Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Matthias Stähli

Network Performance Obfuscation

Semester Thesis SA-2019-04
February 2019 to June 2019

Tutor: Roland Meier
Supervisor: Prof. Dr. Laurent Vanbever

Abstract

Simple network diagnostic tools such as Ping and iPerf allow users to remotely infer network performance properties. Latency, packet loss rate and bandwidth knowledge can serve an attacker as a source of information to select weak targets or to draw conclusions about network activities and geographical allocation. We leverage the recent advances in programmable network devices to deceive selected hosts by manipulating specific traffic types. Our framework provides effective bandwidth throttling to target value, rerouting of packets to reach increased latencies and selectable loss rate. Our evaluation shows that we can worsen the latency, bandwidth and packet loss to specific target values. Furthermore, we do not reveal any obfuscation when the network is examined with iPerf3, Ping or Traceroute.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Performance properties provide valuable insights into a network. They allow the network managing team to detect congestion, false forwarding behaviour and anomalies [8]. If not specifically blocked, these properties can be measured by network users for legitimate reasons, such as debugging their connection. Simultaneously, they serve as a source of information for an attacker: Knowing the topology and performance properties enable an attacker to select weak targets and perform resource optimized but still effective link flooding attacks, such as the Coremelt attack [31]. Additionally, an attacker could draw conclusions about ongoing network activities or the geographical location [26] of network devices.
Obfuscating network performance properties for selected hosts could deceive the view of some, while more trustworthy hosts receive the true values. This could be used to create imaginary bottlenecks that might lure an attacker to choose the wrong target, resulting in an attenuation of a link flooding attack.
Obfuscating performance properties could also be used to cheat on performance tests to cover any installed rate throttling policies or to fudge physical distances.

1.2 The Task

Based on existing work to obfuscate network topology [25] and traffic [24], we set our scope to obfuscate network performance. The main goal of this thesis is to come up with simple ways to deceive selected hosts by manipulating the observed latency, packet loss and bandwidth but without having significant impact on production traffic. The designed framework should:

- Classify different traffic types. We distinguish between measurement traffic examining our performance properties and inconspicuous production traffic.
- Perform obfuscation in the data plane. Enabling us to obfuscate vast amounts of data at line rate.
- Obfuscate the parameters to specific target values.
- Not reveal any ongoing obfuscation.

The main design challenges are:

- The obfuscation must not impact any production traffic
- It is hard to differentiate between production packets and packets that are used to measure the performance.
- Most packets cannot be answered by a host that is not the intended destination.

Our resulting framework will finally be tested by measuring obfuscated performance parameters and comparing them to their target values.

1.3 Related Work

Recent work has been looking at obfuscating the topology [25] and the network traffic [24].

Network topology obfuscation is done by intercepting and modifying path tracing probes directly on the data plane. They are obfuscated in a way to mitigate link flooding attacks while still preserving the practicality of path tracing tools. The main concept is to formulate the obfuscation problem as an optimization problem with multiple objectives, where security requirements are set as hard constraints and usability as soft constraints. Their system NetHide is able to obfuscate large topologies with little impact on utility.

Network traffic obfuscation is done by encrypting the payload and obfuscating the header metadata. iTAP [24] rewrites the packet headers at the network edges directly on the data plane. Their system uses a novel rewriting scheme allowing them to scale while guaranteeing strong anonymity within the premise of a network. The main idea is to combine the anonymity benefits from a single-use ID per flow with the scalability benefits from having a constant ID per host.

1.4 Overview

Chapter 2 contains the theoretical background used to derive our design. It gives a short introduction into software defined networks, introduces the 2 rate 3 color Marker, specifies the network properties and introduces the traffic types of interest. Chapter 3 motivates and describes our design. It provides an overview of our framework and explains the obfuscation tools. We then evaluate the implemented framework in Chapter 4 and discuss our results. Finally, we draw a conclusion in Chapter 5 and provide an outlook for future work.

Chapter 2

Background

2.1 Software Defined Networking

Software defined networking [13] (SDN) refers to the ability to individually control and program network devices through an open interface. The key idea in SDN is to separate the control plane from the forwarding plane.

The *forwarding plane* (or data plane) stays simple and forwards traffic based on the rules set by the control plane. According to [13], the forwarding plane supports basic functions for switching, routing, packet transformation and filtering. Also resources such as filters, meters, markers and classifiers might be supported.

The *control plane* handles complex operations for managing one or more network devices. It has a more global view of the network and is therefore responsible to provide the forwarding plane with forwarding rules.

The goal of this separation is to enable innovation on both planes and to reduce the overall complexity for network management.

2.2 P4

P4 provides a target independent interface to program the forwarding plane as described in 2.1. **P4** is a high level language for **P**rogramming **P**rotocol-independent **P**acket **P**rocessors [9]. P4 specifies how switches should parse, process and forward packets, but does this without any restriction to specific network protocols.

2.2.1 P4 Forwarding Model

The forwarding model (Fig. 2.1) describes how a switch forwards a packet. On packet arrival, a programmable parser extracts the correct header sequence and header values. The extracted header fields are then passed to multiple match+action tables, spread among ingress and egress pipeline. Each packet carries additional information provided by the switch in a meta-data header. Some of these header values are ingress port and time of arrival. Values that can be used to select the correct forwarding rule.

2.2.2 P4 Language Concept

The P4 language concept [9] specifies the main components of a P4 program.

Headers A header defines the sequence and structure of a series of fields. Each field is specified by its width.

