ANALYSIS AND AUTOMATED DISCOVERY OF ATTACKS IN TRANSPORT PROTOCOLS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Samuel C. Jero

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Cristina Nita-Rotaru, Co-Chair

Department of Computer Science

Dr. Sonia Fahmy, Co-Chair

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Dr. Dan Goldwasser

Department of Computer Science

Approved:

Dr. Voicu Popescu by Dr. William J. Gorman

Head of Graduate Program

Soli Deo Gloria

ACKNOWLEDGMENTS

I would like to take this opportunity to thank the people who made this dissertation possible.

First, I would like to thank my advisor, Dr. Cristina Nita-Rotaru, for her guidance, assistance, support, and encouragement during the years I have pursued my PhD. Her comments and insights have been invaluable through this process, both for shaping and presenting my research and for beginning my career. The skills and techniques I have learned from her will, I am certain, be invaluable to my future career in research.

I also want to thank Dr. Sonia Fahmy and Dr. Dongyan Xu for serving on my advisory committee and for their insightful feedbacks on my dissertation proposal. A special thanks goes to Dr. Fahmy for agreeing to be the co-chair of my advisory committee.

I also need to thank Dr. Hyojeong Lee and Dr. Endadul Hoque, both labmates and collaborators, for many insightful discussions and conversations about research and life.

On a personal note, I would like to thank my parents for encouraging me through the entire process of my education. My mother, particularly, deserves a special thanks for the many hours she put in to my early homeschooled education, which instilled in me the love of learning that lead me to this path. A special thanks must also go to my wife, Denise Jero, who has supported me, encouraged me, and put up with me through the last several years of my PhD. I am incredibly grateful for her encouragement, the many hours she has spent listening to me talk about research problems, and the selfless way she has let me pursue this dissertation. A final thanks goes to all my friends, especially in Graduate Intervarsity Christian Fellowship, for the encouragement and conversations that kept me sane through this process.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . viii

LIST OF FIGURES . . . ix

ABSTRACT . . . x

1 INTRODUCTION . . . 1
1.1 Focus and Motivation . . . 2
1.2 Dissertation Contributions . . . 6
1.3 Software Released . . . 7
1.4 Dissertation Roadmap . . . 7

2 TRANSPORT PROTOCOLS . . . 9
2.1 TCP . . . 11
2.1.1 TCP Congestion Control . . . 13
2.2 DCCP . . . 18
2.3 QUIC . . . 20

3 ATTACKER AND ATTACK MODEL . . . 25
3.1 Attacker Capabilities . . . 25
3.2 Attack Goals . . . 26
3.3 Attack Characteristics . . . 28

4 AUTOMATED ATTACK DISCOVERY FOR TRANSPORT PROTOCOL CONNECTIONS . . . 29
4.1 Introduction . . . 29
4.2 Design . . . 34
4.2.1 Overview . . . 34
4.2.2 Attack Injection . . . 36
4.2.3 Attack Strategy Generation . . . 39
4.3 Implementation . . . 42
4.3.1 Overview . . . 42
4.3.2 Attack Proxy . . . 44
4.3.3 State Tracking . . . 44
4.3.4 Parallelism . . . 45
4.4 Results . . . 46
4.4.1 TCP . . . 48
4.4.2 DCCP . . . 52

Page

4.4.3 Benefits of State-based Strategy Generation . . . 55
4.5 Natural Language Processing Pipeline . . . 56
4.5.1 Problem Definition . . . 58
4.5.2 Design . . . 62
4.5.3 Case Study: SNAKE . . . 72
4.5.4 Evaluation . . . 74
4.6 Summary . . . 85

5 AUTOMATED ATTACK DISCOVERY FOR TCP CONGESTION CONTROL . . . 86
5.1 Introduction . . . 86
5.2 TCPwn Attack Model . . . 90
5.2.1 Attacker and Attack Goals . . . 91
5.2.2 Attack, Strategy, Action . . . 92
5.3 Design . . . 94
5.3.1 Overview . . . 94
5.3.2 Abstract Strategy Generation . . . 97
5.3.3 Concrete Strategy Generation . . . 100
5.3.4 State Tracker . . . 104
5.4 Implementation . . . 108
5.4.1 Testing Environment . . . 109
5.4.2 Attack Detection . . . 110
5.5 Results . . . 111
5.5.1 On-path Attacks . . . 114
5.5.2 Off-path Attacks . . . 116
5.6 Summary . . . 120

6 PERFORMANCE AND AVAILABILITY ATTACKS FOR QUIC . . . 121
6.1 Introduction . . . 121
6.2 QUIC in the Presence of Attackers . . . 125
6.2.1 Replay Attacks . . . 126
6.2.2 Packet Manipulation Attacks . . . 127
6.3 Attack Results . . . 128
6.3.1 Replay Attacks . . . 129
6.3.2 Manipulation Attacks . . . 132
6.3.3 Other Attacks . . . 134
6.4 Attack Discussion . . . 135
6.5 Summary . . . 138

7 RELATED WORK . . . 139
7.1 Automated Attack Detection . . . 139
7.2 Transport Protocol Attacks . . . 140
7.3 NLP for Technical Domains . . . 142

Page

8 CONCLUSION 144
REFERENCES 148
VITA 158

LIST OF TABLES

Table Page

4.1 Summary of SNAKE Results 46
4.2 Classes of Attacks Discovered by SNAKE 47
4.3 Protocol Fields 66
4.4 Features for our Entity Mention Identification Classifier 68
4.5 Properties 69
4.6 Relations 69
4.7 Features for our Property/Relation Extraction Classifier 70
4.8 Packet Field Modifications for Test Cases 73
4.9 Dataset Statistics 74
4.10 NLP Document Processing Pipeline Evaluation 76
4.11 Entity Mention Identification Baselines 77
4.12 Property Extraction Baselines 79
4.13 Relation Extraction Evaluation 80
4.14 Coverage Evaluation 83
4.15 Attack Discovery Results 83
4.16 Attacks Discovered 83
5.1 Summary of TCPwn Results 112
5.2 Classes of Attacks Discovered by TCPwn 113
6.1 Attacks on QUIC 129

LIST OF FIGURES

Figure Page

2.1 TCP Header Fields . . . 11
2.2 TCP Connection-level State Machine . . . 12
2.3 TCP New Reno Congestion Control State Machine . . . 14
2.4 DCCP Header Fields . . . 18
2.5 DCCP Connection-level State Machine . . . 19
2.6 HTTPS stack over TCP and QUIC . . . 20
2.7 QUIC Public Header Fields . . . 21
2.8 QUIC Connection-level State Machine . . . 22
3.1 Attacker Types . . . 26
4.1 Design of SNAKE . . . 35
4.2 Attack Injection Algorithms . . . 36
4.3 SNAKE Test Network Topology . . . 42
4.4 NLP Analysis . . . 59
4.5 System Design for Automated Extraction of Protocol Grammars . . . 63
4.6 Example of Zero-shot Learning Classification for Entity Mentions . . . 67
4.7 Examples of Property and Relation Extraction . . . 71
5.1 New Reno and the Optimistic Ack Attack . . . 94
5.2 Design of TCPwn . . . 95
5.3 TCPwn Testing Environment . . . 108

ABSTRACT

Jero, Samuel C. PhD, Purdue University, May 2018. Analysis and Automated Discovery of Attacks in Transport Protocols. Major Professors: Cristina Nita-Rotaru and Sonia Fahmy.

Transport protocols like TCP and QUIC are a crucial component of today's Internet, underlying services as diverse as email, file transfer, web browsing, video conferencing, and instant messaging as well as infrastructure protocols like BGP and secure network protocols like TLS. Transport protocols provide a variety of important guarantees like reliability, in-order delivery, and congestion control to applications. As a result, the design and implementation of transport protocols is complex, with many components, special cases, interacting features, and efficiency considerations, leading to a high probability of bugs. Unfortunately, today the testing of transport protocols is mainly a manual, ad-hoc process. This lack of systematic testing has resulted in a steady stream of attacks compromising the availability, performance, or security of transport protocols, as seen in the literature.

Given the importance of these protocols, we believe that there is a need for the development of automated systems to identify complex attacks in implementations of these protocols and for a better understanding of the types of attacks that will be faced by next generation transport protocols. In this dissertation, we focus on improving this situation, and the security of transport protocols, in three ways. First, we develop a system to automatically search for attacks that target the availability or performance of protocol connections on real transport protocol implementations. Second, we implement a model-based system to search for attacks against implementations of TCP congestion control. Finally, we examine QUIC, Google's next generation encrypted transport protocol, and identify attacks on availability and performance.

1 INTRODUCTION

Transport protocols are an essential component of today's Internet, providing end-to-end delivery of data between applications and implementing guarantees like reliability, in-order delivery, and congestion control. They provide this service not only for end applications but also for other elements of the network infrastructure like BGP. Many of our secure network protocols, like TLS, rely on the guarantees provided by transport protocols, while other transport protocols, like QUIC, provide security guarantees directly.

The essential function of transport protocols is to provide end-to-end delivery of data; however, they also usually provide a variety of guarantees to ease development of applications and protect the network. These include reliable delivery via retransmissions, flow control, in-order delivery, and congestion control. As a result, the design and implementation of transport protocols is complex, with many components, special cases, error conditions, and interacting features. Further, transport protocol implementations often sacrifice simplicity and ease of understanding for improved performance. Hence, implementations are usually written in low level languages like C and make use of error-prone, but highly efficient, constructs, like pointer manipulation and type casting. This leads to a high probability of bugs.

Although there are few transport protocols in common use, there are many different implementations and variants of these transport protocols because of their ubiquitous role in network communication. For example, the `nmap` security scanner is able to detect 5,336 distinct TCP/IP network stack configurations in its most recent version [1]. This includes printers, VoIP phones, routers, and embedded systems, along with general purpose operating systems.

Despite the importance of these protocols and the complexity and number of their implementations, the testing of transport protocol implementations has been mainly

a manual and ad-hoc process [2–4]. This lack of systematic testing for transport protocols and their implementations has resulted in a stream of attacks [2, 3, 5, 6]. Consider TCP, one of the most well studied and well tested network protocols; the list of discovered attacks extends from the mid-1980's to the present day [7–13]. Many of these attacks have been discovered repeatedly or rediscovered again in slightly different contexts.

Prior work has focused on easing the development of manual tests [2, 14] or on enabling deeper testing for implementation crashes by using stateful fuzzing techniques [15–17]. Another line of work has sought to apply model checking techniques to implementations by leveraging techniques like symbolic execution [5, 18] and dynamic interface reduction [19] in combination with concrete attack execution. These works, however, require source code in particular languages and struggle to handle certainly frequently used low-level constructs like type casting, pointer casting, and function pointers. Some require source code annotation.

1.1 Focus and Motivation

We argue that there is a need for the development of automated systems to identify complex attacks in unmodified implementations of transport protocols and for a better understanding of the types of attacks that will be faced by next generation transport protocols. In this dissertation, we focus on improving this situation in three ways. First, we develop a system to automatically search for attacks that target the availability or performance of protocol connections on real transport protocol implementations. Second, we implement a model-based system to search for attacks against implementations of TCP congestion control. Finally, we examine QUIC, Google's next generation encrypted transport protocol, and identify attacks on its availability and performance.

Finding attacks against transport protocol connections. At the core of a transport protocol's utility is its ability to open a connection between two hosts

and exchange the desired data in a reasonable amount of time. Hence, we begin by focusing broadly on attacks targeting the availability or performance of a transport protocol and particularly the ability to establish or maintain a connection with the target implementation. For example, it is possible to blindly inject TCP reset packets and successfully terminate a target TCP connection by using a series of widely spaced acknowledgement numbers. This is known as the TCP Reset attack [13].

Unlike simple implementation crashes, attacks on performance or availability can be complicated to detect. Attacks may cause a degradation in performance, an improvement in performance at the expense of competing flows, the connection to stall completely, or even an incomplete close of the connection that fails to release all resources, eventually leading to resource exhaustion. We focus on being able to automatically discover attacks without modifying the transport protocol implementation under test. Additionally, we model malicious activity by modifying or injecting packets into the network. Thus, we need some algorithm to generate test cases and inject them into each test.

To completely automate testing, we also need to automatically create a description of the protocol's grammar (*i.e.,* its packet fields and the properties of and relations between these fields). Since transport protocols are extensively documented in natural language specification documents, typically RFCs, we believe that Natural Language Processing (NLP) holds significant promise for automatically extracting this information. Unfortunately, most NLP methods are sensitive to the data used at training time and do not adapt easily if applied on data from a different domain. Applying "off-the-shelf" implementations of NLP tools, typically trained on newswire data, or combining them in an ad-hoc way, often results in reduced performance and brittle applications.

We investigate these challenges and identify attack detection techniques based on expected competition and fairness. We also identify a key search space reduction technique that leverages the protocol's connection-level state machine, and then design and implement SNAKE to demonstrate the effectiveness of this approach. Finally,

we create a custom NLP document processing pipeline to automatically extract a transport protocol's grammar from the protocol's specification based on a lightweight zero-shot learning [20] framework.

Finding attacks against TCP congestion control. Congestion control is an essential component of TCP, the transport protocol that underlies the vast majority of Internet services today. Congestion control serves to protect the network from complete congestion collapse, and associated catastrophic throughput drops, and provides fairness between competing applications. This makes it is a prime target for attackers looking to impact the throughput of a flow.

Congestion control attacks can have severe implications for Internet services, including *financial loss*. Consider an attacker who wishes to degrade video quality and streaming experience for a subset of Netflix users. While Netflix recently began to encrypt all of its video traffic with TLS [21], TLS relies on TCP to transfer data across the network. As a result, an attacker can simply launch an attack misleading TCP into believing that the network is congested. This will cause TCP to repeatedly slow down its sending rate, causing rebuffering events and reduced video quality for any Netflix user subjected to this attack. Due to poor streaming experience, the users may consider turning to other video providers.

Unfortunately, techniques focusing broadly on transport protocol performance, like our work in SNAKE, are unable to effectively find attacks on congestion control due to its complex and highly dynamic nature. While a typical attack found by SNAKE might consist of one malicious action, attacks against congestion control typically require a a potentially long sequence of malicious actions spanning several states and transitions, where each action might trigger a new state, which in turn might require a different attack action. Attempting to use SNAKE to find these types of attacks in TCP would require the generation of 1.2×10^{24} test cases, which is impractical for testing in real networks. This search space explosion resulting from the dynamic and iterative nature of congestion control attacks is the key challenge to any automated attack finding system. Additionally, there are a huge variety of

variations and optimizations to congestion control that TCP implementations may include.

We investigate these challenges and develop a state machine model of TCP congestion control. Using this model, we develop a model-based attack search strategy, and then design and implement TCPwn to demonstrate its effectiveness.

Attacks on next-generation, encrypted transport protocols. Google has recently developed QUIC, a next generation transport protocol that provides encryption of all data and most headers as well as the ability to perform 0-RTT connections and dramatically improved acknowledgement information. Further, this protocol is widely deployed in the Internet via Google's Chrome browser, with Google reporting that 85% of all requests from Chrome to Google properties use QUIC, totaling about 7% of Internet traffic [22]. Additionally, there is a very active IETF standardization effort ongoing [23].

Existing work studying QUIC has examined the security guarantees it provides [24, 25] or considered its performance in benign environments [26–30]. Instead, we focus on the performance and availability attacks that an adversary could launch against QUIC since QUIC presents a distinctly different attack surface compared to traditional transport protocols. Thanks to most of the protocol headers being encrypted, attacks on congestion control and connection tear down, which can be used against traditional transport protocols like TCP, are ineffective against QUIC. However, QUIC also introduces 0-RTT connection establishment, which makes heavy use of caching, thereby exposing a host of new information to the attacker.

We manually investigate QUIC and its implementation in Chrome and discover three classes of attacks against its availability, resulting from design choices made to allow enhanced performance. We demonstrate five attacks against QUIC that completely prevent connection establishment.

1.2 Dissertation Contributions

In this dissertation, we improve the state of transport protocol implementation security by developing methods for automatic attack discovery in real implementations and examining attacks against next generation transport protocols. We summarize our key contributions as follows:

- **Attack discovery for transport protocol connections.** We present a system, SNAKE, to automatically identify attacks on transport protocol connection availability and performance in unmodified implementations. We develop a method for identifying attacks on performance based on expected competition and fairness and a novel attack injection technique based on leveraging the protocol's connection-level state machine. We demonstrate the practicality of this approach on five implementations of two transport protocols in four different operating systems, finding 9 attacks, 5 of which were previously unknown. We then develop an NLP document processing pipeline to extract a transport protocol's grammar from its natural language protocol specification document by leveraging the structure and linguistic regularities of the protocol specification document and a zero-shot learning framework which adapts to the specific properties of our domain. This approach allows us to adapt to new protocols easily and effectively. We find this pipeline capable of extracting protocol packet fields with an F-score of 0.74 and finding and linking properties with a success rate of 66%. We further demonstrate that this pipeline enables a reduction in testing effort (from 901 to 819 test cases) over a manually created grammar for TCP, while identifying the same set of attacks.

- **Attack discovery for congestion control.** We model congestion control as a finite state machine and develop a model-based attack strategy generation algorithm that generates possible congestion control attacks by identifying their key characteristics. This algorithm first generates abstract attack strategies from state machine cycles with desirable transitions. These are then converted

into concrete attack strategies by identifying attacker actions that cause the desired state machine transitions. To apply these attack strategies, we develop an algorithm to infer the current congestion control state of a sender by monitoring network packets. We demonstrate the practicality of this approach for finding attacks on real implementations of TCP by creating TCPwn. We test 5 TCP implementations from 4 Linux distributions and Windows 8.1 and find 11 classes of attacks, 8 of which were previously unknown.

- **Performance and availability attacks on QUIC.** We investigate performance and availability attacks against QUIC, Google's new encrypted, performance-optimized transport protocol. Due to the encryption of most protocol headers and the heavy reliance on caching for 0-RTT, QUIC has a very different attack surface than traditional transport protocols like TCP. We discover three classes of availability attacks based on design choices made to optimize performance. We identify and demonstrate 5 attacks against QUIC that completely prevent connection establishment.

1.3 Software Released

We have released both the SNAKE and TCPwn automated testing systems developed over the course of this work under the open-source BSD license. SNAKE can be found at `https://github.com/samueljero/snake` while TCPwn is available at `https://github.com/samueljero/TCPwn`.

1.4 Dissertation Roadmap

The rest of the dissertation is organized as follows. We provide additional background on transport protocols in Chapter 2 and discuss the attacker and attack models we consider in the rest of this work in Chapter 3. Chapter 4 presents SNAKE, our system for automatically finding attacks against transport protocol connections.

Chapter 5 presents TCPwn, our system for automatically finding attacks on TCP congestion control. Our investigation of performance and availability attacks against QUIC is presented in Chapter 6. We then discuss related work in Chapter 7 and conclude this dissertation in Chapter 8.

2 TRANSPORT PROTOCOLS

Transport protocols provide end-to-end communication between two applications running on different hosts. They enable multiple applications to use the same host by introducing the concept of a port and provide protection from data corruption using a checksum. Most transport protocols, with the exception of UDP [31] which provides only unreliable data delivery, provide additional services such as: (1) reliability, (2) ordered delivery, (3) flow control, and (4) congestion control. Providing these services requires the end-hosts to maintain state, which usually requires a connection-oriented protocol. In this work we focus solely on connection-oriented transport protocols due to the fact that they are used for the majority of Internet services and make up the majority of Internet traffic. These protocols consist of three phases: connection establishment, data transfer, and connection tear-down.

Connection establishment. Connection establishment, typically in the form of a handshake, takes place before any data is exchanged between end-hosts and serves to synchronize the state of both parties. During this phase, both hosts exchange sequence numbers, set sequence windows, and allocate buffers.

Data transfer. Once a connection is established, data flows between the two parties. During this phase, the transport protocol may provide a number of additional services.

Reliability is a common service that usually implemented using acknowledgments and retransmissions. The sender uses a buffer to store data that has been sent and includes a sequence number on each packet. Periodically, the receiver sends an acknowledgment to the sender. When the sender receives this acknowledgment, it determines what data has been lost and retransmits this data. Data acknowledged as received correctly is also removed from the sender's buffer. Since there is the possibility of acknowledgments being dropped by the network, the sender includes a timer

to retransmit data if no acknowledgment of sent data has been received after some lengthy time interval.

Ordered delivery is another commonly provided service that guarantees that data sent by one application is received at the other in the same order that it was sent. This is related to reliability and the two are usually implemented together. Implementing ordered delivery also requires a packet sequence number, allowing the receiver to determine the sending order. Packets received out of order are buffered at the receiver until the missing packets are received. The packets can then be delivered to the application in order.

Flow control ensures that a sender does not overwhelm a slow receiver with more data than it can buffer. The goal is for the sender to send at the same rate that the receiver is receiving. Flow control is specified as a sliding window indicating the data that the receiver can currently buffer. The sender is then limited to sending that window of data before receiving an acknowledgment indicating that the window has either slid forward or increased in size.

Congestion control is another common service which serves both to protect against congestion collapse in the network and to provide fairness between competing flows. Congestion collapse occurs when severe network congestion, or over-utilization, results in the network spending the majority of its time sending data that will eventually be dropped. This results in a persistent drop in throughput. *Fairness* ensures that if two flows are competing over the bandwidth on a bottleneck link, they share that bandwidth roughly equally. The networking community has generally understood this to mean that the flows achieve throughput within a factor of two of each other [32,33].

Some transport protocols, like QUIC, may provide security guarantees like *confidentiality* and *authentication* for exchanged data.

Connection tear down. After all data has been transferred, the end-hosts need a way to signal this and agree to release all state about the connection. Like connection establishment, connection tear down takes place through a handshake in

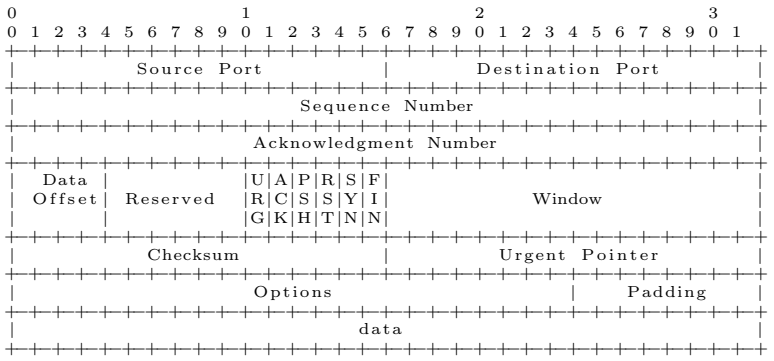


Figure 2.1.: TCP header fields. Each tick represents a bit position [34]

which the two hosts indicate that they are done sending data and are ready to close the connection.

In the following sections we describe in detail the three transport protocols that we will focus on in this work: TCP, DCCP, and QUIC.

2.1 TCP

TCP [34] is the most common transport protocol in use today, underlying the vast majority of Internet traffic, including web, email, instant messaging, and file transfer applications. It provides a reliable byte-stream between end hosts and implements reliability, in-order delivery, and flow control in order to achieve this. It also provides congestion control and attempts to ensure fairness.

A TCP connection is started by a handshake between two end-hosts [34]. This allows both end-hosts to inform each other of their initial sequence numbers and any important options. A similar handshake is performed at the end of the connection to make sure that all data has been delivered before the connection terminates. The full connection state machine is shown in Figure 2.2. Reliability is achieved by having the sender assign a sequence number to each byte of data and having the receiver acknowledge the highest consecutive byte of data it has received [34]. Retransmissions

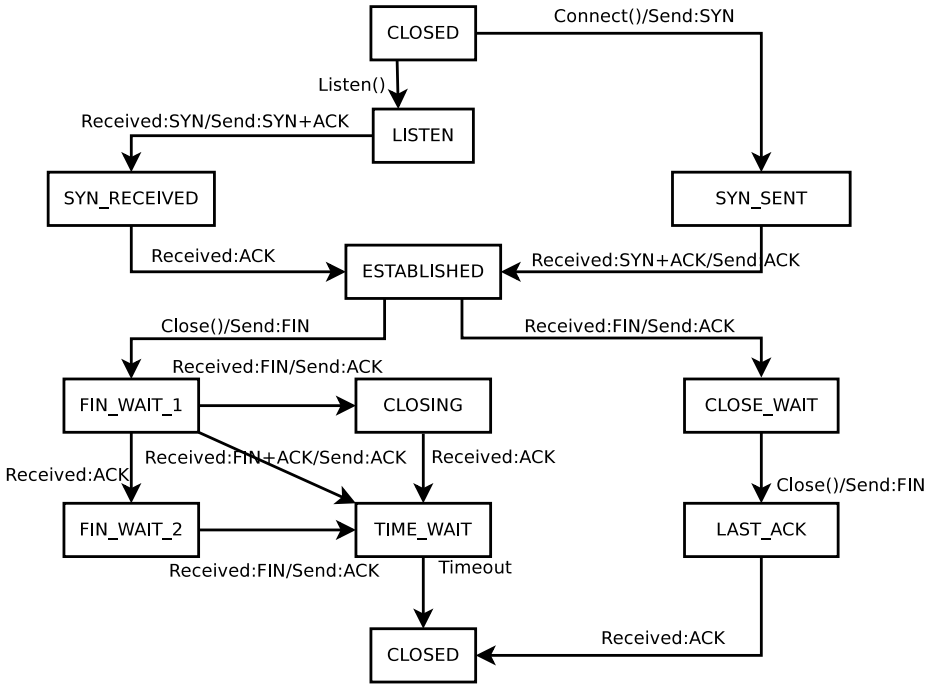


Figure 2.2.: TCP Connection-level State Machine

are triggered either by a retransmission timeout (RTO) or by receiving three duplicate acknowledgments, indicating the reception of packets above some missing bytes [35].

All TCP packets contain a single, common header. (shown in Figure 2.1). This header contains source and destination ports, a sequence number, an acknowledgment number, a set of control bits, a checksum, and options. TCP uses the set of control bits, or flags, in its header to indicate certain types of packets. The packets in the initial handshake are marked with the `SYN` flag; those in the final handshake with the `FIN` flag. Reset packets use the `RST` flag to abruptly terminate a connection after an error. An `ACK` flag indicates a valid acknowledgment field and is set on every packet after the initial `SYN`.

In most TCP connections, only one side of the connection is sending data at any given time. In order to provide feedback to the sender, TCP requires that receivers that are quiescent, that is, not currently sending data themselves, must periodically send an empty TCP packet to supply the sender with a current acknowledgement.

These empty TCP packets are simply TCP packets with no data and are usually called pure acknowledgements, or simply *acknowledgements*.

2.1.1 TCP Congestion Control

TCP provides congestion control to protect the network from congestion collapse and provide fairness between competing flows. We first describe classic TCP New Reno [35,36], and then briefly discuss then discuss optional improvements and variants like SACK [37], DSACK [38], TLP [39], PRR [40], FRTO [41], and others [42, 43].

At a high level, the congestion control of TCP New Reno consists of four phases: (1) slow start, (2) congestion avoidance, (3) fast recovery, and (4) exponential backoff. During the slow start phase the sender is probing the network to quickly find the available bandwidth without overloading the network; once such bandwidth is found, the sender enters a congestion avoidance phase in which the sender can send without causing congestion; in case of congestion and data loss, fast recovery or exponential backoff reduce the rate at which data is sent. The fast recovery phase is intended for less significant events where the beginning of congestion is detected through lost packets and acknowledgments, while the exponential backoff phase deals with more significant events where congestion is detected by the expiration of a large timeout. We present the finite state machine (FSM) model assumed for congestion control in Figure 2.3. Below we describe the associated events, variables, and states.

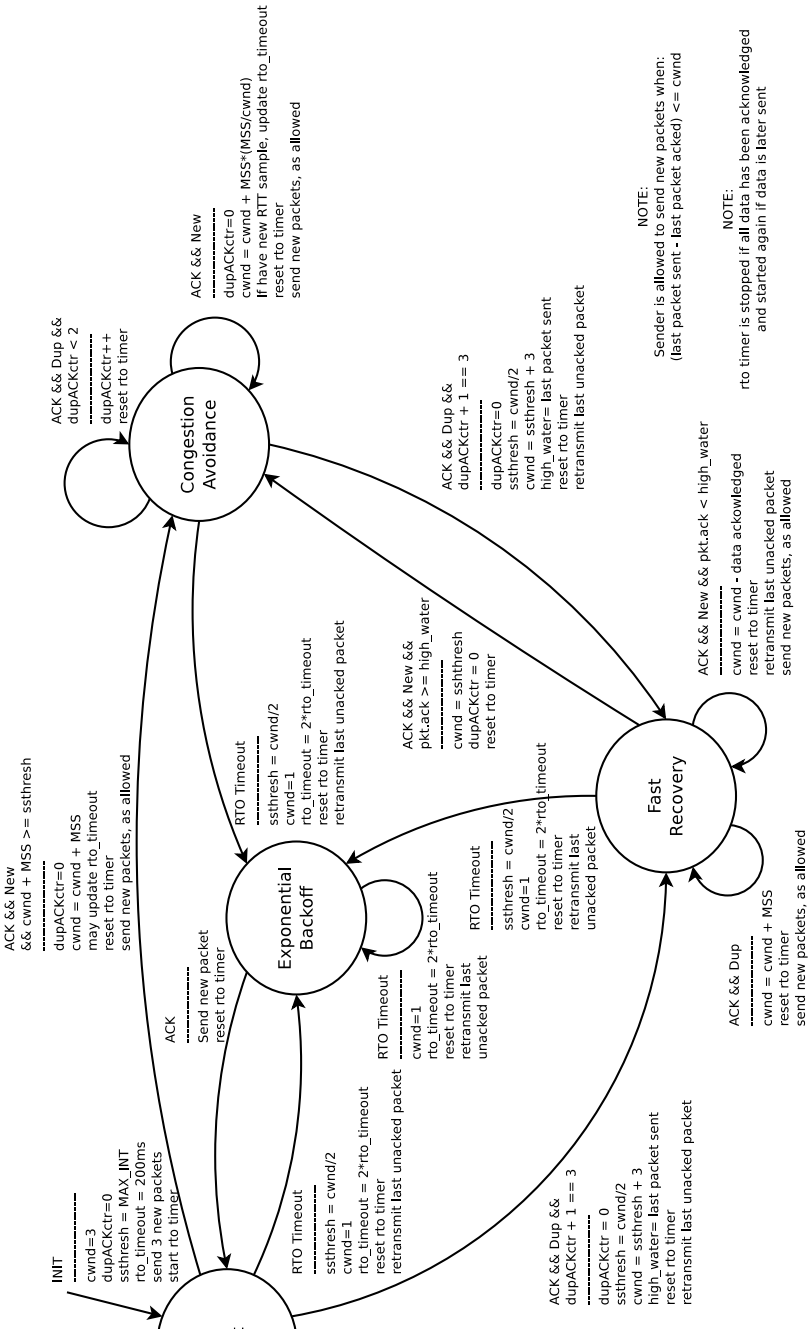


Figure 2.3.: TCP New Reno Congestion Control State Machine

(1) Events. TCP congestion control relies on two events for its operation, the reception of an acknowledgement (ACK) and the occurrence of a timeout (RTO Timeout):

ACK. This event denotes that an acknowledgement packet was received by the sender. TCP acknowledgements are byte-based and *cumulative*, i.e. the receiver acknowledges the highest byte of data at which all prior data has been received. A duplicate acknowledgment, and particularly three duplicate acknowledgments, are used to signal timely information about the network conditions.

RTO Timeout. This event denotes that a timeout occurred when data was outstanding and no acknowledgements were received for several Round-Trip-Times (RTTs). This indicates more severe conditions in the network since the last acknowledgement. This timer is started when new data packets are sent, reset on every acknowledgement, and stopped if all data has been acknowledged.

(2) Variables. The variables capturing the main functionality of congestion control can be grouped into three categorizes: variables related to the amount of data to be sent (`cwnd` and `ssthresh`), variables keeping track of acknowledged data (`dupACKctr` and `high_water`), and variables controlling timeouts (`rto_timeout`).

Congestion window – `cwnd`. This variable represents the number of bytes of data that TCP is allowed to have in the network at any given time. It is modified by TCP congestion control to increase or decrease the sending rate in response to network conditions.

Slow start threshold – `ssthresh`. This variable indicates the value of the congestion window `cwnd` at which TCP switches from slow start to congestion avoidance. TCP uses this information later in the connection by growing the window exponentially up to `ssthresh` after a timeout or idle period.

Duplicate ACK – `dupACKctr`. This variable tracks the number of duplicate acknowledgements received in slow start and congestion avoidance. Receiving three duplicate acknowledgements triggers a transition to fast recovery.

Highest sequence sent – `high_water`. This variable records the highest sequence number sent prior to entering fast recovery. Only once this sequence number has been acknowledged (or a timeout occurred) will fast recovery be exited.

RTO Timeout – `rto_timeout`. This variable indicates the current length of the RTO Timeout. It is usually set to $max(200ms, 2 * RTT + 4 * RTT\_Variance)$. If the RTO timer expires, this value is doubled, resulting in an exponential backoff.

(3) States. We can now describe the state machine from Figure 2.3. The states capture the four high-level phases described before.

