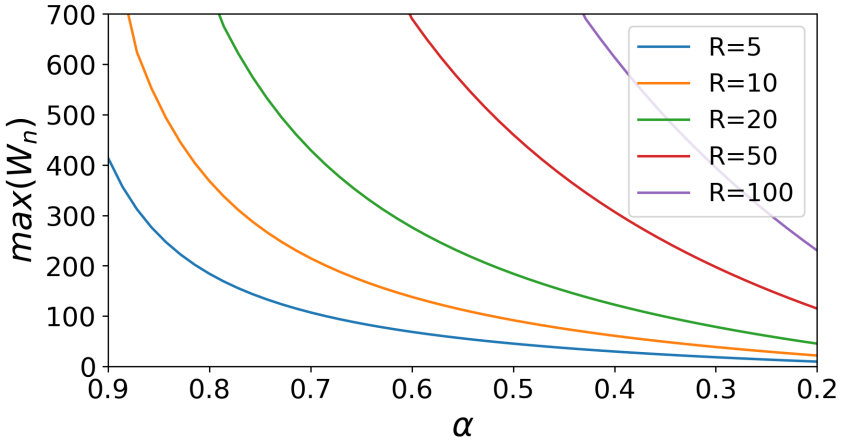


Figure 4.5: Maximum value of W_n depending on α

4.3.4 Behavior under different attack rates

In this experiment, we configure our attack client with different attack rates and $\alpha = 0.85$. α is chosen lower than in the previous experiment because R is higher compared to the attacks rates from the DARPA dataset. Fig 4.6 shows the reaction of the mechanism under different attack rates. There, an attack occurs for $t \in [2000; 3000]$. W_n stabilizes after a quick growth at the beginning of the attack. It also decreases quickly once the attack is finished. The maximum value reached by W_n reflects the attack rate and corresponds to the maximum value determined analytically in 4.3.3.

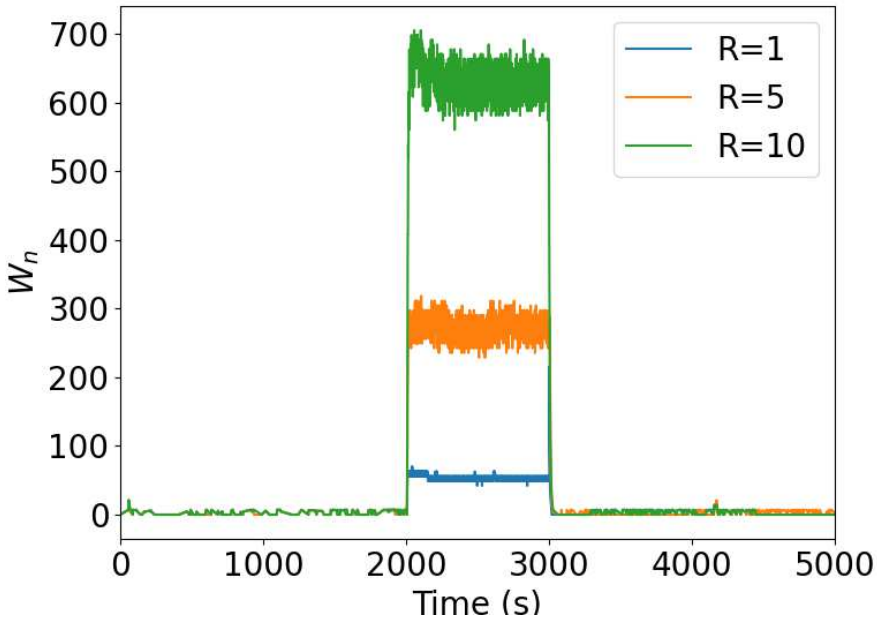


Figure 4.6: Reaction of the mechanism under different attack rates

4.3.5 Complexity analysis

In this subsection, we conduct an analysis on the computation and space complexity of QUIC-Shield. The first step of the algorithm is formatting the packets. It only consists of simple operations for every packet, therefore it has a complexity of O(R) where R is the attack rate. The next step is to hash each packet. Insertion into a hash table is O(1) in time, therefore for R packets we have O(R) time complexity. Since the table has a constant size, its space complexity is O(m). GLR CUSUM computes each cell $|\Theta|$ times. Therefore the time complexity for one execution is O($m * |\Theta|$). Note

that this part does not depend on the number of packets received.

Figure 4.7 is obtained by measuring the CPU usage of an Aioquic server under two attack rates (R=10 and R=20), and the consumption of various CUSUM and GLR-CUSUM settings. We observe that the resource consumption of the mechanism is low compared to the consumption of the QUIC server. Also, GLR-CUSUM only slightly increases the CPU usage compared to conventional CUSUM. As explained previously, the resource consumption of QUICShield increases slower than that of the server.