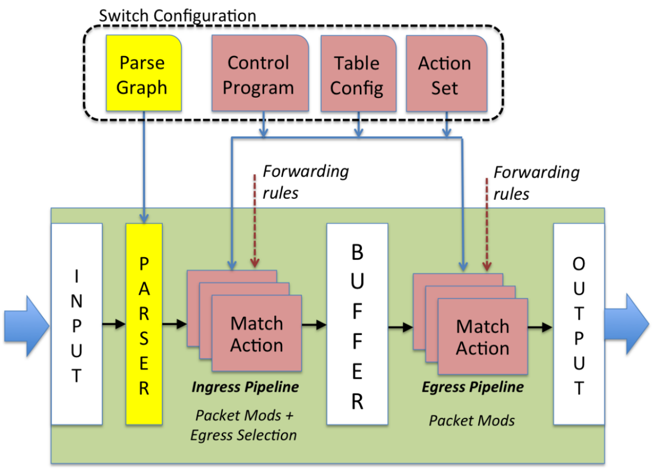


Figure 2.1: P4 Forwarding Model [9].

Header Stack The header stack [33] represents a fixed-size array of headers. The array size is a positive integer that must be known at compile time.

Parsers The parser specifies a graph that knows how to identify headers, validates the sequence of headers and extracts the respective header fields.

Tables Match+Action tables are used to perform packet processing. The user specifies the keys the table may match on and the bounded actions to execute. The keys are a subset of the header or metadata fields [33].

Action functions An action function specifies a compound of primitive actions to build more complex actions, also called compound actions. A set of standard primitive actions can be found in the P4 language specification [33]. Basically those are actions like addition, subtractions, logical operations and bit-shift. No modulo, division or branching is allowed within an action function.

Control Programs The control program is a simple imperative program determining the order of match+action tables applied to a packet. This simplicity prevents the switch from significantly delaying the packet.

Extern Objects and Functions Those are objects and functions provided by the architecture, so they can be vendor specific. E.g. pseudo random number generator, functions for checksum calculations, hash functions [33].

2.3 Two Rate Three Color Marker

A 2 rate 3 color Marker [14], or Meter, can be used to install an artificial bandwidth limit. Given a stream of packets, the Meter marks each packet ether green, yellow or red, illustrated in Fig. 2.2.

The Meter has four different parameters: The Peak Information Rate (PIR) and its associated Peak Burst Size (PBS) are responsible to color packets red. The Committed Information Rate (CIR) and its associated Committed Burst Size (CBS) are responsible for the yellow marks. Both rates are used to determine the color. The PIR has to be bigger or equal than the CIR. Burst size is the number of bytes the meter uses to measure the associated rate. Given those four parameters, the meter works as follows:

The Meter has two token buckets, Tp and Tc with size PBS and CBS respectively. Both token buckets are initially (time t = 0) full, meaning that Tp(t=0) = PBS and Tc(t=0) = CBS. For t>0, the token count for bucket Tp and Tc is incremented by PIR and CIR respectively, up to the maximum bucket size. The pseudo code in Fig. 2.3 explains what happens when a packet of size B arrives at time t.

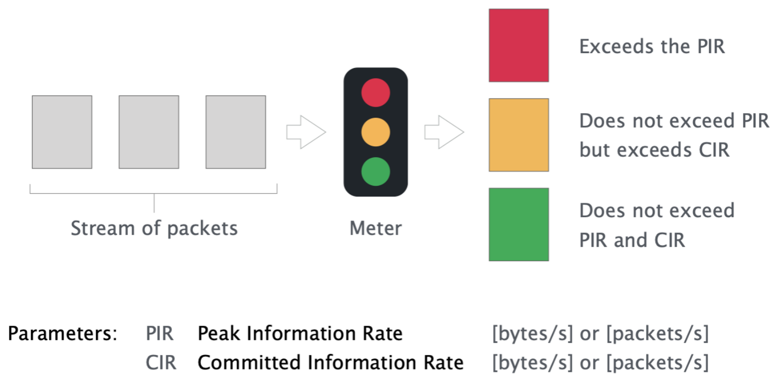
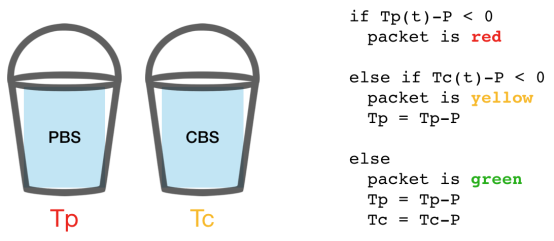

Exceeds the PIR
Does not exceed PIR but exceeds CIR
Does not exceed PIR and CIR
Stream of packets
Meter
Parameters: PIR Peak Information Rate [bytes/s] or [packets/s]
CIR Committed Information Rate [bytes/s] or [packets/s]

Figure 2.2: Packet coloring. The Meter marks a stream of packets with three different colors [35].



```
if Tp(t)-P < 0
  packet is red

else if Tc(t)-P < 0
  packet is yellow
  Tp = Tp-P

else
  packet is green
  Tp = Tp-P
  Tc = Tc-P
```

Figure 2.3: Meter rate calculation. The pseudo code describes how the rate is measured based on the size of two token buckets.

2.4 Network Performance Characterization

Network performance properties characterize a network. This section describes the origin and meaning of latency, packet loss and throughput.

2.4.1 Network Latency

Network latency or delay describes how long it takes a packet to travel from source to destination. The latency can be divided into four major contributors [38], illustrated in Fig 2.4:

Queuing delay describes the time packets spend in the queue of network devices. Typically only one packet at a time can be processed by the router. When packets arrive faster than the router processes them, the queue will start to fill up. This increases the queuing delay for newly arriving packets. The queuing delay is therefore highly varying and a strong indicator for network congestion [37]. Once a queue is full, newly arrived packets will be dropped. Therefore the range of queuing delay can go from zero (empty queue) to infinity (packet dropped).

