

# **QUIC PATROL**

## Protocol Analysis at the TRanspOrt Layer

---

Abdullah Rasool

*July 2, 2018*

Version: v1.1

# Radboud University



## **QUIC PATROL**

### **Protocol Analysis at the TRanspOrt Layer**

Abdullah Rasool

*Supervisor*    **Gergely Alpár**  
Digital Security Group  
Radboud University, Nijmegen & Open University

*Supervisor*    **Joeri de Ruiter**  
Digital Security Group  
Radboud University, Nijmegen

July 2, 2018

# Abstract

With the increasing demand for a more secure and faster web, existing technologies are reaching their limits. The Transmission Control Protocol, which is currently used, introduces latency when setting up a secure connection and is hard to update. Therefore, Google introduced a new transport layer protocol in 2013 called Quick UDP Internet Connections (QUIC). It is designed to be encrypted by default and have low-latency. It is built on top of the User Datagram Protocol (UDP) which makes it possible to deploy changes rapidly so that it can keep up with the developments in modern internet. In this thesis we look at Google's implementation of the QUIC server and perform two types of analyses. First, we learn a model from the implementation and use it to check if it matches the specified requirements. Second, we test the robustness of the implementation.

# Acknowledgement

Writing a thesis is like going to the gym. It is harder to do when you are alone, but when surrounded by others, it boosts the result. Therefore, there are a lot of people I would like to thank.

First of all, I would like to thank my supervisors for their time and effort they spent into me and this thesis. Joeri asked great questions regarding my approach and analysis. It helped me to get a deeper understanding of my research. Gergely helped me write a better thesis, asking great questions and was always very patient and positive. It has been the second time that Gergely has supervised me and I have definitely seen myself grow since then.

I would like to thank my friends for their support during my thesis and sometimes even help me forget it. In particular thanks to Sarah van Meel for our weekly meetings at the TU Eindhoven. These were not only productive but also a lot of fun! I liked our Harry Potter soundtrack trivia, singing *Leef* and discussing everything that happened the last week. Also, many thanks to Tom Sandmann and Matthias Ghering for reading my thesis, answering some of my questions and having nice discussions about technological developments.

I would like to thank my great colleagues from PwC for taking me into their team, providing me another great place to work on my thesis and beating me in a game of pool. Special thanks to Thom Prins for calling me every week to ask and discuss the progress.

I would like to thank my girlfriend, Dagmar van Ham, for dealing with me the past 16 weeks. Writing a thesis is a very intense process and you helped me through it even when I felt like it was impossible to finish it.

At last a very special thanks to my parents. It was them who gave me the opportunity to study in the first place. At the end of the 1990s they made the incredible hard decision to leave their home country (Afghanistan) and start an adventure. At that time, they did not know where you would end up and what would happen during

their travel. I do not say this enough, but I am very grateful for their courage then and their ever since continuing motivation to study hard.

Again, thanks to everyone who helped me the past few months! Even if I forgot to mention you here personally, I could not have done it without you.

Abdullah Rasool  
Amsterdam, June 25th 2018

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions & Overview . . . . .	2
1.2	Related Work . . . . .	4
<b>2</b>	<b>Technical Background</b>	<b>7</b>
2.1	Networking Protocols . . . . .	7
2.1.1	QUIC: Quick UDP Internet Connections . . . . .	9
2.1.2	QUIC Connection in detail . . . . .	13
2.2	State Machine Inference . . . . .	20
2.2.1	L* Algorithm . . . . .	21
2.2.2	Example . . . . .	25
<b>3</b>	<b>Protocol Analysis: Set-Up</b>	<b>30</b>
3.1	State machine inferencing . . . . .	30
3.1.1	Implementing the abstraction component . . . . .	33
3.1.2	Dealing with non-determinism . . . . .	35
3.2	Fuzzing . . . . .	38
3.2.1	Naive fuzzing . . . . .	40
3.2.2	Advanced fuzzing . . . . .	40
<b>4</b>	<b>Protocol Analysis: Results</b>	<b>45</b>
4.1	State machine inferencing . . . . .	45
4.1.1	Without 0-RTT . . . . .	45
4.1.2	With 0-RTT . . . . .	47
4.2	Fuzzing . . . . .	49
4.2.1	Naive fuzzing . . . . .	49
4.2.2	Advanced Fuzzing . . . . .	50
<b>5</b>	<b>Conclusion</b>	<b>52</b>
5.1	Summary . . . . .	52
5.2	Discussion . . . . .	53
5.3	Future work . . . . .	55
	<b>Bibliography</b>	<b>57</b>

<b>A</b>	<b>LibFuzzer code</b>	<b>63</b>
<b>B</b>	<b>Learned model without 0-RTT</b>	<b>64</b>
<b>C</b>	<b>Learned model with 0-RTT</b>	<b>65</b>

# Introduction

Computers are being used for a large variety of tasks. This includes internet banking, buying clothes online or navigating an aircraft at high altitude between waypoints. This requires them to frequently interact with other computers. To ensure successful communications, a well-defined set of rules is needed. This standardizes the exchanged messages even if several vendors are involved. If there is no such standard, it is hard to guarantee that different vendors implement the same message formats, sequence of messages or responses to messages.

Such a predefined set of rules is called a protocol [Tre01]. These rules can be written down as text or it can be presented more graphical. A computer is not able to understand this format directly. It needs to be implemented in software which can then be executed.

The problem with software is that errors can be made, which can cause it to behave incorrectly. This can have negative consequences depending on the protocol and severity of the flaw. Therefore, it is important that implementations of protocols are thoroughly tested to ensure that they are in line with their specification. This is called conformance testing.

However, it is hard to analyse protocol implementations by hand, especially if they stem from complex specifications. An approach which does not require complete manual testing, is state machine inferencing. Here, we infer an abstract representation of the implementation, known as a model or more formally as Mealy machine. This process is also known as model learning or state machine learning. The resulting model can be expressed in a graphical or textual format. In general, it is easier to reason about a model than the implementation.

The computers involved in communication are distributed over the world. There are important reasons for distributing computers [Nad+06]:

- **Functional separation:** Based on functionality / services that are provided.
- **Reliability:** Storing and replicating data or services at different locations.



- **Scalability:** Adding more resources to increase performance or availability.
- **Economy:** Sharing resources with others to lower the cost.

However, there are also challenges with distributed systems. For example, there is no central entity that knows the global state of the system. This is challenging when faults need to be restored or when consensus is needed.

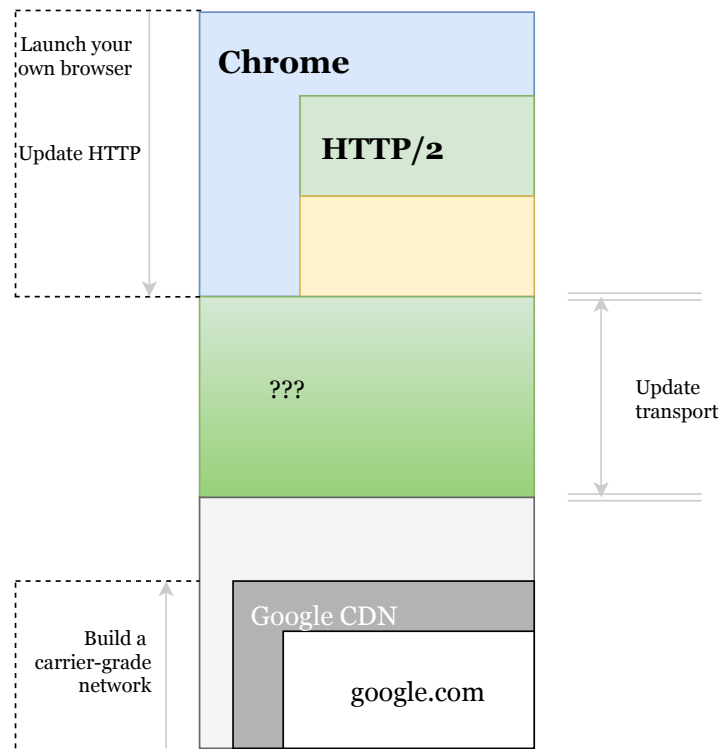
For a major internet company like Google, it is important that they can serve as many users as fast as possible. If we look at the past few years, Google has made much effort to achieve this goal. First, they built a carrier-grade network for their servers to use. This kind of network offers extreme reliability and provides very fast fault recovery through redundancy [Riv09]. Next, they engineered a new browser and were involved in the development of HTTP/2 [Iye13]. HTTP/2 offers performance improvements as it is not blocking, it incorporates header compression to reduce overhead and is represented in binary form rather than textual. The last property makes HTTP/2 more efficient to parse, more compact in transit and are less error-prone [IET17]. More details on HTTP/2 are described in Section 2.1.2.

If we look at Figure 1.1, we can see that the connection between the browser of the user and the Google server was left untouched. This is the level where transport protocols play a role. These protocols are responsible to deliver messages between applications. In order to also enhance the performance at this level, Google developed Quick UDP Internet Connections (QUIC) in 2013 [Lan+17]. It guarantees that messages are delivered and it provides a secure and authenticated channel on which these messages are transported.

## 1.1 Contributions & Overview

In this thesis we analyse the server implementation of QUIC. We perform two different types of analysis. First, we learn a model from the implementation. This model formally known as a state machine or Mealy machine contains a number of states. Between these states there are transitions. The transitions represent the changes in the internal state of the system when it receives certain input. We use the learned model to compare the implementation with its specification. In the second analysis, we check the robustness of the implementation by sending unknown / malformed inputs and see if the implementation is able to handle this.

This thesis is divided in the following chapters:



**Figure 1.1.** Google's performance efforts over the years [Iye13]

- Chapter 2 provides a theoretical background of the QUIC protocol. We describe Google's design decisions, the available packets and their formats. In addition, we mention how state machine inferencing works. We provide an example of a concrete learning algorithm.
- Chapter 3 illustrates the set-up which we used to learn the state machine of QUIC. We describe some obstacles and why some approaches did not work in our study. Additionally, we briefly mention the background of fuzzing, which is another name for robustness testing. We describe the tool we use to fuzz the QUIC server.
- Chapter 4 shows the learned model. We discuss results from state machine inferencing and from the fuzzing.
- We conclude in chapter 5 with a summary, a discussion and mention ideas for future work.

## 1.2 Related Work

The idea to use model learning to analyse software was introduced in [Pel+99]. They want to know whether a given implementation, with unknown internals, satisfies certain specified properties. One of the procedures they mention is an algorithm for learning models using two types of queries, called  $L^*$  [Ang87]. We use this algorithm in this thesis and we discuss it in more detail in Section 2.2.1.

There have been a lot of studies about state machine inferencing and its applications. We would like to mention a few studies that have learned models from implementations of several networking protocols. This is one of the goals of this thesis. First, Paul Fiterău-Broștean performed extensive research in his PhD thesis [FB18] on the topic of model learning for the analysis of network protocols. He applied this technique to learn and check models of SSH implementations and TCP implementations on Windows machines. A similar approach was used to infer models on TLS implementations [Aar+13]. In his PhD thesis, Joeri de Ruiter, learned models from several bankcard vendors [Rui15]. It was shown that there are differences between several vendors while implementing the same specification. Some other interesting fields of application are electronic passports [Aar+10] and industrial control systems [Ker17].

These studies have used an active approach to learn these models. Here, messages are actively being sent and received to the System under Learning (the implementation we want to learn). There are also studies that have tried a passive approach. Here, they learn a model based on the logs of previous executions or use network traces from specific applications [Wan+11]. The downside of using passive approaches is that the learned model is only as good as the data provided by the previous executions of the applications.

A research in 2017 also constructed a state machine from QUIC, but they used a passive approach by looking at the log files created during the execution of the protocol [Kak+17]. The goal was to run QUIC in a large number of different environments (several mobile and desktop devices with different operating systems) and to use the state machine to understand differences across versions of QUIC and in each of the different environments. In this thesis, we perform active learning of the QUIC protocol. We do not use log files but observe the output of the implementation when actively sending input to it.

With state machine inferencing we learn a model and compare it with its specification. This specification describes some data format and what sequence of these messages describe a valid sequence. The implementation of a protocol can not be correct unless it is able to validate input correctly. If the received input is not correct, the

implementation needs to be able to detect this [Sas+13]. Therefore, it is good to test what happens when an incorrect data format is used or when some invalid data is sent.

This is where fuzzing is used to send invalid data and observe what happens. There are two types of fuzzing. First, there is blackbox fuzzing. It is unaware of the data structure of a protocol. This makes it easy to create and fast in execution, since it just creates random data and provides it to the fuzz target. This target is formally known as the System under Test. The other type is whitebox fuzzing. The idea of whitebox fuzzing is to mix fuzz testing with dynamic test generation. It is smarter and more complex than blackbox fuzzing. This increased complexity has a negative effect on the performance of the fuzzer. An example is SAGE which is developed by Microsoft [God+08; God07]. Another example of a whitebox fuzzer is LibFuzzer, which we have used in this thesis to fuzz the QUIC server.

In practice, both types of fuzzers are able to find bugs. Once an application has already been fuzzed by a blackbox fuzzer, it is more effective to use an intelligent, white-box fuzzer to find bugs that are more complex to find.

If we look at QUIC, there has also been some studies [Lyc+15; Car+15] about it. Google claims that QUIC offers performance improvements. Especially if you give a product such a name, you stimulate people to find out if it is as quick as the name suggests. In one study [Meg+16] it was shown that in more than 40% of their scenarios, the page load times significantly improved with the experimental version of QUIC compared to traditional TCP and HTTP/1.x. However, the study also showed that QUIC cannot use the full capacity of high speed links. In addition, network administrators may also limit QUIC traffic due to security concerns as it is built on top of UDP (see Chapter 2).

Another field where QUIC has improvements over existing protocols is security. It provides a secure (authenticated and encrypted) channel by default. There have also been studies about its security. One of the studies found that it was possible to fail the handshake due to an inconsistent state between the client and the server. This makes the QUIC fallback to TCP with TLS. This results in more latency and in the end could be used to mount denial-of-service attacks [Lyc+15].

Another security aspect which was examined is its key exchange. In Chapter 2 we see that QUIC uses a multi-stage key exchange. In 2014 it was shown that the key exchange protocol in QUIC meets the security properties as suggested by the designers [FG14].

One general remark about these studies is that QUIC is still heavily under development. In the past, this meant that the code was changed drastically every few months. This resulted that some studies were outdated before they were published.

# Technical Background

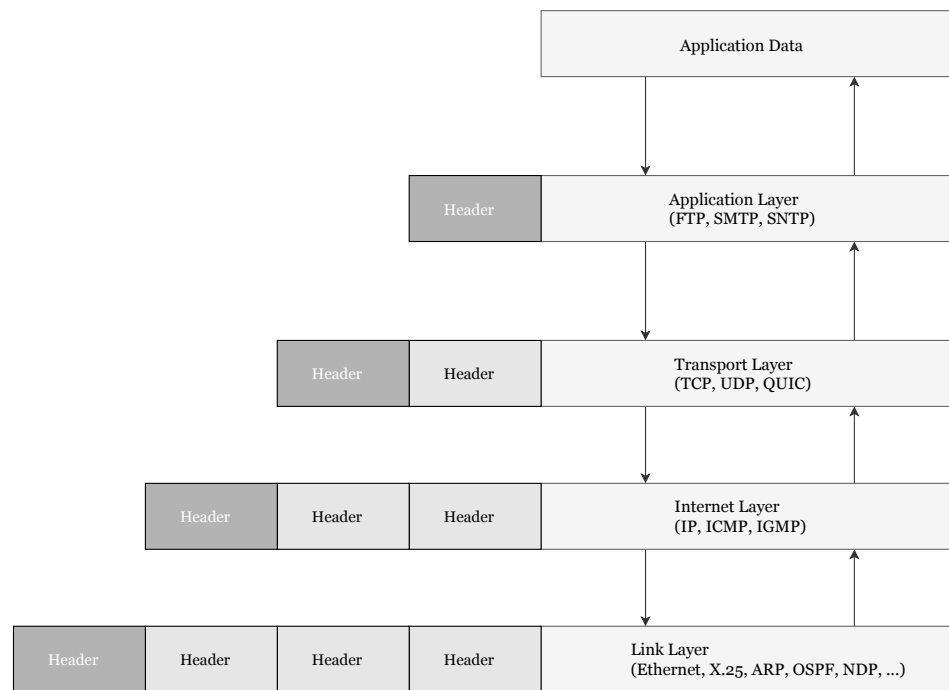
In this chapter we discuss the technical background of Google's QUIC network protocol. Furthermore, we look at the general process of inferring models (i.e. state machines) from implementations. We start with a brief discussion of networking protocols in general.