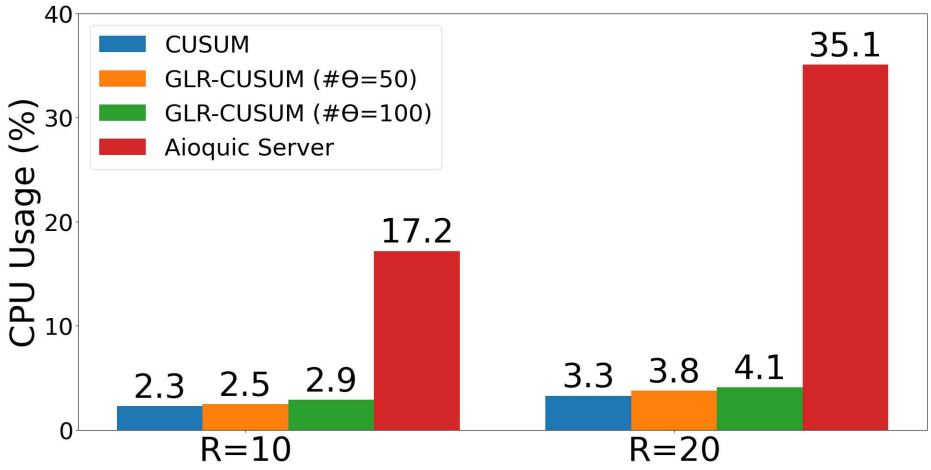


Figure 4.7: Comparison of CPU load for various Θ under different attack rates

Chapter 5

Conclusion

5.1 Work summary

This work mainly addresses the topic of QUIC-Flooding attack and proposes a mechanism to detect those malicious acts.

The evaluation consists in measuring the resource consumption of both the attacker and the victim during an attack. Multiple metrics are monitored such as CPU usage, RAM usage, and bandwidth to observe the behavior of the system under attack. The experimental results show that the CPU is the main bottleneck during QUIC-Flooding attack with an amplification factor varying from 2.2 to 4 depending on the implementation. It is relevant with the theoretical analysis of the attack as the victims have to compute more cryptographic objects than the attacker. Memory usage amplification is high but since the actual usage is very low, it does not represent a danger on the server side. Bandwidth is symmetrical due to conception choices, therefore reflection attacks as we could observe in SYN-Flooding are not possible with QUIC. Three implementations were tested : *aioquic*,*picoquic* and *quic-go*, they were chosen because they are widely used and written in different languages. The different implementations are performing differently due to the performance inherited from the language used as well as from the code design and architecture.

This work introduces QUICShield, a mechanism that detects QUIC-Flooding attacks. It relies on a modified Bloom Table for the storage and uses the GLR-CUSUM statistic method to detect anomalies. Our theoretical analysis explains how the mechanism works in details and proves how it works

in the worst case scenario. Both the simulated and emulated experiments show that it is able to detect the incoming threats accurately. The resource consumption of the mechanism is low compared to the one of the server itself. This is important to avoid adding unreasonable load on a server under attack. We also explain how to choose the parameters of the mechanism appropriately.

5.2 Study limitations

This study's experiments were run on common hardware such as laptops or small size servers. Therefore, the maximum attack rate that was achieved is small compared to the one a real attack would lead to damage large infrastructures. Larger servers may have more optimized hardware chips that compute cryptographic operations more efficiently. This would reduce the amplification factor and make it less interesting for an attacker as he would have to deploy more resources to lead the attack successfully.

Also, even if the implementations pretend to be compliant with the multiple RFCs [17] [18] [19], there is no official acknowledgement of this. The architecture of the code is complex and the size of the code prevents checking for compliance easily. However, to ensure some universality, the implementations we experimented on provide some interoperability tests. If the different implementations are able to communicate with one another without encountering errors, it probably means they abide by the RFC.

5.3 Discussion and future work

In our experiments, CPU was the resource that was drained under QUIC Flooding Attack. It is intuitive as the server needs a lot of computations to handle the packets according to the process described in RFC9000 [17]. It could cause complete service disruption as the server would not be able to process any other incoming packet and would not have enough resources available to run its other processes. The memory also presents a high amplification factor but the usage is much lower so that it is unlikely a threat for most of the systems. Moreover, adding more CPU typically costs more than adding memory. The bandwidth use is not a problem with these stateless protocols as they were designed with security in mind to ensure neutral amplification factor. In addition to that,

the stateless design of the QUIC protocol avoids the full backlog problem encountered during the TCP SYN flood attack to stateful TCP servers.