Processing delay describes the time it takes network devices to process the packet [36]. The processing delay is in the order of microseconds [11]. Compared to other network delays, this value can therefore be ignored.

Transmission delay describes the time it takes to push the packet bits onto the link [39]. Given a link with a data rate of $10\frac{Mbits}{sec}$, it takes $0.1\mu s$ to send one bit. For a packet with 1500 KBytes, this means a delay of $18.75\mu s$.

Propagation delay describes the time it takes for a bit to reach its destination on the transmission medium. It can be seen as a ratio between the link length and the propagation speed of the medium. Electrical signals in copper wire travel at around 2/3 of the speed of light [21].

- Let $c = 300'000\frac{Km}{sec}$ be the speed of light.
- The speed v of an electrical signal in a copper wire results in $\frac{2\cdot c}{3} = 200'000\frac{Km}{sec}$.
- For an air-line link of $x = 100Km$ (Zurich-Bern), we get a propagation delay of $\frac{x}{v} = 0.5ms$.

We see that for wide area networks (WAN), like ISP networks, the propagation delay lies in the order of milliseconds and is therefore a major contributor to the overall latency. Due to its origin, it can also be assumed to be constant per link.

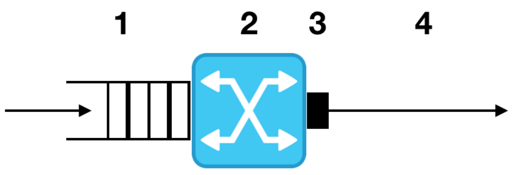


Figure 2.4: Network latency is a composition of multiple delay sources. All packet traversing a switch will first be queued (1. queuing delay), then processed (2. processing delay), transmitted (3. transmission delay) and finally propagated to the next switch (4. propagation delay).

2.4.2 Packet Loss

Packet loss in the network can occur for two reasons [18]: Packets are damaged in transit and get dropped, or somewhere on the path there was not enough buffer capacity. As packet corruption is quite rare, we can assume most losses to be caused by congestion. A lost packet can therefore be seen as a signal from the network to the sender that the network is congested. The sender assumes a packet to be lost after a certain time has past. This threshold time depends mostly on the average RTT of pre-sended packets.

2.4.3 Network Throughput & Bandwidth

At first glance, network throughput and bandwidth seem to be the same thing, but they are not [20]:

Throughput describes the amount of correct packet deliveries within a specific timeframe. Many lost packets will therefore result in low throughput, indicating a low quality network connection. This gives us an important metric to specify network performance. Throughput is mostly measured as an average in bits per seconds (bps).

Bandwidth measures the maximum amount of data that can correctly be send within a specific timeframe. It measures the available capacity of a network connection. Therefore having a high bandwidth does not mean having a high throughput. One way of approximating the bandwidth is to measure the throughput over a certain time range and take the highest value, as it must be closest to the actual bandwidth. A simple tool to measure the maximum reachable bandwidth on IP networks is iPerf [12].

2.5 Network Traffic Types

Next we introduce the traffic types that will later on be classified and manipulated to obfuscate specific hosts.

2.5.1 IPv4

The Internet Protocol version 4 (IPv4) [29] is currently the most widely used network protocol. It is responsible for routing packets through the internet from source IP to destination IP.

2.5.2 TCP & UDP

The Transmission Control (TCP) and User Diagram Protocol (UDP) are both transport protocols. They enable communication between applications running on different hosts.

2.5.3 iPerf

iPerf [12] is a widely used tool for measuring TCP and UDP bandwidth performance on IP networks. This thesis will focus on iPerf using TCP. A simple default iPerf test works as follows:

- An *iPerf server* is installed on the destination and listens on the default iPerf port.
- An *iPerf client* is setup on the source and connects to the server.
- Once a connection is established, the client starts uploading data at maximum sending rate to the server.
- When the upload has finished, the client closes the connection and prints the results.

There are mainly two versions of iPerf: iPerf and iPerf3. iPerf3 is a makeover of iPerf. It is single threaded and has a simpler codebase. iPerf on the other hand is multithreaded. Often iPerf is also called iPerf2, but that's just a newer version without major changes.

2.5.4 Ping

Ping [23] is a tool to test the reachability of a host on an IP network. It sends an ICMP echo request and gets a ICMP echo reply back. Ping measures the Round Trip Time (RTT) of the echo packet. ICMP stands for Internet Control Messaging Protocol [28], a protocol on top of IPv4 used to report information and errors in the network.

2.5.5 Traceroute

Traceroute [22] is a tool used to debug IP networks. It sends a packet with a Time To Live (TTL) of 1. The first hop on the path to target responds with an ICMP [28] expiration message. The response indicates that the packet could not be forwarded because the TTL expired. Traceroute then resends a packet with a TTL of 2, increased by one hop from the previous packet. This lets the second hop on the path respond with an ICMP TTL expired message. This goes on till the specified IP destination is reached. Traceroute records the source of every ICMP TTL exceeded message and can thereby trace the path and the RTT of a packet. This algorithm is often used to debug the network, because every router should be able to send TTL exceeded messages.

Chapter 3

Design

The goal of this thesis is to leverage programmable network devices to obfuscate network performance. This section specifies the actual properties we want to obfuscate and how we do it.