## 2.1 Networking Protocols

Computers often need to communicate with other computers. This communication needs to be standardized in the form of a protocol which defines a set of rules that manage the content and format of the exchanged messages. In addition, a protocol may also specify actions that need to be taken upon certain received messages. If there were no such standards it is up to individual vendors to define message formats. This would reduce interoperability between different vendors [FB18; SG13; BS12].

Once a message is created conforming the rules defined in the protocol, it is sent to the other party. Communication over the internet is divided into five layers. These layers provide some structure to the design of the Internet Protocol (IP). In addition, each layer can focus on solving problems specific to that layer and it can offer certain services to the layers above. We can distinguish five layers, see Figure 2.1:

- **Application Layer:** Network applications such as web browsers with their protocols and application specific data (e.g. HTTP).
- **Transport Layer:** Transports application layer messages between application endpoints.
- **Network Layer:** Moves network-layer packets (datagrams) from one host to another.
- **Internet Layer:** Routes a datagram through a series of routers and switches between the source and the destination.



**Figure 2.1.** Internet Protocol stack [UK18]

- **Link Layer:** Moves individual bits within a frame from one node to the next one.

Figure 2.1 shows the layers of the Internet protocol stack. We can see that messages from higher levels are encapsulated in lower level packets. Each level adds the information it needs in a header and hands the packet over to the lower level.

There are currently two well-known protocols that implement the services of the transport layer. First, we have the Transmission Control Protocol (TCP) [Rfca] which provides a connection-oriented service to the application layer. This includes guaranteed delivery, fragmenting a long message into multiple shorter ones and a congestion-control mechanism. It controls the transmission rate when the network is congested. Next, we have User Datagram Protocol (UDP) [Rfcb] which provides a connectionless service to the application layer. It does not offer reliable delivery or congestion control.

TCP is more commonly used as a transport layer protocol due to the services and guarantees it provides [Cen18]. However, it is hard to deploy changes in TCP. This is because TCP is implemented in the kernel of operating systems. Therefore, making a change in TCP requires an update in operating systems. In addition,

there are middleboxes<sup>1</sup> and legacy systems that make the deployment of updates challenging.

This forms a bottleneck especially now that optimizing for latency and providing encryption at the transport layer became a concern [Rüt+18]. This demand is caused due to an increase in usage of interactive web applications. Paradoxically, encryption over TCP is provided by using Transport Layer Security (TLS) [RD08] on top of TCP which introduces more latency. There are approaches to overcome these drawbacks. However, these cannot be incorporated since updating TCP is not easy because of legacy systems, updating kernels and having middleboxes. Therefore, there is a need for a transport layer which can be easily updated.

### 2.1.1 QUIC: Quick UDP Internet Connections

QUIC is described as an encrypted, multiplexed and low-latency transport protocol, designed to improve transport performance and enable rapid deployment [Lan+17]. We have two parties in QUIC. First, we have the client which can connect to a server and make requests. Next, we have the server which accepts connections and responds to requests. Examples of clients are web browsers (Chrome) and mobile applications (YouTube).

Google added QUIC to Chrome in June 2013 as an optional functionality. It started with very limited features. As of now, it is enabled for almost all Chrome users since it addresses several shortcomings in TCP and offers performance improvements [Lan+17]. QUIC is built on top of UDP in user space<sup>2</sup>, which allows for easier deployment of changes as middleboxes work at a lower level. Additionally, it is possible to update only the user space process, without updating the underlying kernel.

#### Design decisions

There are several design decisions that enable QUIC to address some issues in TCP. One issue is addressed by combining cryptographic and transport handshakes which reduces the set-up latency to 0 round-trip time (RTT) when connecting to the same server. We describe latency of networking protocols by their round-trip time, which is the interval between the sending of a packet and the receipt of its acknowledgement [KP87]. The actual value of one RTT depends on network

---

<sup>1</sup>A device that transforms, inspects, filters or manipulates traffic for other purposes than packet forwarding. For example firewalls, intrusion detection systems, load balancers [Lan+17; Wik].

<sup>2</sup>Where normal processes run.



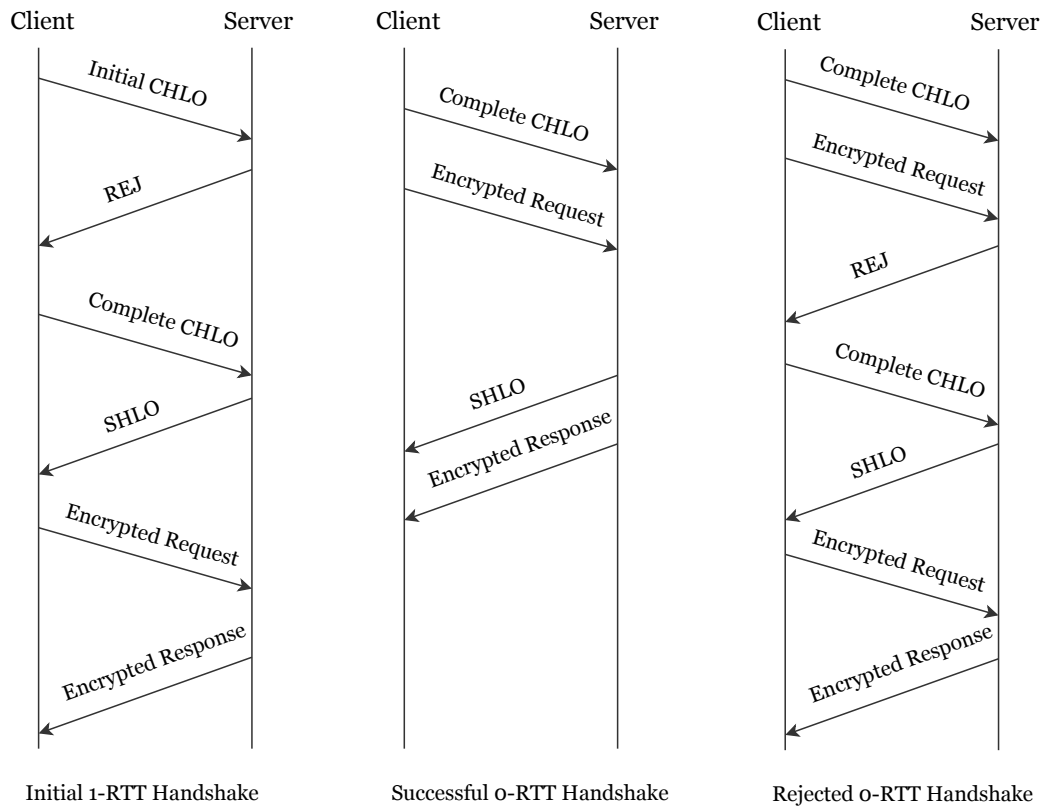
conditions and speed of the client or server. Nevertheless, this metric is used to denote the number of packets that need to be exchanged. Throughout this thesis we use the terms packet, message, request and response interchangeably.

There are three scenarios for connection establishment, also known as handshake. This is shown in Figure 2.2 [LC16; Lan+17]:

- **Initial handshake:** The client initially has little to no information about the server. The client starts with a client hello (CHLO) message which will be rejected by the server. This is done by responding with a rejection (REJ) message. This contains a server configuration with its long-term Diffie-Hellman public value, a certificate chain authenticating the server and a timestamp. The exact content of the messages is discussed in the next section. Now that the client has more information regarding the server, it can send a new complete CHLO message containing its initial tags and the received ones from the REJ message. If the handshake was successful, the server responds with an encrypted server hello (SHLO) message. This message is authenticated and encrypted using the shared key which is computed using the client's and server's public value. The SHLO message includes the server's ephemeral Diffie-Hellman public value, which is used to compute the ephemeral session key.
- **Repeat handshake:** The client has already seen the REJ message in some previous connection establishment. It stored the tags from the REJ message so that it can craft the complete CHLO message at once. Again, if the handshake was successful, the server responds with an encrypted SHLO message. Using the initial shared key, both parties can compute the ephemeral keys to send and receive any further messages. If the client wishes to achieve 0-RTT latency, then it must encrypt the request with the initial keys and send it before it receives an answer from the server. In order to achieve this, the server also stores the client's nonce and its public value such that it can compute the shared key.
- **Failed 0-RTT:** In the case of sending expired server information in the full CHLO, the client receives a REJ message. The handshake continues as if it was an initial handshake.

In Table 2.1 we have summarized the different latencies between QUIC, TCP and TCP together with TLS.

Another issue QUIC addresses is head-of-line blocking. This occurs when a packet is lost in TCP and must be retransmitted. TCP delivers packets in the same order



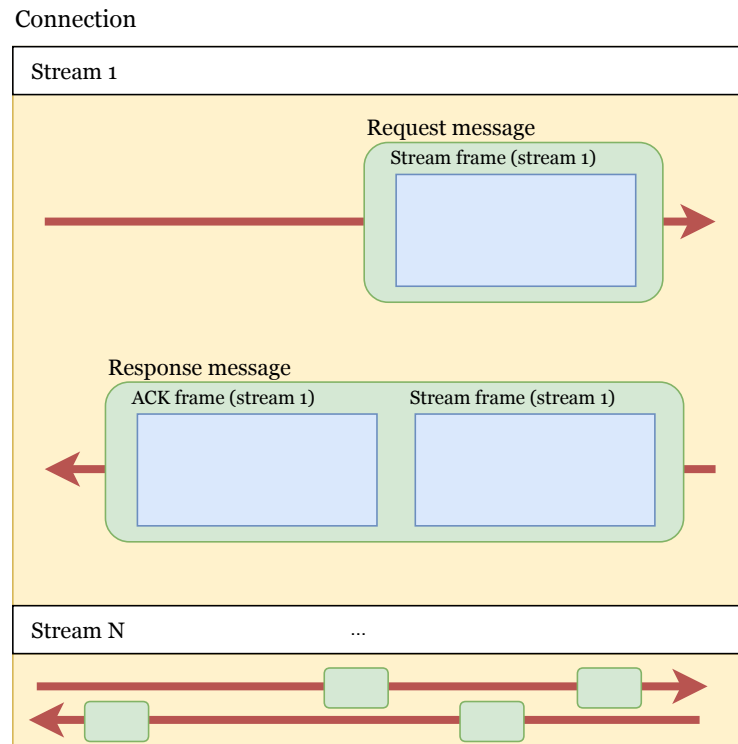
**Figure 2.2.** Overview of different QUIC handshakes [Lan+17].

	TCP	TCP + TLS	QUIC
First connection	1 RTT	3 RTT	1 RTT
Repeated connection	1 RTT	2 RTT	0 RTT

**Table 2.1.** Connection RTT for TCP, QUIC and TCP with TLS [Meg+16].

as they have been sent, therefore all packets must wait until the lost packet is retransmitted and received. QUIC solves this by using a lightweight abstraction called streams, see Figure 2.3. It can be seen as a reliable bidirectional bytestream which can transfer up to  $2^{64}$  bytes in total per stream. Here we see  $N$  streams. In stream 1 there is bidirectional stream with one request message containing a single frame and one response message containing two frames. One of them acknowledges a previous message and one of them is the response data itself. Streams are created implicitly by sending data on an unused stream ID. Closing a stream is explicit by setting the FIN bit in the stream header [Lan+17; LC16].

In general, a single QUIC connection consists of multiple streams. Each stream is cut into frames. We have different types of frames. First, we have regular stream frames that carry data used for connection establishment (e.g. CHLO, REJ). These use the fixed stream 1. Next, we have acknowledgment frames which are discussed later in this chapter. There is also a frame used for congestion control, which we do



**Figure 2.3.** QUIC connection with multiple streams. Each stream has a bidirectional flow of frames with data [Gri13].

not send in our setting since we only have a single user on a local network. Next, there are two frame types to close the connection which are discussed later.

If a packet is lost, it only impacts those streams whose data was carried in that packet. Subsequent data received on other streams is continued to be reassembled and delivered to the application layer [LC16; Gri13]. Each stream has its own ID where odd IDs are used for client-initiated streams and even IDs for server-initiated streams, to avoid collisions.

Packets are authenticated and encrypted using AES-GCM. At the time of writing, Google uses a custom scheme for cryptography [LC16]. Google needed a secure scheme when development started, but TLS1.3 [Res18] was not ready at that time [LC16].

Before the server and the client can encrypt or decrypt, they must first compute the same shared key. This is done in a few steps and depends on whether they are computing the initial or ephemeral key:

- **Initial key computation:**
  1. Perform Elliptic-Curve Diffie-Hellman with the two public values.

2. Perform an HMAC-based Extract-and-Expand key derivation [KE10]. It uses SHA256 as hashing algorithm. As a salt it uses the client nonce concatenated with the server nonce. As info it uses the fixed label `QUIC key expansion` followed by a 0-byte, the connection id, the bytes from the `CHLO` packet, the bytes from the server config and the decoded leaf certificate. The result is 40 bytes composed of two keys (16 bytes) and two initialization vectors (4 bytes). We found that the server nonce is fixed when computing the initial key, even though the server sends its nonce in the `REJ` message. We are not sure why this is fixed, maybe it is present because of some debugging functionality when computing the key.
  3. Diversify the server key and IV by performing another round of hash-based key derivation. Again, it uses SHA256 as algorithm. For salt, the diversification nonce is used, which is present in the `REJ` packet. As info parameter, the fixed label `QUIC key diversification` is used. In contrast to the previous derivation, the length is 20 bytes since only a single key and IV is used.
- **Ephemeral/Forward-secure key computation** has the same steps as the initial key, but without diversifying the key. It uses `QUIC forward secure key expansion` as info parameter in the hash-based key derivation instead of `QUIC key expansion`. Here, we do not observe this fixed server nonce.

Using ephemeral keys results in perfect forward secrecy for a single connection [Cho02; Kra05; LC16], since there is a new key generated per session. Once such a key is no longer used and is erased from memory, there is no way to find this key except by cryptanalyzing the Diffie-Hellman exchange.

### 2.1.2 QUIC Connection in detail

In this section we take a detailed look at a single QUIC connection from establishing a connection, sending messages, acknowledging them, to terminating an existing connection.

There are two versions of the QUIC protocol. First, there is a version by Google [Cyr+16]. It is used in the Chrome browser and the servers of Google. Next, there is the version from IETF [IT18] which will standardize Google's initial work.

We choose to use the Google variant. The reason is that IETF publishes new draft specifications very frequently. Therefore, the implementations of IETF always fall

behind on the documentation and is not used in practice. We are not able to test these unimplemented specifications. In addition, Google has an implementation which is already used on a large scale and is open-source available [Lan+17].

However, there exists a drawback. The specification of Google's variant is not in line with their implementation. The latest version of the specification contains a lot of unfinished sections and does not even mention the latest implemented QUIC versions.

## General Header

We start the description with the general header which is present in plaintext in every packet. It can consist up to six fields. Every header (see Figure 2.4) starts with an eight-bit public flag including:

U: Unused / Reserved bit.

M: Multipath mode.

PNR: Packet number length: two bits to indicate whether a packet number is 1, 2, 4 or 6 bytes.

