DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC

Gaganjeet Singh Reen
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
reen.gagan@gmail.com

Christian Rossow
CISPA Helmholtz Center for Information Security
Saarbrücken, Saarland, Germany
rossow@cispa.saarland

ABSTRACT

QUIC is an emerging transport protocol that has the potential to replace TCP in the near future. As such, QUIC will become an important target for Deep Packet Inspection (DPI). Reliable DPI is essential, e.g., for corporate environments, to monitor traffic entering and leaving their networks. However, *elusion* strategies threaten the validity of DPI systems, as they allow attackers to carefully design traffic to fool and thus evade on-path DPI systems. While such elusion strategies for TCP are well documented, it is unclear if attackers will be able to elude QUIC-based DPI systems. In this paper, we systematically explore elusion methodologies for QUIC. To this end, we present DPIFuzz: a differential fuzzing framework which can automatically detect strategies to elude stateful DPI systems for QUIC. We use DPIFuzz to generate and mutate QUIC streams in order to compare (and find differences in) the server-side interpretations of five popular open-source QUIC implementations. We show that DPIFuzz successfully reveals DPI elusion strategies, such as using packets with duplicate packet numbers or exploiting the diverging handling of overlapping stream offsets by QUIC implementations. DPIFuzz additionally finds four security-critical vulnerabilities in these QUIC implementations.

CCS CONCEPTS

• **Security and Privacy → Intrusion detection systems; Network security; Software and application security**.

KEYWORDS

DPI Elusion, QUIC, Protocol Fuzzing, Differential Fuzzing

ACM Reference Format:
Gaganjeet Singh Reen and Christian Rossow. 2020. DPIFuzz: A Differential Fuzzing Framework to Detect DPI Elusion Strategies for QUIC. In *Annual Computer Security Applications Conference (ACSAC 2020), December 7–11, 2020, Austin, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3427228.3427662

ACSAC 2020, December 7–11, 2020, Austin, USA

ACM ISBN 978-1-4503-8858-0/20/12…$15.00
https://doi.org/10.1145/3427228.3427662

1 INTRODUCTION

Organisations across the globe inspect the encrypted traffic at the periphery of their network. Deep Packet Inspection (DPI) techniques are at the core of such traffic analyses. DPI allows to reassemble and inspect application-layer communication content, and consequently, detect various security-critical incidents such as malware [13], data leakage [38], phishing attacks [37], or remote exploits [36]. With the rising popularity of protocols that are encrypted by default, DPI systems are typically used in combination with man-in-the-middle proxies [27, 39] that assist in intercepting encrypted channels (such as TLS).

However, prior research has demonstrated *elusion* attacks against DPI systems that fool their TCP and/or HTTP [17, 23, 33, 41, 42] inspections. The core reason for such evasion are differences in how the DPI system and the actual data recipient have implemented a protocol (such as TCP). Any slight difference may lead to the fact that the actual recipient of the data and the DPI system reassembled different payload data, although operating on the same sequence of raw packets. There are plenty of reasons why such differences may occur: (i) Protocol specifications (deliberately or not) leave some details out, either not to blow up the standard, to allow vendors to handle certain situation as they see fit, or simply as these details were forgotten. (ii) In order to reassemble the application-layer content, the DPI systems needs to model the state machine of stateful transport protocols (such as TCP). Having said this, DPI systems may choose to simplify this state machine to minimize the overhead [41] and thus, to foster scalability. (iii) Similarly, DPI systems might perform a lower number of checks or less sophisticated checks to validate packets as compared to an endpoint in order to reduce the computational load.

Fortunately, there are just a few relevant protocols that have to be studied to assess and mitigate these evasion attacks. As of now, the vast majority of inspected Internet traffic is HTTPS/TLS-based [22, 25, 27, 39]. As such, it is not surprising that most works in the context of evasion attacks so far have focused on the specifics of TCP [17, 23, 33, 41, 42]. Yet, with the IETF working to standardise the QUIC protocol, coupled with the performance and the security benefits QUIC offers compared to TCP, it is only a matter of time before it is adopted widely as the go-to transport protocol. Consequently, intercepting and analysing QUIC packets using DPI systems coupled with inline proxies will be imperative to organisations to maintain the security of their networks. Unfortunately, the known findings of TCP do not directly translate to QUIC, as the two protocols have vastly different specifications. QUIC provides security features, like authentication and encryption, that are typically handled by a higher layer protocol (like TLS), from the transport layer itself. QUIC also uses a large variety of *packets* and

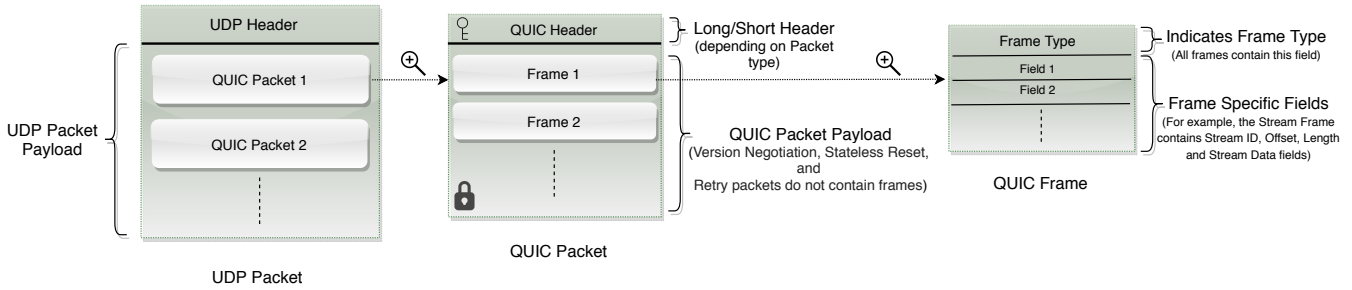

Figure 1: QUIC Packet Structure Overview: QUIC Packets have confidentiality and integrity protection by default. Even parts of QUIC headers are protected (using keys separate from the payload protection keys) [11].

frames , as well as multiple *streams* within a connection, for data exchange between end points as compared to TCP which simply uses packets. Additionally, the wire image of QUIC is integrity protected and reveals much lesser information to network monitors as compared to the wire image of TCP which makes inspecting the traffic more complex. In this paper, we address the problem of DPI elusion attacks for QUIC-based communication. Instead of manually searching for QUIC protocol details that can potentially be used for DPI elusion, we set out for a more methodological (and, hopefully, more complete) way to discover such protocol details. We therefore develop a differential fuzzing framework (DPIFuzz) which can automatically uncover potential differences between implementations which could be used for eluding a DPI system. We design a modular and stateful fuzzer that can generate and mutate sequences of QUIC packets. The fuzzer also tracks the responses of the implementations to the generated sequences. We use the fuzzer to test five popular open-souce QUIC implementations and consequently perform a differential analysis of the behaviour of these implementations to the fuzzed sequences of packets. In order to access the application level data reassembled by the implementations, we create echo servers using these implementations. We also use *Status Codes* to track the status/state of implementations after processing a given sequence of packets. Using DPIFuzz, we uncover two distinct strategies which highlight ambiguities between QUIC implementations. Additionally, we also uncover four security-critical vulnerabilities in the implementations and demonstrate how these could facilitate DPI elusion.

Our findings are of direct importance for DPI users. Eliminating any ambiguities or vulnerabilities that we identified is immensely helpful to organisations that want to reliably monitor QUIC traffic on their network perimeter. At the very least, being aware of such shortcomings for QUIC will help DPI users identify the potential flaws of their monitoring systems in an automated way.

To summarize our contributions:

- We develop DPIFuzz, a structure-aware and modular fuzzing framework which allows (i) automated testing of QUIC implementations by generating and mutating communication streams and (ii) a differential analysis of the behaviour of the implementations to these communication streams.
- We apply DPIFuzz to five popular open-source QUIC implementations. As a result, we are (to the best of our knowledge) the first to report on potential DPI elusion methods for QUIC.
- We additionally uncover and report on four security critical vulnerabilities in the QUIC implementations, demonstrating that DPIFuzz's application is not just restricted to the setting of finding evasion attacks.