Slow Start. In this state TCP rapidly increases its sending rate, as indicated by the congestion window `cwnd`, in order to quickly utilize the available bandwidth of the path while not overloading the network with a huge initial burst of packets. For each acknowledgement acknowledging new data, `cwnd` is incremented by MSS (Maximum Segment Size), which results in a doubling of the sending rate every RTT. TCP exits slow start on the RTO Timeout, after three duplicate acknowledgements—which indicate a lost packet—, or when the congestion window `cwnd` becomes bigger than the slow start threshold `ssthresh`. This last condition indicates that TCP is approaching a prior estimate of the fair-share connection bandwidth. TCP connections start in the slow start state with `ssthresh` set to MAX_INT, such that slow start is only exited on timeout or packet loss, and `cwnd` set to 10, allowing a burst of ten packets to be sent initially.[1]

Congestion Avoidance. In this state TCP is sending close to its estimate of the available bandwidth while also slowly probing for additional bandwidth. Every RTT `cwnd` is increased by one MSS sized packet. In practice, this is done by increasing `cwnd` by a small amount$((MSS * \texttt{cwnd})/MSS)$ for every new ACK received. TCP exits congestion avoidance either on an RTO Timeout or after receiving three duplicate acknowledgments, indicating a lost packet.

Fast Recovery. In this state, TCP is recovering from a lost packet indicated by three duplicate acknowledgements. TCP assumes that packet loss signals network

[1]This initial window was originally 2-4 packets [35], but has been increased to 10 packets in more recent standards [44] and implementations.

congestion, so it cuts its sending rate in half by halving `cwnd`, and retransmits the last unacknowledged packet. `ssthresh` is set to this new value of `cwnd`, providing an approximate bandwidth estimate in case of a timeout. TCP remains in fast recovery until all data outstanding at the time it entered fast recovery has been acknowledged or an RTO timeout occurs. This is achieved by saving the last packet sent in `high_water` upon entry and exiting once this packet has been acknowledged.

In fast recovery, acknowledgement handling is optimized to recover from the loss, avoid expensive RTO timeouts, and return to congestion avoidance as quickly as possible. As a result, duplicate acknowledgements received in fast recovery cause `cwnd` to be increased by one MSS, under the assumption that a duplicate acknowledgement means that a packet was received. This enables TCP to more accurately keep `cwnd` bytes of data in the network, which in turn reduces the likelihood of an RTO timeout. Additionally, an acknowledgement that acknowledges new data but not `high_water` immediately causes retransmission of the last unacked packet, under the assumption that this packet, too, was lost. Once `high_water` is acknowledged, TCP resets `cwnd` to `ssthresh`, undoing the increases resulting from duplicate acknowledgements, and transitions to congestion avoidance.

Exponential Backoff. In this state, TCP is retransmitting a lost packet each time the RTO timer expires. With each timer expiration, `rto_timeout` is doubled, resulting in an exponential backoff between retransmissions. This state is entered from any other state when the RTO timer expires, indicating that data is outstanding in the network but no acknowledgements have been received in `rto_timeout` seconds (at least 2 RTTs). This situation indicates the loss of a large number of packets and, likely, significant changes in network conditions. As a result, `ssthresh` is set to half of `cwnd`, `cwnd` is set to 1 MSS, and the last unacknowledged packet is retransmitted. TCP remains in this state, retransmitting this packet each time the RTO timer expires, until an acknowledgement is received, at which point it transitions to slow start.

Variations and Optimizations. The classic TCP New Reno congestion control algorithm we described above has seen a number of variations and optimizations over

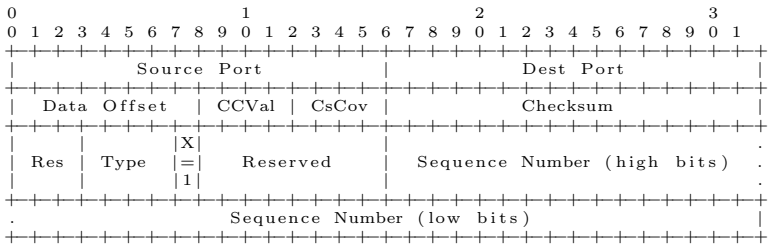


Figure 2.4.: DCCP generic header fields. Each tick represents a bit position [45]

the years. These include SACK [37], DSACK [38], TLP [39], PRR [40], CUBIC [42], and RACK [43]. These variations and optimizations consist of fairly minor changes to the basic New Reno algorithm. SACK [37], for example, provides the sender with additional information about received packets and uses this information to determine when to enter fast recovery. The logic of the decision does not change: fast recovery is entered when three packets above a loss have been received. SACK simply uses a more accurate method to detect this condition. Similarly, PRR [40] modifies New Reno by adopting paced packet sending during the self-loop in fast recovery. TLP [39] introduces a new, faster timeout state before exponential backoff. CUBIC TCP [42] changes precisely how `cwnd` is increased in congestion avoidance and decreased during fast recovery. While these changes affect the performance of TCP in certain network conditions, they follow the same phases of TCP congestion control as New Reno.

2.2 DCCP

The Datagram Congestion Control Protocol (DCCP) [45] was designed for applications that wanted congestion control, but did not want the retransmissions and head-of-line-blocking associated with TCP. Examples of such applications are applications that are highly latency sensitive, such as VoIP, realtime streaming video, and video gaming.

Like TCP, DCCP requires a handshake to setup a connection and another one to tear the connection down. The connection state machine is shown in Figure 2.5.

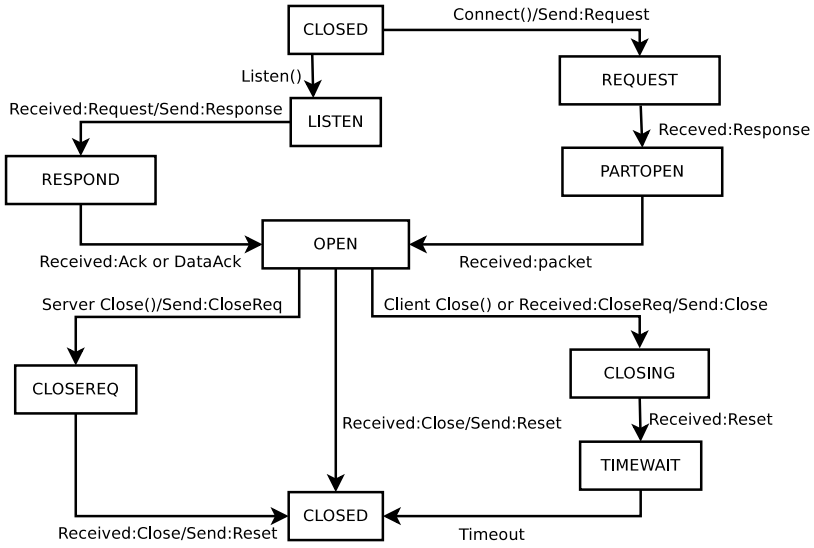
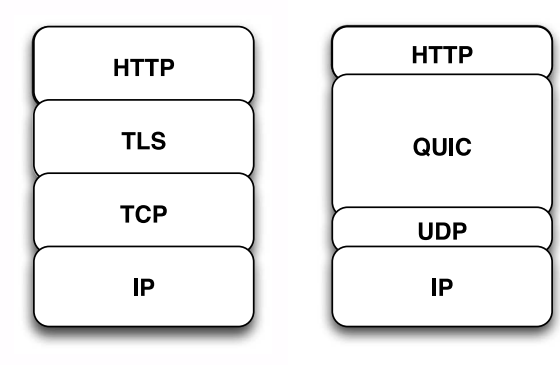


Figure 2.5.: DCCP Connection-level State Machine

However, unlike TCP, DCCP uses different types of packets for these handshakes [45]. Hence, the initial handshake consists of a `REQUEST` and a `RESPONSE` packet while the final handshake consists of a `CLOSE` and a `RESET` packet. The first several fields of all packet types are fixed and are shown in Figure 2.4 while later fields vary according to packet type.

DCCP assigns sequence numbers to packets instead of bytes. Further, every packet increments the sequence number; even pure acknowledgments carrying no data [45]. The receiver acknowledges the highest sequence number received; since DCCP does not retransmit data, a TCP-like cumulative acknowledgment does not make sense. However, this design means that DCCP endpoints can get out of sync after extended bursts of loss and reject valid packets as not within the current sequence window. To mitigate this issue, a third handshake—of `SYNC` and `SYNCACK` packets—is used to exchange the current sequence numbers of both parties and resynchronize the connection [45].

DCCP also features pluggable congestion control modules, known as CCIDs. Two are currently standardized: CCID 2 [46], TCP-like Congestion Control, and CCID 3 [47], TCP-Friendly Rate Control (TRFC). We focus on CCID 2 in this work.



(a) HTTPS over TCP (b) HTTPS over QUIC

Figure 2.6.: HTTPS stack over a) TCP and b) QUIC

It follows TCP's New Reno with SACK congestion control algorithm as closely as possible, although there are several minor changes due to DCCP's packet-based sequence numbers [46].

2.3 QUIC

Quick UDP Internet Connections, or QUIC, is a new transport protocol that was developed by Google, implemented in Chrome in 2013 [48], and now provides service for the majority of requests by Chrome to Google services [22]. QUIC's goal is to provide secure communication comparable to TLS [49] while achieving minimal connection setup latency. In particular, QUIC provides 0-RTT connections, enabling useful data to be sent in the first round trip. In contrast, the equivalent connection using TCP+TLS would require two or three RTT's, depending on whether TLS session resumption [50] was in use. This results in significant and noticeable latency savings that are of significant interest to today's online services and businesses. To achieve this, QUIC provides much of the functionality provided by TCP and TLS in a single protocol and runs on top of UDP, as shown in Figure 2.6. Combining this functionality in a single protocol enables optimizations like 0-RTT connections.

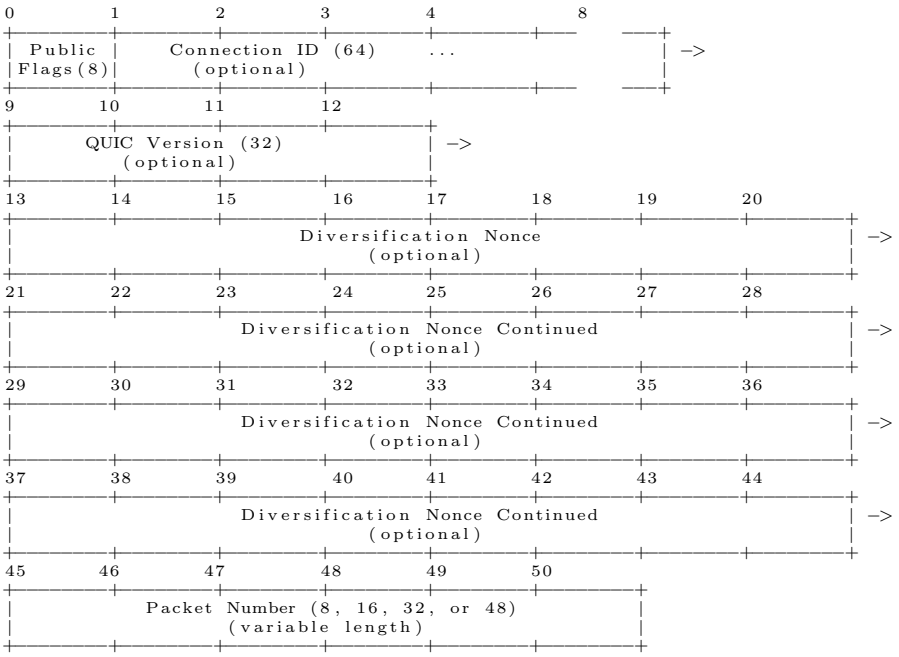


Figure 2.7.: QUIC public header fields. Each tick represents a byte [52]

QUIC also provides encryption of most transport protocol headers, to protect against manipulation attacks, and significantly improved acknowledgement information.

In this work, we focus on QUIC as specified and implemented by Google [51–53].[2] Note that QUIC is currently under active standardization by the IETF [23], and while IETF-QUIC has the same design goals and significant similarity to Google's original QUIC, there are a number of technical differences, including a big-endian packet format and the use of TLS 1.3 [54] instead of a custom cryptographic handshake [51, 53, 55]. The rest of our discussion focuses on Google's original QUIC.

QUIC packets contain a small public header and then a set of frames that are encrypted and authenticated after initial connection setup. The initial public header (shown in Figure 2.7) contains a set of public flags, a unique 64bit identifier for a connection referred to as a connection id or `cid`, a variable length packet number, and optionally a 32bit QUIC version number or a diversification nonce. All

[2]Specifically, QUIC version Q021, from October 2014.

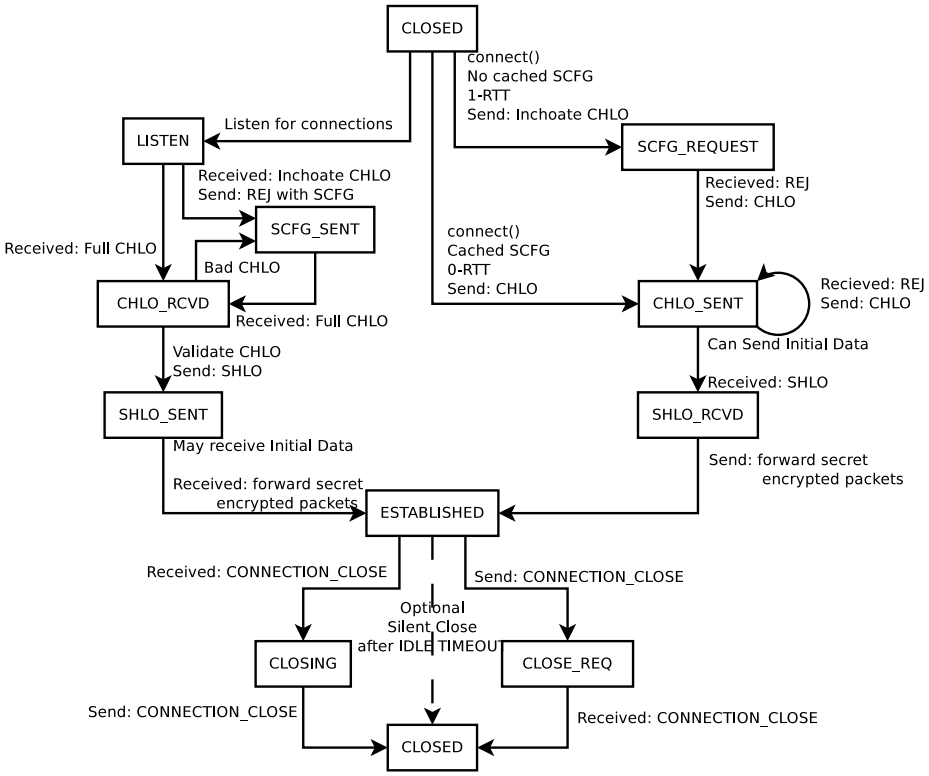


Figure 2.8.: QUIC Connection-level State Machine

other protocol information is carried in control and stream (data) frames that are encrypted and authenticated. Once the connection is setup, QUIC provides multiple byte-streams per connection, to reduce head-of-line-blocking. Reliability is enabled with per-packet sequence numbers and encrypted acknowledgements providing up to 256 SACK blocks [52]. QUIC packet numbers always start at one and are never retransmitted. Instead, missing frames are packaged into a new packet and resent. This separation of packet and byte-stream sequence numbers eliminates retransmission ambiguity. QUIC currently uses the same congestion control algorithms as TCP and combines them with packet pacing to help avoid burst loss [22, 52].

To provide 0-RTT connection establishment, QUIC provides a custom cryptographic handshake [53]. This handshake protocol operates as an exchange of messages over a reserved QUIC byte-steam. The basic idea is to cache important information about the server that will enable the client to determine the encryption key to be used

for each new connection. The client can then encrypt application data without hearing from the server, providing 0-RTT connection establishment. The full connection state machine for QUIC is shown in Figure 2.8.

The first time a client contacts a given server it has no cached information, so it sends an empty `c_hello` message. The server responds with an `s_reject` message containing the server's certificate and three pieces of information for the client to cache. The first of these is an object called an `scfg`, or server config. The `scfg` contains a variety of information about the server, including a Diffie Hellman share from the server, supported encryption and signing algorithms, and flow control parameters. This `scfg` has a defined lifetime and is signed by the server's private key to enable authentication using the server's certificate. Along with the `scfg`, the server sends the client a Source Address Token or `stk` and possibly a Server Nonce or `sno`. The `stk` is used to prevent IP spoofing while the `sno` is used to prevent replay of messages without requiring time synchronization for clients. The `stk` contains an encrypted version of the client's IP address and a timestamp while the `sno` contains an encrypted timestamp and random value.

With this cached information, a client is able to establish an encrypted connection with the server. It first ensures that the `scfg` is correctly signed and that the server's certificate is valid and then sends a `c_hello` indicating the `scfg` its using, any `stk` and `sno` values it has cached, a Diffie Hellman share for the client, and a client nonce. After sending the `c_hello` message the client can create an initial encryption key and send additional encrypted data packets. In fact, to take advantage of the 0-RTT connection establishment it must do so. When the server receives the `c_hello` message, it validates the `stk`, `sno`, and client nonce parameters and creates the same encryption key using the Diffie Hellman share corresponding to the `scfg` and the client's share from the `c_hello` message.

At this point, both client and server have established the connection and setup encryption keys, and all further communication between the parties is encrypted. However, the connection is not forward secure yet, meaning that compromising the

server would compromise all previous communication. This is because the server's Diffie Hellman share is the same for all connections using the same `scfg`. To provide forward secrecy, the server sends an `s_hello` message containing a newly generated Diffie Hellman share after receiving the client's `c_hello` message. Once the client receives this message, client and server derive and begin using a new forward secure encryption key, providing forward secrecy for all data sent after the first RTT.

The security of QUIC, and particularly its new cryptographic handshake, have received significant attention in prior work [24, 25, 56]. [24] focuses on proving the security of QUIC's handshake while [56] provides a formal proof of the security for the protocol as a whole. In particular, the authors show that the protocol preserves the integrity and authenticity of data against an attacker who can initiate protocol connections, observe and modify target connections, and corrupt servers. Additionally, these proofs show that QUIC protects the server from spoofed connections. They note, however, that data sent under the initial encryption key (*i.e.,* 0-RTT data) is *not* forward secret and can be recovered if an attacker compromises the server while the `scfg` is still valid. Additional work [57] has pointed out that it is also possible to replay 0-RTT data to other servers. As a result, it is crucial that data transmitted in the first RTT be idempotent. QUIC prevents 0-RTT data from being replayed later in the same connection and even across connections to the same server. However, it does not protect the same request from being replayed to additional servers that implement the same application (*i.e.,* other servers in a server farm). The work in [25] demonstrates the importance of cross-protocol interactions by pointing out that exposing the same certificate over TLS 1.2 or below and QUIC enables an attacker to mount a Bleichenbacher-attack [58] to forge the signature on a fake `scfg`. This could potentially be used to launch a man-in-the-middle attack on QUIC. Such attacks can be prevented by using different certificates for QUIC and TLS 1.2 (or below) servers running on the same domain.

3 ATTACKER AND ATTACK MODEL

In this section we discuss the attacker capabilities and goals we consider in this work and differentiate between major classes of possible attacks.

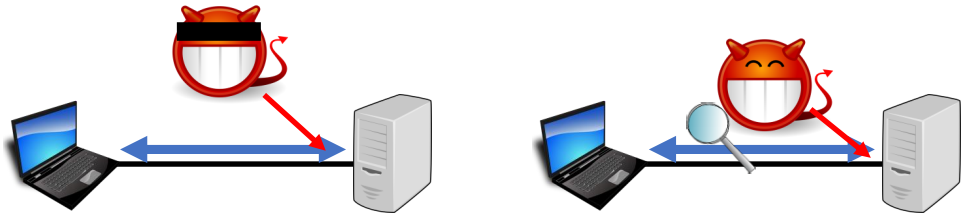
3.1 Attacker Capabilities

A malicious attacker targeting a transport protocol may have a variety of different capabilities for attacking the protocol. We can categorize these attacker capabilities based on the attacker's location in the network relative to the target connection or link: blind, off-path, on-path, or endpoint. We discuss each of these in turn and summarize them in Figure 3.1.

Blind Attackers. The blind attacker knows that some target connection or link exists and seeks to attack it without being able to see any of the target traffic. Such an attacker can inject spoofed packets into the network, but has no knowledge of detailed protocol state. Thus, the attacker has to guess this protocol state or rely on attacks that do not require it.

Off-path Attackers. An off-path attacker has the ability to both inject spoofed packets and observe packets in the target connection or link. This ability is usually obtained by sniffing traffic on the client's local network. By observing the target connection the attacker can gather detailed protocol state for use in injected packets and even race messages from the server.

On-path Attackers. An on-path attacker can modify and control delivery of legitimate packets in some target connection or link as well as inject new spoofed packets. Such an attacker is usually a switch on the path between client and server. Tampering with and injecting packets can be prevented using encryption and au-



(a) Blind Attacker

(b) Off-path Attacker

(c) On-path Attacker

(d) Endpoint Attacker

Figure 3.1.: Attacker Types

thentication of the protocol packets. However, that does not stop this attacker from controlling the delivery of legitimate packets.

Endpoint Attackers. An endpoint attacker is a malicious host who seeks to subvert the protocol. Such an attacker can modify and control delivery of packets arbitrarily as well as send additional packets. Encryption and authentication provide no protection here because the client has the keys needed to encrypt arbitrary packets. For unencrypted protocols, endpoint and on-path attackers are equivalent.

In this work, we consider attackers in all categories.

3.2 Attack Goals

In this work we focus on attacks that target the availability or performance of a transport protocol. We discuss these goals more below:

Compromise Availability (Denial of Service). Attacks that target availability seek to make a particular network service offered over some transport protocol unavailable to its users. This may either deny access to a victim host globally or

be focused only on a particular target connection. These attacks are often referred to as denial of service attacks; however, there are a variety of ways an attacker may go about achieving this goal beyond merely sending more traffic than the victim can process.

Attacks that seek to deny access to the victim globally are likely to focus on overloading or crashing the victim. This can be done by simply overwhelming the victim with traffic or by exploiting a bug in the victim's transport protocol implementation to crash the victim. The attacker may also attempt to manipulate the protocol to cause it to improperly release resources, causing a resource exhaustion attack.

Attacks that target a particular connection are much more focused and precise. The attacker is free to target either end of the connection and can monitor and seek to disrupt the target connection with forged replies or other invalid responses. This would typically cause the target connection to stall or be aborted.

Compromise Performance. Attacks that target performance seek to manipulate the transport protocol to compromise fairness with competing flows. This may be either to increase or decrease the throughput of some target flow. Decreasing the throughput of a target flow can have significant impact at the application level, especially for non-elastic data streams like real-time streaming video, while increasing throughput enables an attacker to artificially increase the share of some target connection beyond what would be considered fair and may be used to increase the damage done by a denial of service attack. Attacks seeking to compromise performance are likely to focus on the protocol's congestion control algorithm.

Attacks that manipulate data or impersonate hosts are not considered in this work. Additionally, while we may identify crashes that compromise availability, we consider further analysis of the exploitability of those crashes as out of scope.

3.3 Attack Characteristics

We differentiate transport protocol attacks based on a couple of key characteristics.

Protocol Specific vs Generic. Attacks on transport protocols can be either generic, applying to all transport protocols, or specific to a particular transport protocol. Dropping all packets in a connection is an example of a generic attack that prevents any transport protocol from establishing a connection. While these attacks are effective, they are also fairly course grained and expected for any network protocol. In contrast, protocol specific attacks rely on specific modifications to a particular protocol, induce much more subtle and unexpected failures, and are harder to find. In this work we focus on protocol specific attacks.

Stateful vs Stateless. Attacks on transport protocols can be either stateful, requiring the attacker to maintain state about the protocol connection, or stateless. Stateful attacks are much harder to execute since they require maintenance and usage of the right state information at the right time; however, they are also usually much harder to identify due to the much larger possible attack space. While we consider both stateless and stateful attacks in this work, we are particularly interested in stateful attacks.

4 AUTOMATED ATTACK DISCOVERY FOR TRANSPORT PROTOCOL CONNECTIONS

Even manually identifying attacks on the performance or availability of transport protocols is a complex task. Attempting to automate this process posses a number of challenges, especially around detecting attacks, search space exploration, and protocol information extraction. In this chapter, we investigate how to automatically identify a broad variety of attacks on transport protocol availability and performance without modifying or making assumptions about the protocol implementation.

4.1 Introduction

Transport protocols provide end-to-end communication in a layered network architecture by implementing guarantees such as reliability, in-order delivery, and congestion control. They are used not only directly by applications, but also by Internet services such as BGP and secure protocols such as TLS.

Providing these guarantees causes the design and implementation of transport protocols to be complex, with many components, special cases, error conditions, and interacting features. Further, many implementations are written in low level languages like C for improved performance and make use of error-prone, but highly efficient, constructs like pointer manipulation and type casting. Unfortunately, transport protocol implementations often sacrifice simplicity and ease of understanding for improved performance, resulting in a high probability of bugs introduced during implementation.

Due to the ubiquitous role of transport protocols in network communication, there are many different implementations of these protocols. For example, the `nmap` security scanner is able to detect 5,336 distinct TCP/IP network stack configurations in its

most recent version [1]. This includes printers, VoIP phones, routers, and embedded systems, along with general purpose operating systems. While many of these may be different configurations of a few common networking stacks, these variations represent different handling of particular network conditions, which often implies the exercise of different code paths.

Despite the importance of these protocols and the complexity and number of their implementations, the testing of transport protocol implementations has been mainly a manual and ad-hoc process [2–4]. This lack of systematic testing for transport protocols and their implementations has resulted in a stream of new bugs and attacks [2, 3, 5, 6]. Consider TCP, one of the most well studied and well tested network protocols; the list of discovered attacks extends from the mid-1980's to the present day [7–13]. Many of these attacks have been discovered repeatedly or rediscovered again in slightly different contexts.

Prior work in testing network protocol implementations has focused on easing the development of manual tests [2, 14] and on enabling deeper testing for crashes by using stateful fuzzing techniques [15–17]. Other work has focused on systematic testing by leveraging techniques like symbolic execution [5, 18] and dynamic interface reduction [19] in combination with concrete attack execution. Many of these techniques require access to the source code and require heuristics to efficiently handle low level constructs like type casting, pointer casting, and function pointers, which are heavily used in network protocol implementations. The major challenge faced by all of these approaches is search space explosion.

In this chapter, we focus on automated attack finding for transport protocol implementations. Specifically, we leverage information about packet formats and the protocol's connection-level state machine to automatically create attack scenarios consisting of malicious actions performed on protocol packets in targeted protocol states. Knowledge of the packet formats enables the generation of malicious packets based on packet type while information about the protocol's connection state machine allows the tracking of the current state of the protocol at runtime. State tracking

is achieved without code instrumentation by monitoring packets sent and received while malicious packet manipulation is achieved using a network proxy. By inferring the current state of the protocol's connection state machine, our method can perform malicious actions on all packets of a particular type in a particular protocol state instead of on individual packets, significantly reducing the search space. The protocol's connection state machine also allows us to identify key points for attack injection in the transport protocol, ensuring wide coverage.

The connection state machine and packet formats are an important part of any protocol specification. Unfortunately, these protocol specifications are usually informal documents written in natural language text, not formal specifications. We, therefore, investigate whether we can leverage Natural Language Processing (NLP) to automatically extract these protocol packet formats and rules (*i.e.,* a protocol grammar) from the natural language specification documents and use these automatically extracted grammars to improve attack finding.

Given the inherent ambiguity of natural language text, extracting a protocol grammar is not a straight-forward task. The writers of protocol specifications often rely on the human reader's understanding of context and intent, making it difficult to specify a set of rules to extract information. This is by no means unique to the computer networks domain, and as a result, the natural language community has shifted its focus over the last decade to statistical methods that can help deal with this ambiguity. Specifically, most NLP methods are sensitive to the data used at training time and do not adapt easily if applied on data from a different domain. Applying "off-the-shelf" implementations of NLP tools, typically trained on newswire data, or combining them in an ad-hoc way, often results in reduced performance and brittle applications. We, therefore, design an NLP framework to extract a network protocol's grammar from its protocol specification document by exploiting the structure and linguistic regularities of documents and leveraging zero-shot learning to enable adapting to new protocols.

Our testing approach works with *unmodified* implementations irrespective of their operating system, programming language, or required libraries. It does not require access to the source code, enabling the testing of a wide range of transport protocol implementations, including proprietary, closed-source systems.

The contributions of this chapter are:

- We present a new approach to search space reduction without instrumenting the code. This approach leverages a description of the protocol's connection state machine to identify critical points in the search space for attack injection and to explore the implementation more thoroughly. We use knowledge of the protocol's packet formats to perform a variety of malicious actions, including packet field manipulation, and apply these malicious actions to packet type, protocol state pairs instead of individual packets, enabling significant state space reduction. We also use the protocol state machine to ensure that we test all protocol states, providing wide coverage.

- We demonstrate our approach with SNAKE, a new tool for finding attacks on unmodified transport layer protocol implementations running in arbitrary operating systems and in realistic networks. SNAKE (State-based Network AttacK Explorer) uses virtualization to run unmodified transport layer implementations in their intended environments and a network emulator to tie these virtual machines together into a realistic, emulated network. The network emulator intercepts and modifies packets, tracks the current protocol state during execution, and uses this information to create packet-based attacks at specific points in the state machine execution. SNAKE is general for use on many transport protocols, requiring only a description of the packet header formats (*i.e.,* the protocol's grammar) and the connection state machine as input.

- We use SNAKE to examine a total of 5 implementations, 2 transport protocols—TCP [34] and DCCP [45]—, and 4 operating systems. We find 9 classes of attacks, 5 of which are, to the best of our knowledge, unknown in the literature.

We also compare our state-based attack search with two baseline approaches and show its effectiveness in search space reduction.

- We define the problem of protocol grammar extraction as a set of NLP tasks. By grammar, we mean protocol header fields and their properties, as well as relations between these fields. We define the learning of protocol fields as an *entity recognition* problem and the learning of properties and relations between fields as a *relation extraction* problem.

- We design an NLP framework to solve these tasks. We minimize the manual supervision effort required for training our NLP framework by exploiting the structure and linguistic regularities of the protocol specification document domain. Unlike previous work that applied transformation rules to the output of NLP tools directly, we propose a lightweight zero-shot learning framework which can adapt to the specific properties of the networking domain. Specifically, we learn a similarity function between textual phrases and protocol fields and relations. The similarity function captures the surface level string similarity, acronyms used in the text to refer to the fields, and anaphoric references ("it","that field") based on their context. This approach allows us to adapt to new protocols by providing different sets of entities. We evaluate the quality of the zero-shot learning process by training it on one set of protocol symbols and testing it on a different set (we use RFCs for GRE [59], IPv6 [60], IP [61], TCP [34], UDP [31], DCCP [45], and SCTP [62]).

- We demonstrate the usefulness of the information extracted by our NLP framework by applying it to SNAKE. We compare three settings: Random, Manual, and NLP-based, where the input packets were generated without protocol semantics knowledge, with manual specification, and with NLP-extracted protocol semantics, respectively. We find that our automatically generated protocol grammars are as effective in identifying attacks as manually created grammars while enabling improved efficiency.

The rest of this chapter is organized as follows. Sections 4.2 and 4.3 present the design and the implementation of SNAKE, respectively. Section 4.4 shows our results, including the attacks we discovered. Section 4.5 discusses our work to further automate SNAKE by leveraging Natural Language Processing to extract information from protocol specification documents automatically. Finally, Section 4.6 summarizes this chapter.

4.2 Design

In this section, we discuss the design of SNAKE. We first provide an overview of our approach, then describe how we utilize the protocol's connection state machine to reduce the search space and generate attack strategies. Finally, we describe the packet-level basic attacks we consider.

4.2.1 Overview

We focus on finding attacks by endpoint or blind attackers on unmodified implementations of transport protocols. We consider availability attacks that target connection establishment as well as resource exhaustion attacks. Additionally, we consider performance attacks resulting in throughput degradation or the compromise of fairness. These attacks can be identified by examining the results of an attempted data transfer. Specifically, connection establishment attacks can be identified by observing a target connection that transfers no data. Resource exhaustion attacks result in incomplete socket cleanup at the server. Throughput degradation attacks and attacks on fairness can be identified by unfair competition between a target connection and its competitor; throughput degradation attacks target the low throughput connection while attacks on fairness target the high throughput connection. All of these attacks can be detected by running the protocol for a relatively short period of time.