CIDL: Connection ID present: indicates whether the 8 byte connection ID is present.

DNP: Diversification Nonce present: indicates the presence of a 32 byte diversification nonce. This nonce is generated by the server and is only used in the server to client direction. It ensures that the server is able to generate unique keys per connection. Otherwise, a repeated CHLO with the same connection ID results in the same initial encryption keys. If we allow the server to generate this nonce, it prevents a client from deriving the same initial keys for two distinct connections since this nonce introduces randomness [IT].

RST: Reset packet: indicates that the packet is a public reset packet.

VER: Version negotiation: if set by the client this means that the header has a version number. This must be set by the client in all packets until the server has agreed to the proposed version. If set by the server, the packet is a version negotiation packet. This lists all the supported versions by the server from which the client can then make a choice.

0	1	2	3	4	5	6	7
U	M	PNR	CIDL	DNP	RST	VER	

**Figure 2.4.** QUIC Header Public Flags

After the public flag, the header continues with the connection ID. QUIC connections are designed to remain established even when a connection is lost. The four-tuple (source IP, source port, destination IP, destination port) traditionally used in TCP is insufficient for this. The third field is an optional four byte QUIC version represented as Q followed by the version number (Q039 for version 39). The fourth field is the optional diversification nonce. The fifth field is the variable-sized packet number. It should always start at 1. The last field is the message authentication hash.

Before the handshake is completed, the messages are sent in plaintext. In order to guarantee integrity during the handshake, a checksum is computed over the content of the packet. This achieves protection against packet corruption. This is done using a non-cryptographic hash function called FNV-128A [Fow+17].

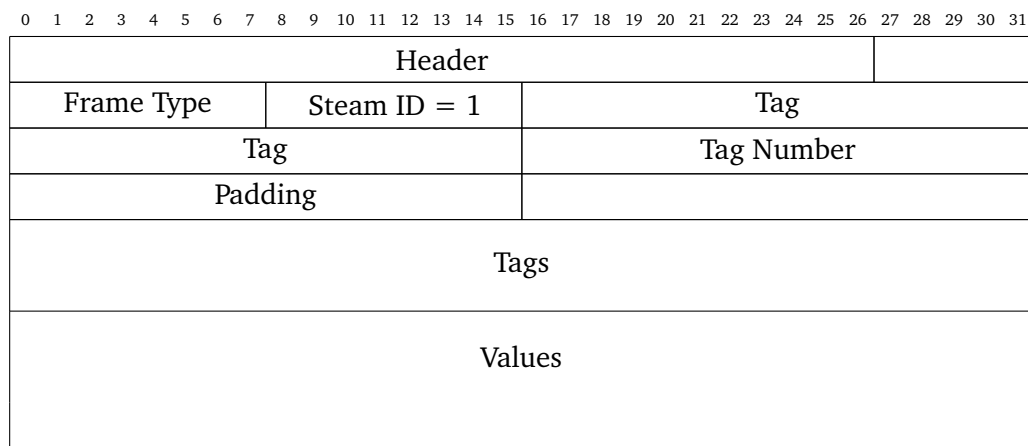
The reason for choosing this hash is unspecified. We believe that the reason has to do with its performance. According to a 2014 study, [Est+14] FNV is one of the most efficient and widely used hash functions. It is also used in Linux, Twitter and several game consoles.

## CHLO

The client starts with sending a client hello (CHLO) message, see Figure 2.5. In the case of an initial CHLO, not all information required by the server is present. The documentation of QUIC was incomplete, so we built this packet based on a different open-source implementation of the client [Cle18].

After the general header, this message is composed of the following four fields. First, there should be a frame type that indicates it is a stream. Moreover, it uses the reserved stream ID for crypto messages. Next, there is the tag that identifies the stream as CHLO. Finally, there is a list of tag-value pairs used for the transport and security handshake. In order to make parsing easier, the number of pairs is displayed before the list itself. These pairs are represented as 7-bit ASCII strings. In our case, we send the following pairs of values to the server in the initial CHLO message [LC16]:

- Server Name Indication: fully quantified DNS name of the server.



**Figure 2.5.** Handshake packets

- Version: what version the clients wishes to use. The client can use Q099 to retrieve a list of supported versions by the server.
- Common Certificate Sets: list of 64-bit FNV hashes of sets of common certificates that the client possesses.
- Proof Demand: a list of tags describing the types of proof acceptable to the client, in order of preference. Currently only X509 certificates are defined.
- Some padding.

Other handshake messages also begin with the general header, then have some frame information and end with a set of tag-value pairs.

## REJ

When sending this initial CHLO message, the server rejects it. The server responds with a REJ message which is composed similarly to the CHLO. The tag identifies that it is a REJ message. In our case it was composed of the following tag-value pairs [LC16]:

- Source Address Token: authenticated-encrypted block that contains at least the client's IP address and a timestamp by the server. Clients can include the source address token in future requests in order to demonstrate ownership of their source IP address. This is used when sending the full CHLO message. If the token has expired, the client needs to request a new one.
- Server Nonce.

- **Proof Signature:** a signature of the server config by the public key in the leaf certificate.
- **Server Config:** message containing the server's serialized config. The config contains an ID, supported key exchange algorithms, supported authenticated encryption algorithms, list of supported QUIC versions by the server.
- **Reasons for sending the REJ.** In our case this was because it was missing the server config ID in the CHLO message.
- **Server Config time-to-live:** duration, in seconds, that the server config is valid for.
- **Encoded certificate chain.**

The elements for the rejection message are optional, but the server must allow the client to make progress [LC16].

## SHLO

The received tags are combined with the initial tags to craft another CHLO message. If the message authentication hash is correct and all required tags are correct and present, the server responds with an encrypted server hello (SHLO) message.

This packet also has the general header and a set of tag-values pairs. There are two important pairs which are used to compute the ephemeral keys. First, there is the public value of the server, this is used when computing the ephemeral shared key. Next, there is the server nonce which is used when performing the hash-based key derivation.

Now that the client has computed the ephemeral keys it can send encrypted requests, receive responses, send acknowledgements and close the connection.

## Acknowledgements

Acknowledgements are sent in a special frame, the ACK frame. This frame is used to inform the other party what packets have been received and which are still missing. It is also used to send some timing information.



The ACK frame is composed of the following fields:

- Frame type: 8-bit value starting with 01 to indicate it is an ACK frame. Next, a bit indicating whether this frame acknowledges more than one acknowledgement range. Next, there is an unused bit. The following two bits indicate the size of the largest acked packet. It can be 1, 2, 4 or 6 bytes long. Finally, there are another 2 bits that indicate the length of the missing packet sequence number delta. This field can have the same lengths as the largest acked.
- Largest acked: the largest observed packet number.
- Delta time: the time between observing the largest packet number and sending this acknowledgement in microseconds. The format is loosely modelled after IEEE 754. This is send to make a better estimation of the actual value of the RTT. This helps in analysing the conditions of the network.
- Variable sized packet number delta.
- Number of timestamps.

We observed an undocumented pattern in the acknowledgements. The first ACK frame always has two additional trailing bytes, whereas later frames do not have this. Contacting the developer of the client did not help as he was not aware of this. The documentation of the acknowledgement frame did not mention this behavior. It is hard to understand it, since both frames use the same frame type. Therefore, it is not possible for the client or server to know whether this frame will have the trailing bytes or not. We feel that this part is outdated from the documentation and has been replaced in the implementation. This made it challenging for us to correctly create and parse this frame type.

After the ephemeral keys have been computed this frame type is send encrypted. During the handshake, it is sent in plaintext.

It is possible to inform the other party that certain packets do not need to be acknowledged any more. In that case a STOP\_WAITING frame can be send. This frame has the type of value 0x08 and has another field that indicates what the lowest packet number is for which it will send acknowledgements. In other words, the other party can stop waiting for acknowledgements lower than the specified packet number.

## HTTP/2 over QUIC

The transport layer moves packets with application-layer content. The most used protocol on that layer is HTTP [Car+15].

Despite its popularity, there are some inefficiencies in HTTP/1.1 that obstruct the creation of a faster internet [Car+15]. For example, HTTP only allows one outstanding request per TCP connection. Using multiple TCP connections, which is done now, to issue parallel requests is counter-productive due to the congestion control used in TCP and it is unfair as browsers are using more network resources [IET17].

This was the reason for Google to create SPDY [BP12] which has been standardized as HTTP/2 [Bel+15].

Some of the design decisions that cause performance improvements are [IET17; Bel+15]:

- It is binary, instead of textual. This makes it more efficient to parse, more compact in transit and less error-prone. This is because there is no need to handle whitespaces, capitalization, line endings and blank lines. In HTTP/1.1 there are four different methods to parse a message. In HTTP/2 there is just a single method to do so.
- It is fully multiplexed. This allows for multiple requests and response messages to be in transit at the same time. This is done similarly to QUIC by using multiple streams with frames that either hold HTTP headers or data, see Figure 2.3 [Gri13].
- It incorporates header compression using HPACK [PR15]. Given a webpage with 80 assets (which is normal in today's web) and each request has 1400 bytes of headers (because of cookies), it takes around 8 round trips to only get the headers across. If we apply compression this could be done in a single roundtrip or within a single packet. This reduces the latency noticeable.

QUIC integrates HTTP/2 mechanisms in order to reduce complexity. For example, this is visible in stream management. QUIC handles most of the stream management. The stream IDs from HTTP/2 are replaced by those from QUIC. It also uses a dedicated stream ID to send the compressed headers used in HTTP/2.

## Connection termination

At the end of the lifetime, the connection should be terminated. This can be done using two different frames. First, there is a `CONNECTION_CLOSE` frame. This can be used to notify that the connection is being closed. If there are streams in transit, they are implicitly closed when the connection is terminated. Second, there is the `GOAWAY` frame. This can be used to notify that the other party should stop using the connection as it will likely be aborted in the future. Current active streams are continued to be processed, but the sender of this stream does not initiate or accept any new streams. Ideally, a `GOAWAY` frame is sent in contrast to a `CONNECTION_CLOSE` frame [Cyr+16].

It is also possible that a connection times out because one of the parties does not respond anymore. The default timeout is 30 seconds, but this can be adjusted by setting the `ICLS` parameter in the connection establishment phase. If there is no network activity for the duration of the timeout, the connection is closed. When this occurs a `CONNECTION_CLOSE` frame is sent. It is also possible to close the connection silently. This happens when it is too expensive to send an explicit close. Examples are when the network is too congested [Cyr+16].

At any time, it is also possible to terminate the connection abruptly. This is done by sending a `PUBLIC_RESET` packet. This is equivalent to TCP RST [Cyr+16].

## 2.2 State Machine Inference

We continue with testing protocol specifications and in particular the method of model learning. We also provide an example of learning a model ‘by hand’.

Testing software is an important aspect of the software development lifecycle, since it enables us to find errors in software by performing experiments. The goal is to gain confidence that during normal operation the system operates as intended. Usually, there is only a limited amount of time for testing and it only allows to show the presence of errors, not the absence of faults [Dij79]. Therefore, we cannot ensure correctness of an implementation [Tre92].

In general, a protocol specification is a detailed document that describes the internals of a protocol. If the description is not formal, the standard might lead to different implementations due to ambiguity. This could result in two versions which are incompatible. Even if the description was formal, differences can emerge due to some parts implemented incorrectly [SL89].

Testing whether an implementation is conform its specification can be done in multiple ways. One of the simplest methods is to derive a series of tests from the specification and apply them to the implementation [FB18]. These tests need to be derived and maintained manually. This is time consuming and the effectiveness depends on whether all corner cases are considered while writing these tests.

Another approach is to use model-based testing. This addresses some of the shortcomings of manual testing. It is able to automatically derive and execute the tests based on a model of the specification. This model covers all important aspects of the specification in a formal way. The problem is that most specifications are textual and at high level. Therefore, deriving a model from its specification is not trivial. Even if we were able to derive it, this model needs to be updated when the specification is adapted [FB18].

Since most specifications do not include a model, we try to automatically generate it from the implementation. This is also called model learning or state machine inferencing. There are multiple methods of doing this. It can be done by analyzing the code or mining the software logs. Depending on the approach, different models can be inferred [Vaa17].

In our approach we consider two conditions when learning the model. First, we apply the approach of black-box learning. In this setting we do not need to have access to the code in contrast to white-box learning. Even though we have access to the code, we chose black-box learning as it is easier to use since we do not need to have thorough understanding of the source code. However, the white-box approach ensures that all the statements in the code is covered. One example of white-box model learning is Predicate-based SYmbolic COMpositional Reasoning (Psyco) where it uses the source code to explore a large number of program execution paths [Mue+17].

The next condition we have is to apply active learning. This means that we are actively performing experiments on the software. In contrast to passive learning where previous runs of the software are analyzed, this has the advantage that we can learn models of the full behavior of the software instead of just specific runs. This increases the completeness of the learned model [Vaa17].

### 2.2.1 $L^*$ Algorithm

In order to learn a model we need two parties. First, there is a learner that wishes to learn an initially unknown empty set  $U$  efficiently. This set is fixed over a known finite alphabet,  $A$ . For example, we have only the letters  $a, b$  in our known alphabet

$A = \{a, b\}$ . The set we want to learn contains only those elements that have an even number of  $a$ 's and an even number of  $b$ 's. In that case, we know that  $aa$  is part of  $U$ , but  $ab$  is not. We call  $aa$  and  $ab$  words, because they are composed of multiple letters from the alphabet.

The learner has information about a finite collection of strings over  $A$ . These can be classified as members or non-members of the set it wishes to learn. Second, we have the teacher which can provide a source of helpful examples.

The learner may ask the teacher for help. However, the learner should not ask for too much help since this makes the learning process inefficient [Ang87]. To realize this, we use a minimally adequate teacher. It can answer two types of questions from the learner. First, there are membership queries where it answers if a given string  $t$  is a member of the unknown set. Second, the learner may ask whether a presented set  $S$  is equal to the initially unknown set  $U$ . If they are not, the teacher responds with a counterexample. If the presented set contained an element which is not in the real set, then the teacher responds with that element as a counterexample. Likewise, if the presented set contains an element which should not be there, it is used as a counterexample [Ang87].

## Basic Definitions

The  $L^*$  algorithm describes how we can efficiently learn the members of the unknown empty set  $U$ . It requires the learner to keep an observation table. In this table information is stored regarding a set of strings over  $A$ . These may or may not be a member of the unknown set  $U$ . The following definitions are applicable to the table:

- **Prefix-closed set:** every prefix of every member of the set is also a member of the set. For example, if 10 is in the set, 1 has to be part of it as well.
- **Suffix-closed set:** if 10 is in the set, 0 has to be part as well.
- $(\cdot)$  denotes **concatenation**. If  $A$  and  $B$  are sets,  $A \cdot B$  is every element of  $A$  concatenated with every element of  $B$ .

Furthermore, the table consists of three objects [Kre]:

- $S$ : nonempty set of finite prefix-closed strings. Initially  $\{\lambda\}$ .

- $E$ : nonempty set of finite suffix-closed strings. Initially  $\{\lambda\}$ .
- $T$ : finite function that maps  $u = ((S \cup S \cdot A) \cdot E)$  to  $\{0, 1\}$ . It returns 1 if  $u$  is a member of the unknown set, otherwise it returns 0.

This table can be visualized as a two-dimensional array as is shown in Table 2.2 (see page 26). The rows are labeled by elements of  $(S \cup S \cdot A)$  and the columns are labeled by elements of  $E$ . The entry for row  $s$  and column  $e$  is equal to  $T(s \cdot e)$  [Kre].

There are two properties for observation tables. First, the table is *closed* if for each string  $t$  in  $S \cdot A$  there exists an  $s$  in  $S$  such that  $row(t) = row(s)$ . Next, an observation table is *consistent* if  $s_1$  and  $s_2$  are elements of  $S$  such that  $row(s_1) = row(s_2)$  and for all  $a$  in the alphabet  $A$  holds  $row(s_1 \cdot a) = row(s_2 \cdot a)$ .