2 BACKGROUND

2.1 QUIC Protocol

QUIC [12] is an encrypted-by-default Internet transport protocol which was originally proposed by Google. QUIC is conceptually similar to a combination of TCP, TLS, and HTTP/2 implemented on UDP and was developed with the intended goal of eventually replacing TCP on the web. Since QUIC runs on top of UDP, it can be distributed as a userspace library that can be easily upgraded. This is in stark contrast to other transport protocols like TCP/UDP which are implemented in operating system kernels and middlebox firmware because of which making significant changes to them is next to impossible [7].

The QUIC IETF working group has been working since late 2016 to standardise the protocol. For the purpose of this paper, we refer solely to draft 27 of the IETF specification [8] of the QUIC protocol. The versions of the implementations considered in the paper are also based on draft 27 of IETF QUIC.

QUIC defines several types of *packets* and *frames*. Figure 1 provides an overview about the structure of QUIC packets. QUIC packets are carried in UDP datagrams. Multiple QUIC packets can be coalesced into one UDP datagram. QUIC packets broadly contain a header (long header or short header) and a payload. The payload of QUIC packets is expressed as a sequence of frames. Draft 27 of the IETF QUIC Protocol defines 20 different types of frames [9] like the Stream Frame, which is used to carry the actual application level payload over a *stream* or the Connection Close Frame, which is used to indicate to an end point that a connection is being closed. Detailed information about the different types of frames and packets can be found in the IETF draft for QUIC [9].

A QUIC connection is a stateful interaction between a client and server. Each QUIC connection starts with a handshake phase during which client and server establish a shared secret using the cryptographic handshake protocol (QUIC-TLS [11]) and decide on an application protocol to use to facilitate the exchange of data. The successful completion of the handshake confirms that both endpoints are willing to communicate and establishes the important

parameters for the connection.
The exchange of information primarily takes place by means of *streams*. In a stream, the protected packets contain stream frames which are responsible for carrying the data as payload. Multiple streams can be used to send data in a connection. Streams in an QUIC connection can be bidirectional or uni-directional, depending on the value of the second least significant bit of the stream ID. QUIC also uses flow control for each stream individually as well as for the connection as a whole. Several transport parameters are defined in the QUIC specification which allow flow control between the client and the server.

2.2 Types of Fuzzing

Fuzzing is the process of providing randomised inputs to programs and observing their behaviour. It has gained immense popularity in the software testing and the security industry owing to the fact that it can detect bugs and security vulnerabilities in an automated way. Several tools [19, 20, 26, 29, 34] already exist to perform fuzzing on a variety of programs and systems and they have been extremely successful in discovering vulnerabilities [1, 5, 10, 29].

Fuzzing strategies can be broadly categorised into [2]:

- Dumb vs. Smart Fuzzing: Dumb fuzzers are unaware about the format of the input that the target expects while smart fuzzers are input format aware.
- Black-box vs. Coverage-guided fuzzing: Black-box fuzzers do not test which branches of the target were covered by fuzzing whereas coverage-guided fuzzers try to maximise the code coverage of the target.
- Generation- vs. Mutation-based fuzzing: Generation-based fuzzers create the input for a target from scratch for every execution of the fuzzer. In contrast, mutation-based fuzzers alter existing data that is input to the fuzzer to modify and thereby create new inputs.

Differential Fuzzing is a testing technique where the same fuzzed input is provided to different yet similar implementations that should behave identical given the same input. Differential fuzzing discovers potential implementation differences by comparing the behaviors and/or responses of the systems under test.

3 GOAL AND SCOPE

3.1 Goal

Our goal is to automatically detect strategies to elude a stateful DPI system. We aim to find sequences of QUIC packets/frames that contain payload that is denylisted (i.e., blocked using specific payload keywords) by the DPI system, which however remains unnoticed by the DPI system. To this end, we aim to reveal sequences of QUIC packets which are reassembled by the server and the DPI system—both of which use different QUIC implementations to reassemble the payload differently. Such sequences reveal a potential point of ambiguity in the two QUIC implementations that can be exploited in elusion attacks. Once we discover such sequences, we analyse them to find the underlying reason for the disparate handling of packets. Finally, we want to generalize the identified differences to demonstrate that attackers can leverage them to elude a DPI system with any denylisted payload.

3.2 Threat Model

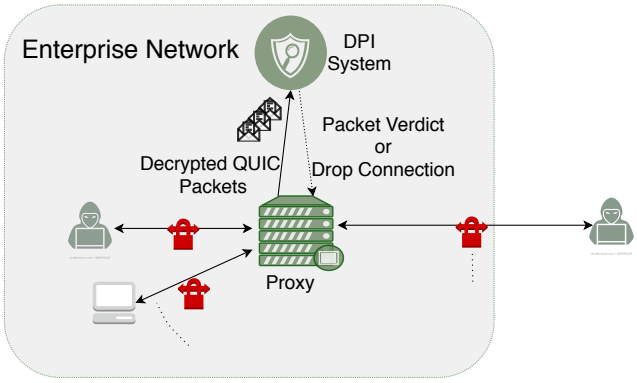

Figure 2: DPIFuzz Threat Model: An in-line proxy intercepts QUIC communication and forwards decrypted QUIC sequences to a separate DPI system for further inspection.

The threat model that we assume for the scope of this paper is depicted in Figure 2. We envision a QUIC-aware monitoring system with two components that tackle the disjoint tasks of (i) decrypting the TLS-encrypted QUIC communication, and (ii) inspecting the decrypted content. This setup is in line with industry-grade DPI systems that intercept TLS-encrypted communication [22, 27, 39], and also corresponds to the recommended setup [21] of monitoring TLS-encrypted communication with open-source DPI systems like Snort/Suricata. In detail, these two components compromise of:

- An inline proxy which establishes a QUIC connection with both the client and the server and forwards the traffic between the two as well as to the DPI system for analysis.
- A stateful, reassembly-based DPI system which reconstructs the streams and analyses the packets being sent for denylisted content. If the organisation prefers intrusion *prevention* over pure detection, the DPI system can optionally send a verdict to the proxy for each packet it receives, or alternatively, send the proxy an asynchronous signal to drop a connection after detecting denylisted content.

Having the proxy and DPI system as separate components provides (i) fault tolerance as it ensures that a fault in the analysis system does not affect the proxy and vice versa, (ii) flexibility to use the system for intrusion prevention or detection depending on the use case and (iii) modularity which allows the DPI systems to be changed or upgraded while using the same proxy and vice versa.

The proxy itself does not attempt to reassemble or analyse the application-level data. However, it will need to keep track of the control data necessary to ensure smooth communication between the client and the server. The proxy might need to keep track of certain frames in the QUIC Payload like the New_Connection_Id frame which is used by endpoints when they want to change the connection ID associated with a connection. It would need to parse the QUIC payload for this; however, in the event that the parsing fails, it will follow a soft fail strategy and forward the packets to the DPI system and the endpoint as expected.