We select an environment that combines virtualization with network emulation. Virtualization allows us to test a wide range of implementations independent of lan-

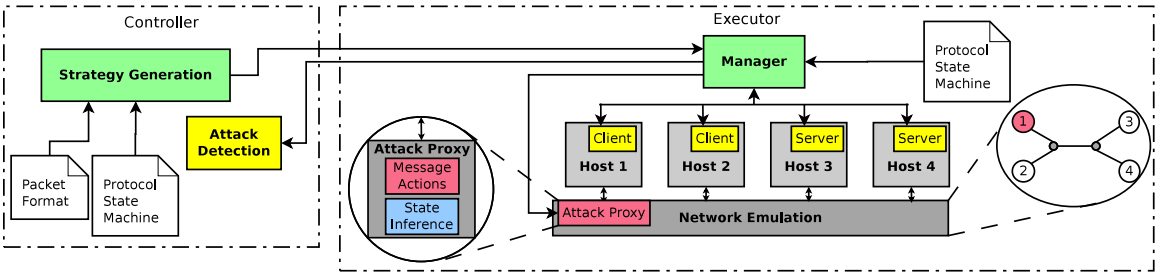


Figure 4.1.: Design of SNAKE

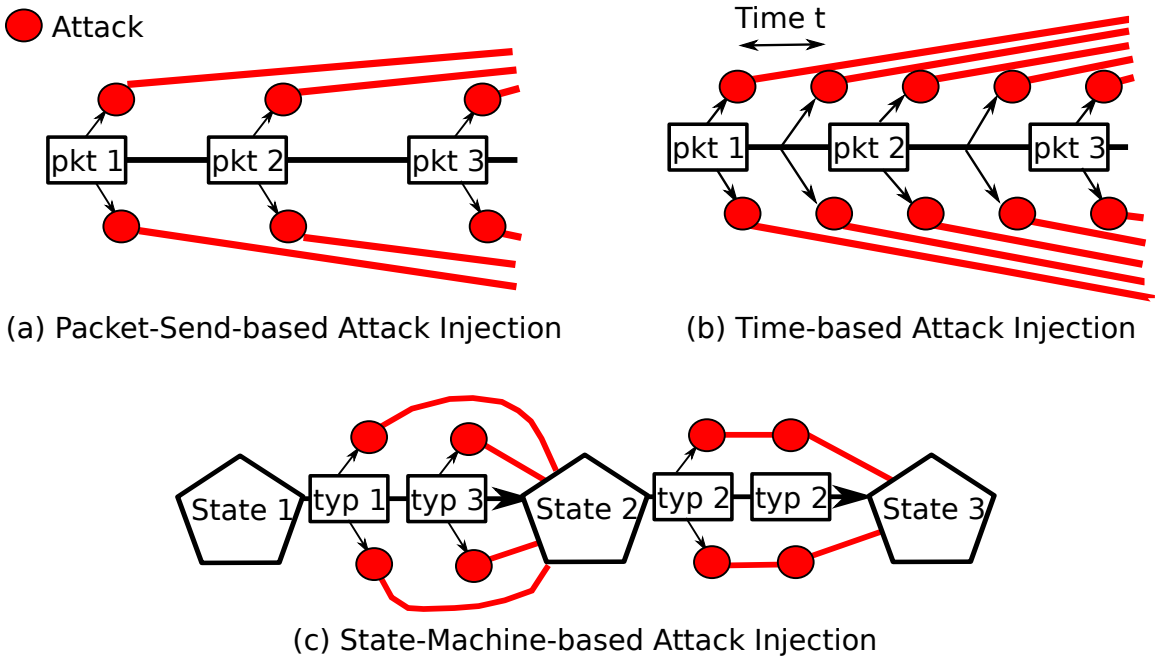
guage, operating system, or access to source code. The network emulation provides us the reproducible measurements and attack isolation needed to detect performance-related attacks. Figure 4.1 presents our system design.

The attack strategies we consider can be created by packet manipulation and injection based on the packet type and the individual packet fields. These strategies are selected from a set of basic attacks derived from information about packet formats. For instance, an attack strategy may be to duplicate packets of type W ten times, or to inject a new packet of type X with field 3 set to Y, or to modify field 5 of packet type Z to 555. Each of these attack strategies are performed in particular protocol states.

To determine what kinds of basic attacks would be most useful, we performed a detailed literature study on transport protocol attacks and identified some common components, or building blocks, used in many of these attacks. Based on this study, we defined a set of packet-based basic attacks that we use to compose attack strategies.

As we do not require access to the source code, our approach relies on intercepting and modifying or injecting network traffic. We place an attack proxy between one of our test hosts and the emulated network. This proxy emulates an on-path or endpoint attacker and intercepts packets to apply basic attacks such as influencing the delivery of packets or modifying the packets flowing through it. We also use the proxy to emulate a blind attacker who injects new packets into the network.

We detect if an attack was successful or not by comparing the connection performance under attack with a baseline generated from a test with no attacks and by



(a) Packet-Send-based Attack Injection

(b) Time-based Attack Injection

(c) State-Machine-based Attack Injection

Figure 4.2.: Attack Injection Algorithms

checking for open sockets on the server after the test completes. Attack strategies that appear successful are tested a second time to ensure repeatability.

4.2.2 Attack Injection

An important aspect of determining an attack search strategy is identifying the attack injection points, that is, the points where attacks can be inserted into a test run.

Send-packet-based attack injection. One simple approach is to have the proxy intercept each packet generated by the client application running in the virtual machine, apply any basic attacks desired, and forward the packet on to its destination. This means that an attack injection point occurs whenever there is a send for a particular packet type, as shown in Figure 4.2(a).

While this approach is relatively simple and can find many attacks, it also results in repeatedly performing attacks that have the same semantics for the protocol, thus

resulting in redundant executions and lengthening the time required to complete the search. In addition, this approach does not work well for blind attackers and fails to find attacks not connected with packet send events in the code. This is particularly problematic for transport protocols because many availability attacks against connection establishment and tear down fall into this category.

Time-based attack injection. One approach to provide support for blind attackers and finer time granularity is to divide the running time into fixed intervals and, for each of these intervals, attempt to inject packets following all basic attacks, as shown in Figure 4.2(b). While this approach is also relatively simple, a small time interval must be used in order to catch many attacks. This will result in testing thousands of strategies that either do not inject attacks or inject many redundant attacks, based on the semantics of the protocol. As a result, this approach also has a high execution time overhead and can take a very long time to complete. Recall that some of our attacks are packet manipulation attacks designed to simulate endpoint attackers. These attacks can only be inserted on a packet send. Nevertheless, this attack injection strategy will attempt to insert them at very small increments throughout the entire test, whether or not a packet send occurs at that point. Time-based attack injection is also overly fine-grained. It will attempt to inject an attack at every possible point in time. However, practical attacks are likely to have much broader timing constraints—on the order of an RTT or between packet sends. Thus the time-based scheme will test numerous attack scenarios that are practically identical.

Protocol state aware attack injection. Our approach to eliminate some of the redundant testing scenarios, support blind attackers, and provide finer granularity for injecting attacks is to take into account the semantics of the protocol when injecting attacks. We can obtain information about the semantics of the protocol from its connection state machine. Many transport protocols have well documented state machines describing their connection lifecycles, and in the absence of such documentation, work in state machine inference may be leveraged [63].

We propose a state-based search strategy that leverages several characteristics of the protocol's connection state machine to reduce the attack search space. Specifically, we inject attacks at specific states in the protocol execution, as shown in Figure 4.2(c). Because the protocol's connection state machine defines key points in the operation of the protocol, this approach allows us to quickly gain wide coverage within the search space by focusing on each of these states. We also treat all attack injection points in the same state in the same manner. This further prunes the number of search paths to be explored. The motivation behind our approach is that two packets of the same type received in the same protocol state usually cause similar results; however, an identical packet received in two different states may cause significantly different results.

In order to apply our *protocol state aware attack injection*, we need a mechanism to infer which state a protocol connection is in. As we do not require access to the source code, we use packet monitoring to infer this state. This is accomplished by a state tracking component (see Figure 4.1) that uses a description of the protocol's connection state machine supplied by the user. This state machine provides information about what packets determine transitions from one state to another. At run time, the state machine tracker infers changes in the connection state machines of each endpoint by observing the packets exchanged and matching them with state transition rules, as shown in Algorithm 1. The state tracking component also keeps track of some basic information about each observed state, including the packet types observed in that state.

Note that this strategy assumes that implementations have correctly implemented the protocol's connection state machine as described in their specification. Existing work on state machine verification [64] could be leveraged to overcome this limitation. However, connection state machines are unlikely to be implemented incorrectly because of their simplicity, high granularity, and importance to the protocols. Taking TCP as an example, the state machine has 11 states in total and all data transfer, and associated retransmissions and congestion control, takes place in a single state

(see Figure 2.2). A mistake in this state machine has a similar impact to getting the packet header formats wrong; while the implementation may work with itself, it will fail simple interoperability tests.

Algorithm 1: SNAKE Connection State Tracking

Input: Connection State Machine Graph $G = (V, E)$ where all $e \in E$ contains $e.recv$ and $e.send$, the packets sent and received during the transition

Output: $h.CurrentState$ variable indicating the current connection state, as seen on host h

```
1  Function Init(h)
2      if ht is Server then
3          h.CurrentState = LISTEN
4      else
5          h.CurrentState = CLOSED
6      return
7  Function OnPacket(p,h,E)
8      foreach e ∈ G.E do
9          if e.from == h.CurrentState then
10             if e.recv == p.type or e.send == p.type then
11                 h.CurrentState = e.to
12                 return
13     return
```

4.2.3 Attack Strategy Generation

Based on the packet types and connection state machine information, we automatically generate attack strategies. For each packet type we generate the basic attacks described below.

We conducted an extensive study of the literature on transport protocol attacks to develop these basic attacks. All of these attacks are conducted by our attack proxy at a packet level, either one packet at a time or considering several packets together.

Malicious Endpoint attacks. The first set of basic attacks we developed interfere with packet delivery or packet content. Packet delivery attacks model a malicious

client who either ignores certain packets entirely or who delays processing packets in order to interfere with the protocol. Packet content attacks model a malicious client who sends packets that contain unexpected or invalid values.

We consider the following *packet delivery attacks*: drop, duplicate, delay, and batch.

Drop: The attack proxy intercepts and drops a packet with a given probability specified as a parameter in percent. This attack may impact many of the core features of transport protocols from connection establishment to connection tear down, depending on when it is applied.

Duplicate: The attack proxy intercepts a packet and then sends multiple copies of it to the destination. The number of duplicates to inject is specified as a parameter. This attack could impact many features of a transport protocol, but fairness and congestion control are particularly vulnerable. Acknowledgment duplication, in particular, can cause fairness problems [11].

Delay: The attack proxy intercepts a packet and then inserts a delay before sending it on. The delay is specified as a parameter in seconds. Depending on the length of the delay, this attack may cause reordering or retransmission situations. It may also interfere with RTT estimation, which is usually a key component of retransmission algorithms.

Batch: The attack proxy intercepts packets and waits some amount of time before sending them all at once. The wait time is a parameter specified in seconds. This attack is designed to find attacks similar to the Shrew and Induced-Shrew attacks [8, 9].

We also consider the following *packet content manipulation attacks*: reflect and lie.

Reflect: The attack proxy intercepts a packet and sends it back to its originating host. This attack models sending an unexpected, but potentially valid, packet. It is particularly likely to disrupt connection establishment and termination. Consider,

for example, the TCP Simultaneous Open Attack where an attacker responds to a SYN packet with another SYN packet [7].

Lie: The attack proxy intercepts a packet and modifies a specified field before sending it on. Modifications supported include setting particular values, setting random values, or adding/subtracting/multiplying/dividing the current value by some factor. The field and the type of modification are parameters. We use a list of modifications chosen based on the field-type to be likely to cause unexpected behavior. These include setting values like 0, the maximum value a field can handle, and the minimum value a field can handle. This attack may impact all of the core features of transport protocols from connection establishment to connection tear down, depending on when and where it is applied.

Blind attacks. The second set of attacks we developed are attacks on a connection by a blind attacker. These attacks spoof packets such that they appear to come from the client or the server in a target connection. We consider the following blind attacks: inject and hitseqwindow.

Inject: The attack proxy injects a new packet into the network. This attack contains a number of parameters describing the fields in the packet, its source and destination, and when it should be injected (in seconds from emulation start). Many parts of a transport protocol may be affected by such an attack, from reliability to connection tear down.

HitSeqWindow: This attack is very similar to inject. Instead of injecting just one packet, the attack proxy injects a whole series of packets with their sequence numbers spanning the whole possible sequence range. This attack is designed to look for attacks similar to the Reset and Syn-Reset attacks on TCP [3, 12].

Note that one can also consider more complex attack strategies that combine the basic attacks described above into strategies consisting of sequences of actions. We currently support only the basic attacks described above.

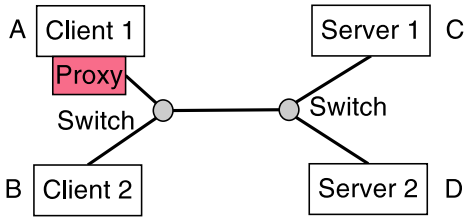


Figure 4.3.: SNAKE Test Network Topology

4.3 Implementation

In this section, we discuss how we implement SNAKE. We first present an overview of the whole platform and then discuss our attack proxy, state tracking, and parallelism in more detail. See also Figure 4.1.

4.3.1 Overview

We separate the functionality of SNAKE into two components: a *controller* that generates attack strategies and one or more *executors* that test the strategies.

The controller generates and selects the attack strategies based on the packet formats and the connection state machine transitions obtained from the protocol specification supplied by the user. An executor first runs a non-attack test and then, for each strategy, runs the attack scenario and reports performance information back to the executor, who determines whether an attack took place or not. SNAKE uses parallelism to run multiple executors concurrently and speed up the attack finding process.

The executor controls the execution of a testing scenario consisting of a set of four virtual machines each running an unmodified instance of the protocol under test. These virtual machines are connected in a dumbbell topology using a network emulator and tap devices. We use KVM as the virtualization environment and NS-3 for network emulation.

A dumbbell topology consists of two machines on each side of a bottleneck link as shown in Figure 4.3. In our setup, the two machines on one side act as servers while the two on the other act as clients. We configured our attack proxy to be between one of the clients and the bottleneck link. The other client makes a connection to a server that we refer to as the *competing connection*, as it will compete with the connection through our proxy for bandwidth on the bottleneck link. This topology allows us to test attacks from both endpoint and blind attackers. An endpoint attacker is usually interested in targeting the fairness of the transport protocol or launching a resource-exhaustion-based denial of service attack against the server while a blind attacker often wishes to terminate or slow a connection between two other hosts. See Section 3 for a more detailed discussion of these types of attackers and attacks.

To determine successful attacks, the controller examines the performance of the client without the attack proxy (Client 2 in Figure 4.3) and the number of connections the server is maintaining at the end of the test. This information is obtained by the executor. Specifically, the executor calculates performance as the quantity of data transferred during the test and queries the OS to determine the number of connections maintained by the server, for example, by using the `netstat` command on UNIX-based systems. After the test completes, the executor sends these metrics to the controller, which compares the received metrics with metrics observed in a non-attack test run.

The executor is implemented as a Perl script that listens for strategies from the controller and then initializes the virtual machines from snapshots, starts the network emulator, configures the attack proxy, and starts the test. Once the test completes, it collects the performance data and any feedback from the attack proxy and sends this back to the controller.

The controller is implemented in a combination of C and Perl and is responsible for choosing strategies to execute and determining attacks based on the performance data returned by the executor. Instead of generating all of the attack strategies at once, we implement our controller to generate them a few at a time in response

to feedback about packet types and protocol states observed by the state tracking component of our attack proxy. This is equivalent to generating all the strategies at once but is a little more flexible.

4.3.2 Attack Proxy

Our attack proxy intercepts all packets along the ingress and egress paths in NS-3. We modify NS-3 to allow us to designate malicious nodes and only intercept packets to or from those nodes. The interception is done in NS-3's tap-bridge module, which connects NS-3 to outside tap-devices serving the virtual machines.

When the attack proxy receives a packet, it examines it to determine the protocol. Protocols not of interest are returned to the tap-bridge for normal processing. For packets of the target protocol, the type of the packet is examined and the sender's protocol connection state is identified from the state tracking system. If there is a matching strategy, the basic attack is performed on the packet. To accomplish this, our proxy needs a description of the protocol's grammar or packet header format. We use a simple language to describe this grammar and then automatically generate C++ code to parse and modify the header.

Our malicious proxy is also capable of injecting packets into the network. Proper packet headers are generated from the protocol grammar as part of our automatically generated C++ protocol processing code, and the resulting packets can then be sent using standard NS-3 packet send mechanisms.

4.3.3 State Tracking

We implement our protocol connection state machine tracking inside the attack proxy. The tracker takes a description of the protocol's connection state machine, written in the `dot` language [65], as input. This description contains the state transitions, including the packets or actions that cause these transitions or result from them. The use of a standardized graph language like `dot` to represent this state

machine enables the use of SNAKE on a variety of two-party protocols simply by swapping out the connection state machine and packet header descriptions.

Our state machine tracker watches the packets that pass through the proxy and uses the state machine transition rules to infer what connection state the client and server are currently in. The state machine tracker also collects some useful statistics about each state in the protocol. This includes what packet types and how many packets were sent and received during each state. It also includes the amount of time each host spent in each state and the number of times it visited that state. These statistics are extracted from the attack proxy by the executor at the end of each test and then sent to the controller along with the performance information.

4.3.4 Parallelism

We have implemented SNAKE as separate controller and executor modules to enable parallelism. These modules can even reside on separate systems, as all communication is done via TCP. Because testing each strategy takes about two minutes this becomes a highly parallel problem, with linear speedup limited only by the amount of processing power that can be thrown at the problem.

Each executor requires significant resources, as it will start four virtual machines and an NS-3 instance. In practice, we found that running about one executor for every six hyperthreads resulted in good performance. The memory requirements per executor depend primarily on the demands of the implementation and operating system under test. In our tests, they ranged around 4-8GB per executor.

Our controller requires little processing power since its primary responsibility is to identify attacks based on the performance information returned by the executors and to supply new attack strategies to the executors. In our experiments, we did not find it necessary to dedicate a core to the controller.

Table 4.1.: Summary of SNAKE Results

Proto	Impl	Strats Tried	Attack Strats Found	On-path Attacks	False Positives	True Attacks	Attack Classes
TCP	Linux 3.0.0	5994	128	82	5	41	4
TCP	Linux 3.13	5717	163	105	10	48	3
TCP	Windows 8.1	5549	137	118	2	17	4
TCP	Windows 95	5013	147	122	3	22	3
DCCP	Linux 3.13	4508	67	27	2	38	3

4.4 Results

We applied SNAKE to test two protocols and a total of five transport protocol implementations on four different operating systems. The two protocols we tested were TCP and DCCP. For TCP, we tested implementations in Linux 3.0.0, Linux 3.13, Windows 8.1, and Windows 95. For DCCP, we focused on the implementation in Linux 3.13. We were able to find attacks on all implementations, including several previously unknown attacks. We discuss these protocols and present our findings below, and summarize them in Tables 4.1 and 4.2.

All of these tests were run on a hyperthreaded 16 core Intel® Xeon® 2.3GHz system with 94GB of RAM. We ran five separate executors simultaneously. Testing each implementation required about 60 hours, but this duration could be decreased by running more executors.

We define successful attacks as strategies that result in an increase or decrease in achieved throughput of at least 50% compared to the non-attack case or that cause the server-side socket to not be released normally after the connection is closed. This throughput threshold is based roughly on the notion that reasonable competition for network flows is achieving throughput within a factor of two of each other [32, 33] as well as on experience.

Table 4.2.: Classes of Attacks Discovered by SNAKE

Proto	Attack	Description	Impact	Operating System	New
TCP	CLOSE_WAIT Resource Exhaustion	Connections hang on server if client exits and resets are dropped	Server DoS	Linux 3.0.0 Linux 3.13	Partially [66]
TCP	Packets with Invalid Flags	The handling of invalid flag combinations could allow OS fingerprinting	Finger-printing	Linux 3.0.0 Win 8.1	Yes
TCP	Duplicate Acknowledgment Spoofing	Frequently duplicating acknowledgments causes sender to increase window faster than normal	Poor Fairness	Win 95	No [11]
TCP	Reset Attack	Brute force a sequence-valid reset	Client DoS	All	No [13]
TCP	SYN-Reset Attack	A sequence-valid SYN causes connection reset	Client DoS	All	No [3]
TCP	Duplicate Acknowledgment Rate Limiting	Occasionally duplicating acknowledgments result in indicated loss and connection slow down	Throughput Degradation	Win 8.1	Yes
DCCP	Acknowledgment Mung Resource Exhaustion	Connection will hang waiting for timeouts to empty send queue if acknowledgments are disrupted	Server DoS	Linux 3.13	Yes
DCCP	In-window Acknowledgment Sequence Number Modification	Connection can be throttled by incrementing sequence number in an acknowledgment, resulting in a forced resync	Throughput Degradation	Linux 3.13	Yes
DCCP	REQUEST Connection Termination	Any packet except Response received in REQUEST state results in connection reset	Client DoS	Linux 3.13	Yes

4.4.1 TCP

We tested TCP in one of its most popular settings, HTTP. Specifically, we utilized a large HTTP download with Apache or IIS running on the servers and `wget` for clients.

For each of our TCP implementations, SNAKE tried between five and six thousand strategies and determined that between 128 and 163 of these (depending on the implementation) resulted in significant performance degradation or potential for resource exhaustion. These attack strategies represent around 3% of the tested strategies.

On-path attacks. Some of the attacks we found, while possible, require an on-path attacker. Strategies like modifying the source or destination ports or the header size do prevent a connection from being established, but these strategies are not possible for blind attackers and an endpoint attacker could simply not initiate a connection. These attacks can be conducted by an on-path attacker. However, as TCP was not designed to handle such attackers, we are not interested in these types of attacks.

False positives. We found a few attacks that were false positive strategies for each implementation. These were related to the *hitseqwindow* basic attack. This attack injects numerous packets in an attempt to get one packet into the sequence window of a target connection. Unfortunately, the injection of such a large number of packets tends to slow down the target connection significantly, irrespective of whether the packets have any malicious impact. We manually inspect the packet captures for attacks using this action to determine why an attack was declared and identify false positives when the reduced performance is caused by the number of packets injected, and not by hitting the target sequence window.

Endpoint and blind attacks. Discarding the false positive and on-path attacks results in a set of between 17 and 48 (depending on implementation) attack strategies. However, many of these strategies are functionally the same attack, just performed on a different field or with a different value. Ultimately, we found a total of six unique

classes of attacks, several of which are effective against multiple implementations. We discuss each of these classes of attacks in detail below.

CLOSE_WAIT Resource Exhaustion Attack (partially known). This class of attack results in connections staying alive on the server in the `CLOSE_WAIT` state for tens of minutes after the client closes them. An attacker can easily initiate hundreds of thousands of such connections before they begin to expire, likely rendering the server unavailable.

`CLOSE_WAIT` is the TCP state that the passive close side of a TCP connection, usually the server, remains in after receiving notification of remote close and while waiting for the local application to close the connection. After the local close, the connection must remain in this state until a `FIN` can be sent.

If a Linux TCP client exits while in the middle of a data transfer (like an HTTP download), Linux will send a `FIN` packet and then not acknowledge any more data on the connection; any further packets will generate a reset. This is valid behavior according to the RFC since the application will never receive this data [34]. If these reset packets are blocked, it will appear to the sending TCP that the whole in-flight window of packets was lost, triggering congestion avoidance and a series of retransmissions that will never succeed.

When the server application eventually closes the TCP connection, TCP will transition to the `CLOSE_WAIT` state where it needs to remain until all outstanding data is acknowledged, including the lost window of packets that were in-flight when the client exited. These packets will never be acknowledged, meaning that TCP is stuck in `CLOSE_WAIT` with (possibly significant) data queued on the socket. Linux will eventually force-close a TCP connection due to lack of delivery, but that requires 15 retries by default, which is between 13 and 30 minutes depending on the RTT [67].

To the best of our knowledge, this attack class is unreported in the research literature. However, system administrators have been aware of similar problems with connections stuck in `CLOSE_WAIT` for many years [66]. SNAKE found this attack on Linux 3.0.0 and Linux 3.13.

Packets with Invalid Flags (new). Recall that the TCP header includes several flags that indicate the packet type. Not all combinations of these flags make sense. For instance, a packet with `SYN+FIN+ACK+RST` flags would indicate a packet starting a connection, closing the connection, acknowledging a packet in the connection, and resetting the connection. This is clearly a nonsensical combination. One would expect a TCP implementation to ignore such invalid packets. However, both Linux 3.0.0 and Windows 8.1 respond to such invalid packets in an active connection.

Linux 3.0.0 attempts to interpret these nonsensical flag combinations as best it can. This results in sending a duplicate acknowledgment in response to a packet with no flags set, a situation that is never valid. We have also observed Linux 3.0.0 attempting to process `SYN+FIN` and `SYN+FIN+ACK+PSH` packets. Note that Linux 3.13 appears to have fixed these problems and no longer responds to such invalid packets.

Windows 8.1 will also process and respond to invalid packets. However, it follows a different approach. If the `RST` flag is set, the connection is reset irrespective of what other flags might also be set. Otherwise, nonsensical flag combinations are ignored.

Responding to packets with invalid flag combinations is not by itself a security issue. We have found no instance where responding to invalid flag combinations achieves something that is not possible with valid flag combinations. However, a target's responses to invalid flag combinations could be used to fingerprint the particular TCP implementation in use, indicating other possible vulnerabilities to exploit. Further, packets with invalid flag combinations may be interpreted differently by end hosts and middleboxes like firewalls and intrusion detection systems, providing a possible way to subvert such middleboxes.

Duplicate Acknowledgment Spoofing (known). This is a classic class of TCP attacks originally discovered by Savage, et al. in 1999 [11]. These attacks operate against naïve TCP implementations where the sender increases their congestion window for every acknowledgment received, without checking for duplicates or checking how much data is currently outstanding in the network. As a result, a receiver can significantly increase its achieved throughput by simply acknowledging packets

multiple times, thereby increasing the sender's congestion window much faster than normal.

These attacks require frequent duplication of acknowledgments to be meaningful, as each acknowledgment only increases the congestion window by a very small amount. In addition, if acknowledgments are duplicated more than three times, TCP will react as if a loss occurred, halve its congestion window, and enter fast recovery. However, in this mode, each acknowledgment received results in a new packet being sent. This simplifies the attack by allowing the attacker to control the sending rate by controlling the acknowledgment rate.

There are mitigations to these attacks, including only allowing the congestion window to be incremented by the number of data segments outstanding in the network. Another option would be a nonce in the TCP header and a sender side register allowing acknowledgment of each nonce only once.

In our tests, SNAKE discovered this attack class against Windows 95 and was able to use it to increase a malicious connection's throughput by a factor of 5. SNAKE did not find this attack class against any other tested implementation, which is expected as this attack class and its mitigations were well known by the time they were released.

Reset Attack (known). This class of attack works by spoofing a large number of resets for a target connection. If one of these resets is sequence-valid, the receiving TCP will reset the connection. The work in [13] showed these attacks to be much more practical than previously supposed by pointing out that a reset packet anywhere in the receive window is sufficient to reset the connection. Thus, one could send packets at receive window intervals, greatly reducing the number of packets required.

In our testing, SNAKE discovered this attack class against all of our TCP implementations. Since these attacks utilize a feature of the TCP specification itself, all implementations should be vulnerable. The only thing implementations can do to protect themselves is to keep their receive window small.

SYN-Reset Attack (known). This attack class is very similar to the Reset Attack discussed above. In this case, the TCP specification says that the receipt of

a sequence-valid `SYN` packet on an active connection should result in the connection being reset. As a result, an attacker can spoof a large number of `SYN` packets at receive window intervals in an attempt to slip one into the target connection's sequence window, resulting in a connection reset. This attack has been known since at least 2009 [3].

In our testing, SNAKE discovered these attacks against all of our TCP implementations. Like the Reset Attack, this attack class utilizes a feature of the TCP specification itself, which makes it difficult for implementations to protect against.

Duplicate Acknowledgment Rate Limiting (new). Duplicate Acknowledgment Rate Limiting is a new class of attack that SNAKE discovered against Windows 8.1. It operates by duplicating `PSH+ACK` packets, which occur only occasionally in the data stream, ten times. This causes duplicate acknowledgments to be sent to the sender by the receiver. After three duplicate acknowledgments, the sender halves its congestion window and retransmits the indicated packet.

So far, this is standard TCP behavior common to all TCP New Reno implementations. However, for a Windows 8.1 server and a Linux 3.0.0 client, we observe a throughput degradation of a factor of 5 compared to the competing flow. Both of the Linux implementations we tested show throughput consistent with normal TCP competition in this scenario; that is, approximately fair bandwidth sharing.

4.4.2 DCCP

For DCCP testing, we used `iperf` to measure throughput. Since DCCP is not a reliable protocol, we measured performance based on server goodput, or actual data received. As DCCP is currently only supported on Linux and is fairly uncommon, we focused our efforts on a single implementation, the Linux kernel 3.13 implementation.

SNAKE tried just over 4,500 strategies against DCCP. Of these, it identified 67 candidate strategies that caused significant performance issues or potential resource exhaustion. This is about 1.5% of the total strategies tested.

On-path attacks. As with TCP, DCCP was not designed to be resilient to on-path attacks. Thus, we exclude all on-path attacks found by SNAKE.

False positives. We also found 2 attacks that were false positives. As with TCP, these attacks are both *hitseqwindow* strategies that attempt to inject packets into a target connection at sequence window intervals. Injecting this quantity of packets tends to significantly slow down the competing target connection, irrespective of any malicious impact of the injected packets. Thus, these strategies tend to fall below our attack threshold.

Endpoint and blind attacks. Discarding the on-path attacks and the two false positives leaves us with 38 strategies that represent actual attacks. However, many of these strategies are functionally the same attack, just repeated on different fields or with different values. Ultimately, we found three classes of attacks; none of which have been reported in the literature. We discuss each of these classes of attacks below.

Acknowledgment Mung Resource Exhaustion Attack (new). This class of attack is possible because a DCCP sender will not close a connection until its send queue is empty. This send queue defaults to 10 packets, but may be much larger for applications like video streaming. As a result, if a connection's congestion control can be persuaded to send at the minimum rate, a connection can be held in an open-but-useless state for a very long time. By repeating this process, one can create an effective resource exhaustion attack that may render the target host unavailable.

Note that DCCP does not retransmit data. As a result, while similar attacks against TCP last until TCP gives up retransmitting a particular packet and resets the connection, DCCP will continue sending at its minimum rate until the application and the human trying to use it explicitly close the connection. Once the application closes the connection, DCCP will send all queued packets and then close the connection and free related resources.

There are several ways to convince DCCP's congestion control to send at its minimum rate. Most of them work by invalidating or dropping the acknowledgments

from the receiver. Modifying the sequence or acknowledgment numbers are very effective because this results in an additional exchange of `SYNC` and `SYNCACK` packets.

In-window Acknowledgment Sequence Number Modification (new). This attack class targets sequence numbers in the receiver's acknowledgment packets. Recall that sequence numbers in DCCP are per-packet and that every packet increments the sequence number; even pure acknowledgment packets.

If the sequence number of one of these acknowledgments is increased, such that it is still sequence valid, the sender will begin to acknowledge this bad acknowledgment number in its data packets. However, when the receiver receives these data packets it will find they acknowledge packets that have not yet been sent. As a result, it will drop these packets and send a `SYNC` in response. The `SYNC` packet will result in a `SYNCACK` packet from the sender, resynchronizing the sequence numbers and allowing the connection to proceed. However, by that point an entire window of packets will have been dropped, resulting in DCCP's congestion control reducing the connection's allowed sending rate. It may even trigger a timeout and subsequent slow start, assuming DCCP's CCID 2 congestion control is in use.

To perform these attacks, an attacker does not have to be an endpoint. It suffices to be able to sniff and spoof network traffic (*i.e.,* an off-path attacker). Such an attacker can inject an acknowledgment with a slightly higher sequence number and trigger this vulnerability.

REQUEST Connection Termination Attack (new). This class of attack is an effective way to terminate a connection during the connection initiation phase. A client enters the `REQUEST` state on initiating a connection, immediately after having sent a `REQUEST` packet to the server, and stays in this state until it receives a `RESPONSE` packet from the server.

The only valid packets in the `REQUEST` state are `RESPONSE` or `RESET`; any other packet results in a reset. Note that both the pseudo-code in RFC 4340 [45] and the Linux 3.13 DCCP implementation perform this packet type check *before* checking the sequence numbers. Thus, it is possible to reset a DCCP connection in the `REQUEST`

state by sending *any* non-`RESPONSE` packet with *any* sequence and acknowledgment numbers.

This makes this attack class exploitable by anyone who can sniff and spoof packets (*i.e.,* an off-path attacker). A blind attacker can also launch this type of attack, if they can guess the connection initiation time (to within an RTT) and the source port.

4.4.3 Benefits of State-based Strategy Generation

Our state-based strategy generation algorithm enabled us to find 9 attacks against 2 transport protocols and a total of 5 implementations. 5 of these attacks were previously unknown. To accomplish this, we required about 60 hours per tested implementation. Removing parallelism, this becomes 300 hours of computation per tested implementation.

By contrast, the time-based attack injection approach discussed in Section 4.2.2 requires trying our malicious strategies at intervals of 5 microseconds, which is roughly the amount of time needed to send a minimum sized TCP packet at 100Mbits/sec. Thus, there are 12 million possible injection points in a 1 minute test connection. For each of these injection points, we would have to test about 60 different malicious strategies resulting from the 8 general malicious actions and the 13 fields in the TCP header. This results in 720 million strategies to test.

At 2 minutes to test each strategy, this would require 24 million hours of computation. At an equivalent level of parallelism, this would take 548 years to complete, which is clearly impractical.

The send-packet-based attack injection approach is more practical. A one minute non-attack test with TCP results in the sending of about 13,000 packets. For each of these packets, we would need to test about 53 different malicious strategies for packet manipulation, resulting in a total of 689,000 strategies. This would require 22,967 hours of computation. At an equivalent level of parallelism, this would take about 191 days.

However, send-packet-based attack injection provides no support for packet injection attacks modeling blind attackers. As a result, it would be impossible to find the Reset and Syn-Reset attacks using this attack injection model.

4.5 Natural Language Processing Pipeline

We now consider how to automatically extract the protocol descriptions that grammar-based fuzzers, like SNAKE, depend on to automatically find attacks. These descriptions encode the layout of packet fields and protocol semantics. For example, consider the TCP protocol (its header is shown in Figure 2.1) where bytes 17 and 18 contain a checksum of the rest of the TCP header. A test packet that contains a modified field and wants to test a particular part of the code must also contain the correct checksum in order to pass the trivial checksum check and reach the desired part of the code. Similarly, bit 6 of byte 14 (the `URG` field) controls whether bytes 19 and 20, the `urgent pointer` field, are interpreted or not. Hence, a test packet to test an `urgent pointer` value of 10, must set bytes 19 and 20 *and* set bit 6 of byte 14 to one.