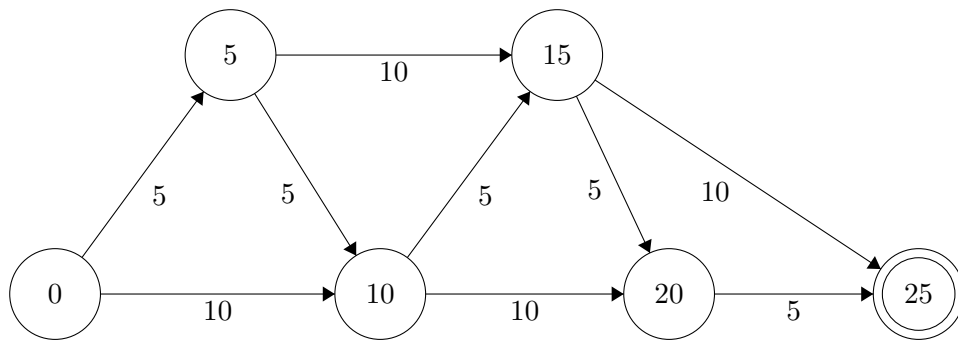
If a table has both properties we can define an acceptor  $M(S, E, T)$  which consists of:

- A state set  $Q = \{row(s) : s \in S\}$ .
- An initial state  $q_o \in Q = row(\lambda)$ .
- Set of accepting states  $F \subseteq Q = \{row(s) : s \in S \wedge T(s) = 1\}$ .
- Transition function  $\delta : \delta(row(s), a) = row(s \cdot a)$ .

This acceptor can be defined as a deterministic finite automata (DFA) which is a model derived from the implementation. A DFA has internal control states and transitions between these states based on certain inputs from the environment [Sil13]. A simple example is a coffee machine that accepts only 5 and 10 cent coins. At this machine coffee costs 25 cents. A possible automaton for this machine is shown in Figure 2.6. Here, the states denote the amount of credit and the transitions what coin is inserted in the machine.

In practice, systems do not distinguish accepting and non-accepting states but rather produce some output when given an input [Raf+05]. This behavior is captured in Mealy machines and can be described similarly to a DFA:

- A set of states  $Q$ .
- An initial state  $q_o \in Q$ .



**Figure 2.6.** DFA of example coffee machine [Sil13].

- $\Sigma$ : finite input alphabet.
- $\Gamma$ : finite output alphabet.
- Transition function  $\delta : Q \times \Sigma \rightarrow Q$ .
- Output function  $\gamma : Q \times \Sigma \rightarrow \Gamma$ .

The difference between a Mealy machine and a DFA is that a Mealy machine does not only move to a new state upon an input symbol, but it also produces an output symbol.

## Main Loop

Here we see the pseudocode of the  $L^*$  algorithm. It produces an acceptor  $M$  given an alphabet  $A$ .

**Data:** Alphabet  $A$

**Result:** Acceptor  $M$

Initialize  $S$  and  $E$  to  $\{\lambda\}$  ;

Ask membership queries for  $\lambda$  and each  $a \in A$  ;

Construct initial observation table  $(S, E, T)$  ;

```
while  $(S, E, T)$  is not closed or not consistent do
  if  $(S, E, T)$  is not consistent then
    find  $s_1, s_2$  in  $S$ ,  $a \in A$  and  $e \in E$  such that  $row(s_1) = row(s_2)$  and
       $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$  ;
     $E \cup a \cdot e$  ;
    extend  $T$  to  $(S \cup S \cdot A) \cdot E$  using membership queries ;
  end
  if  $(S, E, T)$  is not closed then
    find  $s_1 \in S$  and  $a \in A$  such that  $row(s_1 \cdot a)$  is different from  $row(s)$  for
      all  $s \in S$  ;
     $S \cup s_1 \cdot a$  ;
    extend  $T$  by asking membership queries for missing elements ;
  end
end
create acceptor  $M$  ;
query  $M$  for equivalence ;
if teacher responds with yes then
  output  $M$  ;
else
   $S \cup$  counterexample  $t$  and all its prefixes ;
  extend  $T$  by asking membership queries for missing elements ;
  perform mainloop ;
end
```

**Algorithm 1:**  $L^*$  Algorithm [Ang87]

### 2.2.2 Example

We demonstrate the algorithm by providing an example [Kre]. Given a learner that wishes to learn the following unknown set:



$U := \{w \in \{a, b\}^* \mid \text{number of } a\text{'s is even and number of } b\text{'s is even} \}$

Members of  $U$  are  $aabb$ ,  $ab$  and  $\lambda$ , whereas  $aaab$  is not.

## Round 1

We start with initializing both  $S$  and  $E$  to  $\lambda$ . We build the following observation table:

	T1	$\lambda$
$S\}$	$\lambda$	$T(\lambda \cdot \lambda) = 1$
$S \cdot A\}$	$a$	$T(\lambda \cdot a) = 0$
	$b$	$T(\lambda \cdot b) = 0$

**Table 2.2.** Initial observation table.

Every round we must check whether the table is closed and consistent. If that is the case, we can build the acceptor. Next, we query the teacher for equivalence. If the created acceptor is equivalent then we have generated the model.

- Closed: for all  $t \in S \cdot A$  there exists an  $s \in S$  such that  $row(t) = row(s)$ . This is not the case. If we take  $t = a$ ,  $T(t) = 0$ , there is no  $s \in S$  such that  $row(s) = 0$ .

In order to fix this, we must find  $s_1 \in S$  and  $a \in A$  such that  $row(s_1 \cdot a)$  is different from  $row(s)$  for all  $s \in S$ . This is the case for  $s_1 \cdot a = \lambda \cdot a = a$  as  $T(a) = 0$ . Now, we extend  $S$  with  $a$  and query for membership.

## Round 2

After querying for membership, we obtain the following new observation table:

T2	$\lambda$
$\lambda$	1
$a$	0
$b$	0
$aa$	1
$ab$	0

- Closed: we can find  $s \in S$  such that  $row(s)$  is equal to  $row(s \cdot t)$  where  $t \in S \cdot A$ .

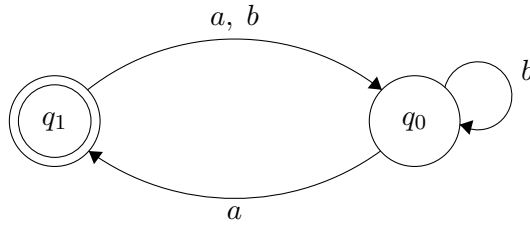
- **Consistent:** there exists  $s_1, s_2 \in S$  such that  $row(s_1) = row(s_2)$  and for all  $a \in A$  holds  $row(s_1 \cdot a) = row(s_2 \cdot a)$ . This holds if we take  $s_1 = s_2 = \lambda$ .

Now we compute the acceptor  $M_1$  as follows:

- Set of states  $Q$ :  $\{row(s) | s \in S\}$ . This results in 0, 1.
- Initial state  $q_0 \in Q$ :  $row(\lambda) = 1$ .
- Set of accepting states  $F$ :  $\{row(s) | s \in S \wedge T(s) = 1\}$ , this results in  $s = \lambda, T(s) = 1$ .
- Transition function can be expressed in the following table.

$\delta$	$a$	$b$
0	1	0
1	0	0

If we visualize it, we get the following DFA:



However, we can see that this model is not correct. We can find the counterexample  $bb$  which has an even number of  $a$ 's and  $b$ 's but is not accepted. The learner receives  $bb$ , adds it and its prefix  $b$  to  $S$ .

### Round 3

The learner continues with the extended  $S := \{\lambda, a, b, bb\}$ , queries for membership and constructs the following observation table:

T3	$\lambda$
$\lambda$	1
$a$	0
$b$	0
$bb$	1
$aa$	1
$ab$	1
$ba$	0
$bba$	0
$bbb$	0

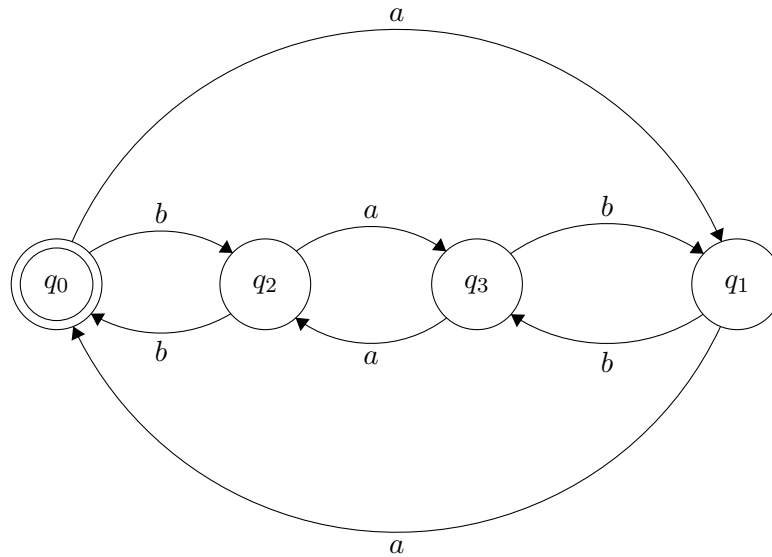
We can easily see that this table is closed. However, it is not consistent. If we take  $s_1 = a$  and  $s_2 = b$  then the first condition where  $row(s_1) = row(s_2)$  holds. However, it does not hold that for all  $a \in A$   $row(s_1 \cdot a) = row(s_2 \cdot a)$ . In particular  $row(aa) \neq row(ba)$ .

We fix the inconsistency by choosing  $s_1, s_2 \in S$ ,  $e \in E$  and  $a \in A$  such that  $row(s_1) = row(s_2)$  and  $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$ . This can be achieved by choosing  $a$  and  $b$  for  $s_1$  and  $s_2$  respectively. If we choose  $a \in A$  we see that the second condition holds  $T(a \cdot a \cdot \lambda) \neq T(b \cdot a \cdot \lambda)$ . We add the value  $a$  to  $E$  and continue with expanding the observation table .

#### Round 4

After expanding  $E$  with  $a$ , we can query for membership and obtain the following observation table:

T4	$\lambda$	$a$
$\lambda$	1	0
$a$	0	1
$b$	0	0
$bb$	1	0
$aa$	1	0
$ab$	1	0
$ba$	0	0
$bba$	0	1
$bbb$	0	0



**Figure 2.7.** Model of unknown set  $U$ .

We can see that this table is closed and consistent. However, if we create a DFA we can still come up with a counterexample. So we continue the iterations until we find a closed, consistent observation table that has a correct model. These steps are not shown here, but the final model of the set of words is shown in Figure 2.7.

Given a DFA it is possible to transform it into a Mealy Machine. This can be achieved by defining a mapping from inputs and outputs of the machine to the letters in the alphabet. However, this increases the size of  $\Sigma$  which is bad from a performance aspect [SG09].

Another option is to tweak  $L^*$  to directly infer a Mealy machine. For this, we need to change the structure of the observation table. Instead of storing 1 or 0 if a given string  $t$  is member of the set, we store the output of the system when provided the string and the current state  $\gamma(q_i, t)$  [SG09; Nie03]. In Chapter 3 we use this approach to infer a Mealy machine from the QUIC implementation.

## Protocol Analysis: Set-Up

In this chapter we describe the analysis we have performed on the server implementation of Google's version of QUIC. We started with learning a model from the implementation. This method is applied since we can test whether the system is conform its specification. In other words, we can test if it satisfies its requirements. These are usually written for positive behavior (e.g. successful connecting to the server).

The next method we used is fuzzing. It is used to capture unwanted features that never made it to the requirements, but were implemented at a later stage of development. It can also be used to test against negative behavior which was captured in negative requirements [Tak09]. Additionally, we also use it to test the robustness of the implementation. Even if the protocol is implemented correctly, if the implementation is not very robust then it can show unexpected behavior when it receives wrong input.

We choose to use fuzzing because of three reasons. First, Fuzzers have proven to be effective in the past by revealing bugs. A good overview can be found in [Zala; LLV18]. Fuzzers were also able to find previously found bugs much faster than manual inspection. This was the case with the Heartbleed vulnerability [MS18; Syn14]. Next, Google uses fuzzing as a testing technique within their software development lifecycle. We believe that they have fuzzed the QUIC implementation. However, their exact method and results are not publicly published. This makes it interesting for us to do it and see whether we can find interesting results. Third, the differences between February and June 2018 in the source code would not result in completely different state machines. Therefore, it is more interesting to apply a new analysis rather than performing the same analysis on a more recent iteration. We came to this conclusion based on the changelog in the forum for the QUIC protocol [Ham18].

### 3.1 State machine inferencing

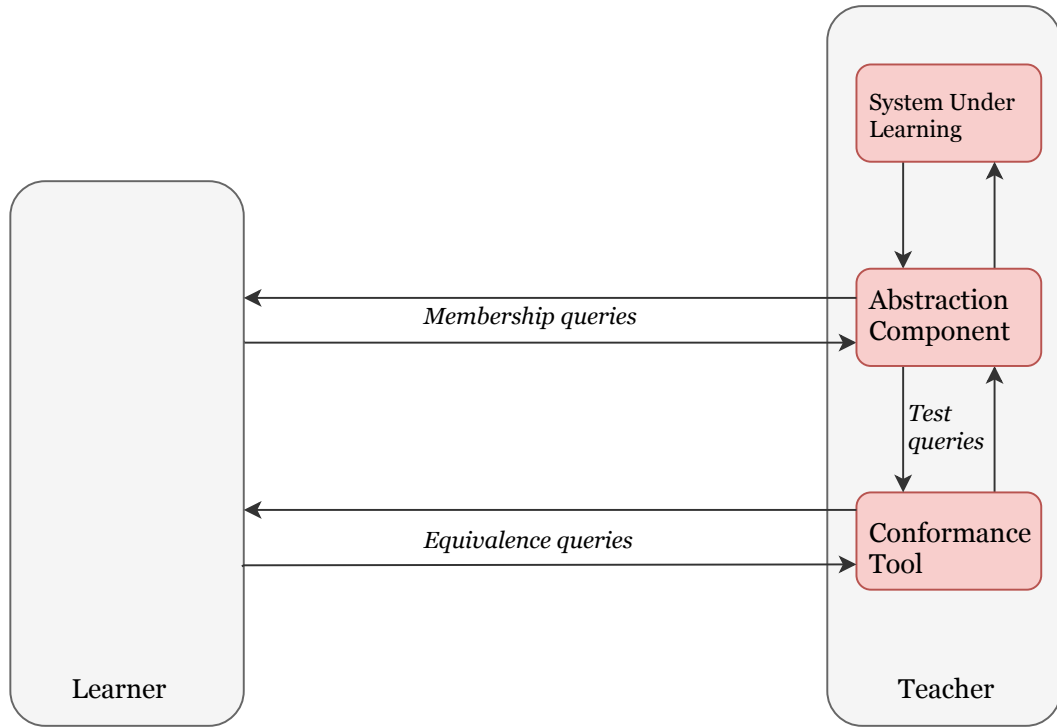
In state machine inferencing we have the learner that communicates with the teacher. At the teacher we have the abstraction component that concretizes abstract messages from the learner or teacher into valid QUIC requests. These messages are then send

to the System under Learning. The learner asks queries to learn more about the initially unknown empty set, until it is able to output the Mealy machine from the QUIC implementation.