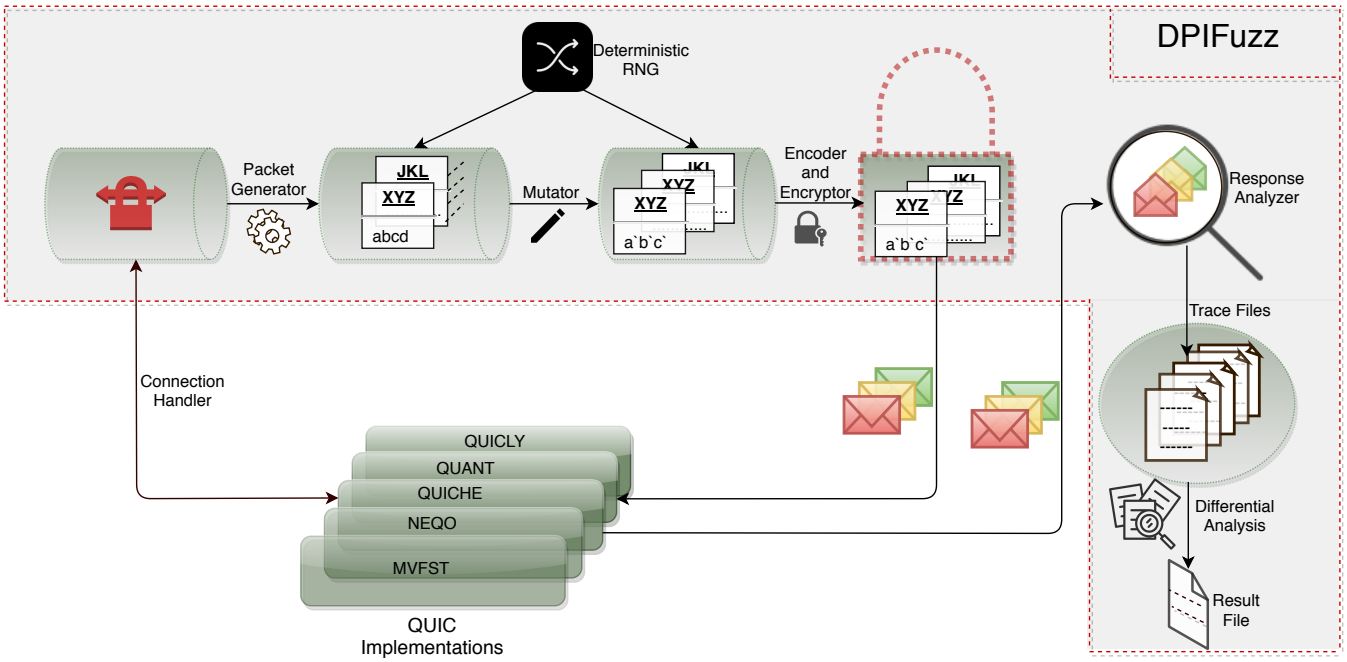

Figure 3: DPIFuzz overview. DPIFuzz is a differential fuzzing framework that executes multiple instances of the fuzzer against different QUIC implementations, and then performs a differential analysis of the resulting trace files to find inconsistencies.

3.3 Elusion Strategies

If we can cause the DPI system to reassemble different payload than the destination server for the same sequence of packets, we can possibly elude the DPI system. We broadly group such elusion strategies into the following three categories:

Insertion Packet/Frame: Some packets/frames might be accepted by the DPI system and rejected by the server (because the edge cases could be handled differently or the checks at the DPI might not be as sophisticated as those in the server implementations). These packets/frames are known as insertion packets/frames [33, 41]. The consequence of an insertion packet/frame could be:

- The insertion packet/frame results in extra application layer payload being registered at the DPI system.
- The insertion packet/frame causes an error/crash on the DPI system, which results in the buffers being flushed, but not on the destination server. As a result, if a blocked keyword is split into parts such that some parts are sent before the insertion packet/frame and some parts are sent after, the server would be able to reassemble the data correctly while the DPI system would never reassemble the entire blocked keyword.
- The insertion packet/frame could affect the state of the DPI system such that while the packet/frame itself does not lead to an observable difference, it causes the DPI system to consequently reassemble different data or run into an error or a crash while the destination server does not.

Evasion Packet/Frame: Some packets/frames might be accepted by the server and rejected by the DPI system (This can happen in cases where the check performed by the DPI are different from those performed by a server). These packets/frames are known as evasion packets/frames [33, 41], and potentially result in the following consequences:

- The evasion packet/frame allows sending data to the server without it being registered at the DPI system.
- The evasion packet/frame could lead to a difference in the state of the DPI and the server implementation such that it eventually manifests as an observable difference between the behaviour of the DPI system and the server implementation.

Ambiguity: A DPI system implementation might also have different rules defined compared to a destination server implementation in order to deal with ambiguous aspects of the protocol specification. As stated by Wang et al. [41], most network protocol specifications are inherently ambiguous because they are written in a natural language like English. Often, some parts of the specifications are deliberately left unspecified, which in turn leads to vendor-specific implementations. This results in accepted packets/frames being reassembled differently or affecting the state of the protocol implementations in different ways.

4 DPIFUZZ ARCHITECTURE

4.1 Challenges Faced

Designing efficient and effective fuzzers for secure and encrypted protocols is challenging. Using a “naive” fuzzer does not lead to valuable results because protocol implementations expect structured packets with specific values for fields as input, in the absence of which the input is simply discarded. Valid inputs are extremely important to guide an implementation into states that are deep in the state space of a protocol [32]. A fuzzer will only be able to

detect meaningful vulnerabilities if it considers the state space of a protocol implementation and then injects unexpected inputs to test how such cases are handled. This is extremely important as the obvious errors/unexpected inputs are usually already handled by the developers using a variety of testing frameworks.

Thus, for protocol level fuzzing, a structure-aware fuzzer is essential to obtain meaningful results. Then again, simply using a structured fuzzer might not be enough for stateful protocols like QUIC where the exchange of data only starts after a successful handshake process.

Taking the above mentioned points into account, we design a modular, stateful, structure-aware, generation+mutation based fuzzer that can actively interact with the server-side implementation under test (IUT).

4.2 Fuzzer Design Overview

Figure 3 shows the general structure of our fuzzing framework DPIFuzz. We design a fuzzer that is split into the following modules:

- Connection Handler: This module establishes a connection with the IUT by completing the TLS handshake. It uses the high level QUIC API provided by QUIC-Tracker [28].
- Packet Generator: This module generates a sequence of QUIC packets without encoding or encrypting them. It supports generating all types of QUIC frames and packets listed in the specification. The design of the Packet Generator module is explored in detail in Section 4.4.
- Mutator: This module mutates the QUIC packets generated by the packet generator. The mutations are defined in Section 4.3. We do not mutate the QUIC packet header.
- Encoder and Encryptor: This module performs the necessary encoding and encrytion of the QUIC packets. It then encapsulates the QUIC packet in a UDP datagram and sends the UDP datagram to the server.
- Response Analyzer: This module analyses the responses that the IUT sends on processing the fuzzed sequence of packets. It creates a trace file that contains:
  - The application level data sent by the IUT as a response to the packets it receives. This response will later be used to gauge if two QUIC implementations differ in their payload reassembly.
  - Status Codes: These indicate the state of an IUT after processing a sequence of packets. Our fuzzer infers the state of the IUT either from the responses that the IUT sends or from the absence of a response. Status codes are (i) *ServerTimeOut*, which indicates that the IUT was in an unresponsive state after processing a sequence of packets[1], (ii) *TLSHandshakeFail*, which indicates that the handshake could not be completed successfully with the IUT, (iii) *ServerDidNotRespond*, which indicates that the IUT was unresponsive right from the initial packet and (iv) *ServerIsAlive*, which indicates that the server was responsive after processing the entire sequence of packets.

[1]We test the responsiveness of the IUT , after processing the fuzzed sequence of packets, by sending a response eliciting packer/frame like a Stream Frame to it and also attempting to establish a new connection with the IUT by sending an initial packet.

Packet flow through the fuzzer: The fuzzing process starts with the Connection Handler module establishing a connection with the IUT. To ensure that the handshake between the client and server completes successfully, we do not fuzz the initial and the handshake packets. Following a successful connection, the Packet Generator creates a sequence of QUIC packets that we wish to send to the server. Since the Packet Generator creates QUIC packets which are not yet encoded or encrypted, we can perform mutations before the packet contents are encoded or encrypted.

Once we have a sequence of packets generated from the Packet Generator, the sequence is passed through the mutator module. We ensure that we do not fuzz the packet header. This fact coupled with fuzzing the packet payload before encoding or encryption ensures that the packets and most of the frames (some of the frames might undergo structure altering mutations) remain structurally valid and allows us to observe the effects that fuzzing different frame field values can have on server implementations. Following this, each packet will either just be encoded (like the initial client packet) or both encoded and encrypted (like the handshake and the data packets). Finally, the fuzzer encapsulates the QUIC packet in a UDP datagram and sends the datagram to the IUT. The IUT response is received at the Response Analyzer based on which a trace file is created.

4.3 Mutations

We broadly define two types of mutations for our fuzzer:

Sequence-level mutations affect the sequence of packets, generated by the Packet Generator (see next subsection), as a whole. We envision three such mutations:

- Shuffle: The order of packets in a sequence is randomly shuffled.
- Duplicate: Packets are randomly selected and then duplicated with varying degrees of duplication
- Drop: Randomly selected packets are dropped from the sequence of packets.