The 0-RTT feature of QUIC is a good way to enhance the performance of the server as it significantly reduces the amount of handshakes necessary. 0-RTT is always optional for the client and he can always request a new handshake if he wants. Therefore it does not change the attacker perspective who can still lead the same attack. However, it allows the server to detect an attack more easily. Indeed, a client repeating multiple complete 1-RTT handshake in a row while 0-RTT is available would be suspicious. If a client has already done the handshake recently, he still has the certificate and 1-RTT handshake would only represent a loss of performance and additional latency. Nevertheless, 0-RTT exposes the server to other vulnerabilities inherent to this feature.

Even though QUIC has multiple RFC documents [17] [18] [19] to provide guidelines, there are multiple different implementations with very different architectures and designs. The results of our experiments show that the efficiency of the process is very different in the variety of implementations An attacker can take down an infrastructure which has 4 times its own computing power. This is a serious issue which was not present in TCP SYN Cookie. The resilience of TCP SYN Cookie is mainly due to the simplicity of its design. The server only computes the cookie and compare it to the one from the packet. There are no additional mechanisms. TCP SYN Cookie [24], on the other hand seems to have only one legitimate version, the one of its designer written for Linux and adapted for other systems. It makes the deployments of this protocols more homogeneous.

The QUICShield design allows a fast and accurate detection as verified in our previous experiments. However, it highlighted the importance of assessing the attack rate expected and the setting of the parameters. As a future work, evaluating the effectiveness of the mechanism with different families of functions would increase its robustness and versatility. Although the detection is accurate in the experiments we led, the limitation of the design at this point is that it does not include solutions for mitigation. A mitigation solution based on the value inside the counters of the Bloom table could be designed. Actually if an IP address is hashed into the index of a cell with a high counter, the packet could be rejected. It would imply a false positive rate of $\frac{1}{m}$ and inability to access the server by these legitimate clients.

Even if the detection mechanism is performing well, the ideal solution would be on the protocol

level. It would need some modifications in order to ensure additional computations on the client's side. The client would need to do as much computation as the server from the very beginning of the handshake. It could be achieved at the same time as the address validation with some cryptographic challenge response based on the value of the token for example. However, as the address validation is not by default enabled, if such solutions are deployed, this mechanism would need to be deployed more often. It would affect the performance, especially for resource-constrained clients. This trade-off between performance and security would need to be evaluated.

We also noticed that there is a guideline on how to set the threshold that decides if the address validation mechanism is enabled or not. It would definitely be interesting to have a good practice analysis on how to set that threshold based on the server's use and the tradeoff desired between security and performance.

Appendix A

List of Publications

Appendix A details the articles this work was published in.

- Benjamin Teyssier, Y A Joarder, and Carol Fung. ”An Empirical Approach to Evaluate the Resilience of QUIC Protocol Against Handshake Flood Attacks.” 2023 19th International Conference on Network and Service Management (CNSM). IEEE, 2023.
- Benjamin Teyssier, Y A Joarder, and Carol Fung. ”QUICShield: A Rapid Detection Mechanism Against QUIC-Flooding Attacks”. 2023 IEEE Virtual Conference on Communications (VCC). IEEE, 2023.

Appendix B

List of acronyms used

Appendix B details all the acronyms used in this work.

- IP : Internet Protocol
- UDP : User Datagram Protocol
- TCP : Transmission Control Protocol
- TLS : Transport Layer Security
- CPU : Central Processing Unit
- DDoS : Distributed Denial of Service
- CUSUM : Cumulative Sum
- GLR-CUSUM : Generalized Likelihood Ratio Cumulative Sum
- DNS : Domain Name System
- NTP : Network Time Protocol
- RTT : Round-Trip Time
- TCB : Transmission Control Block
- H : Siphash keyed hash function

- s_1 : Server's first secret key
- s_2 : Server's second secret key
- s_a : Source address of the packet
- s_p : Source port of the packet
- d_a : Destination address of the packet
- d_p : Destination port of the packet
- ISN_c : Initial Sequence Number from Client
- T : Timestamp
- MSS_i : Maximum Segment Size

Appendix C

Mathematical proofs

C.1 Maximum value of W_n

Let $m \in \mathbb{N}^*, \theta \in \Theta, \alpha \in]0;1[, p \in]0;1[, R \in \mathbb{N}^*$