3.1 Design Goals

We aim to achieve a design that fulfills the following goals:

- Detect IPv4, TCP, UDP, Ping, iPerf and Traceroute traffic. These protocols are ether designed to measure performance properties or used as a basis for alternatives measurement tools.
- The traffic type should be detected as fast as possible.
- Obfuscate the packet loss rate, the latency and the bandwidth by making them worse.
- Apply the obfuscation such that the performance parameters reach specific values.
- Obfuscate all performance parameters in the data plane.
- Hosts should be individually deceivable.

3.2 Design Space

For this project we assume to control a wide area network (WAN). WANs are mostly run by IPSs or large cloud providers. Such networks have a high geographical span, use high speed connections and have high transfer rates. Therefore all packets should stay in the data plane to avoid denial of service attacks on the control plane. The perfect use case for P4: It allows us to inspect any header type, clone, drop and modify packets on the data plane at line rate. To keep the switching fast enough, P4 comes with some restrictions to memory size and operation complexity. No loops, no division and no floating numbers are supported [33]. Overall, P4 is the right programming language to use in this task for network data plane programming.

3.3 Design Overview

This section explains the overall work allocation between control and data plane, between different switches and the reasons behind it.
Our data plane receives the forwarding rules from two controllers: The routing controller and the obfuscation controller. Normal, non obfuscated traffic will be routed based on the forwarding rules provided by the routing controller. As smart routing of production traffic is not the focus of this thesis, we simply use shortest path routing. The routing controller provides the obfuscation controller with: the network topology; the forwarding behaviour for each switch; the link capacities; and the propagation delays. We will further reference to all this information as the

real topology. The obfuscation controller uses the real topology to calculate obfuscation rules. All rules ate then be pushed onto the responsible edge switches. Why edge switch? Well, we immediately want to know what to do with a new packet entering our network. Early classification allows for fast response, and that is of great importance when already one normally routed packet might reveal our obfuscation. The edge switch can directly check for matching rules and apply the corresponding obfuscation. Fig. 3.1 illustrates the function of an edge switch. All packets enter via an edge switch as they are used to interconnect networks. We will revere to all other switches as core switches.

Each edge switch holds all obfuscation rules of IP sources using it as an entry point. Assuming that most of our switches are edge switches, this should somewhat balance the storage overhead per switch. One of the main advantages is that edge switches enable us to drop packets before they enter our network. It also provides the obfuscation controller with full control over the routing. To route the packed differently than specified by the routing controller, the edge switch sets a source routing header provided by the obfuscation controller. The following sections will explain all of this in more detail.

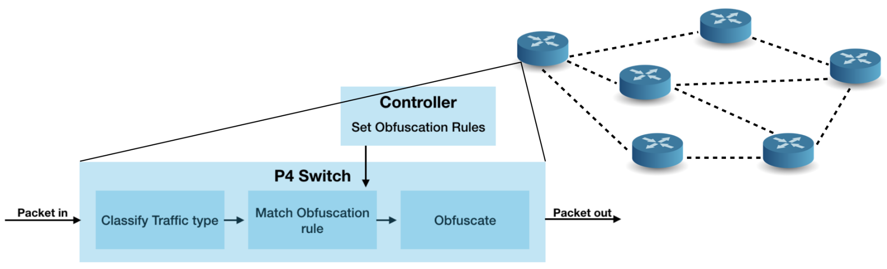


Figure 3.1: Packet forwarding in an edge switch.

3.4 Traffic Classification

We do not just want to obfuscate all traffic of our network. For example, we might want to limit the throughput from one poor paying client, but when she measures the throughput, it should be as good as covenanted. All other traffic should be forwarded as usual. To implement this, we require traffic type classification mechanisms to be able to select certain network protocols and measurement traffic. To handle multi-label classification, we prioritize some types over others. This chapter will explain the protocol features we use for classification and state how we rank multi-labeled traffic types.

3.4.1 IPv4

Most headers work by the same principle: They encapsulate the upper layer protocol, as shown in Fig. 3.2. To specify the upper protocol, headers contain a field specifying the protocol it encapsulates. So when a new packet arrives at the switch, we start by parsing the link layer protocol Ethernet. The Ethernet header [15] contains an Ethernet type field that indicates the protocol it encapsulates in its payload. Therefore an Ethernet packet containing a IPv4 packet in its payload is indicated by the EtherType field having the value 0x0800 [16]. We use this property to classify IPv4 traffic.

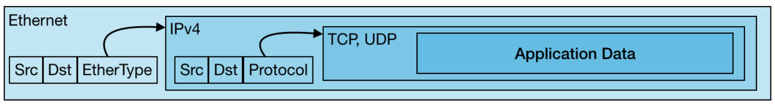


Figure 3.2: This figure shows how layers can specify encapsulated protocols by indicating them trough a dedicated header field.

3.4.2 TCP & UDP

The IPv4 header [29] has a header field, called Protocol, that specifies if TCP, UDP or any other transport protocol encapsulated in the payload. For TCP, the IPv4 Protocol number [6] is 6. For UDP it is 17. We use this property to classify TCP and UDP traffic.

3.4.3 iPerf

Next we will take a look at what fields can be used to classify iPerf and iPerf3 traffic. For simplicity we use iPerf to describe properties that are the same for iPerf and iPerf3.
iPerf can be used in default mode, and often is. After inspecting iPerf traffic using Wireshark[1] [7], we found the following default values:

- Default destination ports: 5001 for iPerf and 5201 for iPerf3 [12].
- Default payload uploaded to server for iPerf (not iPerf3) repeats the sequence '0123456789'. iPerf3 uses randomly generated payload by default.
- iPerf and iPerf3 use the same packet sequence to start the measurement, shown in Fig. 3.3:
    1. First, the client sends a SYN request to establish a connection.
    2. The server accepts the handshake and answers with a SYN-ACK packet.
    3. The client then acknowledges the handshake acceptance by sending an ACK to the server and directly sends two more packets:
    4. A push packet appending some iPerf information. For iPerf, the information has a length of 24 bytes. For iPerf3, it is 37 bytes.
    5. The first maximum sized packet containing data.