If we look at the set-up in more detail, we can see that it consists of four parties, see Figure 3.1:

- First, we have the *System under Learning (SUL)*. In our case this is the QUIC server. It has the task to handle requests made by a QUIC client (typically implemented in browsers). In our set-up the requests are made by the learner in the form of membership queries or equivalence queries.
- Next, we have the *learner*. As was mentioned in the previous chapter, we use the  $L^*$  algorithm to infer a Mealy Machine from a given implementation. It does so by making two types of queries to the SUL. The learner is not a QUIC client. Therefore, it does not know how to make valid requests to the server. In order to solve this, we have a third party.
- This party is *abstraction component*. It is located between the learner, the conformance tool and the SUL. It has the task to receive and transform abstract queries from the learner and the conformance tool to concrete QUIC requests. These are then forwarded to the server. The concrete QUIC response from the server is also abstracted so that the learner and conformance tool understand it.
- Once the learner has made enough membership queries it can build an hypothesis. The *conformance tool* tests whether it is correct. It does so by making random test queries. If the output of these queries match the hypothesis then it was correct. Otherwise, a counterexample is returned. The conformance tool can only make a limited number of tests. Therefore, we are never certain that a learned model is completely correct [Vaa17].

The abstraction component offers some benefits. It provides an abstract representation of the SUL to the learner. This ensures that, in our case, the learner does not consider two REJ messages that only differ in connection IDs and nonces as distinct responses to the same query. This limits the number of states in the model. If we would omit this component, learning models would not scale to realistic applications (e.g. botnets) [Cho+10; Vaa17]. Next, it allows for a simple learner which can be used for different SULs by using different abstraction components.



**Figure 3.1.** State machine inferencing set-up [Vaa17].

The open-source Chrome project, Chromium, provides the QUIC server which we can use as the System under Learning. This does not need any adaptations since the abstraction component provides the server with valid QUIC requests. Although, these requests might be send in an unusual order.

For the abstraction component we decided to create our own version. This is because we did not manage to adapt the original Chromium client to send QUIC messages in arbitrary order. This is required, as the learner queries in random order and based on this, the component should create valid requests. In the next section we describe how we implemented this component.

For the learner we have two options. We can interact directly with the  $L^*$  algorithm. This can be done by using LearnLib [Lea18], which is a Java library that implements the  $L^*$  algorithm. In that case, we must connect to the abstraction component manually and set up the learning algorithm, set up the conformance tool. There are good examples available on [Lea18] to achieve this. Another option is to use StateLearner [Rui18] which is a wrapper for LearnLib developed by Joeri de Ruiter. It allows to create a learner quickly by merely setting some parameters in a configuration file. We chose for the last approach due to its simplicity and speed.

In the next few sections we describe some encountered obstacles and how they were dealt with.

### 3.1.1 Implementing the abstraction component

As was mentioned in the last chapter, Google’s version of the QUIC protocol lacks some documentation. Therefore, creating the abstraction component was a challenge since it needs to be able to craft valid QUIC requests and parse responses from the server. This was even harder, since most QUIC messages (except connection establishment) are encrypted. Therefore, network analyzing tools such as Wireshark<sup>1</sup> are not very useful.

We started with trying to adapt the existing client into an abstraction component. It was hard to find our way in the code. We used a static code explorer called SourceTrail<sup>2</sup> but this did not help. Ironically, this was created by an intern working at Google who also had trouble with the large code base of the Chromium project [Gra16]. We decided not to continue exploring the client.

Another attempt was to use code of another client. However, due to the lack of documentation there are few other clients that implement the latest version of Google’s QUIC [Lar18]. Eventually, we found a ported version of the original client in GoLang [Cle18]. It is developed by Lucas Clemente, who is a software engineer at Google in Zürich. This implementation allowed us to easily add breakpoints and follow the execution of a request with its data.

This helped in understanding the protocol messages and allowed us to manually create the abstraction component. For this, we used a Python library called Scapy. It is able to send, sniff and forge network packets [Bio18b]. It has been used in similar studies in the past with success [Ver16; FB18]. Using this library, the available documentation and thoroughly reverse engineering both QUIC clients, we managed to build the abstraction component.

We were able to set up a connection rather easily. However, key establishment and encryption formed a bottleneck. This is because of the two different key generation steps. Depending on whether the client already received a SHLO message, the key generation differs. Additionally, there is the unknown fixed nonce (as discussed in the previous chapter).

Another reason that made cryptography hard in the abstraction component, was the use of AES-GCM with a 12 byte tag. In general, the tag is 16 bytes long. These shorter tags are not supported by cryptographic libraries for Python (such

---

<sup>1</sup>See <https://www.wireshark.org/>

<sup>2</sup>See <https://www.sourcetrail.com>



as Cryptography<sup>3</sup>). Even though it is allowed to use these shorter tags, the security strongly depends on the tag size [Dwo07]. We believe that the shorter tag was chosen due to performance reasons. In order to perform cryptographic operations with QUIC, we had to use the implementation of Lucas Clemente via sockets [Cle18]. This introduced some latency, however this approach is easier than manipulating existing AES implementations.

Our client is not very advanced. It is able to send messages in arbitrary order but does not implement all the different frame types. We decided to limit to the following messages (i.e. learning alphabet). Our client is also able to acknowledge received packets. Although, the documentation was not clear on these frames we tried to create these frames using the other client from Lucas Clemente.

---

<sup>3</sup>See <https://cryptography.io/en/latest/hazmat/primitives/aead/cryptography.hazmat.primitives.ciphers.aead.AESGCM.encrypt>

Learning symbol	Concrete QUIC Request	Explanation
INIT-CHLO	Initial CHLO request	Starts a fresh new connection and is used when connecting to a previously unknown server.
FULL-CHLO	Complete CHLO Request	Uses the data from the initial CHLO with the missing tags received in a rejection message. This does not create a new connection ID unlike the INIT-CHLO message. Instead, it uses the previous connection ID. If there is no such value, then it defaults to -1.
ORTT-CHLO	Complete CHLO request	Starts a fresh connection but uses the stored tags which were missing from a previous initial CHLO.
GET	HTTP/2 GET Request GET (Stream Frame)	Makes an HTTP/2 GET request for the fixed domain <code>www.example.org</code>
CLOSE	Connection Close Frame	Notifies that the connection is being closed. If there are streams in flight, those streams are all implicitly closed when the connection is closed.

The source code of our abstraction component is available on GitHub, see <https://github.com/aredev/quic-scapy>.

### 3.1.2 Dealing with non-determinism

One important requirement when learning a model is determinism. We want to observe the same output when sending the same input. Otherwise, learning a model will never terminate as it encounters new transitions every time for the same input. For example, when sending the same sequence A B C we always want to observe the same response 1 2 3.

When running the learner for the first few times, we encountered some non-determinism. We can distinguish two types of non-determinism. First, we have it at query level. This means that sending the same sequence of symbols from the input alphabet (in our case, these symbols represent QUIC requests) results in different responses some time. For example, when sending the sequence A B C it may occur that there is a hiccup (e.g. client misses a response) which results in a different response than other times.

In our case we had a second type, which is at symbol level. This means sending a single symbol B does not yield the same response every time or we are not sure what the response to symbol B is.

The non-determinism is caused by two reasons. First, there is some response uncertainty. If the abstraction component makes a request to the server, it does not know whether the next received response is for the newly created request or if it is an acknowledgement, retransmission of a previous message, or something else. Initially, we considered the first received response after sending a request as the response to that request. However, this might not always be the case, as we see later. Second, we are not certain if a server responds at all. This can occur if it received an incorrect message.

We noticed that some packets are retransmitted frequently. The reason for this is twofold. On the one hand, our abstraction component is not as efficient as the original client. We compared the time it takes our implementation with the GoLang version to set up a connection, make an HTTP request and close it. If we take the median<sup>4</sup> from 10 rounds, our implementation is able to do so in 1,94 seconds, compared to 0,22 in GoLang's version. On the other hand, the QUIC server just retransmits fast [SI15]. It does so to increase performance.

Our client is not as efficient as the original client because of three reasons. First, a compiled programming language like C++ or GoLang is faster than a scripted language like Python which we use. Second, for encrypting and decrypting values we use a separate process. In order to communicate with it, we use sockets which introduce some latency. Third, our client in general is not created with efficiency in mind. The goal was to be able to just create valid QUIC requests and send them to the server and parse the responses received. Given the lack of documentation, this was a challenge on itself. This caused the created client to be rather simple.

Dealing with the response uncertainty is tricky, since there is no link between the request and a response. This means we can never link a response to a request with

---

<sup>4</sup>We take the median to ensure that outliers do not affect our results which is the case when taking the average.

complete certainty. In general, we can assume this based on the order in which messages are sent. We could send a request A B C, three times like A B C RST A B C RST A B C. After every iteration the system is reset to its initial state, here shown with RST. If the response would be 1 2 3 1 2 3 1 2 3, we use the majority response to find the probable response to a request. This approach has been used in [FB18; Ver16].

However, we can not use this approach. This is because QUIC uses streams which means that packets might not be delivered sequentially when one of the packets is lost. The problem with sequential delivery is head-of-line blocking (see Section 2.1.1) which was specifically addressed in QUIC [Lan+17; Ham17]. We cannot repeat the requests as mentioned earlier since we have no guarantee that the responses of the server are received in the order in which we sent the requests. Another reason why this would not work is because the StateLearner tool we used, sends queries one-by-one. We do not know the complete query on forehand, such that we could repeat it three times. However, it is possible to adapt the tool, such that it does send the complete query.

If we were able to use the original, advanced QUIC client developed by Google, we could have used this sequential approach. This is because this client is able to parse and reorder the messages, such that it appears like it has been received in that order. Unfortunately, we were not able to use the original client as the abstraction component and our implemented client is basic. Therefore we had to come up with a different solution to deal with the response uncertainty.

One solution for the response uncertainty is to assume a fixed response to a certain request. If the received response does not match the predefined fixed one, it is an invalid answer. This makes the system deterministic but model learning is only effective for normal runs. Responses of unexpected runs are not parsed correctly, as it would only show that the response is invalid but does not provide more information.

Instead, we chose a different approach to gain more confidence in the response. As was mentioned earlier, we are not certain if an observed response is the response to the previous request or if it is something else. In order to increase the level of certainty we repeat a single message three times. By doing this, we can be more certain that a response we observe more often after a certain request is indeed the response to that request. For example, when sending the query A B C, we would send A A A B B B C C C. If we observe the response 1 1 1 2 2 X 3 3 3, then we are more certain that the response to message A is 1, 2 to message B and 3 to message C. Despite the one incorrect response which was received when sending message B, we can still consider 2 as its response because of the other two messages

received. We increase the certainty of a response to a single request. If we apply this repetition to all requests, then we also have certainty of the response to the entire query.

However, this solution is not completely waterproof. It is not possible for every message to be send three times back to back and receive the same response every time. For example, it is not possible to close the same connection three times. Therefore, we had to exclude those requests which we can not make three times without changing the state of the system (e.g. 0-RTT CHLO and Close) and perform some manual filtering for the responses. This is explained in more detail in the next chapter.

Dealing with the issue where the server would not respond at all is easier to address, since we can run a large number (e.g. 10000) of requests and compute the median response time. If during model learning the server does not respond within this time (plus-minus some delta value), we can assume the server will not respond. In our case, we chose to multiply the median value times three. We chose to triple the 'expiration timeout' because of overhead (sockets, sleep periods, database read/writes). By experimenting with the scalar, we found that a multiplication of three was enough. A shorter timeout time would result in more messages being considered as expired.

## 3.2 Fuzzing

The other type of analysis we performed was fuzzing. It is a software testing method used for robustness. It feeds malformed and unexpected data in systems [Tak09]. Examples include integer overflow, underflow, repetition of elements and unexpected elements. Fuzzing is a relatively new in software development, but some software engineers used a similar technique already in the 1980s. They used a tool called *The Monkey* which would behave like an angry monkey, banging on the mouse and keyboard generating random inputs and drag events [Tak+08; Wik17].

Basically, there are only two parties when fuzzing. There is the system which we want to fuzz and the fuzzer. The fuzzer generates input which is fed into the system. If we look in more detail we can observe the following parties involved [Tak+08]:

- **Fuzzer:** Library that crafts unexpected inputs and sends it to the System under Test. Some advanced fuzzers are able to perform some analysis on the response. There are a few types of responses that may stem from a fuzz test.

First, there is a valid response. Next, we have an error response (which can be considered a valid response from a protocol perspective). Third, anomalous response (unexpected but nonfatal reaction). An example is a slowdown or responding with a corrupted message. Last, there is a crash or other failure. Fuzzers can use these responses to tweak the input to make it more ‘effective’.

- **System under Test:** Software that the malformed input is fed to. It is also monitored for unexpected behavior caused by this input.

We can categorize fuzzers in the following categories [Tak+08]:

- **Static and random template-based fuzzer:** These fuzzers have little protocol awareness and have no dynamic functionality. Typically used for simple request-response protocols or file formats.
- **Block-based fuzzer:** These fuzzers can perform little dynamic functionality such as checksum calculation and length values.
- **Dynamic generation / evolution-based fuzzer:** These fuzzers do not necessarily understand the protocol but use a feedback loop from the target system to learn how to mutate their inputs. This feedback loop would show that the input has triggered some new paths in the code which previously have not been provoked.
- **Model-based / simulation-based fuzzer:** These fuzzers have a model or simulation of the system under test. Not only message structures are fuzzed, but also unexpected messages in sequences can be generated.

The goal of fuzzing is to find security-related defects or any other flaws that result in a denial or degradation of service or any other undesirable behavior.

Fuzzers that start with random input and mutate that, are inefficient (e.g. template-based fuzzers). They are only capable of finding naive programming errors. Therefore, it was required to make more intelligent fuzzers (e.g. evolution-based fuzzers) to find bugs that are ‘buried’ deep within the system under test [Tak+08]. For our research we used a random template-based fuzzer and a evolution-based fuzzer. We describe the reasons for these choices in the next sections.

Many software development companies (e.g. Microsoft [God+08] and Google [Rom18]) incorporate fuzzing in their software development lifecycle. This is because it is important to not only focus on the positive requirements but also explore the

challenges that may occur in the negative requirements [Tak09]. By performing such test early in development, the costs of fixing bugs are not as high as when it would be encountered at a production setting.

Fuzzing networking protocols is important for two reasons. First, robust implementations of networking protocols are important as these form the communication infrastructure of the modern time. Therefore, there is a growing burden on the reliability of these implementations. In addition, crafting packets and randomizing them manually is not very practical [Zha+14]. A fuzzer is very helpful during this process.

### 3.2.1 Naive fuzzing

We started with using the built-in fuzzer in Scapy. Since we invested effort and time into building the abstraction component from scratch (see Section 3.1.1), it was trivial to incorporate the fuzzer. We called the `fuzz()` method on our QUIC packets.

This method can be seen as a random template-based fuzzer. Given the structure of a packet with its fields, it replaces any fixed value with a random one [Bio18a]. It does not adapt fields that are computed (e.g. checksums or length fields).

The results are described in the next chapter.

### 3.2.2 Advanced fuzzing

We continue with dynamic generation or evolution-based fuzzers.

In contrast to the `fuzz` function, this type of fuzzer is more advanced. It uses a feedback system such that it can mutate the input according to the behavior of the system. This feedback system is injected while compiling the program and is able to capture branch coverages. This enables it to trigger new interesting execution paths.

We discuss two different type of evolution-based fuzzers. Both take roughly the following steps:

1. Load an initial set of provided test cases (input data) into a queue.
2. Take next input from the queue.

3. Trim the test case to its smallest form such that it doesn't change the observed behavior of the system.
4. Mutate the input using some fuzzing strategies (e.g. bit flips, incrementing or decrementing integer values or replacing hardcoded integers with interesting values such as `MAX_INT` or `-1` [Zal14]).
5. If any of them result in a new state transition, as observed by the feedback system, add it as a new entry in the queue.
6. Go to step 2.

### **American Fuzzy Lop (Afl)**

American Fuzzy Lop is a brute force fuzzer that uses a compile-time feedback system and genetic algorithms to find interesting inputs that trigger new internal states in a specified binary. It has found a number of bugs across a large range of applications [Zala; Zalb].

We are able to apply this method, since we have access to the source code. Therefore, we can 'inject' the feedback system into the source. This feedback system has only a minor impact on the performance [Zalb].