We want to compare :

$$m \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}}}{p^{\frac{\alpha \times R}{m(1-\alpha)}}}$$

and

$$\log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}}$$

We solve :

$$m \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}}}{p^{\frac{\alpha \times R}{m(1-\alpha)}}} \geq \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}} \quad (1)$$

$$(1) \Leftrightarrow m \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}}}{p^{\frac{\alpha \times R}{m(1-\alpha)}}} - \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}} \geq 0$$

$$\Leftrightarrow \log \left(\left(\frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}}}{p^{\frac{\alpha \times R}{m(1-\alpha)}}} \right)^m \right) - \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}} \geq 0$$

$$\Leftrightarrow \log \frac{(1 - e^{-max(\theta) \times \frac{\alpha \times R}{m(1-\alpha)}})^m}{p^{\frac{\alpha \times R}{(1-\alpha)}}} - \log \frac{1 - e^{-max(\theta) \times \frac{\alpha \times R}{1-\alpha}}}{p^{\frac{\alpha \times R}{1-\alpha}}} \geq 0$$

$$\Leftrightarrow \log \frac{(1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}})^m}{1-e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}} \geq 0$$

$$\Leftrightarrow \frac{(1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}})^m}{1-e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}} \geq 1 \quad (2)$$

But we also have :

$$e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}} \geq e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}$$

$$\Leftrightarrow 1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}} \leq 1-e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}$$

And :

$$1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}} \in]0;1[$$

Therefore :

$$(1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}})^m \leq 1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}}$$

Then :

$$(1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}})^m \leq 1-e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}$$

We then have :

$$\forall R \in \mathbb{N}^*,\ m \log \frac{1-e^{-max(\theta)\times\frac{\alpha\times R}{m(1-\alpha)}}}{p^{\frac{\alpha\times R}{m(1-\alpha)}}} \leq \log \frac{1-e^{-max(\theta)\times\frac{\alpha\times R}{1-\alpha}}}{p^{\frac{\alpha\times R}{1-\alpha}}}$$

Appendix D

Measurement scripts

D.1 Memory and CPU usage over time

In order to determine the range of rates we would lead the experiment on, we ran a first experiment in which we increased the attack rate over time until the server was overloaded. Once the server is at 100% of CPU usage, we determined it was the maximum attack rate for our measurment. It is important to note that others processes are running on the device. Therefore the maximum CPU in the figures is less than 100% because it only considers the usage of the QUIC process.

```
#!/bin/bash
for RATE in 1 10 20 50
do
    SECONDS=0
    python3 examples/http3_server.py --certificate tests/ssl_cert.pem \
    --private-key tests/ssl_key.pem --port 4242 & srv_pid=("$!")
    python3 examples/amp_attack.py -r $RATE & cli_pid=("$!")
    while [ $SECONDS -lt 180 ]
    do
        result=$(date +%H:%M:%S)
        cpu_cli=$(ps -p $cli_pid -o %cpu)
        cpu_srv=$(ps -p $srv_pid -o %cpu)
```

```
        result+=","
        result+=$cpu_cli
        result+=","
        result+=$cpu_srv
        echo $result
        echo $result >> results/cpu_time/results_cpu_time_$RATE.csv
        SECONDS+=1
        sleep 1
    done
    pkill python3
done
```

D.2 Memory and CPU usage for different attack rates

```
#!/bin/bash
echo > results/results_quic.csv
for RATE in {1..50..1}
do
    python3 examples/http3_server.py --certificate tests/ssl_cert.pem \
    --private-key tests/ssl_key.pem --port 4242 & srv_pid=("$!")
    python3 examples/amp_attack.py -r $RATE & cli_pid=("$!")
    sleep 10
    result=$(date +%H:%M:%S)
    cpu_cli=$(ps -p $cli_pid -o %cpu,%mem | head -n 2 | tail -n 1)
    cpu_srv=$(ps -p $srv_pid -o %cpu,%mem | head -n 2 | tail -n 1)
    result+=","
    result+=$cpu_cli
    result+=","
    result+=$cpu_srv
```

```
    echo $result
    echo $result >> results/results_quic.csv
    kill -9 srv_pid
    kill -9 cli_pid
done
```

Bibliography

[1] J. Zhang, X. Gao, L. Yang, T. Feng, D. Li, and Q. Wang, "A systematic approach to formal analysis of quic handshake protocol using symbolic model checking," *Security and Communication Networks*, vol. 2021, p. 1–12, Aug. 2021.

[2] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[3] A. Hussain, J. Heidemann, and C. Papadopoulos, "A framework for classifying denial of service attacks," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 99–110, 2003.

[4] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The QUIC transport protocol," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, p. 183–196, ACM, Aug. 2017.