Unfortunately, these protocol descriptions are usually created manually by an expert and are not easily transferable from one protocol to another. As a result, many grammar-based network protocol fuzzers, including SNAKE, suffer from limitations like: (1) time-consuming manual protocol definition, (2) difficulty adapting to new protocols, (3) poor test coverage, and (4) false positives that must be manually triaged. However, we observe that there is an untapped resource of information available for network protocols in the form of natural language specification documents, *e.g.,* RFCs. With the recent interest in using data to solve problems in several fields, we ask the question: "*Can we leverage natural language specifications of protocols to improve protocol fuzzers?*" It seems likely that the information that is manually extracted by human experts today could be extracted in an automated manner with the help of Natural Language Processing (NLP) tools.

Extracting protocol information from natural language text is not a straightforward task. Natural language text has inherent ambiguity and the writers of protocol specifications often rely on the reader's understanding of context and intent. Thus, it is not easy to automate information extraction by simply specifying a set of rules. The natural language community has shifted its focus to statistical methods to address this ambiguity, but domain adaption remains a major challenge. Specifically, most NLP methods are sensitive to the data used at training time and do not adapt easily if applied on data from a different domain. Applying "off-the-shelf" implementations of NLP tools, typically trained on newswire data, or combining them in an ad-hoc way, often results in reduced performance and brittle applications.

Previous work has applied NLP techniques to related problems. WHYPER [68] and DASE [69] apply NLP techniques to identify sentences that describe the need for a given permission in a mobile application description and extract command-line input constraints from manual pages, respectively. The work in [70] used documentation and source code to create an ontology allowing the cross-linking of software artifacts represented in code and natural language on a semantic level. These approaches focus on a small, predefined set of entities; analyze small, structured sentences; and use rule-based approaches. Other works infer protocol specifications using network traces [71–75], program analysis [18, 76–78], or model checking [79, 80]. These approaches rely extensively on input from human experts and do not easily generalize to new software or protocols.

In this section, we study how to improve the coverage and effectiveness of grammar-based fuzzers for network protocols through automated learning of protocol rules from existing textual documentation. We focus on RFCs[1] as this is the most common form in which Internet protocols are specified. They also follow some writing guidelines [81] that render them more amenable to automated learning. We first define the problem of grammar extraction as a set of NLP tasks and then design an NLP framework to solve these tasks. We then evaluate our framework in terms of its ability to extract

[1]An RFC is a formal document from the IETF that is the result of committee drafting and subsequent review by interested parties.

protocol grammars from a set of 7 protocol RFCs and demonstrate the usefulness of the extracted information by applying it to SNAKE.

4.5.1 Problem Definition

In this section we formulate the problem of automating protocol grammar extraction from natural language protocol specification documents as a set of NLP problems. First, we describe background on relevant NLP concepts, and then we define the problem of protocol grammar extraction.

Relevant NLP Background

NLP overview. Natural language processing tasks are organized hierarchically, into low level tasks, defined over words and phrases, and higher level tasks, defined over sentences and even the entire document. The set of tools developed by the NLP community for basic text processing is usually known as the *NLP pipeline*. These tools are chained together, such that outputs of low-level tasks are used as inputs to more advanced tasks [82].

Low level tasks include word and short phrase analysis, such as segmentation, part-of-speech (POS) tagging [83, 84], and entity extraction [84, 85]. More advanced tasks capture long-range relationships between words, either within a given sentence or across multiple sentences. For example, a dependency parser [86,87] constructs a tree connecting the words of a given sentence based on their syntactic dependencies (*e.g.,* subject). A co-reference resolution system connects noun phrases that correspond to the same entity [82].

One of the key NLP challenges is *domain adaption*, accounting for the differences between the training domain used to train tools (typically, this is newswire data) and the test domain, over which the tools are used after training (in our case, technical documents).

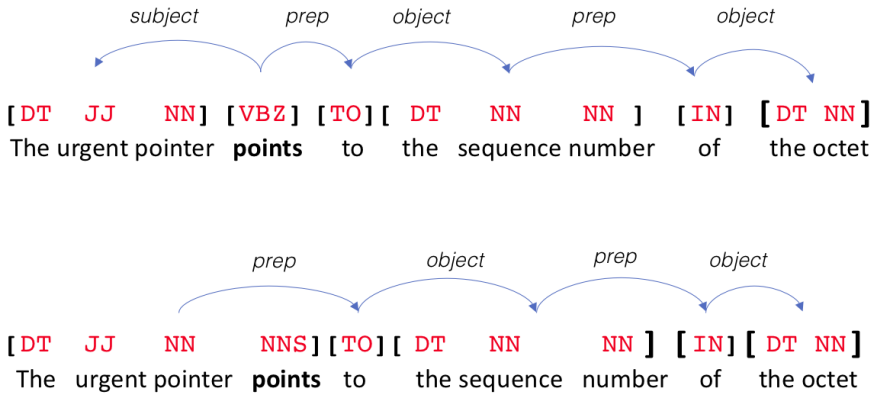


Figure 4.4.: NLP Analysis: two possible outcomes for the senses of the word *"points"* (top: verb, bottom:noun)

To help clarify these concepts, Figure 4.4 describes the output of several NLP tools over the sentence *"the urgent pointer points to the sequence number of the octet"* from the TCP specification [34]. We predict the POS tags for each word, corresponding to determiners (`DET`), nouns (`NN,NNS`), verbs (`VBZ`), and prepositions (`IN,TO`). We also identify phrases (known as chunks, marked with brackets). Finally, we identify syntactic relationships between chunks (*e.g.,* subject and object) using a dependency parser. We can observe the problem of domain adaptation by considering the two interpretations of the word *"points"*. The correct interpretation of this word is as a verb (see top half of Figure 4.4); however, systems trained over newswire data are likely to interpret this word as a noun (*e.g., "the Dow Jones rose by three points"*) as described in the bottom half of Figure 4.4. Note that this mistake propagates to other steps in the pipeline, resulting in incorrect chunking and parsing decisions.

Linking Text to Protocol Entities and Relations. The standard NLP pipeline offers task-independent language analysis. However, many applications (including protocol grammar extraction) require a more advanced analysis, *mapping the raw text to an application-specific ontology.* For example, the phrase *"president Bush"* can refer to two different people. The standard NLP pipeline would identify that the

phrase corresponds to a `PERSON` while an entity linking system [88,89] would map the phrase, based on its context, to an entity in a knowledge-base (*e.g.,* the appropriate Wikipedia entry). Entity linking maps raw entities to a canonical representation defined by an external knowledge-base and is domain-specific (in our case, protocol fields). Using relation extraction the identified concepts can then be connected via a set of specified relations, often expressed as function symbols (in our case, relations between fields).

Zero Shot Learning for Entity and Relation Linking. The traditional, fully supervised, approach for constructing NLP tools can be defined as learning a mapping, `T`$\rightarrow$`E`, from a text t to an output symbol $e_i \in E$ (where E is the set of output symbols, *e.g.,* protocol entity types). Taking this approach would require annotating data for *each protocol separately*, as the set of domain symbols is different for each protocol. This would result in a prohibitively expensive process and would defeat our goal of fully automated protocol grammar extraction. Zero-shot learning [20] addresses this problem as follows. It learns a mapping `<T,E>`$\rightarrow${`t,f`} from a tuple containing the input and output to a Boolean value indicating whether the pair is correct or not. The main observation behind zero-shot learning is that the set of output symbols does not have to be fully specified durning training, and unlike traditional supervised learning, *the system is expected to perform well even over outputs that were not observed during training.* In practice, this can be done by learning a similarity metric, $sim(t, e_i)$, and defining the prediction as: $\arg\max_{e_i \in E} sim(t, e_i)$.

Protocol Grammar Extraction

A network protocol is first and foremost defined by the header attached to transported packets. This header often has fixed size (in bits), where certain parts of it, known as fields, have defined meaning and size. Consider the TCP header presented in Figure 2.1, which has a size of 20 bytes, where bytes 17 and 18 contain a checksum of the rest of the header, meaning that those two bytes can be interpreted as a 16 bit

number. Protocol semantics are defined by the relations that exist between several fields, in the example above, those 16 bits represent the checksum of all other fields in the header. Note that not all fields have a size which is a multiple of 8 bits or 1 byte.

Fields, properties, relations. We consider protocol grammars to have three components: a set of fields that correspond to the header, with each field having a name, a size (*i.e.,* the number of bits in the field), and an order in the packet header; a set of properties that can be attached to a field; and a set of binary relations between these fields.

Specifically, given a header H of n bits, a set of property names P, and a set of relation operators R, we define a field f as the tuple $< name, size, start >$ where *name* is the identifier of the field, *size* represents the size in bits, and *start* represents the starting position in the header H from byte 0. We define a property as the tuple $< f, p >$ to denote that field f has property p, and we define relations as $< f_1, r, f_2 >$ to denote that fields f_1 and f_2 are connected through relation r. As an example, consider the TCP header from Figure 2.1. Fields include `sequence number`, with a size of 32 bits and a starting position of 4 bytes, and `control flags`, with a size of 6 bits and a starting position of 13 bytes and 2 bits. A relation exists between two fields if they are connected to each other. For example, in Figure 2.1, the field `urgent pointer` has meaning only if bit 6 of the `control flags` field is set. A property indicates something about the purpose or characteristics of a field. For example, in Figure 2.1 the field `data offset` indicates the length of the packet header.

We use the term *document* to refer to any protocol specification. For example, such a specification can be an RFC.

Given these notations, we define two NLP problems which help extracting the needed protocol information. The first problem looks at automatically extracting the set of output symbols from text, while the second problem corresponds to learning how to link the document text to the set of extracted output symbols.

Problem 1 - Entity *Type* (Protocol Field) Extraction. Given a set of network protocol documents D, we want to extract in an automated way the relevant protocol field names (including their sizes in bits and order).

Problem 2 - Extracting and Linking *Entity Mentions, Properties, and Relations.* Given a set of protocol documents D, a set of (previously extracted) protocol field names F, a set of property names P, and a set of relations operators R (specific to the networking domain) we want to extract in an automated way the properties of the extracted fields and the relations between them.

4.5.2 Design

We describe the design of our framework for automated extraction of protocol grammars. We first present our approach and then discuss in detail the pipeline we use.

Our Approach

In designing an NLP pipeline to solve the two problems defined in Section 4.5.1 we have two design goals: (1) minimize the manual supervision effort required for training and (2) adapt to new protocols without re-training the system. Previous work that used the output of NLP tools directly via a set of transformation rules does not adapt well to new protocols. Instead, we propose a lightweight zero-shot learning framework which can adapt to the specific properties of the network domain and extract relevant information.

The design of our NLP pipeline is presented in Figure 4.5. The pre-processing step reads in the raw specification documents and normalizes their structure. The entity types extraction task leverages the hierarchical structure of protocol specification documents, like RFCs. We use a hand-tuned rule-based system leveraging RFC specific formatting [81, 90] for identifying and extracting entity types. Note that this

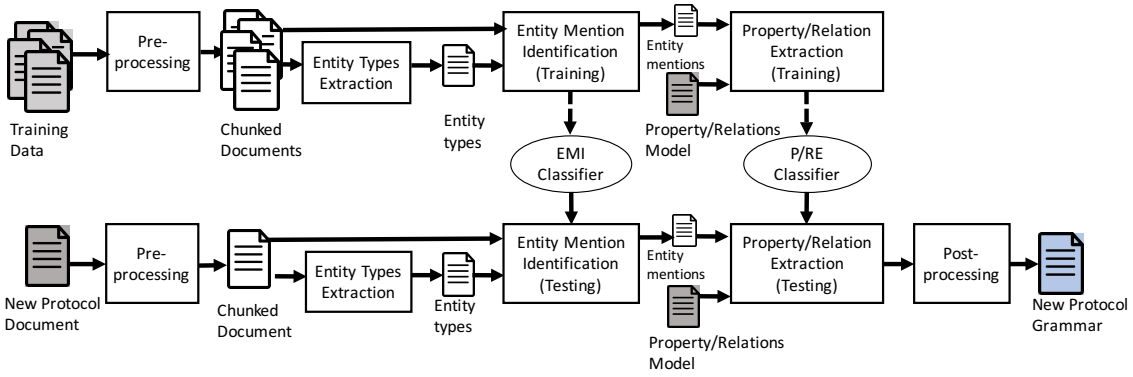


Figure 4.5.: System Design for Automated Extraction of Protocol Grammars

step can be extended to accommodate different structures of protocol documentation or can be replaced with a knowledge-base or domain ontology, when one exists.

For the task of extracting properties and relations, we take an approach where we first locate entity mentions in the document, and then, by examining the context in which they are mentioned, we look for properties and relations. For both parts we use a zero-shot learning approach, where a classifier is trained to look for similarities between document text and a list of things we are looking for. In the first case, this is the list of entity types (extracted from the document structure in the prior step) while in the second case it is a small list of relevant properties and relations. In both cases we developed new classifiers trained on network protocol data instead of using an off-the-shelf NLP system since existing systems are trained over data collected from non-technical domains like newswire, resulting in poor performance in our highly technical domain.

Finally, the goal of the post-processing step is to transform the data extracted in the entity extraction and property/relation extraction tasks into a protocol grammar description. Different tools can then transform this description as needed for different applications and languages. Below we give more details about each step.

Pre-processing

The pre-processing stage takes the raw text of the document and prepares it for the rest of the pipeline while preserving the useful document structure. The output of this stage are *chunks*, where a chunk represents a single grammatical phrase, like a noun phrase or verb phrase, made up of one or more words. The text below shows an example of "chunked" text from an RFC [34].

```
[If] [the ACK control bit] [is set] [this field] [contains] [the value]
[of] [the next sequence number the sender] [of] [the segment] [is
expecting to receive] . [Once] [a connection] [is established] [this]
[is] [always] [sent] .
```

RFC documents are text files formatted with page breaks and page headers to enable printing. We detect these page breaks and page headers and remove them. Additionally, we remove any embedded ASCII art tables and images from the document.

RFCs follow a structure where the document is divided into many sections with each section focusing on a particular topic, which may be a packet field, a protocol state, or a particular action. We find this structure to be very helpful for extracting entity types, and we preserve it by parsing the text itself into a hierarchical structure of sections, each of which has a header line, body text, and possible subsections. We run a set of standard NLP tools from the CoreNLP [82] package over the section bodies and headers to split each section into sentences and then compute part-of-speech (POS) tags for each word and the parse-tree for each sentence. Based on these POS tags and the parse-tree, we split each sentence into chunks.

Even for these simple tasks, we have to make allowances for the mis-match between the newsprint domain that standard NLP tools are trained for and our technical domain. In particular, RFCs capitalize some very common words like MUST, SHOULD, and MAY to give them specific, technical meaning [91]. This confuses CoreNLP's POS tagging, which is trained on newswire. We, therefore, identify these special words, mark them, and convert them to lowercase before doing POS tagging.

Entity Types Extraction

The entity types extraction stage performs named entity recognition. It takes the pre-processed document and extracts the entity types, or packet fields, from the document. In addition to the entity types themselves, it also extracts their size (*i.e.,* number of bits in the field) and their order.

We find that the document structure we preserved in the pre-processing step enables us to extract this information with a simple rule-based system. In particular, each entity type is described *in order* by a section, with the name and size of the entity type as the section title. Hence, we scan all section headers in the document, ignoring high-level section headers that begin with a number. Further, we ignore headers containing function words, as these do not occur in field names. The remaining section headers refer to the entity types and are in packet order. To parse each section header, we check for a colon separating the entity type's name from its size and for commas or the word "and" indicating multiple entity types in a single section header. This results in a list of entity types with their sizes and order. An example of entity types extracted by this method from one of the RFCs in our dataset is below:

```
1,Source Port,16 bits
2,Destination Port,16 bits
3,Sequence Number,32 bits
4,Acknowledgment Number,32 bits
5,Data Offset,4 bits
6,Reserved,6 bits
7,Control Bits,6 bits
8,Window,16 bits
9,Checksum,16 bits
10,Urgent Pointer,16 bits
11,Options,variable
12,Maximum Segment Size Option Data,16 bits
13,Padding,variable
```

Table 4.3.: Protocol Fields

Protocol	Number of Entity Types	Examples of Entity Types
UDP	4	Length, Source Port
TCP	18	Data Offset, Window
SCTP	40	Verification Tag
IPv6	20	Flow Label, Version
IP	22	Flags, Type of Service
GRE	6	Protocol Type, Checksum
DCCP	18	Source Port, Type

Entity Mention Identification

In order to extract properties and relations from documents, we first need to find where the entity types specific to each network protocol are mentioned in the document. Once we find these mentions, we can then look at the context where they are mentioned to identify properties and relations. For this task, the needed inputs are the pre-processed document and the list of entity types. We used the entity types list we extracted automatically from each document, but any ontology consisting of relevant entity types could also be used.

Since entity types vary dramatically both in name and number between documents (*i.e.,* protocols), as shown in Table 4.3, we use a zero-shot learning approach [20]. Instead of training a classifier to identify mentions of specific entity types, we consider the entity type to be a second input and build a classifier to identify references to the given entity type in an input text. The resulting classifier learns a similarity metric between text snippets that takes into account character level similarity, writing style (*e.g.,* capitalization patterns, abbreviations), and relevant context words. This approach allows our classifier to *generalize to previously unseen entity types* that appear in new protocol documents.

Specifically, we define a binary classification problem over all pairs (e_j, c_i) where e_j represents each entity type and c_i represents a chunk in the document text, as shown in Figure 4.6. If the chunk c_i contains a reference to entity type e_j, the pair is

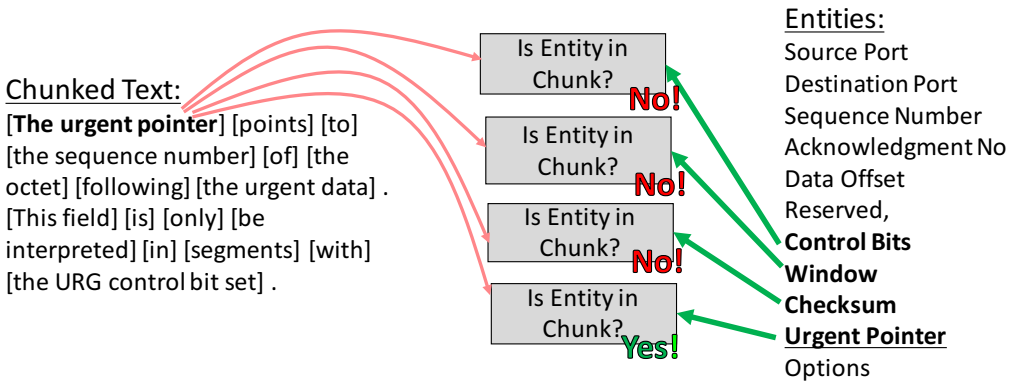


Figure 4.6.: Example of Zero-shot Learning Classification for Entity Mentions

labeled as a positive example. The pair is labeled as a negative example otherwise. This way we learn a similarity score between e_j and c_i that is able to generalize to different entity types. We train an SVM classifier for this problem using the set of binary features shown in Table 4.4. The text below shows an example of the output of our entity mention identification classifier; note that `acknowledgement number` and `sequence number` are entity types that were extracted during the entity types extraction stage.

```
If the ACK control bit is set [(entity mention: Acknowledgement Number)
this field]  contains the value of [(entity mention: Sequence Number) the
next sequence number the sender] of the segment is expecting to receive.
Once a connection is established this is always sent .
```

Property and Relation Extraction

The property and relation extraction stage identifies particular properties of and relationships between entity types and extracts them from the document body. We seek to identify a fixed set of properties and relations; however, the set of entity types related in these ways is not fixed. Because relations and properties are scarce in our dataset (see Table 4.9), we, again, take a zero-shot learning approach and train a binary classifier to identify references to a property or relation (given as input) in a

Table 4.4.: Features for our Entity Mention Identification Classifier

Num	Feature
1	More than two sequential words from the chunk match the section header.
2	A word in the chunk is not compound and matches a section header that is not a function word.
3	Either the entity is subsumed by the chunk, with any additional words being function/jargon words, or the entity contains a word in parenthesis and that word occurs in the chunk.
4	A single word header exactly matches a single word chunk
5	Entity exists in current section header and chunk contains a pronoun phrase like "this field", "they", or "their".
6	Entity is subsumed by the chunk and the next word in the chunk has a specific part-of-speech. We find only Coordinating Conjunctions, Determiners, and Nouns to be useful.
7	Entity is subsumed by the chunk and the prior word in the chunk has a specific part-of-speech. We find only Coordinating Conjunctions, Determiners, Adjectives, Nouns, and Pronouns to be useful.
8	Entity is a single word, the lemma of the entity appears in the chunk, and the next word is one of:"list","field","area","space".
9	Entity is a single word, the lemma of the entity appears in the chunk, and the prior word is "all".

given text. Additionally, we do not initially differentiate between different relations or properties.

Based on an analysis of a wide variety of network protocols, we selected 9 properties and 6 relations to extract. The properties we consider include `checksum`, which marks packet fields containing checksums; `port`, which marks packet fields used for multiplexing different communication channels; and `multiple`, which indicates that a field's value is a multiple of some constant. Relations we consider include `significant`, indicating that some field is only significant if another field has a specific value, and `offset`, indicating that some field encodes an offset from another field. The full list of attributes and relations we consider is shown in Tables 4.5 and 4.6. Note that unlike entity types, which vary between protocols, we look for the same properties and relations in each protocol. We choose these properties and relations

Table 4.5.: Properties

Name	Description (A, B denote entities)	Key phrases
sequence number	A contains a sequence number	data octet, sequence number, acknowledgment number
checksum	A (field) is a checksum	checksum
port	A is used for multiplexing	port
packet type	A denotes different types of packets	type, packet type, packet, control
header length	A indicates length of packet header	length, header, data offset, size
multiple	A's value is is scaled by a constant	integral number
monotonically increasing	A is monotonically increasing	should not be lessened
mbz	A is reserved and currently unused	must be zero, reserved, zeros, zero, zeroes, 0
range	A's values are in a limited range	takes a value of, range

Table 4.6.: Relations

Name	Description (A, B denote entities)	Key phrases
offset	A is an offset from B	offset, indicated in
significant	A is only significant if B has a particular value	significant, only interpreted with, =, valid
field present	A is only present in pkt type B	contain, carry, present, following format, following parameters, packets
contains	A contains B	from left to right, these parameters
greater	A is always > B	greater than
less	A is always < B	must not be greater, less

Table 4.7.: Features for our Property/Relation Extraction Classifier

Num	Feature
1	Maximum % of word overlap between a key phrase and the chunk
2	Maximum % of character overlap between a key phrase and the chunk
3	Longest common substring between a key phrase and the chunk
4	Overlap between a key phrase and the chunk (binary feature)
5	Key phrase is a substring of the chunk (binary feature)
6	Exact match between a key phrase and the chunk (binary feature)
7	Length of the chunk

because they are widely present across network protocols and contain information that is useful for generating test cases. For example, knowing that a field represents a checksum means that we should not spent a lot of time testing random values for that field. Similarly, knowing that some field is only relevant if another field has a particular value enables us to consider these fields together and not waste time testing them separately.

We train a single binary SVM classifier for this problem. This classifier relies on a small, focused, predefined set of key phrases associated with each of the 9 properties and 6 relation types (see Tables 4.5 and 4.6). The features we consider are shown in Table 4.7 and were extracted for a window of [-1, +1] chunks around the current chunk.

This classifier identifies chunks of text that express a relation or property; however, it does not determine which relation/property nor the identify of the arguments (*i.e.,* the entity types involved in the relation or property). Identifying the type of a relation or property is done simply by choosing the relation or property with the maximum key phrase overlap. To determine the arguments of the relation or property, we use the entity mentions identified in the previous stage and two heuristics. For properties, we choose the entity type defined in the title of the section in which the property appears. Since many properties refer to the entity type currently being discussed, this makes sense. For relations, we choose the closest entity type to the left and the

Property extraction:
Section Title: [(entity mention: Data Offset)Data Offset] : 8 bits
Section Text: The offset from the start of the packet 's DCCP [(property keyword: header length) *header*] to the start of its application data area , in 32-bit words . The receiver must ignore packets whose Data Offset is smaller than the minimum-sized header for the given Type or larger than the DCCP packet itself .
Property: Header Length, Data Offset

Relation Extraction:
Section Title: Urgent Pointer : 16 bits
Section Text: This field communicates the current value of [(entity mention: Urgent Pointer) the urgent pointer] as a positive [(relation keyword: offset) *offset*] from [(entity mention: Sequence Number) the sequence number] in this segment . The urgent pointer points to the sequence number of the octet following the urgent data . This field is only be interpreted in segments with the URG control bit set .
Relation: Urgent Pointer, offset, Sequence Number

Figure 4.7.: Examples of Property and Relation Extraction

closest entity type to the right of the relation as the arguments. Figure 4.7 shows an example of the output of this classifier.

Post-processing

This process leverages domain knowledge about how entity types, properties, and relations typically occur in network protocols. We discard any entity types with unknown or variable lengths as well as any following entity types because fields with unknown lengths prevent interpretation of later fields and variable length fields are uncommon in the protocols we consider. We convert the remainder of the list of entity types into a C-struct-style description of the protocol's packet header. In order to do this, a number of complex transformations are required because while packet fields can conceptually be arbitrary length, C is constrained to working with single bytes or multiples of 2, 4, or 8 bytes. Hence, fields that do not match these sizes, or are not aligned, need to be transformed into one or more bitfields.

We post-process the properties and relations by leveraging domain specific knowledge. Since these properties and relations are being used to characterize the protocol, we only need a single $(property, entity)$ or $(relation, entity, entity)$ tuple no matter how many times this property or relation (*i.e.,* tuple) appears in the document. This benefits us significantly because we usually have multiple opportunities to extract each property or relation tuple. In addition, many properties can occur only on a single field in the packet header (*e.g.,* `packet type`, `header length`) while other relations or properties cannot occur in combination (*e.g.,* `packet type` and `sequence number` are mutually exclusive). Finally, if our pipeline was unable to identify key properties like `packet type`, `header length`, and `checksum`, we attempt to guess which fields have these properties based on field names and sizes. Finally, we associate our cleaned properties and relations with the packet fields. This results in a concise and clean protocol description.

4.5.3 Case Study: SNAKE

We demonstrate the usefulness and effectiveness of our automated protocol grammar extraction framework by applying it to SNAKE, providing several benefits including:

(1) **Closing the gap between specification and testing.** By automating the generation of a protocol description, our approach closes the gap between specification and vulnerability finding and eliminates the need for a protocol expert to manually create a description of the protocol for testing purposes. Note that while our approach requires some annotation for training, this is a one time cost; additional protocols can be analyzed with no manual effort. The output produced by our pipeline was easily integrated with SNAKE.

(2) **Optimizing test cases.** The properties and relations that our automated document processing pipeline identifies enable us to optimize test case generation by eliminating some irrelevant tests and providing better manipulation values in other

Table 4.8.: Packet Field Modifications for Test Cases

Field Info	Modifications
1 bit fields	0,1
2 bit fields	0,1,3
3 bit fields	0,2,3,7
4 bit fields	0,2,4,8,15,+1,-1
5 bit fields	0,2,4,8,16,9.31,+1,-1
6 bit fields	0,2,4,16,63,+1,-1
7 bit fields	0,3,4,8,19,64,127,+1,-1
8 bit fields	0,255,random
16 bit fields	0,65535,random
32 bit fields	0,4294967295,random
64 bit fields	0,16777216,random
port property	1025
checksum property	random
mbz property	random
sequence number prop	+1,-1,+10,-10,+100,-100
monotonic increasing prop	+1,+10,+100,+100
range property	4 locations within range
greater relation	+1, +100, +1000, -1000
less relation	-1, -100, -100, +1000
offset relation	+1, -1, +1000, -1000

cases, with the goal of providing more meaningful test cases. For example, from the definition of checksums and protocol ports, we expect that tampering with them will result in modified packets simply being thrown away. Therefore, we apply only a single modification—to confirm expected behavior—to fields identified as checksums or ports (via the `checksum` or `port` properties). In a similar manner, `mbz` (must-be-zero) fields are those with no current functionality, included for padding or extensibility purposes. We apply only a single modification to fields identified as such because we expect these fields to be irrelevant to protocol operation. In other cases, like `sequence number` or `monotonically increasing` fields, we optimize the set of operations we perform. In both cases, adding or subtracting from the current field value is likely to be much more interesting than setting an absolute value, since the absolute values may vary widely between or during tests. The full set of improvements we make to field modifications

Table 4.9.: Dataset Statistics

Statistic	Value
Documents	7
Entity Types	128
Entity Mentions	516
Properties	88
Relations	49

are shown in Table 4.8. This allows us to generate fewer strategies than SNAKE did previously, by pruning uninteresting strategies, while testing a similar amount of the protocol.

(3) **Reducing false positives.** The properties provided by our NLP pipeline enable us to identify and filter test cases that are expected to fail based on the properties associated with modified fields, reducing the test cases that need to be manually triaged. Examples include `header length` or `checksum` fields which contain fundamental details about packets. When these fields are modified, we expect that the resulting packets will to fail to parse.

4.5.4 Evaluation

In this section, we evaluate our document processing pipeline to understand how effective it is at extracting entity types and properties from textual specification documents and how effectively that information can be leveraged to find attacks against protocol implementations.

Dataset. To train the classifiers for our pipeline, as well as develop the rule-based entity type extractor, we annotated a set of training protocol specification documents in the form of RFCs. RFCs are public documents available from the IETF at [92]. They are a common form for protocol specification and are written in plain text following a specific format [81, 90, 91]. We use RFC documents for seven protocols: GRE [59], IPv6 [60], IP [61], TCP [34], UDP [31], DCCP [45], and

SCTP [62]. We selected these RFCs because they specify common transport and network layer protocols, meaning that we expect their designs to have similarities. Additional details about the dataset are shown in Table 4.9.

NLP Pipeline Evaluation

We evaluate our NLP pipeline by examining each of its three stages: extracting the entity types for each protocol, identifying entity mentions, and extracting properties and relations. We compare our use of specialized features and classifiers with simpler approaches based on keyword overlap to understand the additional value added by our approach. We use standard K-way cross validation, where we train on $K-1$ documents and test on the last. Our results are shown in Table 4.10.

Entity Types Extraction. Results for this stage are shown in Table 4.10a. We find that our rule-based extractor is quite effective in most cases, obtaining an average precision of 0.75, recall of 0.72, and F1-score of 0.74.[2]

We note that this rule-based method fails completely for two protocols, UDP and GRE. Both UDP and GRE depart significantly from the normal method for entity type description. Specifically, entity types are described either in paragraph form (UDP) or in numbered headings (GRE), which causes our rule-based system to incorrectly ignore them as not relevant. These kinds of departures from the normal method of description pose significant difficulties for any extraction technique. Fortunately, for all other protocols, our extractor does quite well. This is crucial because errors here have cascading effects down the rest of the pipeline in entity mention identification and property/relation extraction. We are currently looking at machine learning based methods to improve the performance of entity extraction.

Entity Mention Identification. Results for this stage are shown in Table 4.10b. Since identifying entity mentions requires the protocol's entity types, we consider how well this stage performs using both the set of entity types from our annotations and

[2]Recall is the fraction of true instances identified while precision is the fraction of predicted instances that are correct. F1-score is the harmonic mean of recall and precision.

Table 4.10.: NLP Document Processing Pipeline Evaluation

(a) Entity Types Extraction

Protocol	Precision	Recall	F1	Instances
UDP	0	0	0	4
TCP	0.86	0.67	0.75	18
SCTP	0.73	0.78	0.73	40
IPv6	0.82	0.90	0.86	20
IP	0.65	0.64	0.65	22
GRE	0	0	0	6
DCCP	0.94	0.94	0.94	18
Total	**0.75**	**0.72**	**0.74**	**128**

(b) Entity Mention Identification

Protocol	Pipeline Entity Types			Annotated Entity Types			Instances
	Precision	Recall	F1	Precision	Recall	F1	
UDP	0	0	0	0.33	0.14	0.20	7
TCP	0.92	0.59	0.72	0.97	0.71	0.82	41
SCTP	0.57	0.36	0.44	0.69	0.43	0.53	240
IPv6	0.81	0.75	0.78	0.94	0.89	0.92	73
IP	0.93	0.56	0.69	0.82	0.60	0.69	45
GRE	0	0	0	1.0	0.81	0.89	21
DCCP	0.91	0.55	0.69	0.87	0.81	0.84	89
Total	**0.73**	**0.46**	**0.57**	**0.82**	**0.61**	**0.70**	**516**

(c) Property Extraction

Protocol	Found	Linked (Pipeline Mentions)	Linked (Annotated Mentions)	FP Rate	Instances
UDP	1.00	0.0	0.80	0.32	5
TCP	0.92	0.75	0.83	0.33	12
SCTP	0.87	0.55	0.58	0.21	38
IPv6	0.90	0.90	0.90	0.44	10
IP	0.88	0.75	0.88	0.24	8
GRE	1.00	0.0	1.00	0.36	4
DCCP	1.00	0.92	1.00	0.34	11
Total	**0.91**	**0.66**	**0.76**	**0.29**	**88**

Table 4.11.: Entity Mention Identification Baselines

Approach	Precision	Recall	F1	Correct Refs	FP Refs
Overlap $\geq$ 50%	0.18	0.75	0.29	387	1800
Overlap $\geq$ 70%	0.36	0.66	0.47	341	609
Overlap $\geq$ 85%	0.55	0.59	0.57	303	248
Overlap $\geq$ 100%	0.69	0.49	0.57	252	115
RB_1	0.74	0.19	0.30	95	34
RB_2	0.69	0.52	0.59	264	116
Our Approach	**0.82**	**0.61**	**0.70**	**315**	**201**

our extracted entity types from the prior stage. We find that this stage performs quite well, with an overall F1-score of 0.70 when using the annotation entity types. Since the overwhelming majority of chunks are negative instances (*i.e.,* do not contain an entity mention), we tune the classifier to strongly penalize false positives, preferring precision to recall. With our extracted entity types, we observe a decrease in performance across the board, mostly in recall due to missing entity types. Overall F1-score drops to 0.57; however, this is largely due to UDP and GRE where the prior stage failed to find any entity types. Many of the other protocols have F1-scores around 0.70.