We now mention some promising, but unfortunately failed attempts.

Incorporating the instrumentation is done by compiling the code using a special supplied compiler from Afl. In our case compilation would succeed. However, when running the fuzzer after compiling, it would stop directly and mention that the instrumentation was not present in the targeted binary.

To fix it, we used the compile argument `use_afl=true`, which is present in the Chromium build system. However, this did not fix it since it failed on compiling the Afl source code. We then decided to compile the Afl source code separately from the QUIC server. Compilation succeeded but the instrumentation was still not added to the binary according to the fuzzer.

After inspecting the build script, we noticed that this option would only include the Afl source code as a source code dependency for compilation. It would not add any instrumentation to the binary. We were already able to compile the Afl fuzzer manually, so this command was not valuable for us.



It is also possible to use the fuzzer in non-instrumented mode. This mode disables the feedback system, making the fuzzer not very intelligent. After running the fuzzer in this mode, we noticed that it was not able to detect any crashes occurring in the targeted binary. We tested this by raising a segmentation fault. We believe that this is independent of running the fuzzer in non-instrumented mode, as this would only help the fuzzer in mutating the input.

A solution is to remove any signal handlers. However, this is not a trivial task. This obstacle was also observed in a similar study with fuzzing StrongSwan [San18].

We decided not to continue with Afl and try to use a different evolution based fuzzer.

## LibFuzzer

Another fuzzing method which was mentioned on the Chromium repository was LibFuzzer. It was suggested to use for local development, since it does not require any special configuration and gives meaningful output faster than Afl.

Compared with Afl, there are some differences. First, LibFuzzer only works on functions instead of complete binaries. Next, it requires one to write some code that actually calls the function to be fuzzed [Rom18]. These are also reasons why we started with Afl in the first place.

LibFuzzer is an engine for in-process, coverage-guided, white-box fuzzing [MS16]:

- In-process: It does not launch a new process for every test case, but rather mutates inputs directly in memory. This differs from Afl that terminates a process after every test. This impacts the performance significantly, since performing certain required system calls are slow [Zal15]. It is also possible to run Afl in a persistent mode which enabled in-process fuzzing but this is an optional feature.
- Coverage-guided: It measures the code coverage for every input and accumulates test cases that increase the coverage.
- White-box: It uses compile-time instrumentation of the source code.

Writing this function has the benefit that we have more control over the fuzzing process. This enabled us to fuzz a single function. This makes it more efficient and

reliable since we do not need to launch the whole server, which can add overhead and might introduce unexpected behavior. The downside is that we need to know what function we want to fuzz from the source code, this requires some knowledge of the implementation.

We write a fuzz test by implementing a special function. This function receives a data buffer and its length. This is fed into the code we want to test. This is shown in Listing 3.1.

In our case, we want to fuzz the parsing functionality of the QUIC server. It is interesting if we can find a flaw here and trick the server such that a malformed packet is considered to be a valid one and tries to parse it. If there is no proper input validation, it can crash or behave unexpected when trying to parse a packet it does not recognize.

In order to implement this function, we must instantiate the `QuicSimpleServer` class. In the original code, it listens on a socket to incoming connections. However, we cannot use this since we do not use sockets. Therefore, we added a function to the `QuicSimpleServer` that receives the data as a function parameter. This method then calls the internal `ProcessPacket()` method.

**Listing 3.1** LibFuzzer fuzzing code.

```
#include <stddef.h>
#include <stdint.h>

extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data,
                                     size_t size) {

    CallAPIToFuzz(data, size);
    return 0;
}
```

The complete code which was used to fuzz the parser is shown in Appendix A.

LibFuzzer has been used at Google and has found numerous bugs [MS16]. At the time of writing there are 3281 bugs in Chromium that have been found using LibFuzzer. They vary between buffer overflows, usage of uninitialized values and integer overflows. Additionally, it has found bugs in non-Chromium products such as Python, SQLite and Wireshark. LibFuzzer has also been integrated in

Chrome’s Fuzzer Program<sup>5</sup>. It is a program where Google rewards individuals who responsibly disclose vulnerabilities to the Chrome project.

We ran LibFuzzer twice. The first time we do not provide a corpus of sample inputs. Rather, it starts with random input and continues mutating until it finds some unexpected behavior in the function. The second time, we provided a corpus. This was created by exporting 1000 valid and invalid CHLO payloads into a directory. These payloads are generated using our abstraction component from our state machine inferencing analysis (see 3.1.1). The fuzzer can now generate mutations based on the corpus, this makes it more efficient since it is aware of the structure of the input [LLV18].

The results are described in the next chapter.

---

<sup>5</sup>See <https://www.google.com/about/appsecurity/chrome-rewards/>

# Protocol Analysis: Results

In this chapter we describe the results from our analysis with respect to fuzzing and state machine inferencing, as described in the previous chapter. For all experiments we used the the source code from the Chromium repository with commit tag `e611939ed2`. This version stems from early February 2018 and it implements QUIC version 39. This is still used in Chrome's latest stable version `67.0.3396.99`, released on June 22nd 2018. The next version of Chrome will use version 43 of QUIC.

## 4.1 State machine inferencing

We start with describing the results from model learning. We have created two models. We started with modeling the simpler case where the learner does not know about the 0-RTT message. This is simpler because the abstraction component does not need to store the received REJ tags. We continued with learning a model where the learner could also start a connection using a previously received REJ message.

In both tests we use the same default configuration provided by `StateLearner`. It uses  $L^*$  as learning algorithm and uses random queries as equivalence test. The minimum input length is specified at 5 and the maximum is 10. The number of test queries which the teacher should make is set to 100. We use the value 42758 as seed, it was chosen arbitrarily.

### 4.1.1 Without 0-RTT

This model was created in 42 minutes. It required 104 membership queries. The reason for this slow execution is due to some 'sleep' periods of 8 seconds after every reset and 2 seconds between every retransmission that were added to the code. We have added this to ensure that messages in transit are delivered. This lowers the number of wrongly received responses to requests as was mentioned in the previous chapter. These values were chosen arbitrarily and could probably be lower.

The result is a rather simple model with just five states. It is shown in Figure 4.1 and in Appendix B.

We hope to see a path in the model. For example, it should not be possible to receive an HTTP response without setting up the connection by means of an initial CHLO and optionally a full CHLO. This means that the correct path should start with sending a CHLO. This should result in a REJ since the client does not know enough information about the server to send a complete CHLO. With the REJ message, it can send a full CHLO which has enough information. This results in a SHLO message which contains the ephemeral public value of the server. At this point, the connection is successfully established. The client is now able to make GET requests. Since that is an operation which should not change the internal state of the system, it should be possible to make them multiple times and receive the same response each time.

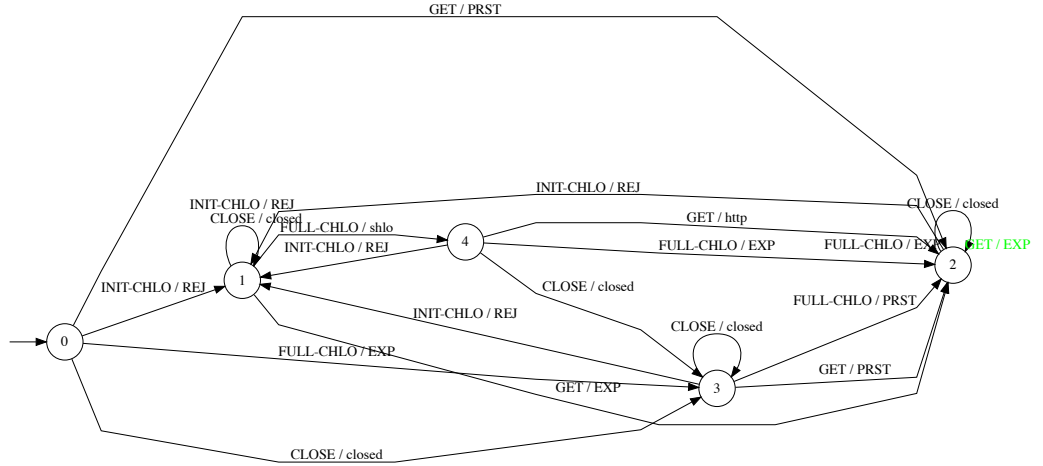
Since we do not have a complete and detailed specification for Google's version of QUIC, we cannot compare the learned model with it. Therefore, we look at transitions we find interesting and need some explanation. All of the expected transitions are present, but we see one interesting transition which stands out.

After the client makes a successful GET request and receives the HTTP response, it is unable to make another GET request on the same stream immediately afterwards. Manual inspection showed that the server responds the second and third time with just an acknowledgment instead of the HTTP response.

This behavior was not mentioned in the documentation. Therefore, we tried to find the cause. First, it could be that requesting the same origin on the same stream ID causes the server not to respond. However, it is not possible to change the stream on which the request is made. QUIC uses stream 3 for transmitting compressed headers for all other streams. This helps in processing of the headers [Cyr+16]. Our second guess was to change the domain and test whether we could make two consecutive GET requests to different domains. However, this requires to send a new CHLO message because the Server Name Indication (SNI) tag needs to be altered to the new domain. Even though we could not verify this, we believe that this behavior is a result of some client side caching that is expected by the server.

We also observe one previously undiscussed result at some responses, namely EXP. It stands for Expired and it is used for when the server did not respond within our expected time limit (see Section 3.1.2).

Another interesting response is with the full CHLO. When sending this message before an initial CHLO, we get an EXP response. However, we would expect it to behave similar to an initial CHLO message and trigger a REJ response. In our implementation of the abstraction component, there is a difference between the initial and full CHLO. In the initial message some tags are not send, they are omitted from the packet payload. The full variant has all required tags and their values



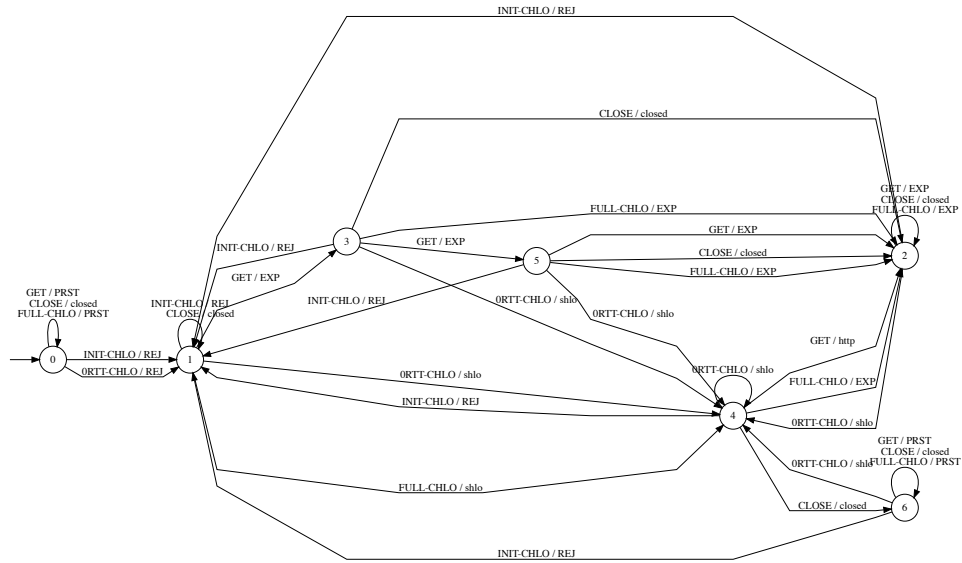
**Figure 4.1.** Learned model of Google's QUIC Server, without 0-RTT.

are initially empty or have some wrong default value. The abstraction component uses the received REJ message to substitute these values with correct ones. As long as it does not receive a REJ message before sending a full CHLO, it continues to send empty/default values with the tags. The QUIC server does not accept these incorrect values and therefore does not respond to these messages which causes them to expire.

There is also a transition where it looks like we can keep closing a connection. Although, it is not possible to close an already closed connection, what it actually means is that a closed connection remains closed when sending a close message. This is a result from our retransmission to ensure consistency while learning the model (see Section 3.1.2). We perform some response parsing/filtering when closing a connection in the abstraction component. This is done to make the model readable, but it requires some explanation. The first time when closing a connection, we just get an acknowledgment. The second and third time we do not get a response, but the messages expire. Instead of responding with EXP to the learner, the abstraction component parses this and transforms it to CLOSED.

#### 4.1.2 With 0-RTT

To learn this model we added one extra symbol (i.e. 0RTT-CHLO) to the input alphabet. This increased the time it took to create this model. The 7 states were created in 73 minutes. It required 225 membership queries and again 100 equivalence queries as this was specified. The model is shown in Figure 4.2 and in Appendix C.



**Figure 4.2.** Learned model of Google's QUIC Server, with 0-RTT.

We expect to see a model that matches our previous assumption. Additionally, there should be a transition after a successful connection establishment (i.e. after receiving a SHLO) that can immediately set up a connection by sending the 0-RTT CHLO. In this case, the client should not receive a REJ message but rather a SHLO. In practice, if the user wishes to achieve the least number of round trip times, it should make a GET request after sending the full CHLO without waiting for the SHLO response. We are not able to model this parallel behavior using our approach.

We can observe the 0-RTT transition. Initially, the client sets up a connection using an initial CHLO message (transition state 0 to 1). Next, it uses the received tags in the REJ message to craft a complete CHLO message which results in a SHLO response (transition state 1 to 4). It is now possible for the client to make a direct request that results directly in a SHLO message (transition state 4 to 4 or state 2 to 4).

While learning this model we encountered some additional non-determinism. A ORTT-CHLO message changes in behavior over the course of three consecutive requests. The first time it triggers a REJ response, the second time it uses that received message to craft a complete CHLO message which results in a SHLO message. The third time it also results in a SHLO, because of the new connection ID it chooses. If we use the approach of the majority response, a ORTT-CHLO message always returns a SHLO, which is not correct.

This problem originated from our implemented abstraction component, since we chose to add received REJ tags directly in the ORTT-CHLO if they are present. Implementing

the ORTT-CHLO in this way was easier. We also believe that it is also the correct functionality of the message.

We could not send the ORTT-CHLO three times back to back, so we have to find a different approach to have response certainty. We perform some manual inspection or apply some response filtering. For example, it is not possible to receive an HTTP response to a initial CHLO message. This could either happen in the implementation of the abstraction component or in the resulted state machine. For example, sometimes a strange response was received on a ORTT-CHLO as a result of a previous retransmission. This response would then be discarded and a correct one was returned to the learner.

The smaller model (see Section 4.1.2) helped us understand the basic run of the protocol. We used this model to decide what responses to consider incorrect and what could actually be considered a valid response in the bigger model. This gives a degree of confidence that our manual correction did not invalidate the learned model.

There was another implementation detail which affected our learned state machine. As was mentioned in the set up chapter, we chose to use the INIT-CHLO and ORTT-CHLO messages to set up a fresh connection (i.e. use a new connection ID). This allowed the transition from state 2 to 1 to be legal. However, if we would have used the previous connection ID, this would have resulted in an expired message. The same behavior is also present in the model without ORTT-CHLO messages.

This is because QUIC accepts packets in a latching fashion. Once an encrypted packet is received, it does no longer accept unencrypted packets [LC16]. But since we use a new connection ID for this, it can start with an unencrypted message.

## 4.2 Fuzzing

In this section we continue with the results from the fuzzing part of the analysis. We conducted two types of fuzzing. We start with the simple fuzzer supplied by Scapy and continue with the results when running LibFuzzer.

### 4.2.1 Naive fuzzing

Since this type of fuzzing is incorporated in Scapy, we can learn a model using the set up mentioned in the previous chapter. This model was generated in 24 minutes and does only contain transitions that have been expired. This means that



the server did not respond to any of the fuzzed messages. This is not an exciting model, therefore it has not been included.