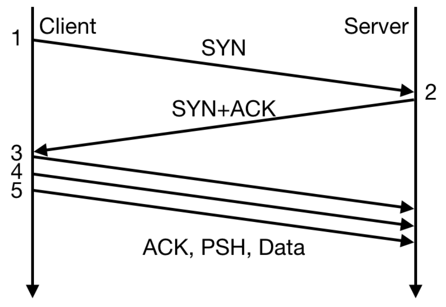


Figure 3.3: Timeline showing the packet transfer during the iPerf starting sequence.

Default ports and payloads can easily be changed, but the starting sequence described above stays the same. We came up with two methods to classify iPerf packets:
One is by matching the default values. The advantage here is that no state needs to be kept for TCP flows, because every packet contains the default values. This provides a fast and efficient way to check if a packet belongs to an iPerf flow.
For non default values we need to track the iPerf starting sequence and eventually mark the flow as iPerf. All subsequent packets will then be classified as iPerf. The main challenge here is to have a memory efficient way for tracking the iPerf starting sequence and to remember the classification result. For our solution we utilize following assumptions and properties:

1. TCP needs one RTT to establish a connection. The connection usually lives much longer than one RTT.
2. The time it takes to identify the starting sequence takes one round trip.
3. Only a small part of all TCP flows will be iPerf flows.

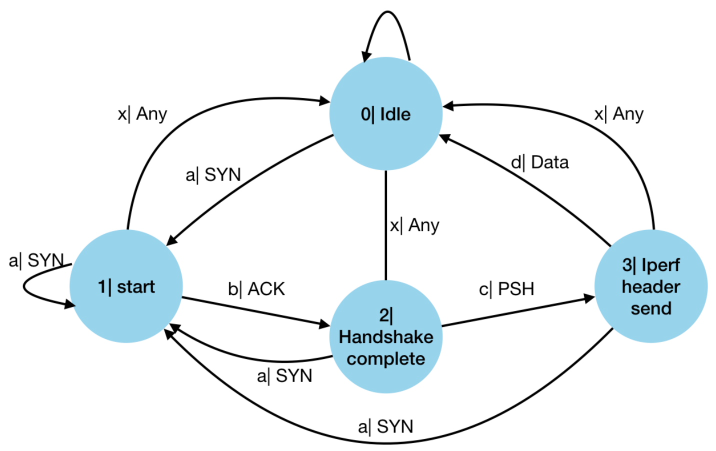


Figure 3.4: State machine of the iPerf starting sequence.

Fig. 3.4 describes the state machine (SM) we use to track TCP flows through their starting sequence. All flows start in idle and move through the SM respectively. Having exactly one state per TCP flow is infeasible due to their large state space. Therefore we use universal hashing [10] to map each flow to one of m buckets. Each bucket contains a number reflecting the current state within the SM. Hashing is necessary to break down the huge amount of possible flows, but it creates collisions. Property 1 allows us to assume that only a few TCP flows are handshaking simultaneously, meaning that almost all collisions will happen to TCP flows being in state 0. When a packet arrives with the SYN flag set, the specific token leaves the idle state towards the starting state 1 (transmission a in Fig. 3.4). Now starts the critical phase because a collision now would lead to a false negative. Property 2 limits this critical phase to only one RRT, lowering the probability of false negatives. The chance of two flows colliding is 1/m. Therefore using more buckets results in a lower chance of false negatives. It is a tradeoff to make between memory overhead and false negative rate.
Once a TCP flow is classified as iPerf (the transmission x from state 3 to 0 in Fig. 3.4), an entry is made in a counting filter [27]. The counting filter is a variation of the Bloom filter that allows counting and deletion. Consequent packets will then check if they have been marked as iPerf. Bloom filter are a space efficient probabilistic data structure used to test if an element is a member of a set. We use counting filter because they allow us to remove entries when an iPerf TCP flow is over. The size of a counting filter determines the false positive rate. Property 3 allows us to assume that the size of the filter can stay quite small while still having a reasonable false positive rate.

3.4.4 Ping

Classifying an ICMP ping packet works the same way as with TCP and UDP: The IPv4 protocol number 1 indicates an encapsulation of an ICMP packet [28]. ICMP messages are differentiated by the Type and Code header fields. Type 8 and Code 0 indicate an echo request packet. Type 0 and Code 0 indicate an echo reply packet. So to classify a ping packet, we match the IPv4 protocol number and the ICMP Type and Code.

3.4.5 Traceroute

To classify a Traceroute packet, we inspect the TTL value. The obfuscation controller knows the path the packet would normally take, and is therefore able to check which hop on the path would respond with an ICMP TTL exceeded message. Knowing the responding hop in advance allows us to set specific obfuscation rules for each hop within a path. This enables us to individually obfuscate each link within the path.

[1]Network protocol analyzer.

3.4.6 Ranking

All these different traffic types work on different OSI-Model [40][2] layers. This allows one packet to get classified as multiple types. Imagine a Traceroute packet send via TCP. It would be classified as IPv4, TCP and Traceroute. To be able to apply only one obfuscation rule to the packet, we rank the classification types. The most specific traffic type here is Traceroute, then TCP and IPv4. So if a rule for Traceroute and IPv4 exists, that both would match with this packet, the Traceroute rule would be taken. Fig. 3.5 shows a podium ranking the priory of each traffic type and section 3.5 will explain how those rules are correctly applied.