We compare our classifier to six simpler baseline approaches in Table 4.11. The first four baselines are simple string matching systems. Here, we measure the overlap between an entity type and the current chunk and classify a chunk as a mention if the overlap is at or above a certain percentage P. The trade-off in these systems is clear, the higher P, the higher the precision and the lower the recall. As we reduce P, recall increases and precision suffers. Best results are obtained using a threshold of between 85% and 100%. None of these systems perform nearly as well as our approach, with the F1-score topping out at 0.57, compared to 0.70 using our approach. Precision is especially problematic, topping out at 0.69 compared to 0.82 with our approach.

The last two baselines are simple rule-based systems. Here we take the same set of features used by our classifier and weigh them manually. In baseline RB_1, we weight each feature by its frequency of occurrence. In other words, for each feature f_j we

calculate pr_j and nr_j. We then give each feature f_j a weight of $+pr_j$ if $pr_j > nr_j$, a weight of $-nr_j$ if $nr_j > pr_j$, and a weight of 0 if $pr_j = nr_j$. We use a weight of $-nr_j$ for the bias term. In the second baseline RB_2, we weight each feature with $+1$ if it occurs more often in positive examples and -1 if it occurs more often in negative examples. In other words, we give each feature f_j a weight of $+1$ if $pr_j > nr_j$ and a weight of -1 if $nr_j > pr_j$ and a weight of 0 if $pr_j = nr_j$. We use a weight of -1 for the bias term. While RB_2 performs better than string matching, it stills performs worse than our classifier. In short, we see value from both our carefully crafted set of features and our use of an SVM classifier.

Property and Relation Extraction. Results for this stage are shown in Table 4.10c. Our classifier does quite well at identifying properties, finding 91% of the properties on average with a 29% false positive rate.[3] Note that for this task we are more concerned with recall, finding all possible properties, than with precision because our post-processing step is able to eliminate many incorrect properties by leveraging domain-specific knowledge. As a result, incorrect properties are much easier to handle than missing ones. The prior stage made the opposite trade off because it is much less likely that a chunk is a mention than that it is not.

Once we have found a property, we need to link it with the relevant entity type. This is a much harder problem and depends on the entity mentions we identified in the previous stage, which depend on the entity types from the stage before. Because of this, we consider the performance of property linking both with entity mentions from annotation and with the entity mentions identified in the prior stage. We observe a success rate of 76% with the annotation entity mentions, with many protocols significantly above that. This drops to 66% when using identified entity mentions from the prior stage. Again, this is partly due to issues with extracting entity types from UDP and GRE in the entity types extraction stage.

To understand how effective our classifier-based property extraction approach is, we compare with six simpler baseline approaches in Table 4.12. The first four baselines

[3]This corresponds to a recall of 0.91 and a precision of 0.71. Our Found/Linked columns are equivalent to recall while the FP Rate columns are $1 - precision$.

Table 4.12.: Property Extraction Baselines

Approach	Found	FP Rate
Overlap $\geq 50\%$	0.90	0.44
Overlap $\geq 70\%$	0.77	0.16
Overlap $\geq 85\%$	0.72	0.13
Overlap $\geq 100\%$	0.72	0.13
RB_1	0.91	0.79
RB_2	0.94	0.79
Our Approach	**0.91**	**0.29**

are simple string matching systems. Here, we measure the overlap between property key phrases and the current chunk and classify a chunk as a property if the overlap is at or above a certain percentage P. These systems tend to have a high success rate and low false positive rate, which is good. However, all perform worse than our approach, with success rates topping out at 77%, compared to 91% with our approach. In the case of 50% overlap, even though we are able to find 90% of the properties, the false positive rate is considerably higher.

The last two baselines are the same rule-based systems we considered for entity mention identification. They take the same set of features used by our classifier and weigh them manually. In baseline RB_1 we weight each feature by its frequency of occurrence while in RB_2 we weight each feature with $+1$ if it occurs more often in positive examples and -1 if it occurs more often in negative examples. While these systems do better than string matching and have an equivalent or slightly higher success rate compared to our classifier, the false positive rate is 79%, high enough to be extremely problematic. In short, we see benefits from using both our carefully crafted set of features and our SVM classifier, especially when it comes to avoiding false positives.

For relations, our classifier is able to identify 80% of them with a 31% false positive rate, as shown in Table 4.13. Interestingly, in many protocols the success rate is 100%, but the performance on the two protocols with the most relations, TCP and SCTP,

Table 4.13.: Relation Extraction Evaluation

Protocol	Relations		
	Found	FP Rate	Instances
UDP	0	0.41	0
TCP	0.67	0.35	6
SCTP	0.71	0.23	28
IPv6	1.00	0.45	3
IP	1.00	0.24	2
GRE	1.00	0.37	4
DCCP	1.00	0.35	6
Total	**0.80**	**0.31**	**49**

is much lower. This is likely due to an uneven distribution of different types of relations in our documents, with SCTP having the vast majority of `field present` relations and TCP having the vast majority of `significant` relations. Given that we only have 49 relations in the entire training set, there are not enough relations in the remaining data to fully train the classifier. For our fuzzing case study with TCP, we find that the vast majority of improvements comes from properties, with even annotated relations providing no significant improvements. Therefore, we do not consider relations further here.

Summary. Overall, we have seen that our document processing pipeline is effective at extracting entity types, entity mentions, and properties from natural language specification documents. We find that our entity types extraction achieves an impressive F1-score of 0.74 despite poor performance on UDP and GRE. Our entity mention identification achieves an F1-score of 0.70 while our property and relation extraction identifies 91% of all properties in the document and links 76% of them.

Fuzzer Evaluation

We now evaluate how effectively the entity types and properties extracted by our NLP pipeline can be used to test real protocol implementations for attacks. We

seek to understand: (1) How much benefit does a protocol grammar description, either automatically extracted or manually created, provide over just blindly fuzzing the protocol? and (2) How effective is testing based on our automatically extracted protocol grammar compared to testing that uses manually defined descriptions?

Fuzzer configurations. We use SNAKE and concentrate on a single protocol, TCP [34], and a single implementation, Linux 3.0.0 in Ubuntu 11.10. We compare three different testing configurations: Random, Manual, and NLP-based.

Random. This is the simplest baseline and uses SNAKE configured with no information about the protocol grammar. It generates tests that randomly replace a random number of the first 20 bytes of packets with random data. We only modify the first 20 bytes to approximate the length of a typical transport protocol header. Note that in any given test the same bytes in all packets are modified. Attack injection is on every packet sent. We generate 1,000 test strategies in this manner to compare with our other testing configurations.

Manual. This configuration uses SNAKE with a manually created protocol grammar as discussed earlier in Section 4.2. Note that we do not use the connection-level state machine in this testing, to better compare with our NLP-based configuration that also lacks a state machine. For each packet type, test strategies are created to inject new messages, modify all packet fields, and apply all delivery actions to those packets. For modifying packet fields, tests modify fields based on their size. During each test, all packets of a particular type are modified, and attack injection is on every packet.

NLP-based. This configuration uses SNAKE but configured with our automatically extracted protocol grammar, derived from extracted entity types and properties. This configuration generates a similar set of tests that injects new packets, modifies the delivery of packets, or overwrites a single field in packets during each test. During each test, all packets of a particular type are modified, and attack injection is on every packet. For each packet type, test strategies are created to inject new messages, modify all packet fields, and apply all delivery actions to those packets. This configu-

ration has more information about packet fields available to it, thanks to our pipeline. We leverage this information to apply better field modifications, as discussed in Section 4.5.3. For example, from the definition of checksums and protocol ports, we expect that tampering with them will result in modified packets simply being thrown away. Thus, we apply only a single modification—to confirm expected behavior—to fields that are identified as checksums or ports (via the `checksum` or `port` properties). We anticipate this resulting in similarly effective testing with a reduced number of strategies.

Metrics. We use the number of test strategies generated to measure the amount of effort required to test an implementation. We measure coverage as the number of unique packet type traces observed. A *packet type trace* records the order in which different types of packets are observed in a TCP flow. Thus, a packet type trace succinctly summarizes a protocol connection and approximates the path traversed through the code. To effectively test a protocol, as many distinct connections, or code paths, as possible should be explored, hence unique packet type traces.[4] Ideally, we want to expend a small amount of effort while achieving high coverage.

The number of attacks identified indicates how many test strategies were reported by the testing configuration as attacks. Unfortunately, many of these attacks are on-path attacks which are not interesting since TCP does not attempt to provide protection against these attacks. Removing these on-path attacks leaves us with the interesting endpoint or blind attacks, which we refer to as *interesting attacks*. Note that many strategies may exercise the same underlying root vulnerability, so we perform a manual analysis of all reported attack strategies to identify the number of *unique attacks* actually identified.

The results from running all three of our testing configurations can be found in Tables 4.14, 4.15, and 4.16.

Random Testing vs Grammar-based Fuzzing. Table 4.14 compares coverage, in terms of unique packet type traces, achieved by all three configurations.

[4]Note that we record packets prior to any possible modification to avoid counting traces where the only different packet is one that was intentionally modified.

Table 4.14.: Coverage Evaluation

	Unique Packet Type Traces	Total Strategies
Random	13	1000
Manual	784	901
NLP-based	713	819

Table 4.15.: Attack Discovery Results

	Reported Attacks	Interesting Attacks	Unique Attacks
Random	996	0	0
Manual	219	63	5
NLP-based	220	69	5

Table 4.16.: Attacks Discovered

Attack	Man	NLP
CLOSE_WAIT Exhaustion	X	X
Reset Attack [13]	X	X
SYN-Reset Attack [3]	X	X
FIN-injection Attack [3]	X	X
Packets with Invalid Flags	X	X

We observe that the manual and NLP-based configurations achieve similar coverage, around 700 unique traces, while random achieves only 13 traces. To achieve this coverage, all three configurations required about 1,000 strategies. Since the number of strategies is directly equivalent to the amount of effort required for testing, we can say that random fuzzing is significantly less efficient than grammar-based fuzzing.

This occurs primarily because in the random test configuration all packet manipulation strategies stall the connection. This is because modifying the packet corrupts the TCP checksum, resulting in the packet being thrown away at the receiver. In order to correct this, the fuzzer would need to know the exact location of the checksum in the packet, which is exactly the information provided by a protocol grammar.

Similarly, all packet delivery strategies in the random test configuration stall the connection because they drop or delay key packets like the TCP SYN. In order to work around this, the fuzzer would need to know the type of each packet, which is also supplied by a protocol grammar. All of these connection stalls generate similar traces and traverse similar code paths, resulting in very poor coverage.

In addition to very poor coverage, Table 4.15 indicates that the random test configuration also generates a lot of reported attacks, but none of them are interesting. This is because each of the connection stalls mentioned above is reported as an attack on availability. Unfortunately, these are on-path attacks and so are not relevant for TCP.

NLP-based vs Manual Configurations. We first consider testing coverage, shown in Table 4.14, and confirm that, thanks to the additional properties provided by our document processing pipeline, the NLP-based configuration generates fewer strategies than the manual configuration. This results in a reduction in the amount of time and effort required for testing. This does result in slightly lower coverage, but only by about 70 traces.

We also consider the attacks that are reported by both testing configurations, shown in Table 4.15. We find that our NLP-based testing system reports one more attack than our manual testing system and that more of those it reports are interesting. Further, we find that both our manual configuration and our NLP-based configuration discover the same set of five attacks, as confirmed by Table 4.16. None of these attacks are new, although one was initially discovered by SNAKE (see Section 4.4). However, they all have serious impacts on TCP connections, ranging from denial of service to server fingerprinting [3, 13].

Summary. Overall, we find that grammar-based fuzzing, like SNAKE, provides significant benefits in terms of efficiency and ability to find attacks and that our automatically generated protocol grammars are as effective in identifying attacks as manually created grammars while enabling improved efficiency. In addition, our NLP document processing pipeline enables completely automated testing.

4.6 Summary

Transport layer networking protocols form an important part of the Internet, yet, to date, their testing has been mostly manual and ad-hoc. This has resulted in a stream of vulnerabilities stretching back to the 1980's. To help remedy this situation, we present SNAKE, a tool to allow systematic testing of unmodified transport protocol implementations, utilizing the protocol's connection state machine to reduce the search space. We demonstrate SNAKE by testing 2 different protocols, TCP and DCCP, and 5 implementations, including both open-source and closed-source systems. We found 9 classes of attacks, 5 of which we believe to be unknown in the literature. To do this testing, SNAKE requires a description of each protocol and its connection-level state machine.

We then design and build an NLP pipeline to extract these protocol descriptions, or grammars, from natural language specification documents automatically. Our NLP pipeline extracts protocol entity types—or packet fields—, properties, and relations from natural language network protocol RFCs using a zero-shot learning approach. We evaluate our ability to extract protocol grammars on a corpus of 7 protocol specification documents and achieve an F1-score of 0.74 for extracting entity types and a success rate of 76% at finding and linking properties. We further demonstrate the value of our approach by applying it to SNAKE and comparing it to using a manual grammar. We find a reduction in the testing effort (from 901 to 819 test cases) while identifying the same set of attacks and doing so in a fully automated manner. We believe that SNAKE and our NLP pipeline can contribute to securing the transport layer of modern network stacks.

5 AUTOMATED ATTACK DISCOVERY FOR TCP CONGESTION CONTROL

In the previous chapter, we introduced SNAKE, a system to automatically find performance and availability attacks on transport protocols. While SNAKE allows us to find many attacks on transport protocols, it has important limitations. In particular, the techniques that SNAKE uses to broadly find a huge range of attacks on a variety of transport protocols are ineffective for finding complex and highly dynamic attacks on complicated guarantees like congestion control that may operate very differently between protocols. In this chapter, we investigate how to automatically find attacks on one of these more complicated guarantees, congestion control, for a particular transport protocol, TCP.

5.1 Introduction

TCP is the protocol that underlies most of the Internet traffic including encrypted traffic via TLS and HTTPS. In addition to reliable and in-order data delivery, TCP has two critical goals – efficient delivery based on network conditions and fairness with respect to other TCP flows in the network. These two goals are achieved by using congestion control mechanisms that cause a sender to adapt its sending rate to the current network conditions (*e.g.,* network congestion) or to the receiver's processing resources (*e.g.,* a slow receiver). Without congestion control, the network can enter a condition where the majority of sent data is eventually dropped, known as *congestion collapse*; such a collapse occurred on the Internet in 1986, causing throughput to drop by a factor of a thousand [93].

TCP congestion control relies on acknowledgement packets from the receiver to explicitly provide the sender with *correct* information about the number of data bytes received (and implicitly about the *real* network conditions). However, TCP does not

have any cryptographic mechanisms to ensure authentication and integrity of sent packets, including acknowledgments. Application-layer secure protocols such as TLS provide no protection for TCP headers or TCP control messages, and network-layer secure protocols such as IPsec [94] require separate infrastructure and protect only up to the tunnel termination point. Thus, an attacker that can intercept acknowledgment packets can modify them without being detected by the intended recipient, who will blindly trust the information they provide. TCP has a protection mechanism against packet injection in the form of a sequence number included on each packet. However, numerous attacks demonstrate that this protection mechanism can be bypassed by blind attackers performing TCP sequence guessing [95–98] or by *off-path* or *on-path* attackers that can observe the target stream. Thus, an attacker can also inject well-crafted acknowledgment packets into a TCP stream without detection. By creating such crafted acknowledgments that propagate malicious information about the data received, an attacker can manipulate TCP congestion control into sending data at rates that benefit the attacker. For example, by creating an acknowledgement that acknowledges data packets prior to receiving them and injecting it into a target stream, an adversarial TCP receiver can persuade the sender to increase its sending rate beyond the rate prescribed by correct congestion control, possibly forcing the network into congestion collapse [11].

Several manipulation attacks against TCP congestion control have been discovered; some of these attacks use external data flows to create the impression of congestion [8,9] and others use acknowledgement packets to directly mislead the congestion control mechanisms [3, 11, 99, 100]. These attacks are more subtle and difficult to detect than traditional crash or control-hijacking attacks. Acknowledgement-based attacks, in particular, do not raise suspicions as long as the packets are consistent with the receiver's state (unlike data that might not assemble properly at the application level). We focus on attacks against congestion control created through maliciously

crafted acknowledgement packets (by fabrication of new ones or modification of existing ones) and refer to them as *manipulation* attacks.[1]

Previous work on attacks against TCP congestion control relied mainly on manual analysis. The only work we are aware of that used automation for finding attacks in TCP congestion control implementations is the work in [5] which relies on the user to provide a vulnerable line of code and then performs static analysis. The vulnerable line of code from the user is critical to ensure the scalability of the approach. In addition, the method is restricted to a specific implementation, language, and operating system.

In this chapter, we aim to automatically discover manipulation attacks on congestion control without requiring the user to provide any vulnerable line of code and without being dependent on specific implementation, language, or operating system characteristics. *Protocol fuzzing* [6, 15, 101] is a well-known approach where packet contents are either randomly generated and injected into the network or randomly mutated in-transit. However, without explicit guidance, given a vast input space, fuzzing fails to concentrate on relevant portions of the source code (*i.e.,* for inducing protocol-compliant behaviors).

Our previous work on testing transport protocols, SNAKE (see Chapter 4), used the protocol's connection state machine to guide the fuzzing process and prune unnecessary executions. However, unlike the attacks SNAKE finds, which usually consist of one action, attacks against congestion control require a potentially long sequence of actions spanning several states and transitions, where each action might trigger a new state, which in turn might require a different attack action. Automatically discovering these combinations at runtime is not practical for scalability reasons. For example, using SNAKE's approach for congestion control would require a search space of about 1.2×10^{24} cases, assuming only 5 types with 4 parameter choices for creating the malicious acknowledgements and 4 possible states for injecting them. Even

[1]Note that attackers can also create the impression of network congestion without manipulating the acknowledgement packets but by using external data flows [8, 9]. We consider such attacks out of scope for this chapter.

limiting this to test at most one manipulation at a time in each state would generate 194,480 cases, which is still impractical for testing in a real network.

To address this scalability challenge while still guaranteeing that we test relevant portions of the code, we use *model-based testing* (MBT) [102], an approach that generates effective test cases based on a model of the program. The approach uses a *model*, an abstract representation of the desired behavior of the program that is typically derived from specifications, to derive *functional tests*. These functional tests contain the same level of abstraction as the model, and are converted to concrete test cases to be tested against the implementation. MBT does not require the source code and guides the testing to concentrate only on relevant portions of the source code.

Our approach. We propose to automatically find manipulation attacks by guiding a protocol fuzzer with *concrete* attack actions derived from *abstract attack strategies*, which are obtained using a model-guided technique inspired by model-based testing. Our model is a finite state machine (FSM) that captures the main functionality of several types of congestion control algorithms used by deployed TCP implementations and is constructed from RFC specifications. We use this abstract model to generate *abstract attack strategies* by exploring the different paths in the FSM that modify state variables controlling throughput, and thus can be leveraged to mount an attack. We then map these abstract strategies to *concrete attack strategies* that correspond to real attacker capabilities; a concrete strategy consists of acknowledgment-packet-level actions with precise information about how the packets should be crafted and the congestion control states in which these actions should be performed. Our approach provides maximum coverage of the model of congestion control while generating an optimum number of abstract strategies. The number of concrete attack strategies is bounded by the number of malicious actions that describe an attacker's capabilities. We consider off-path attackers and on-path attackers; both can sniff traffic and obtain TCP sequence numbers and data that has been acknowledged or sent. However, there is one fundamental difference, an off-path attacker can only inject malicious acknowledgements, but cannot prevent the correct ones from reaching the receiver;

an on-path attacker can modify acknowledgements such that the victim sees only acknowledgments from the attacker.

We created and implemented a platform, TCPwn, to create and inject concrete attack scenarios. The platform combines virtualization (to run different implementations in their native environment), proxy-based attack injection, and runtime congestion control state machine tracking (to inject the attacks at the right time during execution). Our state machine tracking at runtime does not require instrumenting the code. Specifically, we use a general congestion control state machine (*e.g.,* TCP New Reno) and infer the current state of the sender by monitoring the network packets exchanged during fuzzing. While this option is less accurate than extracting the state machine from an implementation's code, it is less complex and more general.

Our model-based attack generation finds 21 abstract strategies that are mapped into 564 (for on-path attackers) and 753 (for off-path attackers) concrete strategies. Each strategy can be tested independently and takes between 15 and 60 seconds. We evaluated 5 TCP implementations from 4 Linux distributions and Windows 8.1, all using congestion control mechanisms that can be modeled as the finite state machine we used to generate abstract strategies. Overall, we found 11 classes of attacks, of which 8 were previously unknown.

The rest of the chapter is organized as follows. We describe our attacker model in Section 5.2. We provide details on the design of our system in Section 5.3 and describe our implementation in Section 5.4. We present our results in Section 5.5 and then summarize this chapter in Section 5.6.

5.2 TCPwn Attack Model

In this section we discuss the attacker capabilities and congestion control attacks that we consider in this chapter.

5.2.1 Attacker and Attack Goals

A typical attacker might be a botnet trying to enhance the power of a DDoS attack by using increased throughput attacks to render TCP flows insensitive to congestion. This gives the attacker the power of a UDP flood with the ubiquity of TCP traffic; perfect for the coremelt attack [103]. Alternately, a nation-state actor could launch decreasing throughput attacks to discourage or prevent use of certain undesirable services.

Decreasing Throughput. In this case, the attacker manipulates the congestion control algorithm of a target connection such that it falsely detects congestion, resulting in a rate reduction. This rate reduction can have significant impact at the application level, especially for inelastic data streams like streaming video.

Example. Consider the *Blind Throughput Reduction Attack* [3]. In this attack, the attacker sends spoofed invalid acknowledgements to the target connection's receiver, which cause the receiver to send duplicate acknowledgements to the sender. These duplicate acknowledgements, when received in the Congestion Avoidance or Slow Start states, mislead the sender about the existence of lost packets and the level of congestion in the network, causing the sender to transition to the Fast Recovery state and slow down (see Figure 5.1). The sender will continue to slow down as long as the attacker emits its spoofed acknowledgements.

Increasing Throughput. In this case, the attacker manipulates the congestion control algorithm such that it perceives significant available bandwidth along with low latency and loss. As a result, the sender rapidly increases its sending rate beyond what is fair to competing connections. Any actual congestion in the network will not be observed, which may be used to damage or deny service to target links or to other connections sharing the same links.

Example. Consider the *Optimstic Ack Attack* [11]. In this attack, the receiver repeatedly sends acknowledgements for data that has not actually been received yet in order to dramatically increase its sending rate and render the sender insensitive

to actual congestion in the network. Acknowledging data not yet received in the Congestion Avoidance, Slow Start, or Fast Recovery states misleads the sender about the data that has been received and the RTT of the connection. As a result, the sender does not react to actual congestion in the network and is unfair to any competing connections.

Target Flows. Any TCP flow that sends more than an initial window (10 packets, about 15KB) of data is vulnerable to these attacks. We focus on bulk data transfers because they result in the widest array of attacks, are easiest to automate, and easiest to explain; however, these attacks are not restricted to such flows. Short transfers, like web pages, are also vulnerable to attacks on congestion control, and flows with a limited bitrate, like streaming video, are vulnerable to decreasing throughput attacks. Interactive flows are vulnerable if their sending rate is limited by congestion control and not by the availability of data from the application.

5.2.2 Attack, Strategy, Action

Congestion control constrains the sender's data-transfer rate, primarily through acknowledgements. Thus, we consider attacks conducted through acknowledgement packets.

Congestion control manipulation attacks. These are attacks conducted by manipulation of TCP acknowledgements in order to mislead congestion control about current network conditions and cause it to set an incorrect sending rate. They can result in either increasing or decreasing the throughput, and sometimes in connection stall. In order to achieve the high-level goals of manipulating congestion control, an attacker applies an *attack strategy.*

Attack strategy. Given a TCP stream, where a sender sends data to a receiver, we define a concrete attack strategy as a sequence of acknowledgment-based malicious actions and the corresponding sender states (as described in Figure 2.3 and Section 2.1.1) where each action is performed.

Malicious actions. A malicious action itself requires an attacker to (1) craft acknowledgements by leveraging protocol semantics to mislead congestion control, (2) infer the state at the sender, and (3) inject the malicious acknowledgment on the path and in the target stream. For example, a malicious action can be to craft an acknowledgment that acknowledges data not yet received and inject it when the sender is assumed to be in Congestion Avoidance.

Crafting malicious acknowledgements. TCP does not use any cryptographic mechanisms to ensure authentication and integrity of packets; thus, an attacker can fabricate packets or modify intercepted ones to have a malicious payload. In order to intercept, the attacker will need to be on the path. Moreover, these crafted acknowledgements are *semantic-aware*, that is, the attacker is aware of the meaning of the bytes acknowledged. For example, in the example above, an attacker will need to know the highest byte of data that was acknowledged in order to acknowledge data that has not been received yet.

Inferring the state machine at the sender. We assume that the attacker can observe the network traffic but does not have access to implementation source code and thus cannot instrument the implementation.

Injecting malicious acknowledgments. This requires an attacker to spoof packets and have knowledge of the TCP sequence number, the only protection TCP has against injection. We do not consider blind attackers here, since, while they can inject spoofed packets into the network, they have no knowledge of sequence numbers or data being acknowledged and thus are restricted to guessing this information. We distinguish between off-path and on-path attackers. An off-path attacker can observe packets in the target connection or link and inject spoofed packets. For example he can sniff traffic on the client's local network — e.g., coffee house Wi-Fi. An on-path attacker can intercept, modify, and control delivery of legitimate packets in some target connection or link, as well as inject new spoofed packets. For example, such an attacker can be a switch on the path between client and server.

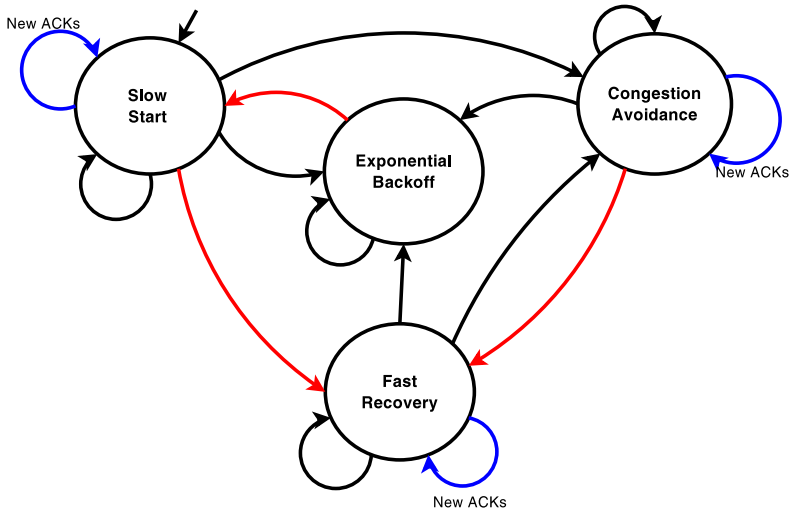


Figure 5.1.: New Reno congestion control and the Optimistic Ack attack. Transitions in blue increase throughput while those in red decrease throughput.

5.3 Design

In this section we describe the design of TCPwn, our automated platform for finding attacks on congestion control. We first provide a high-level overview, then discuss our model-guided attack strategy generation and congestion control protocol state tracking.

5.3.1 Overview

We motivate our approach with the *Optimistic Ack Attack* [11]. Consider its interactions with the congestion control state machine as shown in Figure 5.1. In order to be successful, the attacker must inject packets with an acknowledgement number above the real cumulative acknowledgment number and below the highest sequence number that the sender has sent, and it has to do this in either the Congestion Avoidance, Slow Start, or Fast Recovery states. Each time the sender receives one of these new acknowledgements in those states, it causes a self-loop transition (in blue in Figure 5.1), increasing the congestion window `cwnd`, which directly controls the sending rate.

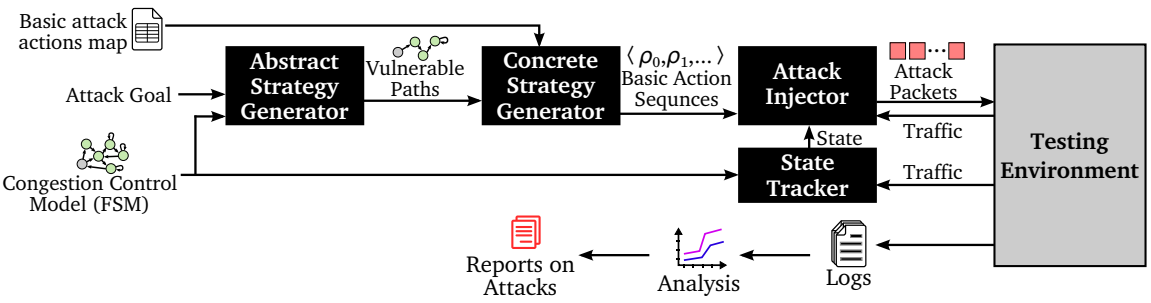


Figure 5.2.: Design of TCPwn

Finding all these transitions (*i.e.,* that impact the sending rate at runtime) is challenging because of the large search space. We address this challenge by using a model-based attack strategy generation algorithm that finds *all possible attack strategies* in a model of congestion control (shown in Figure 2.3). We refer to these as *abstract strategies*. To test them in real implementations, we translate them to *concrete attack strategies*, obtained by mapping the abstract strategies to attack actions corresponding to attacker capabilities and consisting of specific content for a malicious packet and the state in which it will be injected. An attack injector takes these concrete packet-based attack strategies and injects them in our testing environment during an actual execution of the target implementation. Our attack injector requires information about the current congestion control state of the sender. A state tracker determines this current protocol state so that actions can be performed as specified by the strategy. After the execution of each attack, our system collects logs that capture performance metric(s). By comparing the resulting performance with the expected baseline performance, TCPwn identifies whether the strategy indeed leads to a successful attack. Figure 5.2 shows the conceptual design of our system.

Testing strategies with real implementations provides strong soundness properties since any strategy that TCPwn identifies as an attack caused noticeable performance changes in a real TCP connection of the implementation under test. This prevents most classes of false positives, except tests with performance outside of the considered normal range ($>$ 2 standard deviations from average). Our completeness is limited

by the accuracy of the congestion control model and state tracking. Here, we choose to trade off some completeness for the ability to test many implementations and use a generalized congestion control model and inferred state tracking.

Example for TCPwn attack generation. We demonstrate this attack strategy generation approach using the same Optimistic Ack Attack example as above, where the attacker's goal is to increase the sending rate; this can also be expressed as an increase in the sender's `cwnd` variable. Our abstract strategy generator identifies each of the paths in the FSM (Figure 2.3) containing at least one transition that increments the `cwnd` variable. One of the identified paths (say, $\mathcal{P}$) looks as follows:

$\mathcal{P}$: SlowStart $\rightarrow$ FastRecovery $\rightarrow$ CongestionAvoidance $\circlearrowright$

where the self-loop in CongestionAvoidance increments `cwnd` (see Figure 2.3). An *abstract strategy* $\mathbb{S}$ is a projection on the condition of each transition along $\mathcal{P}$ and is represented as the following sequence of (*state*, *condition*) pairs:

(**In**: SlowStart, **Condition**: `ACK && Dup && dupACKctr`$\geq$`3`)

(**In**: FastRecovery, **Condition**: `ACK && New && pkt.ack` $\geq$ `high_water`)

(**In**: CongestionAvoidance, **Condition**: `ACK && New`)$^+$

This strategy $\mathbb{S}$ dictates that when the sender is in SlowStart and is sending data to the receiver, the attacker can send 3 duplicate `ACK`s to the sender so that it moves to FastRecovery. Next the attacker can send the sender 1 new `ACK` (that acknowledges all the outstanding data). As a result, the sender moves to CongestionAvoidance, and the attacker can keep on sending new `ACK`s that optimistically acknowledge all outstanding data even if the receiver has not received it yet. $+$ (the superscript) signifies that the attacker can apply this segment of $\mathbb{S}$ repeatedly.

TCPwn maps $\mathbb{S}$ to several concrete strategies that can be directly tested inside the testing environment running the given implementation. TCPwn relies on a map which associates the abstract network conditions to concrete *basic actions*. For $\mathbb{S}$, TCPwn generates 72 concrete strategies, based on actions mimicking both off-path and on-path attackers. One such concrete strategy is:

(**In**: SlowStart, **Action**: 3 × Inject `Dup-Ack`)

(**In**: FastRecovery, **Action**: Inject `Pre-Ack`)

(**In**: CongestionAvoidance, **Action**: Inject `Pre-Ack`)$^+$

This concrete strategy dictates that when the sender is in SlowStart, the attacker can use the `Dup-Ack` basic action to inject 3 duplicate `ACK`s. Similarly, for acknowledging all the outstanding data in the next step, the attacker can use the `Pre-Ack` basic action. Once the sender is in CongestionAvoidance, the attacker can repeatedly apply `Pre-Ack`. We will describe all supported basic actions in Section 5.3.3.