A sequence can undergo each of these three sub-mutations with a probability α_1, α_2 and α_3, respectively. The values for these probabilities are defined in Table 7 in the appendix.

Packet-level mutations affect an individual packet payload instead of an entire sequence. Every packet in a sequence of packets can undergo packet-level mutations with a probability γ. We distinguish between payload mutations that are defined considering QUIC Packet payload simply as a collection of bytes, and frame mutations that are defined for the individual frames contained in the QUIC packet payload, as outlined below. Once a packet has been selected to undergo packet-level mutations, it could undergo either a payload-level mutation or a frame-level mutation with a probability ω and $1 - \omega$, respectively. It is important to note that after performing the mutations on the packet payload, we update the payload length field in the long headers of packets (if present) to reflect the length of the mutated payload. This ensures that a simple check of comparing the payload length field with actual payload length does not lead to the packet being discarded.

- Payload mutations: These mutations do not take into consideration the structure of the payload or the frames that

make up the payload. They could lead to alterations in the structures of the frames within the packets. The four types of payload mutations are:
  - Repeat payload: A random substring of the entire packet payload is selected and injected at a random position in the packet payload. The existing payload content is not overwritten.
  - Alter payload: In this, we iterate over the payload at a byte level. For each byte, a random decision is made about whether to fuzz the byte or not.
  - Add random payload: A payload with randomised content and a random length lesser than the actual payload length is selected and inserted at a random position in the original payload without overwriting the existing payload content.
  - Drop random payload: A random offset is selected and then a random number of bytes, of length $\leq$ *payload length* $-$ *offset*, starting at that offset is dropped.

  A selected packet will undergo one of the four mutations at random.
- Frame mutations: The individual frames that a packet payload contains are first extracted and then each frame is fuzzed with a probability β.
  As an example, consider the STREAM DATA FRAME which carries the stream-level payload in QUIC. The different fields in this frame are OFF bit, LEN bit, FIN bit, Stream ID, Offset, Length and Stream Data. First, the maximum number of fields that should be fuzzed in the frame is randomly determined. Then, we randomly pick which fields to fuzz. All the other types of frames defined in the QUIC specification are fuzzed based on the same principle.

4.4 Packet Generators

The previous subsection discussed how we can mutate existing sequences of QUIC packets. We will now discuss how to create such streams and packets in the first place. Packet generators can be of two types, which are different in whether or not the testers have control over the sequence of packets being generated:

Randomised: These generators randomly decide which types of frames/packets to create and randomly group frames into packet payload. They fill the frame and packet fields with random but type-accurate values. They attach frames as packet payload without considering whether the specification allows a particular packet to have specific frame types or if a client is even allowed to send frames of a specific type.

Controlled: These generators create specific types of packet sequences that we want to test the IUT with. This allows us to focus more on specific aspects of an implementation that we want to test. The total number of packets and frames, their field values and their ordering can all still be randomised, but we can use the generator to control the type of packets and frames being created as well as the grouping of frames into packet payload. The packets still undergo mutations after this.

In particular, we use three types of controlled generators. These generators create sequences which allow us to specifically target the stream reassembly mechanism of the IUTs.

- Basic Stream Reassembly Generator: This generator creates a sequence that contains a random number of packets with stream frames. Each stream established with the server contains a random number of random-length and random-content QUIC packets that will be sent to the server, and finally, is gracefully terminated. QUIC stream frames correctly encode the stream offets as if data was sent consecutively.
- Flow-Control-Aware Stream Reassembly Generator: In contrast to the previous generator, the sequence generated by this generator also contains packets and frames which affect the flow control parameters established for the streams as well as the connection.
- Overlapping Offset Generator: This generator creates a sequence of packets and frames to specifically test how an implementation deals with overlapping offsets in a stream. The sequence that this generator creates is simply shuffled but not passed through the mutators. This ensures that the values of other fields in a frame are not fuzzed and offset field is the only field affecting the results. Frames which contain overlapping offsets as well as different data for the same offsets are created by randomly deciding on the total length for a stream and then fragmenting it in multiple different ways using randomised data and offsets.

4.5 Differential Analysis

We now use the described fuzzing methodology to automatically search for differences in protocol implementations. To this end, we leverage differential fuzzing, which feeds the same input to similar yet different programs and compares their behavior. In our context, these "similar" programs are semantically equivalent QUIC servers based on different QUIC implementations. Technically, we thus implement simple "echo" servers for the QUIC libraries under test (see Section 5.1 for more details), which reply with the application-level payload they received from the client (i.e., from our fuzzer).
To compare the program behaviors, for every execution (which is bootstrapped with a particular generator and seed, more later) against an IUT, the fuzzer records a trace file containing (i) the application-level data returned by the IUT and (ii) the corresponding status code. The trace files do not necessarily distinguish between all possible states of IUTs; however, they contain enough information to highlight the relevant differences from the perspective of DPI elusion. More details can be readily included in the trace files for a more detailed comparison of IUT states depending on the use case. To speed up comparisons, DPIFuzz computes the hash for all generated trace files (i.e., the reassembled stream payloads and status codes, but not headers). DPIFuzz then compares the hashes of the trace files for all the executions of the fuzzer, with the same seed and generator, for all the listed IUTs. If not all hashes of a given seed are equal, DPIFuzz adds the seed value for the execution, the name of the generator used and a hash map with the names of all the listed IUTs as keys and names of the IUTs their hash differs from, as values to a result file. The differing hashes indicate that the servers under test do not all respond in the same way to the same sequence of packets that were provided as input to them. That is, either the data reassembled by the servers was different and/or

the values assigned by the fuzzer for status codes differed. Thus, the sequence of packets are of interest to us as they can be used to detect elusion strategies.

A slight complication arises from the fact that we cannot simply replay one actual packet capture to all IUTs, as the underlying cryptographic material differs. On the one hand, we want that the sequences sent to all IUTs essentially carry the same payload and follow the same order of packets and frames. On the other hand, packets cannot be fully identical, as QUIC streams are end-to-end encrypted with diverging (and potentially ephemeral) key material. To tackle this, the fuzzer first completes the handshake individually with all the IUTs. All subsequent packets carry the appropriate connection IDs and are correctly encrypted so that all the servers see the same payload after decryption and decoding. This ensures that we can replay "the same" sequence to multiple QUIC instances, i.e., we can now recreate a sequence of randomly generated packets in order to send identical sequencies to multiple implementations. Technically, we use a deterministic random number generator to randomize packet generations and to select mutations. This allows us to easily regenerate the same sequence of packets as long as we know the seed value used to initialise the random number generator. DPIFuzz thus selects and records a new cryptographically secure seed value after each fuzzer execution. This seed initializes the deterministic random number generator used in the fuzzer.

Finally, we outline the inputs that DPIFuzz expects. DPIFuzz requires an ***IUTList***, which specifies the IUTs whose behaviour will be compared, a ***GeneratorList***, which specifies the packet generators that will be used in the fuzzer, the value $\mathbf{N_s}$, which determines the number of times we execute the fuzzer with the each specified generator against each IUT with different seed values, and the value **ParallelExecutions**, which determines the number of fuzzer instances that can be executed in parallel.

5 RESULTS

5.1 Experiment Methodology

We implement our entire framework using Golang. For our experiments, we consider five actively developed, open source implementations[2] of QUIC; namely, QUICHE [15] by Cloudflare, MVFST [16] by Facebook, QUANT [31] by NetApp, NEQO [30] by Mozilla and QUICLY [18] by Fastly.

In order to access the data reassembled by the DPI system and the server, we create echo servers using the QUIC implementations[3]. The echo servers allow us to easily validate whether or not there are ambiguities in how QUIC libraries handle certain packet streams. Having said this, they do not necessarily capture the full logic of more complex QUIC applications—hence our findings represent a lower bound of all potential QUIC implementation ambiguities. From these five QUIC libraries, we create all potential pairs, i.e., exhaust all possible combinations of systems being used for DPI inspection and data reception, respectively. We run DPIFuzz simultaneously against servers of all five implementations, i.e., we compare the responses of all the servers against each other. We then analyse the results with the assumption that a DPI system is based on one of the implementations under consideration in the result.