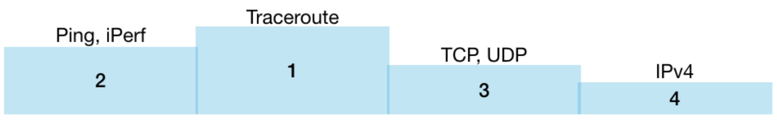


Figure 3.5: Podium showing the priority ranking of different classification types

3.5 Matching Obfuscation Rules

Once a packet is classified, the edge switch needs to check for any matching rules. The keys to macht on are specified in Fig. 3.7. One of our goals is to only deceive selected hosts. A simple approach would be to specify one rule for each source destination pair, but this doesn't scale. Potential use cases would mostly care to specify rules for a few sources to many destinations, or multiple sources to few destinations. P4 allows us to match on IP ranges [33] which, given our assumption, will shrink the number of rules per switch. Each range does only contain IP's that use the same edge switch as an entry point. Each source having only one entry point is a feasible assumption for WAN. Fig. 3.6 shows such a WAN, visualizing groups of hosts only entering via one edge switch.
Generally, IP ranges can overlap. P4 handles those collision cases by requesting a priority per range. We use the priority parameter to specify the rank of classification types and leave the collision avoidance to the obfuscater.
Each rule uses ternary logic to specify what packet type should be considered. Matching the TTL value exactly allows us to detect Traceroutes and specify individual rules for each hop inside our network. This way we can obfuscate each link on the path differently.

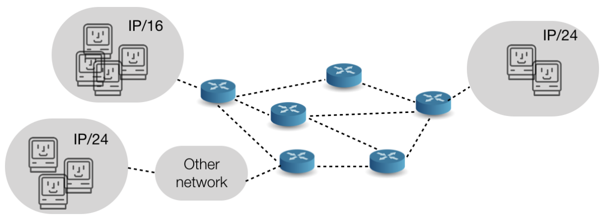


Figure 3.6: Example of a WAN.

```
key = {
    hdr.ipv4.srcAddr : range;
    hdr.ipv4.dstAddr : range;
    meta.ping : ternary;
    meta.iperf : ternary;
    meta.tcp : ternary;
    meta.udp : ternary;
    meta.ipv4 : ternary;
    hdr.ipv4.ttl : ternary;
}
```

Figure 3.7: Keys used to match classified packets with rules.

3.6 Latency Obfuscation

Once a packet is classified and matches an obfuscation rule, we change the performance properties respectively. To increase the latency, a packet needs to spend more time inside the network. We can do this by rerouting the packet on a longer path. This section will show how to reroute a packet and then explain which paths we consider.
We will further refer to latency as the propagation delay, the time a packet spends on a link. By path latency we mean the sum of propagation delay of all links on that path. The target latency

[2]Model partitioning a communication system into abstract layers.

specifies the time we want our packet to be inside the network. We ignore any other delay, because they are ether relatively small or vary heavily, but only in an worsening way.

3.6.1 Source Routing

Source routing allows the sender of the packet to specify how the packet should be routed [32]. Depending on the target latency, we might route each packet differently. Having all routing information for all obfuscation rules on all switches would waste memory, which is already restricted. Instead we store the routing information on top of the packet. We do this by creating a source routing header shown in Fig. 3.8. The header is constructed as an header stack, where each layer specifies what egress port to use, and if we reached the bottom of the stack. Each switch looks at the first layer in the header stack, sets the egress port respectively and deletes the layer. Once the bottom of the stack is reached, the switch will remove the source routing header and the packet will be forwarded as normal IPv4 traffic.

```
header srcRoute {
    bit<7> port;
    bit<1> bottom_of_stack;
}
```

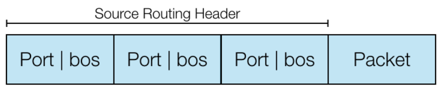


Figure 3.8: Specification of the source routing header, dictating each switch the egress port to use.

3.6.2 Simple Path Routing

A simple path is a path with no repeated nodes. There might be multiple simple paths for each source destination pair. Simple path routing picks the path with the latency closest to the target latency.
The routing method will not create congestion on its own, meaning that it creates no loops filling up any switch queue. This is useful if we want to reroute a large flow on a sparsely used network.
One disadvantage is that the packet might be routed completely different than the path it would normally take. This will affect the usability of network debugging tools such as Traceroute. To find all simple paths we use a modified depth first search algorithm [5].

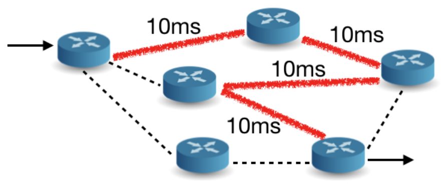


Figure 3.9: Simple path example

3.6.3 Same Path Routing

To make our network more debuggable while still obfuscating the latency, we use Same path routing. By Same path we mean to keep the packet on the same path it would normally take. Each link on that path can be used multiple times. We combine the links to give us a path latency closest to the target latency. The network becomes more debuggable because our packets never leave the intended path. Therefore any malfunctioning switch on that path can be correctly identified.
The Same path closest to target latency is found in the following way: We reduce the real network to only include the normally taken path. Next we run the Any path routing algorithm, explained in 3.6.4, on the reduced network.