5.3.2 Abstract Strategy Generation

We now describe in detail the core of our approach. We observe that a successful attack will (1) trigger a transition that causes an increase or decrease in the congestion window `cwnd` and (2) traverses a cycle in the congestion control state machine.

Changes to `cwnd`. The congestion window, `cwnd`, adjusts the sending rate of TCP to avoid congestion collapse and provide fairness.[2] This variable controls the amount of data allowed in the network at any given time, which directly corresponds to TCP's allowed sending rate. As a result, any attack on congestion control will have to impact this variable to have any impact on the network traffic. There may be attacks on TCP that do not manipulate this variable, but these are not attacks on TCP's congestion control.

Further, congestion control modifies `cwnd` frequently during the course of its normal operation. These modifications are done on many transitions of the congestion control state machine and either increase or decrease `cwnd` depending on the transition. As a result, an attacker can increase or decrease `cwnd`, and therefore TCP's

[2]This is true for all congestion control algorithms except Google's new BBR [104] congestion control. This includes Reno [35], New Reno [36], CUBIC [42], Compound TCP [105], and Vegas [106], among others. However, BBR does maintain a variable containing the explicitly computed allowed sending rate, which has similar properties for our purposes. As BBR's public release was concurrent with this work, we do not consider it further here.

sending rate, merely by inducing TCP to follow specific normal transitions in the congestion control state machine.

State Machine Cycles. Successful congestion control attacks traverse a cycle in the congestion control state machine. This is due to the highly dynamic and cyclical nature of congestion control where a sender often traverses the same set of states many times over the course of a connection and multiple state transitions in a single second are common. As a result, the impact on `cwnd` from a single transition is quickly diminished by other transitions. For an attack to be effective and achieve measurable, lasting impact, an attacker has to frequently induce TCP to follow some desirable transition. Such a series of desirable transitions will form either a cycle or a unique path in the state machine. Given the relatively small size (under 10 states) of the congestion control state machine and the frequency of state transitions, anything but the shortest connections would require a cycle to achieve a sufficiently long series of desirable transitions.

Note that these characteristics are necessary but not sufficient for an attack on congestion control. For instance a cycle may contain two manipulations to `cwnd` that balance each other out, or a cycle may not be triggerable by the attacker.

Our abstract strategy generator takes as input an FSM model of congestion control and a description of the desirable transitions. In our case, a desirable transition is one that modifies `cwnd`. It outputs a list of all paths with cycles that contain a desirable transition and can therefore be used by an attacker to achieve his goal. This list includes the transitions in each path as well as the conditions that cause each transition. We use a modified depth-first traversal to enumerate all paths in the FSM. We formally define the abstract strategy generation problem and our algorithm below.

State Machine Model. We define a model $\mathcal{M}$ describing the state machine of the congestion control algorithm as a tuple $(\mathcal{S}, \mathcal{N}, \mathcal{V}, \mathcal{C}, \mathcal{A}, \sigma, \mathcal{T})$. $\mathcal{S}$ is a finite set of states $\{s_0, \ldots, s_n\}$, and the initial state is $\sigma \in \mathcal{S}$. $\mathcal{N}$ represents a finite set of network events (*e.g.,* `ACK` signifies the reception of a TCP acknowledgment). $\mathcal{V}$ is a finite

set of variables including both some fields of a received packet and some program variables. For instance, `New` means the received `ACK` acknowledges some new data and `cwnd` indicates the program variable that represents congestion window size. $\mathcal{C}$ represents a finite set of conditional statements such that each element $c \in \mathcal{C}$ is a quantifier-free first order logic (QF-FOL) formula [107] over $\mathcal{V}$ (*e.g.,* `dupAckCtr` < 2). $\mathcal{A}$ represents a finite set of assignment statements (*i.e.,* protocol actions) over a subset of $\mathcal{V}$ (*e.g.,* "`cwnd = 1`" means the congestion window is set to 1). In addition, $\mathcal{N}$, $\mathcal{V}$, $\mathcal{C}$, and $\mathcal{A}$ are pairwise disjoint. $\mathcal{T}$ represents the transition relations such that $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{N} \times \mathcal{C} \times 2^{\mathcal{A}} \times \mathcal{S}$.

Let $\psi : \mathcal{T} \mapsto \mathcal{S}$ and $\xi : \mathcal{T} \mapsto \mathcal{S}$ be two maps indicating the *source* and *target* of a transition. For example, if a transition $t \in \mathcal{T}$ begins at s_b and ends at s_e, then $\psi(t) = s_b$ and $\xi(t) = s_e$. Let $\lambda : \mathcal{T} \mapsto \mathcal{N} \times \mathcal{C}$ and $\aleph : \mathcal{T} \mapsto \mathbb{P}(\mathcal{A})$ be two maps to indicate the triggering conditions and the set of actions of a transition, respectively. Now we define a path as follows.

Definition 5.3.1 *Path: A* **path** *P in $\mathcal{M}$ is a sequence of pairs of states and transitions $\langle (s_{i_0}, t_{j_0}), (s_{i_1}, t_{j_1}), \ldots, (s_{i_k}, t_{j_k}) \rangle$, where $k \geq 0$; each $s_{i_x} \in \mathcal{S}$ for $0 \leq x \leq k$ and $s_{i_0} = \sigma$ (the initial state); $\forall y\ [t_{j_y} \in \mathcal{T} \wedge \psi(t_{j_y}) = s_{i_y} \wedge \xi(t_{j_y}) = s_{i_{(y+1)}}]$ where $0 \leq y \leq k-1$; $t_{j_k} \in \{\mathcal{T}, \perp\}$ and $[\psi(t_{j_k}) = s_{i_k} \wedge \xi(t_{j_k}) \in \{\mathcal{S}, \perp\}]$. In addition, $\forall r, s[r \neq s \rightarrow s_{i_r} \neq s_{i_s} \wedge t_{j_r} \neq t_{j_s}]$, where $r, s \in \{0, 1, \ldots, k\}$.*

In other words, a path P starts at σ and moves to the state s_{i_1} by taking the transition t_{j_0}. By following the sequence, P finally reaches at s_{i_k}. The last segment of P (*i.e.,* (s_{i_k}, t_{j_k})) is special as it determines the existence of a cycle. If P contains a **cycle**, then $[t_{j_k} \neq \perp \wedge t_{j_k} \in \mathcal{T}]$ and $\exists z[\xi(t_{i_k}) = s_{i_z}]$, where $z \in \{0, 1, \ldots, k\}$. When P has no cycle, $t_{j_k} = \perp$ and $\xi(t_{j_k}) = \perp$.

Definition 5.3.2 *Vulnerable path: Given a vulnerable action $\alpha \in \mathcal{A}$, a path P in $\mathcal{M}$ is a* **vulnerable path** *if P has a segment (s_{i_x}, t_{j_x}) such that $\alpha \in \aleph(t_{j_x})$, where $x \in \{0, \ldots, k\}$ and $k \geq 0$.*

Definition 5.3.3 *Abstract strategy: Given a vulnerable path P in $\mathcal{M}$ such that $P = \langle (s_{i_0}, t_{j_0}), \ldots, (s_{i_k}, t_{j_k}) \rangle$ for some $k \geq 0$, the corresponding* ***abstract strategy*** $\mathbb{S}$ *is defined as $\langle (s_{i_0}, \lambda(t_{j_0})), (s_{i_1}, \lambda(t_{j_1})), \ldots, (s_{i_k}, \lambda(t_{j_k})) \rangle$, where $\lambda(t_{j_x}) \in (\mathcal{N} \times \mathcal{C})$ if $t_{j_x} \in \mathcal{T}$ or $\lambda(t_{j_x}) = \perp$ if $t_{j_x} = \perp$ for each $0 \leq x \leq k$.*

Abstract Strategy Generator. Given $\mathcal{M}$, a directed *multigraph*[3] with cycles, and the attacker's goal $\alpha \in \mathcal{A}$, the Abstract Strategy Generator aims to find all the vulnerable paths in $\mathcal{M}$ with respect to α. We devise the algorithm shown in Algorithm 2, which begins the search from the function `VulnerablePathFinder`. Intuitively, the algorithm traverses the entire graph in a depth-first fashion, starting at the initial state $\sigma \in \mathcal{S}$. For each transition $t \in \mathcal{T}$ such that $\psi(t) = \sigma$, the algorithm initializes a new path P, appends (σ, t) to P, and recursively continues its exploration of the subgraph rooted at $\xi(t)$. For P, the recursion stops when it encounters a cycle (line 13) or a terminating state (line 15). If any of these stop conditions is met, the algorithm checks if P is a vulnerable path with respect to α; if so, it adds P to the set of the vulnerable paths (line 20). Unlike traditional depth-first traversal, the algorithm restores the subgraph rooted at $\xi(t)$ by marking it unvisited (line 28) in order to find all possible vulnerable paths w.r.t. α. Upon termination, the algorithm returns the set of vulnerable paths w.r.t. α (line 10) identified during the exploration. This set of vulnerable paths contain our abstract strategies. We generate our abstract strategies $\{\mathbb{S}\}$ by taking projections on the conditions of the transitions along each path.

5.3.3 Concrete Strategy Generation

An abstract strategy just specifies a path in the FSM that can lead to an attack. However, there are usually several ways in which this path can be concretely achieved at runtime. Concrete strategy generation takes our abstract strategies and converts

[3]A multigraph permits multiple edges between a pair of vertices

Algorithm 2: TCPwn Abstract Strategy Generator

Input: Multigraph $\mathcal{M} = (\mathcal{S}, \mathcal{N}, \mathcal{V}, \mathcal{C}, \mathcal{A}, \sigma, \mathcal{T})$, ψ, ξ, λ, $\aleph$ and a vulnerable action $\alpha \in \mathcal{A}$

Output: All vulnerable paths with respect to α

```
1  VulnerablePaths ≔ ∅                                  /* to store vulnerable paths */
2  Function VulnerablePathFinder(M, α)
3      root ≔ σ
4      Mark root as visited
5      foreach transition t such that ψ(t) = root do
6          Create a new path P
7          P ≔ P‖(root, t)                              /* concatenating */
8          v ≔ ξ(t)
9          RecursiveSearch(v, P, α)
10     return VulnerablePaths

11 Function RecursiveSearch(v, P, α)
12     base_case ≔ false
13     if v is already visited then                     /* reached a cycle */
14         base_case ≔ true
15     else if exists no t such that ψ(t) = v then
                                                        /* v is a terminating state */
16         base_case ≔ true
17         P ≔ P‖(v, ⊥)                                 /* concatenating */
18     if base_case is true then
19         if P is a vulnerable path w.r.t. α then
20             VulnerablePaths ≔ VulnerablePaths ∪ P
21     else
22         Mark v as visited
23         foreach transition t such that ψ(t) = v do
24             v′ ≔ ξ(t)
25             P′ ≔ P                                   /* creating a copy */
26             P ≔ P‖(v′, t)                            /* concatenating */
27             RecursiveSearch(v′, P′, α)
28         Mark v as unvisited
29     return
```

them into sets of basic message-based actions that can be applied by our attack injector in particular states of the FSM.

Our concrete strategy generator considers each abstract strategy individually and iterates through each transition in that strategy. Each of these network conditions is *mapped* to a basic action that the attacker can directly utilize to trigger that network condition in that state. This results in a set of $(state, action)$ pairs which we call a *concrete strategy*. A transition condition may be triggered by multiple basic actions, in which case this mapping results in a set of basic actions that could be applied in that state to cause the next transition. Our generator creates one concrete strategy for each combination of actions from these sets. Note that we require a domain expert to provide the mapping of network conditions to basic actions since it relies on domain knowledge. This mapping only needs to be updated when the state machine model changes or new actions are added; generating concrete actions for a given implementation is completely automated.

We developed our set of basic actions based on an extensive study of TCP and known congestion control attacks. We also sought to restrict the information required by our attack injector primarily to message format and current congestion control state information, for practicality. We consider two categories: injection of acknowledgements, which captures the capabilities of an *off-path attacker*, and modification of acknowledgements, which captures the capabilities of an *on-path attacker*.

Injection of acknowledgements (off-path attacker). This type of action injects new spoofed acknowledgement packets for either the client or server of a target connection. Since congestion control algorithms usually rely on acknowledgements to indicate lost packets and to gradually increase the sending rate, injecting additional acknowledgements may cause significant issues for congestion control at fairly low cost to an attacker. This type of action parallels the capabilities of off-path attackers. We support a number of different ways of injecting acknowledgements:

(1) Duplicate Acknowledgements (param: dup_no, delay, offset) — Injecting many acknowledgements with the same acknowledgement number as an apparent set of du-

plicate acknowledgements. This enables an off-path attacker to slow down a connection. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of duplicates injected (2, 10, 1000), the spacing between these duplicates (1ms), and offset from the current acknowledgement number (0, 3000, 90000).

(2) Offset Acknowledgements (param: num, delay, data, offset) – Injecting a series of acknowledgements with an acknowledgement number offset from the legitimate acknowledgement number. Acknowledges either less or more data than is acknowledged by the receiver. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of acknowledgements injected (10000, 50000), the spacing between these acknowledgements (1ms, 2ms), any bytes of data included (0, 10), and any offset from the current acknowledgement number (0, 100, 3000, 9000, 90000).

(3) Incrementing Acknowledgements (param: num, delay, data) — Injecting a series of acknowledgements where the acknowledgement number increases by a variable amount each time. Congestion control expects these acknowledgements to indicate the successful receipt of new data and will act accordingly. This action assumes that target connection's sequence and acknowledgement numbers are known or can be guessed. Parameters control the number of acknowledgements injected (50000), the spacing between these acknowledgements (1ms), and the amount the acknowledgement number is incremented with each packet (9000, 90000).

Modification of acknowledgements (on-path attacker). This type of action changes the manner in which acknowledgements for the sequence space are sent. To do so, it requires an on-path attacker because the genuine acknowledgement packets need to be modified. This action leverages the key role that acknowledgements play in TCP congestion control. Not only are they used to determine loss via duplicate acknowledgements, but they are used to clock out new data packets (the "conservation of packets" principle [93]) and increase the sending rate. We support a number of manipulations to the sequence of acknowledgements for a data stream:

(1) Division (param: chunk_size) — Acknowledge the sequence space in chunks much smaller than a single packet. This splits a single acknowledgement packet into many acknowledgement packets that acknowledge separate ranges. A parameter controls the number of bytes to acknowledge in a single chunk (100). This technique has been known to cause significant and unfair increases in sending rate with overly-trusting senders who assume that one acknowledgement represents one packet [11].

(2) Duplication (param: dup_no) — Duplicate acknowledgements of chunks of the sequence space repeatedly. A parameter controls the number of duplicate acknowledgements to create (1, 4, 100). This breaks the assumption that each acknowledgement received corresponds to a packet that left the network. The Dup Ack Attack leverages this assumption during Fast Recovery to trick the sender into sending new data packets at the same rate as incoming duplicate acknowledgements [11].

(3) Pre-acknowledging (param: none) — Acknowledging portions of the sequence space that have not been received yet. This hides any losses, preventing slow downs, and effectively shrinks the connection's RTT, allowing faster than normal throughput increases. This is referred to as the Optimistic Ack Attack [11].

(4) Limiting (param: none) — Prevents the acknowledgement number from increasing. This generates duplicate acknowledgements but also prevents any new data from being acknowledged. This is likely to stall the connection and lead to an RTO.

5.3.4 State Tracker

In order to test a strategy against an implementation, TCPwn needs to know the state of the sender with respect to congestion control. This is not an easy problem as there are several implemented congestion control algorithms such as Reno [35], New Reno [36], CUBIC [42], Compound TCP [105], and Vegas [106]. Implementations may also choose to include an Application Limited state, adjustable `dupACKctr` thresholds, and optional enhancements like SACK [37], DSACK [38], TLP [39], F-RTO [41], and PRR [40]. Additionally, we desire to do this *without* modifying the sender or

making assumptions about what kind of debugging information it makes available. Finally, key variables that determine the state of the sender (like `cwnd`, `ssthresh`, and `rto_timeout`) are not exposed by the sender and are not readily computable from network traffic. Further, the observed behavior of an implementation depends not only on events observable from the network but also on internal events like the fullness of the driving application's buffer. As one example, it is impossible, by looking only at the network traffic, to distinguish an RTO event from an idle application that suddenly decided to send a single packet of data.

To overcome these challenges, we choose to *approximate* the congestion control state machine by focusing on its core states and *assume* a bulk transfer application that always has data available to send. This is practical because nearly all TCP congestion control algorithms contain the same basic core set of states from TCP New Reno (see Figure 2.3) with the differences being in terms of small changes in the actions done on each transition or the insertion of extra states. For example, CUBIC TCP simply modifies the additive increase and multiplicative decrease constants on the transitions to Fast Recovery and Congestion Avoidance. Similarly, TLP adds a single state before Exponential Backoff. It is entered using a slightly smaller timeout and sends a single new packet to try and avoid an expensive RTO. Assuming a bulk transfer application enables us to make assumptions about application behavior when needed.

We developed the novel algorithm shown in Algorithm 3 to track the sender's congestion control state using only network traffic. We find that this algorithm works well even when used with implementations containing complex state machines and enhancements that we approximate using only TCP New Reno. Our algorithm detects the Fast Recovery state even when the `cwnd` reduction is CUBIC's 0.8 factor and not the 0.5 used by New Reno. It still identifies retransmitted packets and enters Fast Recovery even if SACK is in use and Fast Recovery was triggered via SACK blocks. TLP is a case where our approximation fails, but even here we misclassify a tail-loss-

Algorithm 3: State Tracking for TCPwn

```
1  Function Init()
2      Start timer intervalTimer to expire every sub_rtt ms (10ms)
3      priorPkt = curPkt = now()
4      urgEvent = false
5      state = UNKNOWN
6  Function OnPacket(p)
7      update dataBytes, dataPkts, ackBytes, ackPkts, seqHigh, highAck, curPktType and
         rexmits based on p
8      if curPkt < now()− max_burst_gap then
9          lastIdle = now()
10     priorPkt = curPkt
11     curPkt = now()
12     Reset timer packetTimer to expire in max_burst_gap ms (5ms)
13     if rexmits > 0 then
14         urgEvent = true
15         Reset timer packetTimer to expire now
16 Function OnTimer()
17     if urgEvent or curPkt >max_burst_gap or lastIdle > 4∗sub_rtt then
18         urgEvent = false
19         if curPktType is SYN then
20             state = INIT
21             return
22         if curPktType is FIN or RST then
23             state = END
24             return
25         curRatio = dataBytes / ackBytes
26         pktSpace = curPkt − priorPkt
27         if dataPkts > 0 and (pktSpace > 200ms) then
28             state = EXP_BACKOFF
29         else if state == FAST_RECOV and ackHigh < ackHold then
30             state = FAST_RECOV
31         else if rexmits > 0 or (ackBytes == 0 and ackPkts > 3) then
32             ackHold = seqHigh
33             state = FAST_RECOV
34         else if (curRatio + priorRatio)/2 > 1.8 then
35             state = SLOW_START
36         else if (curRatio + priorRatio)/2 > 0.8 then
37             state = CONG_AVOID
38         else if state == EXP_BACKOFF and curRatio < 0.1 then
39             ackPkts = 0
40         else
41             priorRatio = 0.8 ∗ curRatio + 0.2 ∗ priorRatio
42             return
43         priorRatio = curRatio
44         ackPkts = ackBytes = dataPkts = dataBytes = rexmits = 0
```

probe as an RTO. This is only a minor issue because both states are entered via by timeouts and trigger the transmission of a single packet.

The core idea of our algorithm is to take a small (sub-RTT) time slice and observe the packets received and sent by an implementation. If about twice as many bytes of data have been sent as acknowledged, the state is inferred to be Slow Start and the sending rate is increasing exponentially. If about an equal number of bytes have been sent and acknowledged, the state is inferred to be Congestion Avoidance since the sender is maintaining a steady sending rate. If fewer bytes have been sent than acknowledged or there are retransmitted packets, the state is inferred to be Fast Recovery, and if no packets are received and only a few packets are sent, then an RTO event was observed and the sender is in state Exponential Backoff.

Our algorithm uses two timers, the first fires every `sub_rtt` seconds and the second fires `max_burst_gap` seconds after each packet unless reset. This first timer handles the case where TCP is operating at high speed and has packets in flight constantly while the second handles the case where TCP has not yet reached peak efficiency and is sending packets in bursts and then waiting for their acknowledgements before sending more. We experimentally set `sub_rtt` to 10ms and `max_burst_gap` to 5ms based on a network with an RTT of around 20ms.

Whenever either of these timers expires, the algorithm determines whether TCP is sending data smoothly or in bursts. If TCP is sending data in bursts and it has been less than `max_burst_gap` seconds since the last packet, this timer expiration is ignored. Otherwise, the state inference is updated. If the most recent packet was a SYN, FIN, or Reset, then the connection state is INIT or END. Otherwise, we compute the ratio of sent to acknowledged data and the space between the two most recent packets, and use this information to determine what state the sender is in based on the intuition presented above. We then reset our data sent and data acknowledged counters. For the Slow Start and Congestion Avoidance state, we average the ratios from the last two sampling periods as we found experimentally that this helped to

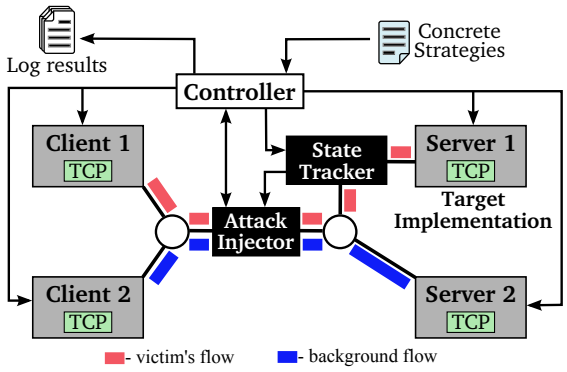


Figure 5.3.: TCPwn Testing Environment

produce more accurate results. Finally, if the ratio is less than 0.8, a situation that should never occur, we ignore this sample and do not reset our counters.

5.4 Implementation

This section discusses how we implemented TCPwn and apply it to real implementations of TCP to identify attacks.

TCPwn is able to test real implementations easily, despite diverse programming languages, operating systems, hardware support, and libraries. Given the different variants of TCP congestion control algorithms, features, and optimizations [37–43] any implementation has to make choices about what configuration and combination of features will be provided. This leads to minor differences in congestion control behavior between implementations which can enable or prevent particular attacks or even attack classes. Further, TCP implementations are typically written as part of the operating system in a low level language like C and highly optimized for performance, leading to an increased probability of implementation bugs.

5.4.1 Testing Environment

We developed a testing environment (Figure 5.3) which leverages virtualization for both client and server, enabling us to run a wide range of implementations, independent of operating system, programming language, libraries, or availability of source code.

We connect four virtual hosts into a dumbbell topology with two clients on one side, two servers on the other, and a single bottleneck link between. When each client connects to one of the servers, this topology provides an environment where two flows have to compete for bandwidth on the bottleneck link. This competition is precisely what an attacker must influence in order to either increase or decrease the throughput of his target flow. We connect the virtual machines together with Linux tap devices and bridges. We artificially cap the bandwidth on the bottleneck link and introduce a 10ms delay, using Linux traffic control. This gives us a virtual network based on the widely used Linux networking stack that supports throughput in excess of 800Mbits/sec.

One of the servers runs the target TCP implementation under test. The other hosts run a standard TCP stack and serves simply to complete the test harness and generate necessary traffic. To generate traffic, our tests use file transfers over HTTP. This simplifies setting up the target implementation, as HTTP servers are available for a wide variety of operating systems and implementations.

The Attack Injector is implemented as a proxy placed in the middle of the bottleneck link. It intercepts all packets in the target connection and applies any on-path basic actions. It can also inject new packets into the network to emulate an off-path attacker. The proxy also measures connection length and amount of data transferred for attack detection and is implemented in C++.

The State Tracker component is also implemented as a proxy and is placed in our testing environment as near to the target sender as possible. This proxy observes the packets sent and received by the sender over small timeslices to automatically

infer the current state of the sender's TCP congestion control state machine. We experimentally set `sub_rtt` to 10ms and `max_burst_gap` to 5ms based on a network with an RTT of around 20ms. This proxy is also implemented in C++.

This whole environment is controlled and coordinated by a Controller script that takes a concrete strategy from our strategy generator, orchestrates the virtual machines, applications, Attack Injector, and State Tracker components to test that strategy, collects the results, and returns them for analysis.

5.4.2 Attack Detection

The goal of an attacker targeting congestion control is to impact throughput. We distinguish between four cases for a target connection that are the observable outcome of an attack:

- *Benign*: no attack occurs.
- *Faster*: the sender sends at a faster rate that is should; throughput is larger than the benign case; this corresponds to a sender bypassing congestion control to send faster.
- *Slower*: the sender is made to send at a slower rate that what the network conditions will allow; the throughput is smaller than a benign connection.
- *Stall*: the connection has stalled and will never complete; this corresponds to the case where the attacker made the connection to stall.

Measuring the time it takes to transfer a file at the application layer is not sufficient because it does not allow us to distinguish between two cases: sending faster or connection stalled. Both appear, in some cases, as stalled because the TCP receiver has blocked reassembling data, while all data has already been sent. Thus, the first metric we use is the time it takes to transfer and acknowledge all data packets at the TCP level, referred to simply as $Time$ below.[4] Thus, we measure the time from when

[4]This definition explicitly excludes trailing RTO retransmissions that are never acknowledged.

the SYN packet is sent to the last set of data carrying packets that arrive within a second of each other.

The time needed to transfer the data at the TCP level is not sufficient to accurately classify attacks because it does not capture the case when the connection stalls out part way through due to an attack and the file has actually not been transferred in its entirety. To detect this case, we use a second metric, the amount of data transferred in the connection at the TCP-level, referred to as $SentData$ below.

We perform 20 tests transferring a file of size $FileSize$ without any attacks being injected to create baseline average and standard deviation values of $TimeBenign$ and $stddev$. Then, using the $Time$ and $SentData$ metrics defined above, our detection works as follows:

```
if Time is > (TimeBenign + 2*stddev):
    Attack: Slower
else if Time is < (TimeBenign + 2*stddev):
    if SentData >= (0.8*FileSize):
        Attack: Faster
    else:
        Attack: Stall
else:
    Benign
```

5.5 Results

We tested five different implementations of TCP in five operating systems: Ubuntu 16.10, Ubuntu 14.04, Ubuntu 11.10, Debian 2, and Windows 8.1. The tests were run on a hyperthreaded 20 core Intel® Xeon® 2.4GHz system with 125GB of RAM. We configured the bottleneck link to be 100Mbits/sec, with a 20ms RTT, and generated traffic for both the target and competing TCP connections with a 100MB HTTP file download for all implementations except Debian 2. Due to limitations with the virtualized NIC, Debian 2 was limited to 10Mbits/sec, so we also limited the bottleneck

Table 5.1.: Summary of TCPwn Results

Implementation	Attacker	Tested	Marked	FP	Attacks
Ubuntu 16.10 (Linux 4.8)	On-path	564	38	3	35
Ubuntu 14.04 (Linux 3.13)	On-path	564	37	1	36
Ubuntu 11.10 (Linux 3.0)	On-path	564	16	6	10
Debian 2 (Linux 2.0)	On-path	564	3	0	3
Windows 8.1	On-path	564	9	1	8
Ubuntu 16.10 (Linux 4.8)	Off-path	753	466	8	458
Ubuntu 14.04 (Linux 3.13)	Off-path	753	448	9	439
Ubuntu 11.10 (Linux 3.0)	Off-path	753	564	10	554
Debian 2 (Linux 2.0)	Off-path	753	425	0	425
Windows 8.1	Off-path	753	471	3	468
Total		6585	2477	41	2436

link to that same rate with a 20ms RTT while traffic generation used a 10MB file. We used the Apache webserver for Linux and IIS on Windows.

Testing each implementation took about 13 hours for the on-path testing and 21 hours for the off-path testing, using only 6 cores. Testing each strategy is independent and takes between 15 and 60 seconds. With 48 cores running eight testing environments (each needs 6 cores), the on-path testing could have been completed in 1.6 hours and the off-path testing in 2.6 hours.

Over all the tested systems, we tested 6,585 strategies and found 2,436 attacks, which we classified into 11 classes. 8 of these classes were previously unknown in the literature. We summarize the attacks in Tables 5.1 and 5.2.

While this analysis was performed manually, we observe that it is amenable to automation. In our results, three classes of attacks—Optimistic Ack, Desync, and Ack Lost Data—make up the majority of marked strategies. An automated classification of these three categories leaves only 281 (11%) strategies to manually examine.

Table 5.2.: Classes of Attacks Discovered by TCPwn

Num	Attack	Attacker	Description	Impact	Impl	New
1	Optimistic Ack	On-path	Acking data that has not been received	Increased Throughput	ALL	No [11]
2	On-path Repeated Slow Start	On-path	Repeated cycle of Slow Start, RTO, Slow Start due to fixed ack number during Fast Recovery	Increased Throughput	U16.10, U11.10	Yes
3	Amplified Bursts	On-path	Send acks in bursts, amplifying the bursty nature of TCP	Increased Throughput	U11.10	Yes
4	Desync Attack	Off-path	Inject data to desynchronize sequence numbers and stall connection	Connection Stall	ALL	No [99]
5	Ack Storm Attack	Off-path	Inject data into both sides of connection, creating ack loop	Connection Stall	D2, W8.1	No [100]
6	Ack Lost Data	Off-path	Acknowledge lost data during Fast Recovery or Slow Start	Connection Stall	ALL	Yes
7	Slow Injected Acks	Off-path	Inject acks for little data slowly during Congestion Avoidance	Decreased Throughput	U11.10	Yes
8	Sawtooth Ack	Off-path	Send incrementing acks in Congestion Avoidance/Fast Recovery, but reset on entry	Decreased Throughput	U16.10, U14.04, U11.10, W8.1	Yes
9	Dup Ack Injection	Off-path	Inject >= 3 duplicate acks repeatedly	Decreased Throughput	D2, W8.1	Yes
10	Ack Amplification	Off-path	Inject acks for lots of new data very rapidly during Congestion Avoidance or Slow Start	Increased Throughput	U16.10, U14.04, U11.10, W8.1	Yes
11	Off-path Repeated Slow Start	Off-path	Repeated cycle of Slow Start, RTO, Slow Start due to increased duplicate ack threshold	Increased Throughput	U11.10	Yes

5.5.1 On-path Attacks

We only consider attacks resulting in increased throughput for some target connection to be of interest to this attacker. Our model-guided strategy generation produced 564 strategies based on the basic actions described in Section 5.3.3. As shown in Table 5.1, our system marked between 3 and 38 of these strategies (depending on implementation). A few of these marked attacks were false positives, due to the imprecision of testing with a real network and real implementations. In particular, while our target connection typically incurs its first loss within 0.5 seconds of starting, due to competing with the background connection, in these false positive tests the first loss in the target connection does not occur until after at least 1.5 seconds. Since TCP continues to increase its sending rate until it gets a loss, this results in an unusually high sending rate. This longer time to loss is not attributable to any basic action applied, but is simply a result of variations in packet arrival and departure times, packet processing delays, operating system scheduling, and other random variations. The remaining marked strategies are real attacks against a TCP implementation. We identified between 3 and 36 of these, depending on the implementation. Through manual analysis, we grouped these into 3 classes (Table 5.2), two of which are previously unknown in the literature.

On-path Repeated Slow Start (new). These attacks operate by repeatedly inducing an RTO followed by Slow Start. Thanks to Linux's choice to use a short RTO timer, the rapid increase in sending rate during Slow Start balances out the idle period needed to cause an RTO and in many tests actually provides a higher average sending rate. This is partly due to the significant impact this attack has on competing connections because of the repeated, rapid sending periods that end in a loss for both connections. These repeated losses cause the competing connection to slow down repeatedly. We found this attack class against both Ubuntu 11.10 and Ubuntu 16.10. For both implementations, this behavior is best induced by preventing

an increase of the cumulative acknowledgement in Fast Recovery, preventing recovery of losses and causing an RTO. We believe this attack to be unknown in the literature.

Amplified Bursts (new). This class of attack operates by collecting acknowledgement packets and then sending them together in a burst. This additional burstiness often causes more frequent losses in the competing connection which causes it to slow down and our target flow to increase its throughput. We found this attack class against Ubuntu 11.10 with a strategy that collected acknowledgement packets to send them in bursts during Congestion Avoidance and optimistically acknowledged data during Slow Start, increasing the size of `cwnd`. It is interesting to note that without our model-guided strategy generation we would have been extremely unlikely to find this attack. This is because delaying acknowledgements and sending them in bursts is only a good idea during Congestion Avoidance. During Slow Start, `cwnd` is small enough that there may not be enough acknowledgements in flight to make a single burst, leading to a connection stall. Similarly, in Fast Recovery, the sender needs to get acknowledgements as soon as possible so that it can recover from the loss and keep sending data. Delaying acknowledgements and collecting enough for a single burst tends to cause the connection to stall.

This attack bears significant resemblance to the Induced-Shrew Attack [8]. However, that attack seeks to manipulate a TCP connection to cause catastrophic throughput reduction on other competing connections while maintaining a minimal sending rate itself. Instead, the Amplified Burst attack focuses on increasing the throughput of our target connection; we believe this attack to be previously unknown in the literature.