To model an inline proxy that forwards the packets to a DPI system for further inspection, our differential fuzzing module makes sure that the same sequence of packets are sent to all the IUTs. To ensure that in the event different data is returned by the two echo servers, the difference is actually caused by an implementation level difference and not a design difference in the echo servers, we design and configure the echo servers in the same way. We use identical values for the QUIC transport parameters in the servers as these parameters can affect stream data reassembly. The echo servers send the data received on a stream back only if they are able to reassemble the stream data completely, i.e., the reassembled data is contiguous and the stream has been closed. The servers attempt to respond on the same streams for which they reassemble the data. If a stream is unidirectional in nature, the write operation for that stream will simply fail and the data is discarded (this does not affect our results because we ensure that our generators create bi-directional communication streams).

5.2 Experiment Setup

To model a proper client server architecture while removing the effects of network latency and unintentional packet reordering, we run our echo servers on a locally hosted virtual machine using VirtualBox and run the client on our actual local machine. This ensures that (i) the implementations being compared are fed the exact same logical sequence of packets and (ii) the reassembly differences that DPIFuzz uncovers are a consequence of design differences in the implementations and not of any other reason. The experiments were run on a machine with a Quad-Core Intel i5 processor with a 16GB RAM. The virtual machine used was allocated 4 cores with 8GB of RAM. The operating system used for both the virtual machine and the local system is Ubuntu 18.04. Running the tests locally also ensures that no servers in production are harmed. Table 6 in the appendix contains the values used for the transport parameters in the servers. Values of input parameters for the Differential Fuzzing Module are:

- ***IUTList*** is initialised with the IP address as well as the port number for the 5 IUTs.
- ***GeneratorList*** is defined as "Basic Stream Reassembly, Flow-Control-Aware Stream Reassembly, Overlapping Offset"
- $\mathbf{N_s}$ is set to 200
- ***ParallelExecutions*** is set to 5.
- The probability values used for various mutations are listed in Table 7 in the appendix.

5.3 DPI Elusion Results

In total, we use our framework to create 600 ($|\mathbf{GeneratorList}|^*\mathbf{N_s}$) unique sequences, i.e., we run our fuzzer against each IUT, with each of the 3 ($|\mathbf{GeneratorList}|$) specified generators, with 200($\mathbf{N_s}$) different seed values. It takes approximately 2.5 hours for all the sequences to finish executing and for the comparison results to be generated. If run without parallelization, the same process would take around 12.5 hours (considering the parallelization factor of 5). We summarise the differences uncovered by these sequences in

[2]Versions of implementations as available on 29/05/2020

[3]We do not create an echo server for the QUANT and the NEQO implementation due to the absence of a well defined API and documentation. Although we perform testing on their fully fledged servers supporting QUIC protocol, we do not consider these servers directly for our reassembly based results.

Seed Value	Generator	QUICHE Reassembled Data	MVFST Reassembled Data	QUICLY Reassembled Data
4373445819122772715	Basic Stream Reassembly	2Z?b"?@N5?#48SZyBp????;yGJ-+$0P7cdWYaBMetcem=+@Wf-Sja1xZwhYKFFA26AN&YI_	yGJ+$0P7cdWYaBMetcem=+@-WfSja1xZwhYKFFA26AN&YI_	yGJ+$0P7cdWYaBMetcem=+@-WfSja1xZwhYKFFA26AN&YI_
7253654666463259418	Overlapping Offset	(;k]Nx[CV@g@mc'jZP	(;k]Nx[CV@g@mc1jZP	(;k]>.gXhn%@mc1jZP

Table 1: Reassembly Differences

Packet No.	Stream Frame Payload	Stream Offset	Payload Length	No. of overlapping offsets	Stream Finbit	QUICHE Reassembled Data	QUICLY Reassembled Data	MVFST Reassembled Data
1	'jZP	14	4	0	True	_____'jZP	_____'jZP	_____'jZP
2	x[	5	2	0	False	_____x[_____'jZP	_____x[_____'jZP	_____x[_____'jZP
3	@mc1	11	3	1	False	_____x[____@mc'jZP	_____x[____@mc1jZP	_____x[____@mc1jZP
4	(	0	1	0	False	(____x[____@mc'jZP	(____x[____@mc1jZP	(____x[____@mc1jZP
5	CV@g	7	4	0	False	(____x[CV@g@mc'jZP	(____x[CV@g@mc1jZP	(____x[CV@g@mc1jZP
6	k]N	2	3	0	False	(_k]Nx[CV@g@mc'jZP	(_k]Nx[CV@g@mc1jZP	(_k]Nx[CV@g@mc1jZP
7	>.g	4	3	3	False	(_k]Nx[CV@g@mc'jZP	(_k]>.gCV@g@mc1jZP	(_k]Nx[CV@g@mc1jZP
8	Xhn%	7	4	4	False	(_k]Nx[CV@g@mc'jZP	(_k]>.gXhn%@mc1jZP	(_k]Nx[CV@g@mc1jZP
9	;	1	1	0	False	(;k]Nx[CV@g@mc'jZP	(;k]>.gXhn%@mc1jZP	(;k]Nx[CV@g@mc1jZP

Table 2: Overlapping Offset Data Reassembly. Packets 3, 7 and 8 contain overlapping offsets and highlight the diverging behaviour of the implementations.

Table 8 in the appendix. We use the seed values in the result file generated by DPIFuzz to regenerate sequences of packets for which the behaviour of the implementations differs. We then manually inspect them to find the underlying reasons for the differences in implementation behaviour. Analysing these sequences, we uncover 6 logically different DPI elusion results which we summarise next.

5.3.1 *Reassembly Differences:*

In this category, we discuss the results which lead to different data being reassembled (for the same sequence of input packets) by the IUTs. Using the two exemplary results in Table 1, we highlight how attackers can exploit differences in reassembly strategies to elude a DPI system:

- **Exploiting Packets with Duplicate Packet Numbers:** The first way to elude DPI systems uncovered by our fuzzer is based on the idea of inserting packets with duplicate packet numbers that are ignored by some IUTs, in accordance with the IETF specification of QUIC, but not by others.
  Consider the first entry in Table 1. The MVFST server or the QUICLY server acts as the DPI system and the QUICHE server acts as the destination server. While the destination server reassembles data from two streams (stream ID 12 and 16 in our concrete sequence), the DPI server only reassembles the data from one stream (stream ID 16). After receiving packets containing stream frames of one stream (ID 16), the servers receive a packet with a duplicate packet number. This packet has undergone the "Repeat Payload" mutation at the fuzzer side because of which the frames in the payload become structurally invalid. While the destination server is configured to drop packets with a duplicate packet number, the DPI server attempts to process the packet. This results in a "Frame Format Error" on the DPI server and the connection is closed. Any data sent after this(stream 12 data) is not reassembled by the DPI server. When used with an inline proxy, it will still be able to inspect the data in the individual packets and maybe reassemble stream 12 data independently, but will not be able to combine it with contents reassembled from stream 16.
  This concrete evasion instance can be trivially generalized to fool the DPI in missing any blocked keyword(s). Let us assume that the text "BLOCKED" is denylisted by the DPI system. We use two streams (IDs 0 and 4) to send this data from the client to the server. We create the frames such that stream 0 frames reassemble to "BLO" and stream 4 frames reassemble to "CKED". After the packets containing stream 0 frames, we insert a packet with a duplicate packet number, with a randomised payload (structurally invalid frames) , into the sequence of packets. We then insert packets containing the stream 4 frames and send this sequence of packets to the server. The DPI server reassembles the packets as "BLO" and therefore fails to detect the denylisted word. The destination server reassembles the packets as "BLOCKED" and is thus able to elude the DPI.