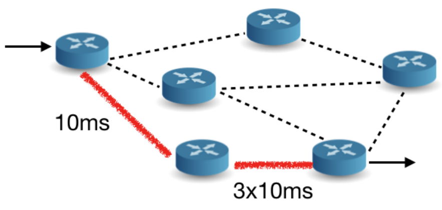


Figure 3.10: Same path example

3.6.4 Any Path Routing

Both previous methods have their legitimacy, but they are quite restricted in how close they can get to the target latency. Any path routing allows to take any possible path that has the latency closest to the target latency. This can lead to a path that contains multiple loops, thus sensible to congestion when rerouting large flows. For small packets like Pings or Traceroutes this is assumed to work fine.

The algorithm that finds the optimal path closest to target latency is a modification of the depth first search algorithm called depth limited search [30]. By limiting the number of hops in our path we can significantly decrease the complexity of our search problem. The optimum path is found by exploring all possible paths. Each node in our network knows the shortest path and the minimal number of hops towards the destination. This allows us to cut off an exploration once we are certain our current path will not improve upon the best path we already have.

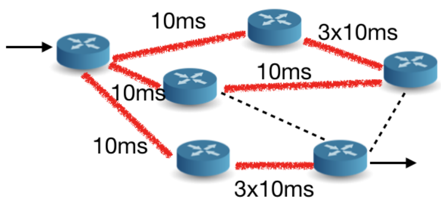


Figure 3.11: Any path example

3.7 Packet Loss Obfuscation

As described in 2.4.2, the packet loss is defined as the rate of packets not reaching the destination correctly. We assume the network to have zero loss rate. To increase the loss rate, we can simply drop more packets. To do this, we need some way to track the packet drops and to keep the actual loss rate fixed on the target loss rate. We do this by leveraging the uniform distribution property of the pseudo random number generator [34] provided as an external function in P4. For all packets with a loss rate set, we draw a random number between 1 and 1000, and check if it is below the rate. If yes, we drop it. All numbers are equally likely to be drawn, which then leads to a packet being dropped by the probability of our loss rate. This method allows us to drop packets on average at the specified rate, without having to count the packets or track the rate.

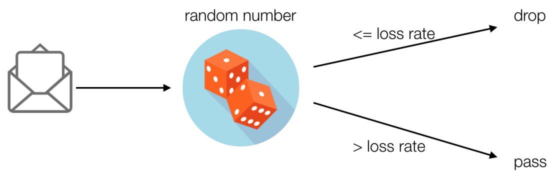


Figure 3.12: Packet loss decision-maker. A pseudo random number generator determining which packets to drop.

3.8 Throughput Obfuscation

Bandwidth is the maximum amount of data successfully reaching the destination per time. It is measured in data per time, e.g. Megabits per second (Mbps). Therefore lowering the bandwidth means to decrease the amount of data per time making it to the receiver. We can do this by dropping packets. Normally a network only drops packets when it becomes congested or the capacity of the link is reached. We simulate these effects by rate limiting the traffic with a 2 rate 3 color Marker. This section will first look at how we choose the Meter parameters for bandwidth throttling. Additionally we will examine an idea to manipulate the TCP window for bandwidth obfuscation.

3.8.1 Choice of Meter Traffic Parameters

Our goal is to throttle the bandwidth to a specific target rate. We will use an example to understand how the Meter parameters are chosen to reach that target rate.
Imagine having a link with capacity of 10Mbps that we want to throttle to 5Mbps. This already provides us with the value to use for PIR and CIR. But how should we chose the burst size?

The burst size is the number of bytes used to measure the rate. This means that first 'burst size many' bytes pass the Meter without any restriction. Let us consider the two buckets in Fig. 2.3 again. As long as the buckets are not empty, bytes can pass our Meter. Each passed byte subtracts a token from the bucket. Bytes are added to the bucket at Information Rate. When more bytes arrive than the Information Rate can add, we eventually start marking packets as yellow or red. Marked packets can then be dropped. So if the bucket size is too low, too many packets will be subject to rate limiting. If the bucket is too big, too few packets will be rate limited. This leads to a minimum burst size of one Maximum Transmission Unit (MTU), because all bytes of one packet arrive at once. The upper limit is determined by how many bytes we allow to blast trough. We chose to have something tracking a long term average, and something to catch large blows: We want the CIR to be measured over a time period of one second. On a 10Mbps link, this results in 10Mbit = 1.25Mbyte Committed Burst Size. With a Maximum Transmission Unit (MTU) of 1500 Bytes, the CBS lets trough $\frac{1.25MByte}{1500Byte}$ ≈833 packets. Because only a small number of packets might already reveal our bandwidth obfuscation, we need something to catch large traffic blows. When we look at TCP, it almost sends a full window at once. The TCP window field [19] has 16 bits specifying the amount of bytes. Therefore the larges window is 65K bytes, without window scaling[3]. 65K bytes are 43.33 MTU's. We set the Peak Burst Size to 66K bytes, equating to 44 packets. So TCP can send at most one full window at a time before the rate is applied.

Both of these values are chosen by an educated guess, but as we will see in the evaluation section they seem to work quite well.

3.8.2 TCP Window Size Manipulation

For a TCP [17] connection, the sending rate is negotiated between the sender and receive. The sender announces what it can send, and the receiver what it can receive. TCP does that by a mutual agreement specified in the TCP receiver window field [19]. TCP will never send more than what is specified in the receiver window field. The network has no say in what rate it can or wants to allow. The only way for a network to communicate to the sender for changing the sending rate is by dropping packets [18]. Programmable network devices open the possibility for the network to change the receiver window field, and therefore manipulate the TCP sending rate remotely. This allows for rate limiting of the sender without having to drop any packets and therefore without pretending the link to be congested. At first we pursued this idea as it seemed to have great potential. But we removed it of our final framework due to the following reasons:

- To assign a specific bandwidth to a TCP flow we would need to calculate the TCP window respectively. This can only be done by having some knowledge of the RTT. The RTT though is not known and would have to be measured and stored for the whole session.