[5] Y. Cui, T. Li, C. Liu, X. Wang, and M. Kühlewind, "Innovating transport with quic: Design approaches and research challenges," *IEEE Internet Computing*, vol. 21, no. 2, pp. 72–76, 2017.

[6] Y. A. Joarder and C. Fung, "A survey on the security issues of quic," in *2022 6th Cyber Security in Networking Conference (CSNet)*, pp. 1–8, 2022.

[7] M. Nawrocki, R. Hiesgen, T. C. Schmidt, and M. Wählisch, "QUICsand: Quantifying QUIC Reconnaissance Scans and DoS Flooding Events," in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 283–291, Nov. 2021. arXiv:2109.01106 [cs].

[8] J. Mirkovic and P. Reiher, *Internet Denial of Service: Attack and Defense Mechanisms*. Prentice Hall, 2004.

[9] D. C. MacFarland, C. A. Shue, and A. J. Kalafut, "Characterizing optimal dns amplification attacks and effective mitigation," in *Passive and Active Measurement* (J. Mirkovic and Y. Liu, eds.), (Cham), pp. 15–27, Springer International Publishing, 2015.

[10] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Hell of a handshake: Abusing TCP for reflective amplification DDoS attacks," in *8th USENIX Workshop on Offensive Technologies (WOOT 14)*, (San Diego, CA), pp. 463–478, USENIX Association, Aug. 2014.

[11] M. Nawrocki, P. F. Tehrani, R. Hiesgen, J. Mücke, T. C. Schmidt, and M. Wählisch, "On the interplay between TLS certificates and QUIC performance," in *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '22, (New York, NY, USA), pp. 204–213, Association for Computing Machinery, Nov. 2022.

[12] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A performance perspective on web optimized protocol stacks: Tcp+tls+http/2 vs. quic," in *Proceedings of the Applied Networking Research Workshop*, pp. 1–7, ACM, July 2019.

[13] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[14] D. M. Divakaran, H. A. Murthy, and T. A. Gonsalves, "Detection of syn flooding attacks using linear prediction analysis," in *2006 14th IEEE International Conference on Networks*, vol. 1, pp. 1–6, 2006.

[15] W. Chen and D.-Y. Yeung, "Defending against tcp syn flooding attacks under different types of ip spoofing," in *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, pp. 38–38, 2006.

[16] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.

[17] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport." RFC 9000, May 2021.

[18] M. Thomson and S. Turner, "Using TLS to Secure QUIC." RFC 9001, May 2021.

[19] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control." RFC 9002, May 2021.

[20] D. J. Bernstein, "Syn cookies, 1996," *URL http://cr. yp. to/syncookies. html*, 2016.

[21] J.-P. Aumasson and D. J. Bernstein, "Siphash: a fast short-input prf." Cryptology ePrint Archive, Paper 2012/351, 2012. https://eprint.iacr.org/2012/351.

[22] L. Xie, S. Zou, Y. Xie, and V. V. Veeravalli, "Sequential (quickest) change detection: Classical results and new directions," *IEEE Journal on Selected Areas in Information Theory*, vol. 2, no. 2, pp. 494–514, 2021.

[23] B. Teyssier, Y. A. Joarder, and C. Fung, "An empirical approach to evaluate the resilience of quic protocol against handshake flood attacks," in *2023 19th International Conference on Network and Service Management (CNSM)*, pp. 1–9, 2023.

[24] W. Eddy, "TCP SYN Flooding Attacks and Common Mitigations." RFC 4987, Aug. 2007.

[25] J. Lainé, "aioquic." https://github.com/aiortc/aioquic, 2019.

[26] P. octopus, "picoquic." https://github.com/private-octopus/picoquic, 2019.

[27] J. Lainé, "aioquic." https://github.com/microsoft/msquic, 2020.

[28] M. Seeman, "quic-go." https://github.com/quic-go/quic-go, 2017.

[29] B. Teyssier, Y. A. Joarder, and C. Fung, "Quicshield: A rapid detection mechanism against quic-flooding attacks," 2024.

[30] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.

[31] G. Lorden, "Procedures for reacting to a change in distribution," *The Annals of Mathematical Statistics*, vol. 42, no. 6, pp. 1897–1908, 1971.

[32] M. Basseville and I. V. Nikiforov, *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., 1993.

[33] L. Xie, S. Zou, Y. Xie, and V. V. Veeravalli, "Sequential (quickest) change detection: Classical results and new directions," 2021.

[34] W. Chen and D.-Y. Yeung, "Defending against tcp syn flooding attacks under different types of ip spoofing," in *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*, pp. 38–38, 2006.