- **Exploiting Stream Offset Overlaps:** Another way to elude DPI systems discovered by our fuzzer is based on the idea

Packet No.	Stream Frame Payload	Stream Offset	Stream Finbit	QUICHE Reassembled Data (Destination Server)	QUICLY Reassembled Data (DPI system)
1	OCKED	2	True	__OCKED	__OCKED
2	BLIN	0	False	BLOCKED	BLINKED

Table 3: DPI elusion using Overlapping Offsets (Case 1)

Packet No.	Stream Frame Payload	Stream Offset	Stream Finbit	MVFST Reassembled Data (Destination Server)	QUICLY Reassembled Data (DPI system)
1	OCKED	2	True	__OCKED	__OCKED
2	INKED	2	False	__OCKED	__INKED
3	BL	0	False	BLOCKED	BLINKED

Table 4: DPI elusion using Overlapping Offsets (Case 2)

Seed Value	Generator	Implementation	Error Description
5224880393376231849	Basic Stream Reassembly	MVFST	Null Pointer Dereference
6867396659762739268	Flow-Control-Aware Stream Reassembly	QUANT	Heap use after free
3544824671711368728	Flow-Control-Aware Stream Reassembly	QUICLY	Null Pointer Dereference
8969571667189322506	Basic Stream Reassembly	NEQO	Assertion Failed

Table 5: Summary of IUT Crashes

that implementations might handle receiving data at overlapping offsets in different ways. We consider the second entry of Table 1 for this section. An analysis of the packets sent to three servers (QUICHE, QUICLY and MVFST), as shown in Table 2, reveals the reason for diverging data reassembly.
Looking at Packets 3, 7 and 8, we can see that when the QUICHE server receives a payload at an offset that it has already received data for, it simply ignores the new data. The QUICLY server on the other hand replaces the payload with the new payload. The analysis of the MVFST implementation is more involved than these cases. Packet 3 contains the first stream frame with data overlapping with the already reassembled data. Here, the MVFST server replaces the existing data with the new data that it receives. Packet 7 is the next packet that carries overlapping data. In this case, however, the MVFST implementation does not replace the data. Further analysis reveals that MVFST does not replace the existing data if the starting offset of the payload in a stream frame already has existing data. Any unoccupied positions that lie within the range of this new frame payload will be filled with characters from the new payload for those locations. However, if the starting offset of the frame payload is not already occupied, the contents of this frame will replace all the existing characters that it overlaps with.
According to Draft 27 of IETF QUIC protocol, data at a given offset must not change if it is sent multiple times and an endpoint MAY treat receipt of different data at the same offset within a stream as a connection error [6]. The behaviour of QUICHE seems to be consistent with the specification as it does not change the data at an offset once received. QUICLY seems to completely ignore the case of repeating offsets and the MVFST implementation seems to be partially consistent with the specification but does not consider all possible cases of overlapping offsets.
We can generalize this fuzzing sequence to elude the DPI system in missing any blocked keyword(s). Let's assume that the text "BLOCKED" is supposed to be denylisted by the DPI system. We demonstrate two DPI elusion cases:

- If QUICLY is used for DPI and QUICHE used for receiving data, Table 3 depicts the packets we send and the data reassembled by them which consequently allows eluding the DPI system.
- Similarly, if QUICLY is used for DPI and MVFST used for receiving data, Table 4 depicts the packets we send and the data reassembled by them which consequently allows eluding the DPI system.

5.3.2 *QUIC Implementation Bugs and Vulnerabilites:*

In addition to strategies that lead to different data being reassembled by semantically equivalent QUIC implementation servers, our fuzzer revealed several security-critical vulnerabilities in the tested QUIC implementations. In this section, we discuss the results in which a sequence of packets lead to a crash on one implementation but not on another.

We detect the errors by observing the status code values in the trace generated for each execution of the fuzzer. These errors can be trivially leveraged in order to evade a DPI system using the buggy QUIC implementation. If a server crashes, the execution at the client will timeout at the client and the fuzzer will assign the value "ServerTimeOut" to the status code.

These sequences of packets could effectively be used to flush the buffers storing the reassembled data in the DPI system mid way through data transfer, thereby allowing blocked content to go undetected. On top of this, and possibly more important, these

application crashes illustrate that our fuzzing framework can also be readily applied to finding security vulnerabilities in QUIC implementations. In the following, supported by Table 5, we describe the bugs that our fuzzer revealed in detail.

- QUANT: The QUANT server runs into a heap use after free error[4]. Using address sanitizer [24] reveals that the implementation tries to access the state of a stream, i.e., check whether a stream is closed or not by calling "q_is_stream_closed ()" after the memory allocated to the stream has already been freed.
- NEQO: The NEQO server runs into an Assertion failed error[5]. A Connection is already in the "Closed" state and the NEQO server calls close () function on this connection to try close it again.
- QUICLY: The QUICLY server runs into a Segmentation Fault[6]. The reason for the Segmentation Fault is a null pointer dereference [3]. When the server receives a stream frame from a client with a stream id not permitted by the QUIC specification, the "QUICLY_get_ingress_max_streams ()" function tries to access the value stored in a NULL pointer causing the server to run into a segmentation fault.
- MVFST: The MVFST server tries to dereference a null pointer. This causes the server to run into a Segmentation Fault[7] and crash. When we send a stream frame to the server such that the *Offset* field has a non-zero value but value of the *OFF bit* is set to 0 (which indicates that the Offset field is absent), the *FIN bit* is set to true and the *Payload* field empty, the MVFST implementation is unable to handle such a packet and runs into a Segmentation fault.

Coordinated Disclosure: We disclosed these security-critical vulnerabilities to the developers of the implementations and provided them with as much information as possible to assist them in fixing the root causes for these problems.

6 DISCUSSION

Man-in-the-middle interception techniques are widely used in enterprise controlled [27, 39] networks. With the development of an encrypted by default protocol like QUIC which has less information visible to the network than TCP [4], interception techniques will become even more important for monitoring purposes. These are the situations where a framework like DPIFuzz can be used to identify the potential limitations of the DPI system. Leveraging our results, we can conclude that:

- Programmers implementing a protocol specification often fail to account for all the possible types of inputs that an implementation might encounter [32].
- Programmers often fail to handle properly all the possible state transitions that the state machine of an implementation might encounter.
- All protocol specifications have some degree of ambiguity and have parts that are deliberately left unspecified and the programmers are free to handle these cases as they see fit. These are potential points of divergence in the behaviours of the implementations.

Implications of DPIFuzz for DPI systems: Our main finding is that DPI systems that parse and reassemble QUIC packets in a different manner than the actual recipient are prone to elusion and differential fuzzing is an effective technique to uncover elusion strategies for such systems. As of writing this paper, we are not aware of any actual DPI systems for QUIC. Designing performant DPI systems for QUIC will be a challenge owing to the fact that QUIC uses TLS 1.3 for providing security. TLS 1.3 makes selective MITM proxy interventions for DPI harder because of the encrypted Server Hello messages (which contain the server certificates used to verfiy server identity). Additionally, the ability of end points to change connection IDs as well as migrate connections would make monitoring of connections more challenging for DPI systems. Having said this, given the expected rapid deployment of QUIC, DPI systems will have to be developed for QUIC based traffic in the near future. To test the validity and effectiveness of our approach, we used open source QUIC implementations as a DPI system. Given that actual DPI systems are usually limited in functionality and resources, our findings likely just represent the lower bound of elusion sequences that DPIFuzz will be able to find for any upcoming full-fledged QUIC DPI system.

Limitations and Future Work: DPIFuzz uses a smart logic-based fuzzer. However, as it is not coverage guided, it does not attempt to maximize the code coverage of the implementations being tested and different sequences of fuzzed packets likely repeatedly end up testing the same code parts. Also, we currently do not track how much of the implementation code we cover owing to the variance in the languages used to code the implementations. Extending the functionality of the fuzzer to make it coverage conscious would definitely improve the efficiency and effectiveness of DPIFuzz.

Additionally, while analysing the results we encountered sequences which registered crashes according to their trace files. However, when replaying these sequences individually to the IUTs for analysis, we noticed that not all crashes were reproducible. The reason for this is that since DPIFuzz executes multiple fuzzer instances in parallel, if an IUT has multiple active connections and one of them causes it to crash, the remaining open connections also register a crash. Also, once an IUT crashes, there is a certain amount of delay before it is automatically restarted. If our fuzzer sends a sequence to the IUT within this delay window, it registers a “TLSHandshakeFail” value for the status code even though the handshake does not fail when the sequence is replayed to the IUT. This can be avoided by replaying the according candidates in a non-parallel version of DPIFuzz, at the cost of a temporal performance slowdown.