Optimistic Ack (known). This class of attack operates by optimistically acknowledging data that the receiver has not received and acknowledged yet. This reduces the effective RTT of the connection, allowing TCP to increase its sending rate faster, and hides lost packets, preventing TCP from slowing down in response to congestion. By hiding lost packets, the receiver will not receive the complete data transfer, but this may be acceptable if the data stream can tolerate losses or if the

attacker does not care about the data, *i.e.,* is simply conducting a denial of service attack.

This attack class was first identified in [11]. Unfortunately, the mitigations proposed require non-backwards-compatible modifications to TCP, such as inserting a random nonce into each packet. As a result, this attack class is still present in modern TCP implementations, and we found many instances of it in all 5 of the implementations we tested. In our tests, this attack usually caused the target connection to consume all available bandwidth up to the network and/or sending system capacity. This left the competing connection starved for bandwidth, often doing repeated RTOs, and with throughput near zero for the duration of the attack.

5.5.2 Off-path Attacks

An off-path attacker can observe network traffic but cannot directly modify such traffic. As a result, they are limited to injecting new (possibly spoofed) packets into the network. In addition to increasing throughput, possibly as part of a denial of service attack, an off-path attacker might be interested in decreasing the throughput or stalling some target connection.

Our model-guided strategy generation produced 753 strategies based on injecting spoofed packets. As shown in Table 5.1, our system marked between 425 and 564 of these strategies (depending on implementation) as attacks. A few of these marked attacks turned out to be false positives. These are mostly cases where, due to imprecision from testing real implementations, the target connection does not see its first loss for an abnormally long time, leading to a higher sending rate than normal. We present a summary of the attack classes found in Table 5.2.

Ack Lost Data (new). This class of attacks contains a wide range of operations that cause lost data to be perceived as acknowledged at some point in the connection. This occurs when an attacker injects a spoofed acknowledgement packet acknowledging data above the current cumulative acknowledgement when the network is about

to enter Fast Recovery. In this case, at least some of the lost data will be deemed acknowledged by the victim, causing that data to never be retransmitted. At this point, anything the sender retransmits or sends will not cause the receiver to increase the cumulative acknowledgement and the connection permanently stalls. We found a wide variety of strategies in this attack class against all implementations we tested.

Slow Injected Acks (new). These attacks operate by injecting spoofed acknowledgements that increase their acknowledgment number at a slow and constant rate. As these acknowledgement packets are injected, each one causes TCP to send a few packets—equivalent to the amount of data acknowledged—, due to TCP's self-clocking design. This essentially causes TCP to bypass congestion control and *cwnd* entirely and send at the rate at which the spoofed acknowledgements are acknowledging data: $ack\_amount * injection\_frequency$. This rate can be made much slower than TCP would otherwise achieve. Additionally, due to the spoofed acknowledgements, any real acknowledgements for data will be considered old and ignored. We found this class of attacks against Ubuntu 11.10 and believe it to be unknown previously in the literature.

Sawtooth Ack (new). These attacks also operate using spoofed acknowledgements that increase their acknowledgement number at a steady pace. However, these packets may acknowledge more data and occasionally reset their acknowledgment number to the true cumulative acknowledgement point. This starting over, typically at a state transition from Congestion Avoidance to Fast Recovery or back, results in a long string of spoofed acknowledgements with increasing acknowledgement numbers that eventually reaches the previous high acknowledgement, at which point the sender begins sending new data. This causes a very prominent sawtooth pattern in a time sequence graph of the connection. Due to the increasing number of acknowledgements that must be sent to reach the highest acknowledgement each time, the sending rate of a connection under this type of attack continuously decreases. We found this class of attacks against Ubuntu 16.10, Ubuntu 14.04, Ubuntu 11.10, and Windows 8.1 using a variety of strategies. In our tests, this attack usually resulted

in approximately a 12x reduction in throughput. The attacker is required to expend approximately 40Kbps to keep the attack going.

Dup Ack Injection (new). This class of attack operates by repeatedly injecting three or more spoofed duplicate acknowledgements into the target connection in hopes of spuriously triggering Fast Recovery and slowing the connection down. We have found this class of attack to be very effective against Windows 8.1 and Debian 2. Newer Linux versions are not vulnerable to this attack due the use of DSACK [38] to detect spurious retransmissions and a mechanism to dynamically adjust the duplicate acknowledgement threshold needed to trigger Fast Recovery [108]. In our tests, this attack often resulted in approximately a 12x reduction in throughput when using Windows 8.1 or Debian 2. The connection repeatedly enters Fast Recovery and needlessly retransmits significant data. The attacker needs only 40Kbps of bandwidth to launch this attack.

Ack Amplification (new). This class of attack operates similarly to Slow Injected Acks. Instead of sending spoofed acknowledgements with increasing sequence numbers slowly, the attacker sends them very quickly. Each one causes the sender to send a large burst of packets, effectively bypassing congestion control and `cwnd` completely. This effect is even more pronounced in Slow Start, where the sender can send two bytes for every one acknowledged. Additionally, since any losses are masked by the spoofed acknowledgements, TCP will never slow down. This results in a very powerful class of attack where an attacker can cause the target connection to consume all available bandwidth up to the network and/or sending system capacity by simply sending acknowledgements at around 40Kbps. In our tests, the competing connection was left starved for bandwidth, with throughput near zero, and often doing repeated RTOs for the duration of the attack. The low bandwidth required of the attacker makes this ideal for a denial of service attack. We found a wide variety of strategies in this attack class against Ubuntu versions 16.10, 14.04, 11.10, and Windows 8.1.

Off-path Repeated Slow Start (new). This class of attacks is very similar to the On-path Repeated Slow Start attacks discussed previously. The difference is that

instead of repeatedly inducing Slow Start by preventing acknowledgements in Fast Recovery from acknowledging new data, we instead inject duplicate acknowledgements to increase Linux's duplicate acknowledgement threshold to the point where Fast Recovery is never entered and an RTO occurs instead. From there, we enter Slow Start and repeat. As in the on-path version, the rapid increase in sending rate during Slow Start balances out the idle period needed to cause an RTO and in many tests actually provides a higher average sending rate. These attacks also significantly impact competing connections due to the repeated, rapid sending periods that end in a loss for all connections. These repeated losses cause competing connections to slow down repeatedly.

We found this attack against Ubuntu 11.10. We believe this attack to have been previously unknown in the literature.

Desync Attack (known). This class of attacks operates by spoofing packets containing a few bytes of data to both sender and receiver in the target connection. If a host is not currently receiving data, this injected data will incorrectly cause its cumulative acknowledgement number to increase. All future packets by this host will then have an acknowledgement number higher than anything the other host sent. and will be ignored, causing an unrecoverable connection stall.

These attacks were first identified by [99]. The only known mitigation is encryption to prevent access to the sequence numbers of the packets. We identified many instances of this attack class against all tested implementations and in all congestion control states.

Ack Storm Attack (known). Ack Storm attacks are similar to Desync Attacks. The difference is that while only one half of the connection is desynchronized in Desync Attacks, both sides become desynchronized by Ack Storm Attacks. As before, we spoof packets with a few bytes of data to both sender and receiver in the target connection. However, in this case, both sides are idle, so cumulative acknowledgments at both sender and receiver are increased and both sides send new acknowledgements. Unfortunately, since neither side actually sent any data, both will

consider these acknowledgements invalid. As it happens, the TCP specification [34] requires that a host receiving an invalid acknowledgement should respond with a duplicate acknowledgement. This leads to an infinite storm of acknowledgements between both sides of the connection, as each responds to the invalid acknowledgements from the other. Additionally, as in Desync Attacks, the target connection itself is stalled and no further data can be transferred.

This is a known attack, first identified by [100]. One mitigation to this attack is to ignore invalid acknowledgements if they show up too frequently. Unfortunately, neither Debian 2 nor Windows 8.1 provide this mitigation, enabling us to discover this attack with several different strategies.

5.6 Summary

Today, the testing of congestion control and the discovery of attacks against it is mostly a manual process performed by protocol experts. We developed TCPwn, a system to automatically test real implementations of TCP by searching for attacks against their congestion control. TCPwn uses a model-guided attack generation strategy to generate abstract attack strategies which are then converted to concrete attack scenarios made up of message-based actions or packet injections. Finally, these concrete attack scenarios are applied in our testing environment, which leverages virtualization to run real implementations of TCP independent of operating system, programming language, or libraries. We evaluated 5 TCP implementations including both open- and closed- source systems, using TCPwn. We found 2,436 attack strategies which could be grouped into 11 classes, of which 8 were previously unknown.

6 PERFORMANCE AND AVAILABILITY ATTACKS FOR QUIC

Next generation transport protocols are introducing security features like data and header encryption and performance optimizations like 0-RTT connections that have important implications on the kinds of attacks they are vulnerable too. Some of these features, like encryption, reduce the attack surface significantly, while others, like 0-RTT connections based on caching, open up whole new areas that have not been previously explored. In this chapter, we investigate one next generation transport protocol, Google's QUIC, and identify attacks on its performance and availability.

6.1 Introduction

The proliferation of mobile and web applications and their performance and security requirements have exposed the limitations of current secure transport protocols. Specifically, secure protocols like TLS [49] have a relatively high connection establishment latency overhead, causing user unhappiness and often resulting in a decreased number of customers and financial losses. As a result, several efforts [48, 62, 109, 110] have gone into designing new transport protocols that have low latency as one of the major design goals, in addition to basic security goals such as confidentiality, authentication, and integrity.

One of the most promising of these protocols is QUIC [48], a secure transport protocol developed by Google and implemented in Chrome in 2013. QUIC integrates ideas from TCP, TLS, and DTLS [111] in order to provide security functionality comparable to TLS, congestion control comparable with TCP, as well as minimal round-trip costs during connection setup/resumption and in response to packet loss. Some of the major design differences from TLS are not relying on TCP in order to eliminate redundant communication and the use of initial keys to achieve faster

connection establishment. QUIC also includes techniques similar to to TCP Fast Open [112], TLS Snap Start [113], and forward error correcting codes.

QUIC has already seen significant deployment, being supported by all Google services and the Google Chrome browser; as of 2016, more than 85% of Chrome requests to Google servers use QUIC [22]. In fact, given the popularity of Google services, QUIC now represents a substantial fraction (estimated at 7% [114]) of all Internet traffic. As a result, it is critical to understand its performance and availability guarantees in the presence of attackers, especially considering that QUIC is envisioned mainly for web content delivery and mobile applications.

Previous work examining QUIC has investigated its security guarantees [24,25] or its performance in benign environments [26–30], but has not investigated its performance and availability in the presence of attackers. There exists a significant body of work looking at attacks on the performance and availability of older transport protocols like TCP [3, 5, 7–13]; however, these attacks all assume the ability to observe, modify, or inject packets in the connection under attack. QUIC's use of encryption prevents packet modification or injection in the vast majority of cases as well as protecting against observation of packet data and acknowledgement information. This complicates and restricts possible attacks. However, the caching of detailed information about a server provided by QUIC's server config (`scfg`), which enables 0-RTT connections, is unheard of in existing protocols and provides a wealth of possible information for attackers. This opens up a whole new mode of attack unavailable in traditional transport protocols like TCP or DCCP.

In this chapter, we investigate QUIC's performance and availability in the presence of attackers and the root causes underlying this behavior. Due to the use of encryption, which effectively obscures any user data and most protocol state, we are limited to considering the connection setup packets, which are sent in plain text. We focus our investigation on the cacheable information in QUIC used to achieve 0-RTT connections, as 0-RTT seems to be an important performance motivator for this and other next generation transport protocols. We consider attacks by on-path attackers,

with ability to modify packets; off-path attackers, with the ability to observe and spoof packets; and blind attackers, who are limited to only injecting packets blindly into the connection.

Through this investigation, we show that the very mechanisms used in QUIC to achieve 0-RTT connections, such as unprotected fields on handshake packets and the use of publicly available information on both client and server sides, can be exploited by an adversary to attack QUIC's availability during the connection establishment handshake. In particular, we identify two classes of attacks on QUIC's availability based on replaying cached information or modifying unprotected information in the packets used for the connection establishment handshake. We then successfully implement 5 attacks against the Chromium implementation of QUIC.[1] Four of these attacks prevent a client from establishing a connection with a server, compromising availability, while the fifth is a resource exhaustion denial of service attack against QUIC servers. Note that these attacks only compromise the availability, not the security of QUIC. In all cases, we found the attacks easy to implement and completely effective. In many cases, the client is forced to wait for QUIC's ten-second connection establishment timeout before giving up.

Our results suggest that the techniques used in QUIC to minimize latency may not be useful in the presence of malicious parties. Although these weaknesses are not completely unexpected, they are of significant concern to the QUIC team at Google who have been developing a dedicated monitoring infrastructure to try to address them [115]. Moreover, these issues appear to be fundamental limitations in 0-RTT connections that rely on caching, an important result for next generation transport protocols.

We note that some of these attacks are similar to known attacks against TLS and TCP and investigate this similarity. However, TLS and TCP make no general promises about their performance in the presence of adversaries. We find that even

[1]Specifically, QUIC version Q021, from October 2014.

if QUIC's performance may not be perfect, it is not worse than that of TLS in the worst case, and is much better in the absence of adversaries.

To summarize, the contributions of this chapter are:

- We investigate QUIC's performance and availability in the presence of attackers, focusing on ways to leverage the use of caching during the connection establishment handshake to attack the connection. Later phases of the connection are protected by encryption, which prevents the modification and injection of packets as well as blinds the attacker to most protocol state. Our investigation considers a variety of attackers, including on-path, off-path, and blind attackers.

- We identify two classes of attacks on QUIC's availability based on replaying cached information or modifying unprotected information in the packets used for the connection establishment handshake. These classes of attacks are of particular interest to other next generation transport protocols because they are general to 0-RTT protocols relying on caching or protocols with any unprotected packet fields. We further identify multiple attacks on QUIC within these classes.

- We demonstrate 5 attacks compromising the availability of QUIC clients or servers running the Chromium QUIC implementation. Four of these attacks prevent a client from establishing a connection with a server, compromising availability, while the fifth is a resource exhaustion denial of service attack against QUIC servers. Note that these attacks only compromise the availability, not the security, of QUIC.

The rest of this chapter is organized as follows. Section 6.2 presents our investigation of possible attacks against QUIC while Section 6.3 implements the attacks we discovered and discusses their impacts on a real QUIC implementation. Section 6.4 then discusses these attacks and examines their similarity to existing attacks on TCP and TLS. Finally, we summarize this chapter in Section 6.5.

6.2 QUIC in the Presence of Attackers

In this section we investigate QUIC's performance and availability in the presence of attackers. Due to QUIC's use of encryption during later phases of the connection, we focus on the connection establishment handshake. In particular, we concentrate on attacks possible by leveraging that cacheable information used to provide 0-RTT connections or unprotected packet fields used during the handshake. QUIC's cacheable information consists of three components: 1)the server config or `scfg` that contains important information about the server including a Diffie Hellman share, supported encryption and signing algorithms, and flow control parameters; 2) the source-address token or `stk`; and 3) the server nonce or `sno`. The `scfg` contains all the information about the server needed to establish a 0-RTT connection and initial encryption key while the `stk` is used to prevent IP spoofing and the `sno` is used to prevent packet replay attacks. These components are unique to transport protocols providing 0-RTT connections and contain a wealth of information about a server.

Also important to our investigation are unprotected packet fields in QUIC. With so much of the packet encrypted or authenticated during the connection (*i.e.,* any data, any acknowledgment information, the contents of the `scfg`), it is important to pay attention to those fields that are not authenticated immediately. These are fields that may be able to be manipulated or spoofed by an attacker. These fields include the connection id or `cid`, QUIC version number (if present), and public flags. Additionally, a number of fields are considered as opaque byte-strings by the client but authenticated by the server. These include the `stk` and `sno`.

Our investigation of these fields and components led to the identification of two classes of attacks that compromise QUIC's availability. The first of these classes requires only an off-path attacker and operates by replaying QUIC's cacheable information to the client or server, misleading the other party about the progress of the connection. The second of these classes modifies unprotected packet fields, requiring an on-path attacker, to create different ideas about connection state on each side of

the connection. Both of these classes prevent QUIC connection establishment. Persistent failure to establish a QUIC session could further result in a fall-back to TCP, defeating QUIC's purpose of minimizing latency while securing the transport layer. In the remainder of this section we discuss these two classes of attacks and multiple specific attacks within each class.

6.2.1 Replay Attacks

Once at least one client establishes a session with a particular server, an attacker could learn the public values of that server's `scfg` as well as the source-address token value `stk` corresponding to that client during their respective validity periods. An attacker could then replay the server's `scfg` to the client and the source-address token `stk` to the server, misleading in either case the other party. This requires an off-path attacker, with the ability to observe and inject packets.

Server Config Replay Attack. An attacker can replay a server's public `scfg` to any other clients sending initial connection requests to that server while keeping the server unaware of such requests from clients. Thus, these clients believe they have enough information to establish an initial connection with the server. When combined with a random `stk` or `sno`, which a client cannot verify, this leads to a server not recognizing the client and rejecting the their packets. While data confidentiality is not affected, the clients would experience additional connection establishment latency and waste computational resources deriving an initial key.

Source-Address Token Replay Attack. An attacker can replay the source-address token `stk` of a client to the server that issued that token many times to establish additional connections. This action would cause the server to establish initial keys and even final forward-secure keys for each connection without the client's knowledge. Any further steps in the handshake would fail, but an attacker could create a denial of service attack on the server by creating many connections on behalf

of a many different clients and possibly exhausting the server's computational and memory resources.

Ironically, these attacks stem from parameters whose main purpose was to minimize latency by enabling 0-RTT connections. These attacks are more subtle than simply dropping QUIC handshake packets because they mislead at least one party into "believing" that everything is going well while causing it to waste time and resources deriving an initial key.

Resolving these types of attacks seems to be infeasible without reducing the `scfg` and `stk` parameters to one-time use, because as long as these parameters persist for more than just a single connection, they can be used by an attacker to fake multiple connections while they remain valid. However, such restriction would prohibit QUIC from ever achieving 0-RTT connection establishment, the primary motivation for using these parameters.

6.2.2 Packet Manipulation Attacks

An on-path attacker with access to the communication channel used by a client to establish a connection with a particular server could flip bits of any unprotected parameters, leading to different connection state at client and server. Of particular interest are unprotected fields that are used to derive encryption keys. There are two of these: the connection id `cid` and the source-address token `stk`. Modifying these parameters leads the client and server to derive different initial keys which ultimately leads connection establishment to fail. For a successful attack, the attacker has to make sure that all parameters modified in this way seem consistent across all sent and received packets with respect to any single party but inconsistent from the perspective of both parties participating in the handshake.

This type of attack does not raise concerns over the confidentiality and authenticity of communication that is encrypted and authenticated under the initial encryption key, because even though the initial keys are different, they are not known by the

attacker. Note also that if both parties do not agree on an initial key, they cannot establish a final encryption key in QUIC because the final `s_hello` message is encrypted and authenticated under the initial key. Therefore, these attacks do not compromise the confidentiality and authenticity of communication encrypted and authenticated under the final key.

These packet manipulation attacks are smarter than just dropping QUIC handshake packets because the client and server progress through the handshake while having a mismatched conversation, resulting in the establishment of inconsistent keys. This causes both parties to waste time and resources deriving keys and other connection state. In particular, the server performs all the processing required for a successful connection, unlike in attacks that simply drop QUIC handshake packets.

A simple strategy for mitigating this type of attack would be to have the server sign all such modifiable fields in its `s_reject` and `s_hello` packets. However, this would incur the cost of computing a digital signature over all such modifiable parameters, which would in turn open another opportunity for a denial of service attack in which the adversary, with IP spoofing, could send many initial connection requests on behalf of as many clients as it desires.

6.3 Attack Results

We have implemented the attacks on QUIC's availability described in the previous section and discuss their results here. We target the Chromium implementation of QUIC[2] in our attacks, as this is the canonical implementation. Our attacks were developed in python using the scapy library.[3] We summarize our attacks, their properties, and impacts in Table 6.1.

[2] `https://chromium.googlesource.com/chromium/src.git`. We tested QUIC version Q024 from git revision `50a133b51fa9c6a3dc2b82ce9fedcf074859cd13` from October 1, 2014.
[3] `http://www.secdev.org/projects/scapy/`

Table 6.1.: Attacks on QUIC

Attack Name	Type	Attacker	Impact
Server Config Replay Attack	Replay	Off-path	connection failure
Source-Address Token Replay Attack	Replay	Off-path	server DoS
Connection ID Manipulation Attack	Manip	On-path	connection failure; server load
Source-Address Token Manipulation Attack	Manip	On-path	connection failure; server load
Crypto Stream Offset Attack	Other	Off-path	connection failure

6.3.1 Replay Attacks

Replay attacks use values designed to be cached by the client, like the server config `scfg` and the source-address token `stk`, to mislead either the client or the server into believing that a connection is being established correctly. As these attacks require snooping on legitimate connections, they require an off-path attacker.

Server Config Replay Attack. To conduct this attack, an attacker must first collect a copy of the target server's `scfg`. This can be done either by actively establishing a connection to the server or by passively listening for a client to attempt a connection. In either case, the server's `scfg` can be readily collected from a full, 1-RTT QUIC connection handshake.

Once the attacker has `scfg`, he waits for the target client to attempt to start a connection. When the attacker sees a `c_hello` message from the client, he can respond with a spoofed `s_reject` message using the collected `scfg` and randomly generated `stk` and `sno` values. Similar `s_reject` messages are the proper response to a client that either does not have a cached copy of the server's `scfg` or has a copy that is no longer valid. We assume that the attacker is closer to the client than the server is so that the `s_reject` message reaches the client prior to the response from the legitimate server. When the client receives this spoofed `s_reject` message, it promptly sends a new `c_hello` message using these new `scfg`, `stk`, and `sno` values.

When the real server receives this new `c_hello` message, it will attempt to validate it. However, the `stk` and `sno` values were randomly generated by the attacker and so are almost certain to fail the validation. In response to this failure, the server generates a new `s_reject` message containing `scfg` and new `stk` and `sno` values.

This new `s_reject` message provides the client with valid `stk` and `sno` values so another `c_hello` message could correctly complete the connection. However, when testing this attack, we found two further issues, the combination of which will always result in the connection terminating abnormally. The first issue is that each QUIC packet includes an entropy bit in its header and QUIC acknowledgment frames include a hash of these bits along with a list of unseen packets. The goal of this mechanism is to prevent Optimistic Ack attacks [51]. In our case, an acknowledgment frame will typically be included with the client's second `c_hello` message acknowledging the spoofed `s_reject` message. If the entropy bit in the attacker's spoofed `s_reject` message does not match the entropy bit in the server's real response, then the entropy hash in this acknowledgement will not validate and the server will abruptly terminate the connection.

The second issue is that a single QUIC connection provides multiple byte-streams for data transfer, and the QUIC handshake takes place within a special byte-stream reserved for connection establishment. This implies that all the `c_hello`, `s_reject`, and `s_hello` messages we have mentioned so far occur within the context of this byte-stream and have offset and length attributes. As a result, if the attacker's `s_reject` is not exactly the same size as the server's response, then this byte-stream is effectively broken. Any further messages from the server will be at offsets either above or below the client's position in the byte-stream. These messages will either be dropped or buffered forever. After ten seconds the client will abruptly terminate the connection because it is unable to complete the handshake.

In our tests, the combination of these two issues completely prevented the establishment of any QUIC connections. Connection attempts always terminated after either half a second, in the case of an entropy bit mismatch, or ten seconds, if the en-

tropy bits matched, but the byte-stream was corrupted. Our python implementation requires that the attacker be about 20ms closer to the client than the server is, in order to create an `s_reject` message and have it reach the client before the server's legitimate response. However, with an optimized C implementation, this requirement could be significantly reduced.

Source-Address Token Replay Attack. The `stk` token is supposed to prevent packet spoofing by ensuring that a connection request originates at the IP address claimed. The `stk` is created by the server as part of the `s_reject` message. It contains the client's IP address and the current time, both encrypted. A client must present a valid `stk` in its `c_hello` message in order to perform a 0-RTT connection. However, the `stk` token must be presented prior to encryption being established. This means that any attacker who can sniff network traffic can collect `stk` tokens that can be used to spoof connection requests from a specific host for a limited period of time, by default 24 hours.

This attack operates by sniffing the network for `s_reject` messages from the target server. Each `s_reject` message contains a new `stk` being sent to some client. For each new `stk` seen, our attacker grabs the `stk`, the `scfg`, and the client's IP address and starts repeatedly spoofing 0-RTT connection attempts with random `cid`s from this client.

When the target server receives these requests, they appear to be legitimate 0-RTT connection requests. The `stk` will validate because the `stk` is replayed from a legitimate connection with an actual client at the spoofed IP address. As a result, the server will create a new connection for this request. This includes creating initial and forward-secure encryption keys and sending an `s_hello` message. At this point, the server believes it has completed connection establishment with the spoofed client.

In our tests, we used separate virtual machines for the attacker and server. We found that a single attacker starting with a single `stk` and sending packets at 200KB/sec was able to completely overwhelm our test server. The 2.4 GHz Intel® Xeon® CPU dedicated to our server was pegged at 100% utilization, and the operating system's

out-of-memory killer eventually killed the server process after it exhausted the 3GB of memory allocated to the server's virtual machine.

It seems apparent that the QUIC server implementation in Chromium has no limitation on the number of connections that can be established from a single IP address. While we do not believe that this is the server implementation that Google uses in production, it is the only open-source QUIC server available. Additionally, much of the QUIC code is a library that we expect would be used by any production QUIC server. Note, however, that even if a limit on the number of connections from a single IP were added, this attack can inflate the number of connections to the server by this maximum number for *every* observed QUIC client.

6.3.2 Manipulation Attacks

Manipulation attacks subvert key agreement by causing the client and server to agree on different keys. This is done by modifying unprotected packet fields that are used as input to the key derivation process. Two fields, the connection id `cid` and source-address token `stk`, seem particularly interesting. We developed attacks against both of these parameters. Note that modifying packet fields requires an on-path attacker.

Connection ID Manipulation Attack. In this attack, the attacker is positioned on the path between the client and the server and re-writes the `cid` such that the client and server see different values. The handshake proceeds as normal, with the client requesting the `scfg`, if it does not have a cached copy, and then sending a `c_hello` message. This `c_hello` is processed by the server and an `s_hello` message sent in response. At this point, the server believes the connection has been successfully established. However, when the client receives the `s_hello` message sent by the server, it will fail to decrypt. This is because the `cid` is an input to the encryption key derivation process. Since the attacker changes the `cid`, the client and server will compute different encryption keys.

Unfortunately, decryption failure is not a sign of catastrophic handshake failure because it can be caused by reordering. In particular, packets encrypted with the forward-secure key will fail to decrypt prior to the reception of the `s_hello` message, which may be delayed due to reordering. As a result, packets failing decryption are buffered until the handshake completes. With the bad `s_hello` message buffered, the client will eventually timeout and retransmit its `c_hello` message. This process will repeat until the client's 10 second timer on connection establishment expires. At that point the connection will be terminated.

An error message will be sent to the server when the connection is terminated. However, this message will be encrypted with the initial encryption key, and thus the server will fail to decrypt it and will queue it for later decryption. Since it cannot decrypt the error message, the server will retain the connection state until the idle connection timeout expires. This timeout defaults to 10 minutes.

Source-Address Token Manipulation Attack. The goal of this attack is to prevent a client from establishing a connection, either denying access to the desired application or forcing the client to fall back to TCP/TLS. It requires an attacker positioned on the path between the client and the server who re-writes the `stk` such that the client and server see different values. It is important that the server always see the value it initially sent because it will validate `stk` later. To the client, however, `stk` is simply an opaque byte string.

Any attempted connection request will proceed as normal, except that the attacker silently changes the `stk` values seen by client and server. The client requests the `scfg` from the server, which replies with the current `scfg` and an `stk` value. The client then sends a full `c_hello` to initiate the connection. The server receives and processes this `c_hello` and sends an `s_hello` message in response.

When the client receives this `s_hello` message sent by the server, it will fail to decrypt. This is because `stk` is an input into the encryption key derivation process, and the attacker has changed the `stk` value seen at the client. As a result, the client and server will compute different encryption keys.

However, as mentioned previously, a decryption failure is not a sign of catastrophic handshake failure because this could happen due to reordering, if packets encrypted with the forward-secure key were received before the `s_hello` message. Hence, the client buffers the bad `s_hello` message for later decryption. Eventually the client times out and retransmits the `c_hello` message. This process will repeat until the client's 10 second timer on connection establishment expires. At that point the connection will be terminated.

The client will notify the server that it terminated the connection, but, unfortunately, this message will be transmitted encrypted with the initial encryption key. Hence, the server will be unable to process it and will continue to retain the connection state. This state will only be removed when the idle connection timeout expires, by default after 10 minutes.

We found that this attack effectively prevented all targeted QUIC connections. Further, all targeted connections experienced a 10 second delay before timing out.

6.3.3 Other Attacks

While developing and testing the Server Config Replay Attack, we discovered an additional attack against QUIC. This attack results from QUIC treating handshake messages as part of a logical byte-stream.

Crypto Stream Offset Attack. Recall that handshake messages are part of a logical byte-stream in QUIC. As a result, by injecting data into this byte-stream an attacker is able to break the byte-stream and prevent the processing of further handshake messages. The attack results in preventing a client from establishing a connection using QUIC, either denying access to the desired application or forcing the client to fall back to TCP/TLS.

We create the attack by injecting a four character string into this handshake message stream. This injection is sufficient to prevent connection establishment. Our attacker listens for `c_hello` messages and responds with a spoofed reply containing

the string "REJ\0" in the handshake message stream. As observed before, this breaks connection establishment because any messages from the server will now start at the wrong offset in the handshake message stream. Hence, they will be discarded or buffered indefinitely.

A connection that is attacked in this manner will either be terminated by the server because of an entropy bit mismatch or be timed out by the client after 10 seconds.

Note that an attacker requires very little information to launch this attack. No information is needed from the client's `c_hello` message, QUIC packet sequence numbers always start from 1, and the `cid` can be omitted from any packet other than the client's `c_hello`. As a result, all an attacker needs to launch this attack is knowledge of when a connection attempt will occur and the 4-tuple (server IP, client IP, server port, client port) involved. Of this 4-tuple, three items are already known: the server's IP, the client's IP, and the server's UDP port. If an attacker can guess the client's UDP port and when it will make a connection attempt, he can launch this attack completely blind.

In our tests, the ephemeral UDP port range was still too large to brute force within an RTT, at least with our python attacker. However, if the attacker can narrow the port range sufficiently, then an optimized C implementation could probably conduct this attack completely blind.

6.4 Attack Discussion

In this section we discuss how some of the attacks we found against QUIC relate to prior attacks on TCP and TLS. We find that attacking QUIC is not easier than attacking TLS over TCP.

Source-Address Token Replay Attack. This QUIC attack is similar to the TCP SYN Flood attack [116] where the attacker sends numerous spoofed TCP SYN packets to a server to overwhelm it and cause DoS. The QUIC attack does almost

the same thing, but the attacker is limited in the IP addresses he can use for spoofed packets. However, the impact of each spoofed packet is larger because QUIC needs to create encryption keys after receiving the initial packet.

The classic mitigation to SYN Flood is SYN Cookies, opaque tokens passed to the client by the server in the SYN-ACK and returned by the client on the final handshake ACK [116]. A SYN-Cookie encodes enough information so that the server does not need to keep state between the SYN and the final ACK and can serve as a proof that the client resides at its claimed IP address. The server creates the connection state structures only after the cookie is returned by the client, making it more difficult to overwhelm the server with spoofed connection requests.

An `stk` serves a similar purpose in preventing spoofed packets. However, SYN Cookies are single use, limiting their time and IP address validity [116]. This prevents an attacker from using a SYN Cookie to spoof multiple TCP connections. While the `stk` could be made single use, this would severely limit the cases where QUIC could successfully establish a 0-RTT connection.

QUIC Manipulation Attacks. These QUIC attacks are similar to the SSL Downgrade attack [117] against a modern TLS implementation. In both cases, an on-path, man-in-the-middle attacker modifies packet fields and the attack is not discovered until the end of the handshake, after key generation and multiple RTTs.

SSL Downgrade works against SSL connections where both endpoints have SSL versions less than SSL 3.0 enabled. The goal is to downgrade the connection to an older, less secure version of SSL [117]. Basically, the attacker rewrites the connection request to indicate that the client only supports an older version of SSL, often version 2.0. The server and client then establish an SSL 2.0 connection, which the attacker can presumably compromise.

SSL 3.0 adds protection against this attack by adding a keyed hash of all the handshake messages to the Finished message and requiring the receiver to verify this hash [117]. This defense is effective, but the attack will only be detected at the end of the handshake.

Our QUIC Manipulation Attacks have a similar outcome where the attack only becomes apparent at the end of the handshake, when the keys generated by the client and server do not match. Thus, the connection fails after a timeout, and the client may fall back to TCP/TLS. Since QUIC is designed to provide much lower latency for connection initiation than TCP/TLS, this compromises one of QUIC's main goals.

As discussed in section 6.2.2, one simple mitigation would be to sign all modifiable fields in the server's `s_reject` and `s_hello` messages. However, this introduces signature computation overhead and a possible denial of service attack.

QUIC Crypto Stream Offset Attack. This attack is similar to the TCP ACK Storm Attack [100] in that both result in the inability to transfer any more data over the target byte-stream and are caused by an attacker inserting data into the byte-stream.

The TCP ACK Storm Attack requires an off-path attacker who can observe a TCP ACK packet of the target connection and then spoof data-bearing packets to both the client and the server. This data will be received and processed by the client and server and both will increase their ACK numbers as a result. Unfortunately, when an ACK is eventually sent by either client or server, it will appear to acknowledge data that the other side has not yet sent. TCP will drop such packets and send a duplicate ACK. At this point, the TCP byte-stream is effectively broken; no more data can be transferred because all packets will have invalid ACK numbers.