We can explain this because of our naive method of fuzzing. It simply randomizes all fields that are not computed. Examples of computed fields are checksums and lengths. This means that the TAG field, which indicates whether the payload is, for example, a CHLO, is also randomized.

When the server receives this kind of input, it cannot match the TAG to any type it is aware of. Therefore, the server is unable to process the request and does not respond to any of the messages. This causes the timer to expire and mark all responses as expired.

This method of fuzzing did not result in crashes or any other unexpected behavior.

## 4.2.2 Advanced Fuzzing

Since the basic type of fuzzing did not find any weaknesses in the `ProcessPacket` function, we tried a more advanced fuzzer namely LibFuzzer. We fuzzed the server three different times.

First, we fuzzed without supplying a corpus. This means that the fuzzer started with random inputs and mutated it until it observed new states in the execution. As was mentioned in the previous chapter, this method of fuzzing is not very efficient as it can take a lot of time before it is aware of the complex packet structure.

We terminated the fuzzer after it had generated  $2^{23}$  different inputs. At that time it had covered 4536 code blocks. This is a straight-line sequence of code with only a single entry point and a single exit point [GCC]. Additionally, it used 4188 different signals to evaluate the code coverage. The fuzzer executed at 295 iterations per second and used 224 Mb of RAM.

This method did not find any weaknesses in the targeted function either. However, there were two inputs that caused the `ProcessPacket` function to take a long time to finish. These inputs were `ac1` with 28 seconds, `q5` with 2029 seconds of execution.

Afterwards we were not able to reproduce the behavior of slow responses to certain inputs. We executed the fuzzer on our private laptop and believe that hibernating and resuming it caused this effect. We can also support this, because the second

Method	Inputs	Coverage	Features	Speed	Memory (Mb)
No corpus	8388608	4536	4188	295	224
Corpus	8388608	4544	4188	252	231
Corpus, no reduction	8388608	4544	4188	268	230

**Table 4.1.** Summary of fuzzing tests

fuzzing attempt was performed over night. During that time, the laptop remained running and we did not receive any inputs that caused a slow execution.

We continued with providing a corpus with 2000 entries to the fuzzer. After  $2^{23}$  different inputs we terminated the process. At that time it had covered 4544 code blocks. It has used 4188 different signals. Executed at 252 iterations per second and used 231 Mb of RAM.

This time the fuzzer did not find any issues in the ProcessPacket function.

We can observe from the logs and the algorithm description that the fuzzer tries to reduce the input that triggers a certain state. We tried to fuzz the server without this reduction phase. This is possible by running LibFuzzer with the `reduce_inputs=0` option. We applied this, because we are not necessarily interested in the shortest input but rather in any input that can trigger unexpected behavior at the system under test. By disabling the reduction phase, the fuzzer can generate more different inputs. One of these inputs might trigger some unexpected behavior.

However, the results are almost similar to the one with the reduction enabled. The only difference is the higher number of executions per second compared when the reduction phase was enabled. This can be explained since the fuzzer only needs to create new inputs and send it to the fuzzing target instead of performing the reduction step between the input mutation.

The results of all three fuzzing attempts are shown in the Table 4.1.

# Conclusion

In this chapter we summarize the key concepts of this thesis. We put the thesis in a broader context. In addition, we discuss our approach and mention some ideas for future work.

## 5.1 Summary

QUIC is a fairly new transport layer protocol developed by Google. It was introduced for several reasons. First, it improves performance by reducing the number of round trips needed to set up a connection between two application endpoints. QUIC is able to establish a connection with an unknown origin with 1 RTT. If the client previously has connected to the same origin this can be reduced to 0 RTT. In TCP this is 1 RTT, regardless of any previous connections. Another reason to introduce QUIC was the need for establishing secure (authenticated and encrypted) connections faster. To achieve this in TCP, we need to use TLS which adds another 2 RTT. With QUIC, connections are secured by default and does not increase the RTT. The third reason for introducing QUIC is the need for deploying changes rapidly in network protocols. The internet continues to evolve, but updates in TCP or UDP requires updating the kernel in Operating Systems, which is not very flexible. Therefore, QUIC is built in user-space on top of UDP. This makes it easier to deploy changes which enables QUIC to better keep up with the developments of modern internet.

There are currently two versions of the protocol. First, there is Google's version. It has been used in production by Google for several years. The source code is available as part of the Chromium project. However, the documentation is not up-to-date with the implementation. The next version is the draft from IETF. IETF is standardizing QUIC so that it can be used by other parties and may replace TCP in the long run. The drafts are updated very frequently. This causes the implementations to fall behind on the draft specifications.

We chose to use Google's version since it is used in a production setting, which makes it more interesting to conduct research on. Similarly, IETF's latest draft version is not implemented and is not used in practice.

During the implementation of a specification it is possible that some errors are made. This can be caused because the specification was unambiguous or it did not cover all edge cases. This can result in different implementations that are incompatible. Another effect is that it can produce errors on some unexpected input.

Therefore, it is important that implementations are thoroughly tested. We performed two different analyses of the protocol implementation. First, we performed state machine inferencing. It allows to test whether an implementation matches the positive requirements as specified in the documentation. We do so by learning a model of the implementation. Next, we performed some robustness testing against unexpected or malformed input in the form of fuzzing. This type of testing is used for negative or unhandled requirements.

We could not fully compare the inferred state machines with its specification due to the lack of detail. We found one transition in the state machine which was not explicitly mentioned in the specification. With respect to fuzzing, we did not find any inputs that caused a crash or any other disruption of service in the server implementation.

## 5.2 Discussion

In this section we discuss our why we did not find any weaknesses in the input validation and other deviations in the learned model. We start with fuzzing and finish with state machine inferencing.

As mentioned, the implementation of QUIC was part of the Chromium project. It aims to build a safer, faster and more stable method of using the web. This project is maintained by Google, which at its core is a software engineering company. They spent a lot of time and effort into finding new ways of creating secure software and testing whether software already built is secure. They have published many articles<sup>1</sup> regarding security in software development. Therefore, we can say that it is a company that has security in mind when developing software. In general, this makes it harder to find weaknesses in their applications.

Google integrates published findings in their daily operations. For example, they have implemented fuzzing as a new type of unit testing since traditional testing is not enough. It enables them to write more code. This code needs to be written faster, but it still needs to be correct, stable and secure. This can be achieved by

---

<sup>1</sup>See <https://security.googleblog.com>.

using fuzzing, it also helps to keep the development costs under control, since more bugs can be found earlier in the development process [Vyu17].

Despite their efforts, it does not mean that Google's products are without bugs. They use strategies such as fuzzing to find flaws. For example, in their Machine Learning framework TensorFlow they have found many bugs using LibFuzzer.

One reason we did not find weaknesses in the input validation is that previous versions of QUIC has already been successfully researched. Especially, if we consider that it only exists for a few years. In 2015 it was found that connection establishment could be failed with simple bit flipping some parameters and sending certain packets multiple times during the handshake [Lyc+15]. This was addressed afterwards by Google. Now that they perform fuzzing during their software development cycle, it is becoming harder to find such flaws as an external party.

Not only academic researchers or software developers at Google, but anyone can fuzz Google's open-source projects. They offer the tools, computation power and reward people for responsibly disclosing found vulnerabilities. This resulted in 3292 bugs in Chromium found using only LibFuzzer. This way Google's projects are tested multiple times by different parties. This does not guarantee that the software is completely without bugs, but it gives more assurance. Google also offers their computation power so that it can be used by others to fuzz their own projects.

Regarding state machine inferencing, we believe that Google internally has more documentation regarding QUIC and also has a state machine. However, with the upcoming standardization by IETF they probably do not want to publish documentation which will be outdated within few months anyway. In addition, they can use QUIC in their production setting as a sort of laboratory in which they can test new features. Some of them can then be used in the standardized version of QUIC. Documenting every test feature and afterwards removing it, is a cumbersome process.

We can conclude that the official server has implemented proper input validation. We were not able to find any weaknesses when fuzzing it using multiple approaches. This is because they incorporate fuzzing in their software development lifecycle. Additionally, some weaknesses with respect to input validation were already found and fixed. With respect to model learning, we found that it is hard to deal with non-determinism in this complex networking protocol. In the case of QUIC, speed and encryption are important and deeply integrated within the protocol. It is not possible to disable these two features as this would leave us with a slightly polished version of UDP. This would make model learning easier, as we do not need to worry on retransmissions and computing correct cryptographic keys. Therefore, we suggest to use the original client when inferring models of advanced protocols

such as QUIC. In our study we invested time and effort into building our client. In the end, we noticed that our client was not as efficient and advanced as the original client. Therefore, it suffered from retransmissions and response uncertainty which we had to handle by implementing certain tricks. Another lesson we learned is that we should have analysed the QUIC client by Lucas Clemente instead of reading the documentation. The documentation is incomplete and not very specific on certain aspects (e.g. acknowledgements). The client implementation was more helpful, since we could see it running with data and see what computations it made.

## 5.3 Future work

In this section we describe some ideas for future work.

One idea is to perform the same analysis on a more recent implementation of QUIC. However, the changelog showed very minor adaptations between the version in February which we have used and the current iteration. So we do not expect a difference between our learned model and the one from a more recent iteration.

Another idea is to handle non-determinism in a different way. Our implemented client was basic, not very efficient and suffered from retransmissions. Therefore, we implemented some tricks and performed manual inspection to handle non-determinism. For a next step, it would be better to spend more time on analysing an existing client and adapting it, such that it can be used as a learner. It is also possible a more advanced custom client.

In this thesis we looked at Google's QUIC implementation but with the standardization of QUIC in progress, it is interesting to infer a state machine from an implementation of the standardized version of QUIC as well. It can show some internal differences between the two versions. From the draft specification we know that there are some naming differences as well as changes in the header layout.

Other than comparing differences, we can infer a state machine to match against requirements. We assume that the standardized specification will be more thorough and therefore is more suitable for checking whether it matches certain requirements. It is also possible to add model checking to this process. This allows us to check certain properties of the learned model.

Another step is to look at the client side of the protocol. We can infer the state machine from the client implementation and fuzz the client.

Other vendors will also make implementations of QUIC when the standardized specification is released. It is interesting to learn models from those different implementations and check whether there are differences between distinct vendors. If that is the case, this can be used to enhance the specification (e.g. by removing ambiguous sections). While performing our study, there was only one vendor that implemented the latest version of QUIC.

There are also other fuzzers than Afl and LibFuzzer. Two interesting ones are VUzzer and AutoFuzz. The first one is an application-aware evolutionary fuzzer. It was shown that VUzzer was able to find bugs with less inputs compared to Afl. It does so by analyzing application behavior. The second one is more specifically created for network protocol implementations as it functions as a man-in-the-middle. It is able to learn message syntax, fields and types by applying techniques from bioinformatics. It was able to find bugs in implementations of the file transfer protocol (FTP).

We did not use them for this study because we did not get them up and running. For example, AutoFuzz only works on Windows machines and VUzzer does not have extensive documentation on how to use them. Another reason was the limited time we had to perform fuzzing.

In this thesis we have studied QUIC and even though we did not find new bugs, it still remains important to *go on patrol*, conduct research and test whether applications are secure. Especially, with initiatives like Google's fuzzing program, it does not only help software developers and users but can make you rich as well.

# Bibliography

- [Aar+10] Fides Aarts, Julien Schmaltz, and Frits Vaandrager. “Inference and abstraction of the biometric passport”. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Springer. 2010, pp. 673–686 (cit. on p. 4).
- [Aar+13] Fides Aarts, Joeri de Ruiter, and Erik Poll. “Formal Models of Bank Cards for Free”. In: vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 461–468 (cit. on p. 4).
- [Ang87] Dana Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and computation* 75.2 (1987), pp. 87–106 (cit. on pp. 4, 22, 25).
- [Bel+15] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015 (cit. on p. 19).
- [Bio18a] Phillippe Biondi. *Fuzzing()*. Documentation <https://scapy.readthedocs.io/en/latest/usage.html?highlight=fuzz>. 2018 (cit. on p. 40).
- [Bio18b] Phillippe Biondi. *About Scapy*. Documentation <http://scapy.readthedocs.io/en/latest/introduction.html>, accessed on June 14th, 2018. 2018 (cit. on p. 33).
- [BP12] Mike Belshe and Roberto Peon. *SPDY protocol Draft 3.1*. Protocol specification, <http://www.chromium.org/spdy/spdyprotocol/spdy-protocol-draft3-1>. 2012 (cit. on p. 19).
- [BS12] Jørn Braa and Sundeep Sahay. *Integrated health information architecture: power to the users: design, development and use*. Matrix Publishers, 2012 (cit. on p. 7).
- [Car+15] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. “HTTP over UDP: an Experimental Investigation of QUIC”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 609–614 (cit. on pp. 5, 19).
- [Cho+10] Chia Yuan Cho, Eui Chul Richard Shin, Dawn Song, et al. “Inference and analysis of formal models of botnet command and control protocols”. In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 426–439 (cit. on p. 31).
- [Cho02] Pete Chown. *Advanced encryption standard (AES) ciphersuites for transport layer security (TLS)*. Tech. rep. 2002 (cit. on p. 13).
- [Cle18] Lucas Clemente. *Quic-Go*. Code repository <https://github.com/lucas-clemente/quic-go>. 2018 (cit. on pp. 15, 33, 34).



- [Cyr+16] Britt Cyr, Jeremy Dorfman, Ryan Hamilton, et al. *QUIC Wire Layout Specification*. Tech. rep. Google, 2016 (cit. on pp. 13, 20, 46).
- [Dij79] Edsger Dijkstra. “Structured programming”. In: *Classics in software engineering*. Yourdon Press. 1979, pp. 41–48 (cit. on p. 20).
- [Dwo07] Morris J Dworkin. *Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC*. Tech. rep. 2007 (cit. on p. 34).
- [Est+14] César Estébanez, Yago Saez, Gustavo Recio, and Pedro Isasi. “Performance of the Most Common Non-cryptographic Hashfunctions”. In: *Software Practice & Experiments* 44.6 (June 2014), pp. 681–698 (cit. on p. 15).
- [FB18] Paul Fiterau-Brostean. “Active Model Learning for the Analysis of Network Protocols”. PhD thesis. Radboud University, Nijmegen, 2018 (cit. on pp. 4, 7, 21, 33, 37).
- [FG14] Marc Fischlin and Felix Günther. “Multi-stage key exchange and the case of Google’s QUIC protocol”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1193–1204 (cit. on p. 5).
- [Fow+17] Glenn Fowler, Landon Curt Noll, Kiem-Phong Vo, Donald E. Eastlake 3rd, and Tony Hansen. *The FNV Non-Cryptographic Hash Algorithm*. Internet-Draft draft-eastlake-fnv-14. Work in Progress. Internet Engineering Task Force, Dec. 2017. 119 pp. (cit. on p. 15).
- [GCC] GNU GCC. *15.1 Basic Blocks*. Documentation, <https://gcc.gnu.org/onlinedocs/gccint/Basic-Blocks.html>, Accessed June 13th, 2018. (cit. on p. 50).
- [God+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. “Automated whitebox fuzz testing.” In: *The Network and Distributed System Security Symposium (NDSS)*. Vol. 8. 2008, pp. 151–166 (cit. on pp. 5, 39).
- [God07] Patrice Godefroid. “Random testing for security: blackbox vs. whitebox fuzzing”. In: *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*. ACM. 2007, pp. 1–1 (cit. on p. 5).
- [Gra16] Eberhard Grater. *Why working on Chrome made me develop a tool for reading source code*. Blogpost, <https://medium.com/@egraether/why-working-on-chrome-made-me-develop-a-tool-for-reading-source-code-7111ba21a6f0>. July 2016 (cit. on p. 33).
- [Gri13] Ilya Grigorik. *High Performance Browser Networking: What every web developer should know about networking and web performance*. " O'Reilly Media, Inc.", 2013 (cit. on pp. 12, 19).
- [Ham17] Ryan Hamilton. *Out of order delivery in QUIC*. Forum response, [https://groups.google.com/a/chromium.org/d/msg/proto-quic/A6zwxh9A4so/xiCvL\\_57AwAJ](https://groups.google.com/a/chromium.org/d/msg/proto-quic/A6zwxh9A4so/xiCvL_57AwAJ), Accessed on June 22nd, 2018. Oct. 2017 (cit. on p. 37).
- [Ham18] Ryan Hamilton. *ETA for v42 in Chrome?* Forum response [https://groups.google.com/a/chromium.org/d/msg/proto-quic/uGuzC1pz1EI/pAW\\_zQd\\_AQAJ](https://groups.google.com/a/chromium.org/d/msg/proto-quic/uGuzC1pz1EI/pAW_zQd_AQAJ), Accessed June 14th, 2018. May 2018 (cit. on p. 30).