7 RELATED WORK

Fuzzing: Most related fuzzing works so far have focused on discovering software vulnerabilities, which is not our prime focus. Instead, we aim to discover ambiguities in network protocol implementations. We are aware of just a few related works that have designed fuzzers for secure network protocols, as described next.

[4]https://github.com/NTAP/quant/issues/61
[5]https://github.com/mozilla/neqo/issues/571
[6]https://github.com/h2o/quicly/issues/347
[7]https://github.com/facebookincubator/mvfst/issues/135

In 2012, Tsankov et al. [32] designed a fuzzer for the IKE (Internet Key Exchange) protocol. They used fuzz operators to fuzz the payloads, messages and fields in the IKE protocol. The design of their fuzzer differs significantly from ours, though, as their fuzzer is not a client in itself but instead sits between the client and the server. Additionally, they do not use the Sequence level mutators and do not allow embedding logic into their fuzzer.

A fuzzing-based differential black-box testing approach was used by Walz and Sikora [14] in 2015 to test the handshake phase of TLS implementations. The fact that they only fuzz the initial ClientHello message which is not encrypted makes their approach significantly different and less detailed than our fuzzing methodology.

A concurrent work by Pham et al. [40] uses a mutational approach coupled with server state feedback to create a coverage-guided, greybox fuzzer for protocol implementations. Instead of generating packets, they replay variations of previously captured packets and use the server response codes to identify the states exercised by a message sequence. They explore FTP and RTSP implementations using their work. However, they do not demonstrate the applicability of their work to protocols like QUIC where previously captured traffic cannot be directly used to establish a connection with, or test, a server. Additionally, the server state machine-learning algorithm used relies on server response codes and does not work for implementations that do not generate response codes.

Deep Packet Inspection Elusion: All the way back in 1998, Ptacek et al. [33] proposed the idea of insertion and evasion attacks on Network Intrusion Detection Systems (NIDS) and highlighted several implementation level differences in TCP and IP Protocols. In 2013, Khattak et al. [35] used the same approach to uncover several vulnerabilities in GFW.

We do levearage similar principles of insertion and evasion packets but, we (i) are the first to explore them for the QUIC protocol, and (ii) find such packets in an automated way.

A genetic algorithm based automated approach to detect packet manipulation based evasion strategies was used by Bock et al. [23] in 2019. They used genetic algorithms to generate packet manipulation strategies based on basic packet manipulation primitives (drop, tamper headers, duplicate and fragment), and then apply these strategies to user input. This is in contrast to our work where our fuzzer actually generates input sequences (instead of strategies) and mutates those in a randomised manner. The primary aim of their work was to automate "censorship evasion" which differs significantly from the purpose of our work of allowing DPI users to test the robustness of ther DPI systems by uncovering DPI elusion strategies that attackers might leverage to get restricted or malicious content in and out of protected networks.

Another approach to detect automated DPI evasion strategies was developed by Wang et al. [41] in 2020. They used Selective Symbolic Execution to explore TCP implementations and discover insertion and evasion packets which could effectively "de-synchronise" the state machines of the DPI middlebox and the implementation being tested. Their approach, however, is limited by path explosion (even though they use just three symbolic packets) and they make pruning decisions based on their domain knowledge to tackle path explosion. For a protocol like QUIC which has a large variety of frames and packets, the number of symbolic packets needed to explore an implementation would be much more as compared to TCP and consequently, the path explosion much worse.

These approaches by Wang et al. [41] and Bock et al. [23] detect elusion strategies in an automated way, but they do not demonstrate applicability to the QUIC protocol. To the best of our knowledge, we are the first (i) to use the concept of Differential Fuzzing to automate detection of DPI elusion strategies, (ii) to explore DPI elusion strategies for the QUIC protocol, and (iii) to design a modular and structure-aware fuzzer for QUIC.

8 CONCLUSION

In this paper, we have presented a differential fuzzing framework which allows detecting DPI elusion strategies for the QUIC protocol, when being inspected by an stateful DPI system, in an automated way. We test the framework against multiple open source implementations of the QUIC protocol and demonstrate techniques which can allow DPI elusion, thereby proving the effectiveness of our approach. DPIFuzz would enable organisations to test their QUIC traffic monitoring systems and uncover possible elusion strategies that attackers might use. As a consequence, it will help improve the security of enterprise networks.

9 AVAILABILITY

The code for our differential fuzzer is open source[8] and can be freely used and extended by (i) organisations to test the robustness of their DPI implementations, and (ii) DPI users to identify potential shortcomings in their methodology.

REFERENCES

[1] 2015. *A collection of vulnerabilities discovered by the AFL fuzzer (afl-fuzz).* https://github.com/mrash/afl-cve

[2] 2017. *Fuzzing Basics.* Retrieved May 11, 2020 from https://docs.microsoft.com/en-us/security-risk-detection/concepts/fuzzing-basics

[3] 2019. *CWE VIEW: Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors.* Retrieved May 19, 2020 from https://cwe.mitre.org/data/definitions/1200.html

[4] 2019. *Manageability of the QUIC Transport Protocol.* Retrieved May 22, 2020 from https://quicwg.org/ops-drafts/draft-ietf-quic-manageability.html

[5] 2020. *Hongfuzz Found Bugs.* Retrieved May 7, 2020 from https://github.com/google/honggfuzz#trophies

[6] 2020. *Overlapping Offsets.* Retrieved June 06, 2020 from https://quicwg.org/base-drafts/draft-ietf-quic-transport#section-2.2-4

[7] 2020. *QUIC, a multiplexed stream transport over UDP.* Retrieved May 11, 2020 from https://www.chromium.org/quic

[8] 2020. *QUIC: A UDP-Based Multiplexed and Secure Transport.* Retrieved May 11, 2020 from https://quicwg.org/base-drafts/draft-ietf-quic-transport

[9] 2020. *QUIC: Packets and Frames.* Retrieved June 06, 2020 from https://quicwg.org/base-drafts/draft-ietf-quic-transport#name-packets-and-frames

[10] 2020. *Syzkaller Found Bugs.* Retrieved May 7, 2020 from https://github.com/google/syzkaller/blob/master/docs/linux/found_bugs.md

[11] 2020. *Using TLS to Secure QUIC.* Retrieved May 11, 2020 from https://datatracker.ietf.org/doc/draft-ietf-quic-tls/

[12] Alyssa Wilk et al. Adam Langley, Alistair Riddoch. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM '17: Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* https://dl.acm.org/doi/10.1145/3098822.3098842

[13] Nour-Eddine Lakhdari Mourad Debbabi Amine Boukhtouta, Serguei A. Mokhov and Joey Paquet. May 2016. Network malware classification comparison using DPI and flow packet headers. In *Journal of Computer Virology and Hacking Techniques, vol. 12, no. 2.* Springer, 69–100. https://link.springer.com/article/10.1007%2Fs11416-015-0247-x

[8] https://github.com/piano-man/DPIFuzz

[14] Axel Sikora Andreas Walz. March-April 1, 2020. Exploiting Dissent: Towards Fuzzing-based Differential Black-Box Testing of TLS Implementations. In *IEEE Transactions on Dependable and Secure Computing*. IEEE.

[15] Cloudflare. 2018. *Quiche: Savoury implementation of the QUIC transport protocol and HTTP/3.* https://github.com/cloudflare/quiche

[16] Facebook. 2018. *MVFST: A client and server implementation of IETF QUIC protocol in C++ by Facebook.* https://github.com/facebookincubator/mvfst

[17] Arash Molavi Kakhki Arian Akhavan Niaki David Choffnes Phillipa Gill Alan Mislove Fangfan Li, Abbas Razaghpanah. November 2017. lib•erate, (n): a library for exposing (traffic-classification) rules and avoiding them efficiently. In *IMC '17: Proceedings of the 2017 Internet Measurement Conference*. ACM, 128–141. https://dl.acm.org/doi/10.1145/3131365.3131376

[18] Fastly. 2017. *Quicly: A QUIC implementation, written from the ground up to be used within the H2O HTTP server.* https://github.com/h2o/quicly

[19] Google. 2015. *honggfuzz: Security oriented software fuzzer.* https://github.com/google/honggfuzz

[20] Google. 2015. *syzkaller: An unsupervised coverage-guided kernel fuzzer.* https://github.com/google/syzkaller

[21] Erik Hjelmvik. 2020. *Sniffing Decrypted TLS Traffic with Security Onion.* https://www.netresec.com/?page=Blog&month=2020-01&post=Sniffing-Decrypted-TLS-Traffic-with-Security-Onion

[22] Jeff Jarmoc. 2012. SSL/TLS Interception Proxies and Transitive Trust. In *Black Hat Europe*. blackhat.