In much the same way, injection of data into a QUIC handshake stream disrupts the stream offsets and prevents any further handshake negotiation. This eventually results in connection timeout. Although a byte-stream is a convenient abstraction, it does not appear to be a good fit for handshake data. A message stream, or sequence of messages, would be less prone to disruption in this manner.

6.5 Summary

QUIC is a new, next generation transport protocol that has seen significant adoption and now makes up about 7% of Internet traffic. Unlike other transport protocols, it encrypts user data and most protocol state and offers 0-RTT connection establishment for improved performance. We investigate performance and availability attacks against QUIC, focusing on QUIC's use of caching to achieve 0-RTT and the impact of unprotected packet fields. Our analysis identifies two classes of attacks on QUIC's availability, based on replaying cacheable information or modifying unprotected packet fields. These attack classes appear to be general to 0-RTT protocols relying on caching and unprotected packet fields. We further identify and demonstrate 5 attacks on QUIC within these classes. Four of these attacks prevent a client from establishing a connection with a server, compromising availability, while the fifth is a resource exhaustion denial of service attack against QUIC servers. Note that these attacks only compromise the availability, and not the security, of QUIC; however, they significantly impact QUIC's goal of low latency connections and deny access to resources available over QUIC.

7 RELATED WORK

In this section, we present related work organized by topic.

7.1 Automated Attack Detection

Prior work has looked at automatically finding attacks on network protocols. Fuzzing has been the predominant approach in this research direction. While random fuzz testing [118] is often effective in finding interesting corner case errors, the probability of "hitting the jackpot" is low because it typically mutates well-formed inputs and tests the program on the resulting inputs. To overcome this inherent challenge of fuzzing, a set of works like SNOOZE [14], KiF [15], and EXT-NSFM [16] leverage the protocol state machine to cover deeper portions of the search space. KiF uses the state machine to bias fuzzing towards unexplored search space while SNOOZE uses it to reach deeper locations before beginning fuzzing, and EXT-NSFM uses it to determine what part of the protocol to fuzz without unnecessarily restarting the application. These and similar fuzzing tools primarily search for crashes or other fatal errors.

Several other research efforts [5, 18, 19, 119–121] leverage program analysis, for example, symbolic execution, to find vulnerabilities in protocol implementations. MAX [5] focuses on finding performance attacks mounted by a compromised participant in two-party protocols. However, MAX relies on user specified information about interesting lines of code to limit the search space during symbolic execution. Similarly, SymbexNet [119] tests two-party protocols by executing one party symbolically to operate on symbolically marked input packets. Thus it can generate high-coverage test input packets for the implementation, whose responses are verified against manually derived rules from the specification. Many of these techniques re-

quire access to the protocol implementation source code in a specific language and most do not consider malicious parties.

MACE [18] combines symbolic execution with concrete execution to infer the protocol state machine and use it as a search space map to allow deep exploration for bugs. The inferred state machine represents the external interactions of the protocol (*e.g.,* the sequence of exchanged messages). While this state machine captures transitions caused by distinct messages types, it is unable to identify transitions caused by other characteristics of the exchanged messages. MACE searches only for crashes or other fatal errors in protocol implementations and does not consider malicious parties.

Turret [6] provides a platform for finding performance attacks against intrusion tolerant distributed systems. Turret inserts a malicious proxy in front of an unmodified implementation to simulate a malicious attacker and uses a greedy search strategy to look for the malicious actions that cause the largest impact in system performance. While parts of our approach are similar, transport protocols require a very different set of malicious actions than the multi-party, attack-resistant, application-layer protocols that Turret targets. Additionally, since intrusion tolerant distributed systems are designed to be attack resistant, Turret is able to use a greedy search strategy that looks for actions that cause performance impacts and then combines them. In contrast, transport protocols are not designed with intrusion tolerance in mind, which causes a greedy search strategy to fail completely. Instead we have to develop search strategies based on models of the protocol's behavior.

7.2 Transport Protocol Attacks

There has been significant prior work on finding attacks on transport protocols. One of the earliest works in this area is [10] which identified the problems with predictable initial sequence numbers in TCP. The authors demonstrated that predictable initial sequence numbers enable a blind attacker to spoof TCP connections.

Another early work in this area was [99] which identified the Desynchronization Attack against TCP. This attack, identified by manual analysis of TCP, causes the sender and receiver to become desynchronized with respect to the location of the cumulative acknowledgement, resulting in a connection stall.

Another important work in this area is [11] which considered the ways that a malicious receiver could cheat TCP's congestion control. An expert analysis of the protocol identified and demonstrated three attacks that such a receiver could launch: Optimistic Ack, Ack Division, and Dup Ack Spoofing. These attacks allow an endpoint or on-path attacker to increase the throughput of a target connection by modifying how it acknowledges data, either acknowledging more data than it should, acknowledging it in many little pieces, or repeatedly acknowledging the same data. Ack Division and Dup Ack Spoofing has since been widely mitigated by applying Appropriate Byte Counting [122] and similar implementation-level mitigations.

Another well known set of attacks are the SYN-flood [116], Reset [13], and SYN-Reset [3] TCP attacks. The SYN flood attack operates by overwhelming the target with a huge number of SYN packets, preventing legitimate connections from being established, while the Reset and SYN-Reset attacks allow a blind attacker to abruptly terminate a target connection. They operate by brute forcing the sequence and acknowledgement numbers, which is practical due to large receive window sizes.

The work in [9] and [8] introduced another pair of attacks (the Shrew and Induced Shrew attacks) against TCP's congestion control. These attacks offer a blind attacker with a means to degrade the throughput of TCP connections along some target link while expending minimal bandwidth in an attempt to avoid detection. Both of these attacks were again identified manually by protocol experts.

A security analysis of TCP commissioned by the British Government [3] identified two additional attacks available to a blind attacker. These are the Blind Flooding Attack and the Blind Throughput Reduction Attack. Both operate by sending spoofed acknowledgements which will cause the receiver to send a duplicate acknowledgment if the packet is out of the acceptable sequence window.

The work in [100] identified the Ack Storm Attack where the injection of data into a target TCP connection prevents further data transfer and generates an infinite series of acknowledgements, as both parties respond to what they consider to be an invalid acknowledgement with an acknowledgement. This attack was similarly identified by expert protocol analysis.

More recently, a number of works [95–98] have demonstrated that inferring TCP sequence numbers is feasible under certain circumstances. These works leverage some form of side channel to enable a blind attacker to determine the sequence and acknowledgement numbers needed to inject data into some target TCP connection. They were identified by expert analysis.

7.3 NLP for Technical Domains

A variety of works have looked at applying NLP to extract information from technical domains. DASE [69] applies NLP techniques to identify input constraints from code comments describing file formats; regular expressions are also used to extract command line arguments from man pages. These constraints are used to improve test case generation for symbolic execution. Input constraints are identified using a simple rule-based system applied to a typed dependency parse of each code comment.

The work in [123] also applies NLP to code comments, but with the intention of creating simplified versions of highly optimized library functions to ease static and dynamic analyses. It attempts to extract input and output conditions from semi-structured JavaDoc comments. The system constructs a parse tree and then uses pattern matching to generate possible function implementations. The authors find it necessary to process the output at several points in the process to deal with inaccuracies introduced by NLP techniques trained on newswire.

ARSENAL [124] is designed to take requirements documents for safety-critical systems and turn them into Linear Temporal Logic (LTL) formulas that can be auto-

matically analyzed to ensure that desired properties hold. Again, a rule-based system is applied to a typed dependency parse of each sentence to create a corresponding LTL formula. A complex pre-processor, based on regular expressions, is required to ensure that domain specific terminology is handled correctly.

Other works have looked at mobile application permissions and privacy [68, 125, 126], enhanced source code search and cross-linking [70, 127], requirements document and API checking [128–131], and trouble ticket classification [132]. For mobile applications, WHYPER [68] applied NLP to extract required permissions from descriptions of mobile applications while [125] examined application privacy policies to extract what the policy allows applications to do with the data they collect.

SWordNet [127] seeks to improve software code search by identifying semantically related words, which may be domain specific. This is done by extracting semantic pairs of words from code comments and function names. Similarly, [70] used documentation and source code to create an ontology allowing the cross-linking of software artifacts represented in code and natural language on a semantic level.

ACRE [128] leverages NLP to extract access control rules from design documents written in natural language. This is done using a type dependency parse tree and a set of pattern-based rules. Pandita, et al.'s work in [129] creates formal API specifications from natural language, often fragmentary, API descriptions. Here, available structured information, like parameter and method names can be leveraged. Similarly, Doc2Spec [130] extracts formal specification rules from API specifications to search for bugs in application code. NLP is used to identify resources used by APIs based on an ontology.

NetSieve [132] analyzes free-form text in trouble tickets to identify problem symptoms and suggest resolutions. It identifies domain-specific phrases based on text statistics and maps them into an ontology. A rule-based system then extracts $(problem, action, resolution)$ tuples which can be applied to new tickets.

8 CONCLUSION

Given the importance of transport protocols, it is crucial to insure that implementations of these protocols operate securely and reliably and that we understand the types of attacks to which these protocols are exposed. This has previously been done via painstaking manual analysis of individual protocol implementations by networking experts, which is extremely time consuming and inefficient. As a result, there has been a steady steam of attacks against transport protocols in the literature. This dissertation addresses this situation by providing novel techniques for automatically searching for attacks in real, unmodified implementations of transport protocols and by providing a better understanding of the types of attacks that are faced by next generation transport protocols.

As a first step towards automated testing of real transport protocol implementations, we presented SNAKE, a system to automatically and broadly search for attacks on the performance or availability of arbitrary transport protocol implementations. SNAKE uses a novel attack injection technique to generate test cases by leveraging the protocol's connection state machine to focus testing on key protocol locations as well as a new attack detection technique based on expected competition and fairness, to detect attacks that are not as obvious as implementation crashes. To provide completely automated testing, we also developed an NLP framework to extract a description of a protocol's grammar automatically from a natural language specification document. To do this we use a zero-shot learning approach that learns a similarity function between textual phrases and protocol fields and relations, enabling adaptation to new protocols, and rely on the structure and linguistic regularities of the protocol specification documents in our domain to minimize the amount of training required.

We have provided a concrete implementation of SNAKE and demonstrated its effectiveness using five implementations of two transport protocols in four different operating systems, finding 9 classes of attacks. While skilled researchers manually analyzing implementations have previously discovered some of these attacks, SNAKE was able to find all of these attacks automatically and without human intervention. Additionally, we used our NLP-pipeline to automatically extract grammars from 7 protocols and found that our pipeline was capable of extracting protocol packet fields with an F-score of 0.74 and finding and linking properties with a success rate of 66%. We further demonstrated the value of automatic grammar extraction by applying our pipeline to SNAKE and comparing it to using a manual grammar. We find a reduction in the testing effort (from 901 to 819 test cases) while identifying the same set of attacks and doing so in a fully automated manner. We, therefore, conclude that SNAKE can significantly improve the process of testing and securing implementations of transport protocols.

Although SNAKE finds many types of attacks, it is ineffective at finding attacks on congestion control due to the highly complex and dynamic nature of the congestion control algorithms. Therefore, we proposed TCPwn, a system to automatically search for attacks on implementations of TCP congestion control. TCPwn models congestion control as a finite state machine and uses a model-based attack strategy generation algorithm that generates possible congestion control attacks by identifying their key characteristics. This algorithm first generates abstract attack strategies which are then converted into concrete attack strategies by identifying attacker actions that cause the desired state machine transitions. These strategies are then applied to real implementations of TCP with the help of an algorithm to infer the current congestion control state of a sender from network traffic.

We have provided a concrete implementation of TCPwn and demonstrate its effectiveness using five TCP implementations from different operating systems, finding 11 classes of attacks. Again, some of these classes of attacks have been previously discovered by skilled researchers manually inspecting implementations. However, TCPwn

finds all these attack classes automatically and without human intervention. We, therefore, conclude that TCPwn can be applied to significantly improve the security of TCP congestion control implementations.

Next generation transport protocols like QUIC present distinctly different attack surfaces compared to traditional transport protocols like TCP. This stems from the heavy use of encryption to protect user data as well as most protocol headers, rendering third-party attacks on congestion control and connection tear down ineffective. Additionally, QUIC has been optimized for low latency thanks to 0-RTT connections, which make significant use of caching. This exposes a host of new information to the attacker. We have, therefore, studied QUIC, looking for attacks on its availability and performance with the goal of understanding the types of attacks that impact next generation transport protocols. Our manual investigation revealed two classes of attacks against the availability of QUIC, both of which result from design choices made to allow enhanced performance. We futher identified and demonstrated 5 attacks within these classes compromising the availability of QUIC clients or servers running the Chromium QUIC implementation.

Future Work. There are several compelling directions to pursue for future work. First, automated testing is limited by the amount of information about the protocol that is available. Increasing this to include further information about expected protocol behavior, the meanings of particular fields, or protocol algorithms can help to improve testing. While we have made progress on using NLP to tackle this problem, we have made no more than an initial attempt. We would like to be able to extract not only protocol grammars, but also state machines and algorithms as well as to parse other sources, like blog-posts, for protocol information and possible bugs. Second, the ability to automatically search for attacks on congestion control using TCPwn raises interesting questions about the security of alternative congestion control algorithms like BBR [104] and TFRC [32]. In particular, we hope that these algorithms are less susceptible to manipulation, but certainly no more so, than classic New Reno. While TCPwn provides a good basis for such a comparison, these new

congestion control algorithms will require new, currently unknown, state inference algorithms due to their significantly different state machines. Finally, the development of automated testing for QUIC implementations seems particularly valuable given the interest in the protocol. Our evaluation clearly identifies several classes of attacks to which QUIC is vulnerable as well as some of the challenges to automated testing. In particular, methods to search for attacks leveraging cached information would need to be developed.

REFERENCES

REFERENCES

[1] Gordon Lyon. Nmap, 2017. `http://nmap.org/`.

[2] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. Packetdrill: Scriptable network stack testing, from sockets to packets. In *USENIX Annual Technical Conference*, pages 213–218. USENIX, 2013.

[3] Centre for the Protection of National Infrastructure. Security assessment of the transmission control protocol. Technical Report CPNI Technical Note 3/2009, Centre for the Protection of National Infrastructure, 2009.

[4] Vern Paxson, Mark Allman, Scott Dawson, William Fenner, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP implementation problems. RFC 2525 (Informational), March 1999.

[5] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govidan, and Madanlal Musuvathi. Finding protocol manipulation attacks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, pages 26–37. ACM, 2011.

[6] Hyojeong Lee, Jeff Seibert, Endadul Hoque, Charles Killian, and Cristina Nita-Rotaru. Turret: A platform for automated attack finding in unmodified distributed system implementations. In *IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*, pages 660–669. IEEE, 2014.

[7] Biswaroop Guha and Biswanath Mukherjee. Network security via reverse engineering of TCP code: Vulnerability analysis and proposed solutions. *IEEE Network*, 11(4):40–48, 1997.

[8] V. Kumar, P. Jayalekshmy, G. Patra, and R. Thangavelu. On remote exploitation of TCP sender for low-rate flooding denial-of-service attack. *IEEE Communications Letters*, 13(1):46–48, 2009.

[9] Aleksandar Kuzmanovic and Edward Knightly. Low-rate TCP-targeted denial of service attacks and counter strategies. *IEEE/ACM Transactions on Networking*, 14(4):683–696, 2006.

[10] Robert Morris. A weakness in the 4.2 BSD unix TCP/IP software. Technical report, AT&T Bell Leboratories, 1985.

[11] Stefan Savage, Neal Cardwell, David Wetherall, and Tom Anderson. TCP congestion control with a misbehaving receiver. *ACM SIGCOMM Computer Communication Review*, 29(5):71, October 1999.

[12] Joe Touch. Defending TCP against spoofing attacks. RFC 4953 (Informational), July 2007.

[13] Paul Watson. Slipping in the window: TCP reset attacks. Technical report, CanSecWest, 2004. http://bandwidthco.com/whitepapers/netforensics/tcpip/TCPResetAttacks.pdf.

[14] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmer, and Giovanni Vigna. SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr. In SokratisK. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security Conference*, volume 4176 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2006.

[15] Humberto Abdelnur, Radu State, and Olivier Festor. KiF: A stateful SIP fuzzer. In *International Conference on Principles, Systems and Applications of IP Telecommunications*, pages 47–56. ACM, 2007.

[16] Jiajie Wang, Tao Guo, Puhan Zhang, and Qixue Xiao. A model-based behavioral fuzzing approach for network service. In *3rd International Conference on Instrumentation, Measurement, Computer, Communication and Control (IMCCC)*, pages 1129–1134. IEEE, 2013.

[17] Petar Tsankov, Mohammad T Dashti, and David Basin. SECFUZZ: Fuzz-testing security protocols. In *7th International Workshop on Automation of Software Test (AST)*, pages 1–7, 2012.

[18] Chia Y Cho, Domagoj Babic, Pongsin Poosankam, Kevin Z Chen, Edward X Wu, and Dawn Song. MACE: Model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *USENIX Security Symposium*. USENIX, 2011.

[19] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[20] Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, and Tom M Mitchell. Zero-shot learning with semantic output codes. In *Advances in neural information processing systems*, pages 1410–1418, 2009.

[21] Randall Stewart, Scott Long, Drew Gallatin, Alex Gutarin, and Ellen Livengood. The Netflix tech blog: Protecting Netflix viewing privacy at scale. http://techblog.netflix.com/2016/08/protecting-netflix-viewing-privacy-at.html, 2016.

[22] Ian Swett. QUIC deployment experience @Google. https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf, 2016.

[23] Jana Iyenger and Martin Thomson. QUIC: A UDP-based multiplexed and secure transport. draft-ietf-quic-transport-10 (Internet Draft), 2018.

[24] Marc Fischlin and Felix Günther. Multi-Stage key exchange and the case of Google's QUIC protocol. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[25] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS# 1 v1. 5 encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[26] Chromium Blog. A QUIC update on Google's experimental transport. `http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html`, April 2015.

[27] Peter Megyesi, Zsolt Krämer, and Sandor Molnár. How quick is QUIC? In *Proceedings of the IEEE International Conference on Communications (ICC)*, May 2016.

[28] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. HTTP over UDP: an experimental investigation of QUIC. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2015.

[29] Somak R Das. Evaluation of QUIC on web page performance. Master's thesis, Massachusetts Institute of Technology, 2014.

[30] 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)*, pages 290–303. ACM, 2017.

[31] Jon Postel. User datagram protocol. RFC 768 (Standard), August 1980.

[32] Sally Floyd, Mark Handley, Jitendra Padhye, and Joerg Widmer. TCP friendly rate control (TFRC): Protocol specification. RFC 5348 (Proposed Standard), September 2008.

[33] Joerg Widmer and Mark Handley. TCP-friendly multicast congestion control (TFMCC): Protocol specification. RFC 4654 (Experimental), August 2006.

[34] Jon Postel. Transmission control protocol. RFC 793 (Standard), 1981.

[35] Mark Allman, Vern Paxson, and Ethan Blanton. TCP congestion control. RFC 5681 (Draft Standard), September 2009.

[36] Tom Henderson, Sally Floyd, Andrei Gurtov, and Yoshifumi Nishida. The NewReno modification to TCP's fast recovery algorithm. RFC 6582 (Proposed Standard), 2012.

[37] Ethan Blanton, Mark Allman, Lili Wang, Ilpo Jarvinen, Markku Kojo, and Yoshifumi Nishida. A conservative loss recovery algorithm based on selective acknowledgment (SACK) for TCP. RFC 6675 (Proposed Standard), 2012.

[38] Ethan Blanton and Mark Allman. Using TCP duplicate selective acknowledgement (DSACKs) and stream control transmission protocol (SCTP) duplicate transmission sequence numbers (TSNs) to detect spurious retransmissions. RFC 3708 (Experimental), 2004.

[39] Nandita Dukkipati, Neal Cardwell, and Yuchung Cheng. Tail loss probe (TLP): An algorithm for fast recovery of tail losses. draft-dukkipati-tcpm-tcp-loss-probe-01 (Internet Draft), 2013.

[40] Matt Mathis, Nandita Dukkipati, and Yuchung Cheng. Proportional rate reduction for TCP. RFC 6937 (Experimental), 2013.

[41] Pasi Sarolahti, Markku Kojo, Kazunori Yamamoto, and Max Hata. Forward RTO-Recovery (F-RTO): An algorithm for detecting spurious retransmission timeouts with TCP. RFC 5682 (Proposed Standard), 2009.

[42] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for fast long-distance networks. RFC 8312 (Informational), 2018.

[43] Yuchung Cheng, Neal Cardwell, and Nandita Dukkipati. RACK: a time-based fast loss detection algorithm for TCP. draft-ietf-tcpm-rack-03 (Internet Draft), 2018.

[44] Jerry Chu, Nandita Dukkipati, Yuchung Cheng, and Matt Mathis. Increasing TCP's initial window. RFC 6928 (Experimental), 2013.

[45] Sally Floyd, Mark Handley, and Eddie Kohler. Datagram congestion control protocol (DCCP). RFC 4340 (Proposed Standard), 2006.

[46] Sally Floyd and Eddie Kohler. Profile for datagram congestion control protocol (DCCP) congestion control ID 2: TCP-like congestion control. RFC 4341 (Proposed Standard), 2006.

[47] Sally Floyd, Eddie Kohler, and Jitendra Padhye. Profile for datagram congestion control protocol (DCCP) congestion control ID 3: TCP-friendly rate control (TFRC). RFC 4342 (Proposed Standard), 2006.

[48] Google QUIC Team. QUIC at 10,000 feet. `https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU`, 2013.

[49] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246 (Proposed Standard), August 2008.

[50] Joseph Salowey, Hao Zhou, Pasi Eronen, and Hannes Tschofenig. Transport layer security (TLS) session resumption without server-side state. RFC 5077 (Proposed Standard), January 2008.

[51] Jim Roskind. Quick UDP internet connections: Multiplexed stream transport over UDP, 2013. `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit`.

[52] Adam Langley, Britt Cyr, Jeremy Dorfman, Fedor Kouranov, Ian Swett, Jana Iyengar, Charles Krasic, Jo Kulik, Ryan Hamilton, Jim Roskind, Robbie Shade, Satyam Shekhar, Cherie Shi, Raman Tenneti, Victor Vasiliev, Antonio Vicente, Patrik Westin, Alyssa Wilk, Dale Worley, Fan Yang, Dan Zhang, and Daniel Ziegler. QUIC wire layout specification, 2017. `https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/`.

[53] Adam Langley and Wan-Teh Chang. QUIC crypto, 2016. `https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit`.

[54] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. draft-ietf-tls-tls13-26 (Internet Draft), 2018.

[55] Martin Thomson and Sean Turner. Using transport layer security (TLS) to secure QUIC. draft-ietf-quic-tls-10 (Internet Draft), 2018.

[56] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? Provable security and performance analyses. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[57] Eric Rescorla. [TLS] 0-RTT and anti-replay. IETF TLSWG mailing list posting, 2015. `https://www.ietf.org/mail-archive/web/tls/current/msg15594.html`.

[58] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer, 1998.

[59] Dino Farinacci, Tony Li, Stan Hanks, David Meyer, and Paul Traina. Generic routing encapsulation (GRE). RFC 2784 (Proposed Standard), March 2000.

[60] Stephen Deering and Robert Hinden. Internet protocol, version 6 (IPv6) specification. RFC 2460 (Draft Standard), December 1998.

[61] John Postel. Internet protocol. RFC 791 (Internet Standard), September 1981.

[62] Randall Stewart. Stream control transmission protocol. RFC 4960 (Proposed Standard), September 2007.

[63] Yipeng Wang, Zhibin Zhang, Danfeng Daphne DD Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security (ACNS)*, pages 1–18. Springer-Verlag, 2011.

[64] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 155–168. USENIX, 2004.

[65] Emden Gansner, Eleftherios Koutsofios, and Stephen North. Drawing graphs with *dot*, 2006. `http://www.graphviz.org/Documentation/dotguide.pdf`.

[66] WebHosting Talk. DOS attack – hosting security and technology, 2004. `https://www.webhostingtalk.com/showthread.php?t=293069`.

[67] Oskar Andreasson. TCP variables, 2002. `https://www.frozentux.net/ipsysctl-tutorial/chunkyhtml/tcpvariables.html`.

[68] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: towards automating risk assessment of mobile applications. In *USENIX Security Symposium*, pages 527–542. USENIX, 2013.

[69] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. DASE: document-assisted symbolic execution for improving automated software testing. In *37th IEEE/ACM International Conference on Software Engineering, (ICSE)*, pages 620–631, 2015.

[70] René Witte, Qiangqiang Li, Yonggang Zhang, and Juergen Rilling. Text Mining and Software Engineering: An Integrated Source Code and Document Analysis Approach. *IET Software*, 2(1):3–16, 2008.

[71] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. Prospex: Protocol specification extraction. In *IEEE Symposium on Security and Privacy*, pages 110–125. IEEE, 2009.

[72] Yipeng Wang, Zhibin Zhang, Danfeng Daphne Yao, Buyun Qu, and Li Guo. Inferring protocol state machine from network traces: A probabilistic approach. In *Applied Cryptography and Network Security*, pages 1–18. Springer, 2011.

[73] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 621–634, 2009.

[74] 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 (CCS)*, pages 426–439. ACM, 2010.

[75] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 199–212, 2007.

[76] Nupur Kothari, Todd Millstein, and Ramesh Govindan. Deriving state machines from TinyOS programs using symbolic execution. In *Information Processing in Sensor Networks (IPSN)*, pages 271–282. IEEE, 2008.

[77] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Network and Distributed Systems Security Symposium (NDSS)*, 2008.

[78] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 317–329, 2007.

[79] David Lie, Andy Chou, Dawson Engler, and David L Dill. A simple method for extracting models from protocol code. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 192–203. IEEE, 2001.

[80] James C Corbett, Matthew B Dwyer, John Hatcliff, Shawn Laubach, Corina S Păsăreanu, Ro Bby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the International Conference on Software Engineering*, pages 439–448. IEEE, 2000.

[81] John Postel. Instructions to RFC authors. RFC 1543 (Informational), October 1993.

[82] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[83] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Conference on Empirical Methods in Natural Language Processing*, 1996.

[84] Xuezhe Ma and Eduard Hovy. End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 1064–1074, 2016.

[85] Erik F Tjong Kim Sang and Fien De Meulder. Introduction to the CoNLL-2003 shared task: Language-independent named entity recognition. In *Proceedings of the 7th Conference on Natural language learning at HLT-NAACL 2003*, pages 142–147. Association for Computational Linguistics, 2003.

[86] Joakim Nivre. Non-projective dependency parsing in expected linear time. In *Proceedings of the Association for Computational Linguistics (ACL) and Asian Federation of Natural Language Processing (AFNLP) Joint Conference*, pages 351–359. Association for Computational Linguistics, 2009.

[87] Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. Globally normalized transition-based neural networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2016.

[88] Xiao Cheng and Dan Roth. Relational inference for wikification. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1787–1796, 2013.

[89] Andrea Moro, Alessandro Raganato, and Roberto Navigli. Entity linking meets word sense disambiguation: A unified approach. *Transactions of the Association for Computational Linguistics*, 2:231–244, 2014.

[90] Heather Flanagan and Sandy Ginoza. RFC style guide. RFC 7322 (Informational), September 2014.

[91] Scott Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119 (Best Current Practice), March 1997.

[92] Internet Engineering Task Force. IETF - RFCs. `https://www.ietf.org/standards/rfcs/`, 2018.

[93] Van Jacobson. Congestion avoidance and control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.

[94] Stephen Kent and Karen Seo. Security architecture for the internet protocol. RFC 4301 (Proposed Standard), 2005.

[95] Zhiyun Qian, Z. Morley Mao, and Yinglian Xie. Collaborative TCP sequence number inference attack: How to crack sequence number under a second. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 593–604, 2012.

[96] Zhiyun Qian and Z. Morley Mao. Off-path TCP sequence number inference attack – How firewall middleboxes reduce security. In *IEEE Symposium on Security and Privacy*, pages 347–361, 2012.

[97] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. Off-path TCP exploits: Global rate limit considered dangerous. In *USENIX Security Symposium*, pages 209–225, 2016.

[98] Yossi Gilad and Amir Herzberg. Off-path attacking the web. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, pages 41–52, 2012.

[99] Laurent Joncheray. A simple active attack against TCP. In *USENIX Security Symposium*, 1995.

[100] Raz Abramov and Amir Herzberg. TCP ack storm DoS attacks. In *IFIP International Information Security Conference*, pages 29–40. Springer, 2011.

[101] Joeri de Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[102] Mark Utting and Bruno Legeard. *Practical model-based testing: A tools approach*. Morgan Kaufmann, 2010.

[103] Ahren Studer and Adrian Perrig. The coremelt attack. In *European Symposium on Research in Computer Security*, pages 37–52, 2009.

[104] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September-October:20–53, 2016.

[105] Murari Sridharan, Kun Tan, Deepak Bansal, and Dave Thaler. Compound TCP: A new TCP congestion control for high-speed and long distance networks. draft-sridharan-tcpm-ctcp-02 (Internet Draft), 2009.

[106] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In *Conference on Communications Architectures, Protocols and Applications*, pages 24–35, 1994.

[107] Greg Nelson and Derek Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[108] The Linux Kernel Community. /proc/sys/net/ipv4/* variables. `https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`, 2017.

[109] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and K. K. Ramakrishnan. Towards a SPDY'ier mobile web? In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, pages 303–314. ACM, 2013.

[110] Bryan Ford. Structured streams: A new transport abstraction. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 361–372. ACM, 2007.

[111] Eric Rescorla and Nagendra Modadugu. Datagram transport layer security version 1.2. RFC 6347 (Proposed Standard), January 2012.

[112] Yuchung Cheng, Jerry Chu, Sivasankar Radhakrishnan, and Arvind Jain. TCP fast open. RFC 7413 (Experimental), December 2014.

[113] Adam Langley. Transport layer security (TLS) snap start. draft-agl-tls-snapstart-00 (Internet Draft), 2010.

[114] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Joanna Kulik, Jim Roskind, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 183–196. ACM, 2017.

[115] Alyssa Rzeszutek Wilk, Jo Kulik, Fedor Kouranov, and Assar Westerlund. Google QUIC team, Personal communication, 2014.

[116] Wesley Eddy. TCP SYN flooding attacks and common mitigations. RFC 4987 (Informational), August 2007.

[117] Jeremy Clark and Paul C. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pages 511–525. IEEE Computer Society, 2013.

[118] Barton Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12), 1990.

[119] JaeSeung Song, Cristian Cadar, and Peter Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, 2014.

[120] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *IEEE International Conference on Information Processing in Sensor Networks*, pages 186–196, 2010.

[121] Radu Banabic, George Candea, and Rachid Guerraoui. Finding trojan message vulnerabilities in distributed systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 113–126, 2014.

[122] Mark Allman. TCP congestion control with appropriate byte counting (ABC). RFC 3465 (Experimental), 2003.

[123] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for Java API functions. In *Proceedings of the 38th International Conference on Software Engineering*, pages 380–391. ACM, 2016.

[124] Shalini Ghosh, Daniel Elenius, Wenchao Li, Patrick Lincoln, Natarajan Shankar, and Wilfried Steiner. ARSENAL: automatic requirements specification extraction from natural language. In *Proceedings of the 8th International Symposium on NASA Formal Methods*, pages 41–46. Springer-Verlag New York, Inc., 2016.

[125] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven M Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[126] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 1025–1035. ACM, June 2014.

[127] Jinqiu Yang and Lin Tan. SWordNet: Inferring semantically related words from software context. *Empirical Software Engineering*, 19(6):1856–1886, 2014.

[128] John Slankas, Xusheng Xiao, Laurie Williams, and Tao Xie. Relation extraction for inferring access control rules from natural language artifacts. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014 Annual Computer Security Applications Conference (ACSAC), pages 366–375. ACM, 2014.

[129] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language API descriptions. In *Proceedings of 34th International Conference on Software Engineering (ICSE)*, June 2012.

[130] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring specifications for resources from natural language API documentation. *Automated Software Engineering Journal*, 18(3–4):227–261, 2011.

[131] Reut Tsarfaty, Ilia Pogrebezky, Guy Weiss, Yaarit Natan, Smadar Szekely, and David Harel. Semantic parsing using content and context: A case study from requirements elicitation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1296–1307. Association for Computational Linguistics, October 2014.

[132] Rahul Potharaju, Navendu Jain, and Cristina Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.

VITA

VITA

Samuel Jero received a combined Bachelor of Science and Master of Science in computer science from Ohio University in 2013. He received his PhD in computer science from Purdue University in 2018. During his time at Purdue, he was a member of the Network and Dependable Systems Security Lab and was affiliated with the Center for Education and Research in Information Assurance and Security (CERIAS). He was also a recipient of the Purdue University Andrews Fellowship and the Purdue Bisland Dissertation Fellowship. His work has received a variety of awards including a best paper award from the IEEE Conference on Dependable Systems and Networks (DSN) in 2015, the Applied Networking Research Prize from the IRTF in 2016 and 2018, and the Cisco Network Security Distinguished Paper Award from the Network and Distributed Systems Security Symposium (NDSS) in 2018. His research focused on automatically finding performance and availability attacks in transport protocol implementations and identifying attacks against next generation transport protocols. He has also done research on the security of Software Defined Networks (SDNs) and using Software Defined Networking to enhance the security of networks.