- [IT] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-00. Work in Progress. Internet Engineering Task Force. 45 pp. (cit. on p. 14).
- [IT18] Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-11. Work in Progress. Internet Engineering Task Force, Apr. 2018. 105 pp. (cit. on p. 13).
- [Iye13] Jana Iyengar. *QUIC - Redefining Internet Transport*. Presentation. 2013 (cit. on pp. 2, 3).
- [Kak+17] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. “Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols”. In: *Proceedings of the 2017 Internet Measurement Conference*. IMC ’17. London, United Kingdom: ACM, 2017, pp. 290–303 (cit. on p. 4).
- [KE10] Dr. Hugo Krawczyk and Pasi Eronen. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. RFC 5869. May 2010 (cit. on p. 13).
- [Ker17] Max Kerkers. “Assessing the security of IEC 60870-5-104 implementations using automata learning”. MA thesis. University of Twente, 2017 (cit. on p. 4).
- [KP87] Phil Karn and Craig Partridge. “Improving round-trip time estimates in reliable transport protocols”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 17. 5. ACM. 1987, pp. 2–7 (cit. on p. 9).
- [Kra05] Hugo Krawczyk. “Perfect forward secrecy”. In: *Encyclopedia of Cryptography and Security*. Springer, 2005, pp. 457–458 (cit. on p. 13).
- [Kre] Anja Krenn. *Learning Regular Sets from Queries and Counterexamples*. TU Graz. Slides used [http://www.ist.tugraz.at/\\_attach/Publish/Akswt2/krenn.pdf](http://www.ist.tugraz.at/_attach/Publish/Akswt2/krenn.pdf) (cit. on pp. 22, 23, 25).
- [Lan+17] Adam Langley, Alistair Riddoch, Alyssa Wilk, et al. “The QUIC transport protocol: Design and Internet-scale deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM. 2017, pp. 183–196 (cit. on pp. 2, 9–11, 14, 37).
- [Lar18] Eggert; Lars. *Implementations*. Wiki on Github <https://github.com/quicwg/base-drafts/wiki/Implementations>, Accessed on June 14th, 2018. June 2018 (cit. on p. 33).
- [LC16] Adam Langley and Wan-Teh Chang. *QUIC Crypto*. Tech. rep. Google, 2016 (cit. on pp. 10–13, 15–17, 49).
- [Lea18] LearnLib. *LearnLib*. Code repository <https://github.com/LearnLib/learnlib>. 2018 (cit. on p. 32).
- [Lyc+15] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. “How secure and quick is QUIC? Provable security and performance analyses”. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 214–231 (cit. on pp. 5, 54).
- [Meg+16] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. “How quick is QUIC?” In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6 (cit. on pp. 5, 11).

- [MS16] Max Moroz and Kostya Serebryany. *Guided in-process fuzzing of Chrome components*. Blogpost, <https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html>, Accessed on June 12th, 2018. Aug. 2016 (cit. on pp. 42, 43).
- [MS18] Matt Morehouse and Kostya Serebryany. *libFuzzer Tutorial*. Read me <https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md>, Accessed on June 14th, 2018. May 2018 (cit. on p. 30).
- [Mue+17] Malte Mues, Falk Howar, Kasper Luckow, Temesghen Kahsai, and Zvonimir Rakamarić. “Releasing the PSYCO: Using Symbolic Search in Interface Generation for Java”. In: *ACM SIGSOFT Software Engineering Notes* 41.6 (2017), pp. 1–5 (cit. on p. 21).
- [Nad+06] Krishna Nadiminti, Marcos Dias De Assunção, and Rajkumar Buyya. “Distributed systems and recent innovations: Challenges and benefits”. In: *InfoNet Magazine* 16.3 (2006), pp. 1–5 (cit. on p. 1).
- [Nie03] Oliver Niese. “An integrated approach to testing complex systems”. PhD thesis. Dortmund University, Dec. 2003 (cit. on p. 29).
- [Pel+99] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis. “Black box checking”. In: *Formal Methods for Protocol Engineering and Distributed Systems*. Springer, 1999, pp. 225–240 (cit. on p. 4).
- [PR15] Roberto Peon and Herve Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. May 2015 (cit. on p. 19).
- [Raf+05] Harald Raffelt, Bernhard Steffen, and Therese Berg. “Learnlib: A library for automata learning and experimentation”. In: *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*. ACM. 2005, pp. 62–71 (cit. on p. 23).
- [RD08] Eric Rescorla and Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008 (cit. on p. 9).
- [Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Internet-Draft draft-ietf-tls-tls13-28. Work in Progress. Internet Engineering Task Force, Mar. 2018. 156 pp. (cit. on p. 12).
- [Rfca] *Transmission Control Protocol*. RFC 793. Sept. 1981 (cit. on p. 8).
- [Rfcb] *User Datagram Protocol*. RFC 768. Aug. 1980 (cit. on p. 8).
- [Riv09] Wind River. *10-Point Checklist for Building Carrier Grade Network Elements*. White paper [https://www.windriver.com/solutions/network-equipment/carrier-grade-checklist\\_0909.pdf](https://www.windriver.com/solutions/network-equipment/carrier-grade-checklist_0909.pdf). Sept. 2009 (cit. on p. 2).
- [Rom18] Eric Roman. *Getting Started with libFuzzer in Chromium*. Blogpost, [https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/getting\\_started.md](https://chromium.googlesource.com/chromium/src/testing/libfuzzer/+/HEAD/getting_started.md), Accessed on June 12th 2018. Apr. 2018 (cit. on pp. 39, 42).
- [Rui15] Joeri de Ruiter. “Lessons learned in the analysis of the EMV and TLS security protocols”. PhD thesis. Radboud University Nijmegen, 2015 (cit. on p. 4).

- [Rui18] Joeri de Ruiter. *StateLearner*. Code repository <https://github.com/jderuiter/statelearner>. 2018 (cit. on p. 32).
- [Rüt+18] Jan Rütth, Ingmar Poesse, Christoph Dietzel, and Oliver Hohlfeld. “A First Look at QUIC in the Wild”. In: *CoRR* abs/1801.05168 (2018). arXiv: 1801.05168 (cit. on p. 9).
- [San18] Tom Sandmann. *Fuzzing strongSwan: a job from start to DNF*. Internship report. 2018 (cit. on p. 42).
- [Sas+13] Len Sassaman, Meredith L Patterson, Sergey Bratus, and Michael E Locasto. “Security applications of formal language theory”. In: *IEEE Systems Journal* 7.3 (2013), pp. 489–500 (cit. on p. 5).
- [SG09] Muzammil Shahbaz and Roland Groz. “Inferring mealy machines”. In: *International Symposium on Formal Methods*. Springer. 2009, pp. 207–222 (cit. on p. 29).
- [SG13] Ronak Sutaria and Raghunath Govindachari. “Making sense of interoperability: Protocols and Standardization initiatives in IOT”. In: *2nd International Workshop on Computing and Networking for Internet of Things*. 2013 (cit. on p. 7).
- [SI15] Ian Swett and Jana Iyengar. “QUIC loss recovery and congestion control”. In: *Internet-Draft, Intended status: Informational, Internet Engineering Task Force* (2015) (cit. on p. 36).
- [Sil13] Alexandra Silva. *Languages and Automata: Lecture notes*. 2013 (cit. on pp. 23, 24).
- [SL89] Deepinder P. Sidhu and T-K Leung. “Formal methods for protocol testing: A detailed study”. In: *IEEE transactions on software engineering* 15.4 (1989), pp. 413–426 (cit. on p. 20).
- [Syn14] Synopsys. *The heartbleed Bug*. <http://heartbleed.com/>, Accessed on June 15th, 2018. Apr. 2014 (cit. on p. 30).
- [Tak+08] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008 (cit. on pp. 38, 39).
- [Tak09] Ari Takanen. “Fuzzing: the Past, the Present and the Future”. In: *Actes du 7ème symposium sur la sécurité des technologies de l’information et des communications (SSTIC)*. 2009, pp. 202–212 (cit. on pp. 30, 38, 40).
- [Tre01] Jan Tretmans. *An overview of OSI conformance testing*. Article, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.1165&rep=rep1&type=pdf>. 2001 (cit. on p. 1).
- [Tre92] G.J. Tretmans. *A Formal Approach to Conformance Testing*. 1992 (cit. on p. 20).
- [UK18] Technology UK. *The TCP/IP Protocol Stack*. Image from <https://www.technologyuk.net/telecommunications/internet/tcp-ip-stack.shtml>. 2018 (cit. on p. 8).
- [Vaa17] Frits Vaandrager. “Model learning”. In: *Communications of the ACM* 60.2 (2017), pp. 86–95 (cit. on pp. 21, 31, 32).
- [Ver16] Patrick Verleg. “Inferring SSH state machines using protocol state fuzzing”. MA thesis. Radboud University, Feb. 2016 (cit. on pp. 33, 37).

- [Vyu17] Dmitry Vyukov. *Fuzzing: The New Unit Testing*. Slidedeck, <https://www.slideshare.net/DmitryVyukov/fuzzing-the-new-unit-testing>. Feb. 2017 (cit. on p. 54).
- [Wan+11] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. “Inferring protocol state machine from network traces: a probabilistic approach”. In: *International Conference on Applied Cryptography and Network Security*. Springer. 2011, pp. 1–18 (cit. on p. 4).
- [Wik] Wikipedia. *Middlebox*. Definition from <https://en.wikipedia.org/wiki/Middlebox> (cit. on p. 9).
- [Wik17] Wikipedia. *Monkey testing*. Definition from [https://en.wikipedia.org/wiki/Monkey\\_testing](https://en.wikipedia.org/wiki/Monkey_testing), Accessed on June 14th, 2018. Dec. 2017 (cit. on p. 38).
- [Zala] Michal Zalewski. *american fuzzy lop (2.52b)*. <http://lcamtuf.coredump.cx/afl/>, Accessed on June 12th 2018 (cit. on pp. 30, 41).
- [Zalb] Michal Zalewski. *American Fuzzy Lop: Readme*. <http://lcamtuf.coredump.cx/afl/README.txt> (cit. on p. 41).
- [Zal14] Michal Zalewski. *Binary fuzzing strategies: what works, what doesn't*. Blogpost <https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html>, Accessed June 14th, 2018. Aug. 2014 (cit. on p. 41).
- [Zal15] Micha Zalewski. *New in AFL: persistent mode*. Blogpost <https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html>, accessed on June 22nd, 2018. June 2015 (cit. on p. 42).
- [Zha+14] Hua Zhang, Zhao Zhang, and Wen Tang. “Improve Peach: Making Network Protocol Fuzz Testing more Precisely”. In: *Applied Mechanics and Materials*. Vol. 551. Trans Tech Publ. 2014, pp. 642–647 (cit. on p. 40).
- [Cen18] Center for Applied Internet Data Analysis. *Analyzing UDP usage in Internet traffic*. <https://www.caida.org/research/traffic-analysis/tcpudpratio/>, Accessed on June 22th, 2018. June 2018 (cit. on p. 8).
- [IET17] IETF HTTP WG. *HTTP/2 Frequently Asked Questions*. <https://http2.github.io/faq/>, Accessed June 21th, 2018. Dec. 2017 (cit. on pp. 2, 19).
- [LLV18] LLVM Project. *libFuzzer a library for coverage-guided fuzz testing*. <https://llvm.org/docs/LibFuzzer.html>, Accessed on June 12th, 2018. Feb. 2018 (cit. on pp. 30, 44).

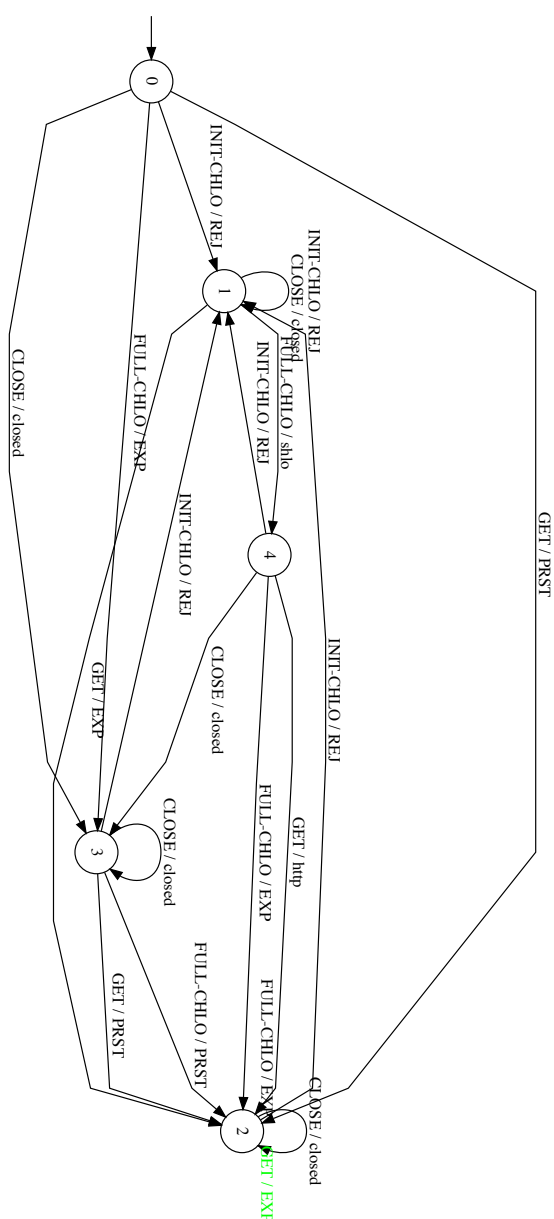
## Appendix A: LibFuzzer code

```

1  std::unique_ptr<net::ProofSource> CreateProofSource(const base::
    FilePath& cert_path, const base::FilePath& key_path) {
2      std::unique_ptr<net::ProofSourceChromium> proof_source(
        new net::ProofSourceChromium());
3      return std::move(proof_source);
4  }
5
6  extern "C" int LLVMFuzzerTestOneInput(const uint8_t* data, size_t
    size) {
7      // Instantiate the QuicSimpleServer
8      base::AtExitManager exit_manager;
9      base::MessageLoopForIO message_loop;
10
11     net::QuicConfig config;
12     net::QuicHttpResponseCache response_cache;
13     response_cache.InitializeFromDirectory("/Users/
        abduallahrasool/Documents/chromium/quic-data/www.
        example.org");
14
15     QuicSimpleServer simple_server(
16         CreateProofSource(base::FilePath("net/tools/quic/
            certs/out/leaf_cert.pem"), base::FilePath("net
            /tools/quic/certs/out/leaf_cert.pem")),
17         config,
18         net::QuicCryptoServerConfig::ConfigOptions(),
19         net::AllSupportedVersions(),
20         &response_cache
21     );
22
23     //Call our custom server function that receives the data
        as parameter instead of from a socket
24     simple_server.ParseMessageForFuzzing((char*) data,
        static_cast<int>(size));
25
26     return 0;
27 }

```

## Appendix B: Learned model without 0-RTT



# Appendix C: Learned model with 0-RTT