[23] et al. Kevin Bock, George Hughey. 2019. Geneva: Evolving Censorship Evasion Strategies. In *CCS*. ACM.

[24] Konstantin Serebryany. 2011. *Address Sanitizer.* Google. https://github.com/google/sanitizers/wiki/AddressSanitizer

[25] Erling Ellingsen Collin Jackson Lin-Shung Huang, Alex Rice. [n.d.]. Analyzing Forged SSL Certificates in the Wild. In *IEEE Symposium on Security and Privacy*.

[26] LLVM. 2015. *libFuzzer: a library for coverage-guided fuzz testing.* http://llvm.org/docs/LibFuzzer.html

[27] Mohammad Mannan Louis Waked and Amr Youssef. 2018. The Sorry State of TLS Security in Enterprise Interception Appliances. (2018).

[28] Quentin De Coninck Maxime Piraux and Olivier Bonaventure. 2018. Observing the Evolution of QUIC Implementations. In *EPIQ'18: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. https://doi.org/10.1145/3284850.3284852

[29] Michal Zalewski. 2019. *american fuzzy lop.* Google. https://lcamtuf.coredump.cx/afl/

[30] Mozilla. 2019. *Neqo: An Implementation of QUIC written in Rust.* https://github.com/mozilla/neqo

[31] NetApp. 2016. *Quant: QUIC implementation for POSIX and IoT platforms.* https://github.com/NTAP/quant

[32] Mohammad Torabi Dashti Petar Tsankov and David Basin. 2012. SECFUZZ: Fuzz-testing Security Protocols. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST 2012)*. Zurich, Switzerland.

[33] Thomas Ptacek and Timothy Newsham. [n.d.]. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. ([n. d.]).

[34] Ashish Kumar Lucian Cojocar Cristiano Giuffrida Herbert Bos Sanjay Rawat, Vivek Jain. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS Symposium 2017*.

[35] Philip D. Anderson Vern Paxson Sheharbano Khattak, Mobin Javed. 2013. Towards Illuminating a Censorship Monitor's Model to Facilitate Evasion. In *3rd USENIX Workshop on Free and Open Communications on the Internet (FOCI '13)*.

[36] D. Smallwood and A. Vance. 2011. Intrusion analysis with deep packet inspection: increasing efficiency of packet based investigations. In *International Conference on Cloud and Service Computing*. IEEE, 342–347.

[37] K. Xiong T. Chin and C. Hu. 2018. Phishlimiter: A phishing detection and mitigation approach using software-defined networking. In *IEEE Access, vol. 6*. Springer, 42516–42531.

[38] R. Tahboub and Y. Saleh. 2014. Data leakage/loss prevention systems (dlp). In *World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE, 1–6.

[39] Roelof Du Toit. 2017. Responsibly Intercepting TLS and the Impact of TLS 1.3. (2017).

[40] Abhik Roychoudhury Van-Thuan Pham, Marcel Böhme. 2020. AFLNET: A Grey-box Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[41] et al. Zhongjie Wang, Shitong Zhu. 2020. SYMTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *NDSS Symposium 2020*.

[42] Zhiyun Qian Chengyu Song Srikanth V. Krishnamurthy Zhongjie Wang, Yue Cao. November 2017. Your state is not mine: a closer look at evading stateful internet censorship. In *IMC '17: Proceedings of the 2017 Internet Measurement Conference*. ACM, 114–127. https://dl.acm.org/doi/10.1145/3131365.3131374

APPENDIX

Table 6 contains the transport parameter values used in the servers. Table 7 contains the values for mutation probabilities defined in Section 4.3. Table 8 contains a summary of the differences uncovered by our fuzzer between different pairs of implementations.

Transport Parameter	Value
initial_max_data	1048576
initial_max_stream_data_bidi_local	66560
initial_max_stream_data_bidi_remote	66560
initial_max_stream_data_uni	66560
initial_max_streams_bidi	2048
initial_max_streams_uni	2048
max_idle_timeout	60000
max_packet_size	1500
ack_delay_exponent	3

Table 6: Transport Parameter Values

Probability Parameter	Value
α_1, α_2	$\frac{2}{3}$
β, γ	$\frac{1}{2}$
ω	$\frac{1}{3}$
α_3	0

Table 7: Probability Values

Seed Value	Generator	IUT 1	IUT2	Difference Category
4373445819122772715	BSR	QUICHE	MVFST	Duplicate Packet Number
4373445819122772715	BSR	QUICHE	Quicly	Duplicate Packet Number
7253654666463259418	OO	QUICHE	MVFST	Overlapping Offset Handling
7253654666463259418	OO	MVFST	QUICLY	Overlapping Offset Handling
7253654666463259418	OO	QUICHE	QUICLY	Overlapping Offset Handling
5224880393376231849	BSR	MVFST	QUICHE	Null Poninter Dereference
5224880393376231849	BSR	MVFST	QUICLY	Null Poninter Dereference
5224880393376231849	BSR	MVFST	NEQO	Null Poninter Dereference
6867396659762739268	FCSR	QUANT	QUICHE	Heap use after free
6867396659762739268	FCSR	MVFST	QUICHE	Heap use after free
6867396659762739268	FCSR	NEQO	QUICHE	Heap use after free
3544824671711368728	FCSR	QUICLY	QUICHE	Null Poninter Dereference
3544824671711368728	FCSR	QUANT	QUICLY	Null Poninter Dereference
3544824671711368728	FCSR	MVFST	QUICLY	Null Poninter Dereference
3544824671711368728	FCSR	NEQO	QUICLY	Null Poninter Dereference
4471396090777236879	FCSR	QUANT	QUICHE	Heap use after free
4471396090777236879	BSR	QUANT	QUICLY	Heap use after free
4471396090777236879	FCSR	QUANT	MVFST	Heap use after free
4471396090777236879	BSR	NEQO	QUANT	Heap use after free
8969571667189322506	BSR	NEQO	QUICHE	Assertion Failed
8969571667189322506	BSR	MVFST	NEQO	Assertion Failed
8969571667189322506	BSR	QUANT	NEQO	Assertion Failed
8969571667189322506	BSR	QUICLY	NEQO	Assertion Failed
6399819732713312401	BSR	QUICHE	QUICLY	Duplicate Packet Number
6399819732713312401	BSR	QUICHE	MVFST	Duplicate Packet Number
98269818471122413	FCSR	QUICHE	QUICLY	Null Pointer Dereference
3738301969304892419	FCSR	QUICHE	NEQO	Assertion Failed
3738301969304892419	FCSR	QUANT	NEQO	Assertion Failed
3738301969304892419	FCSR	MVFST	NEQO	Assertion Failed
8566626253336265389	OO	MVFST	QUICLY	Overlapping Offset Handling
8566626253336265389	OO	MVFST	QUICHE	Overlapping Offset Handling
8566626253336265389	OO	QUICHE	QUICLY	Overlapping Offset Handling
423731078537465151	BSR	QUICHE	QUANT	Heap use after free
423731078537465151	BSR	MVFST	QUANT	Heap use after free
423731078537465151	BSR	QUICLY	QUANT	Heap use after free
2730155016155308010	BSR	QUICHE	NEQO	Assertion Failed
2730155016155308010	BSR	QUANT	NEQO	Assertion Failed
2730155016155308010	BSR	MVFST	NEQO	Assertion Failed

Table 8: Summary of differences uncovered by the fuzzer. *BSR* -> Basic Stream Reassembly, *FCSR* -> Flow-Control-Aware Stream Reassembly, *OO* -> Overlapping Offset