[3]Window scaling is a TCP extension for higher performance [19]

- This means that per TCP flow we would need to store a state. This does not scale due to the number of possible flows.
- Also obfuscating a link capacity for a source destination pair should fairly split the bandwidth among all flows through that link, as TCP congestion control would do. When using TCP window manipulation though, we would need to know, recalculate and update all windows of all flows sharing that obfuscated link.

Chapter 4

Evaluation

In this chapter, we evaluate the accuracy of our obfuscation framework and look at the impact on a network. We first describe the setup on which the framework is implemented and run. Then we evaluate the quality of our performance obfuscation.

4.1 Setup

All experiments are conducted on a virtual machine running Ubuntu 16.04 LTS 64 bit on a MacBook Pro with a 3.1 GHz Intel Core i7 and 8 GB of RAM.
On the virtual machine, we use Mininet [4] to create and run virtual networks. The behavioral model lets us compile P4 programs to a P4 software switch, which can then be used within Mininet. For this project we use the second version of the behavioral model (bmv2) [1]. As a target architecture for the bmv2, we use the simple_switch architecture [2]. Throughout our evaluation, we will emulate multiple different networks with Mininet. One reason is to show that our framework adapts to different networks, but mainly because we are only interested in measuring specific properties and fit the network respectively.

4.2 Packet Loss Obfuscation

Evaluating the packet loss rate is quite straight forward: We install an obfuscation rule for a source destination pair with a target loss rate (red line in Fig. 4.1). Next we sample the packet loss obfuscation by sending 1000 Pings from source to destination and compare it with the number of received answers. There is no other traffic going on in the network, which means that no loss can occur due to congestion, as can be seen in the first 10 units of Fig. 4.1.

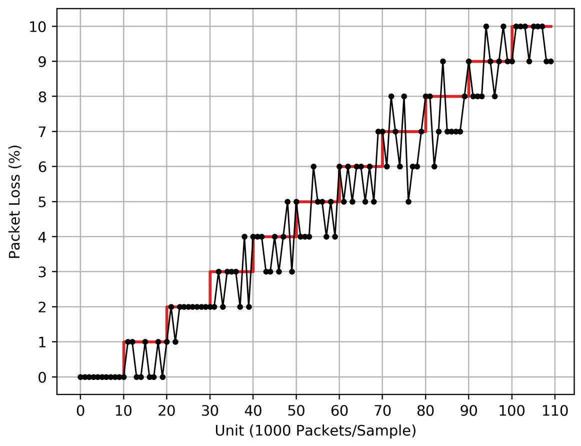


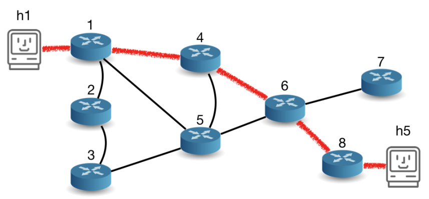
Figure 4.1: Packet loss obfuscation. Measuring the packet loss rate by counting the number of answered Ping requests. On average, the loss rate is below or on the target loss, but rarely overshoots.

4.3 Latency Obfuscation

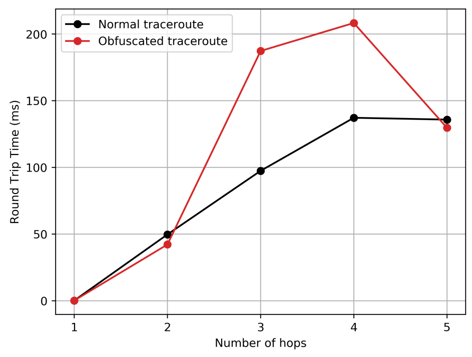
Latency obfuscation allows us to stretch individual links or paths. This section will evaluate the obfuscation of individual links and then look at the accuracy of latency routing and its overhead on a network. For this evaluation, the source header stack can only specify a maximum of 10 hops.

4.3.1 Delay Individual Links

Next we check if individual link latencies can be obfuscated. Therefore we use Traceroute to measure the RTT to each hop on the path. We consider the path from Host h1 to Host h5 in Fig. 4.2a.
To show that individual link delays can be obfuscated, we run a Traceroute without any obfuscation first (black line in Fig. 4.2b)). Next, we set an obfuscation rule increasing the latency to hop 3 by 80ms and to hop 4 by 60ms. When repeating the Traceroute (red line), we can see that the RTT to hop 3 and 4 increased as expected. Notice that the latency to hop 5 (h5) remains unchanged.



(a) Network showing the traced path.



(b) RTT measured by Traceroute to each hop on the path from h1 to h5.

Figure 4.2: Obfuscation of individual links to deceive Traceroute.

4.3.2 Accuracy of Latency Routing

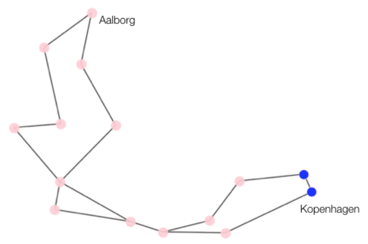
Next we evaluate how close *Any path routing* can find a path to target latency. To make things more realistic, we pick a real network from Topology Zoo [3]. Two directly connected nodes are then chosen as source S and destination D, marked blue in Fig. 4.3a. Started from the shortest path between S and D, the target latency is evenly increased. For each step, the Any path routing algorithm finds the path with a latency closest to target latency. Fig. 4.3b illustrates the result and additionally shows how many hops are needed to reach the target latency.

4.3.3 Overhead of Latency Routing

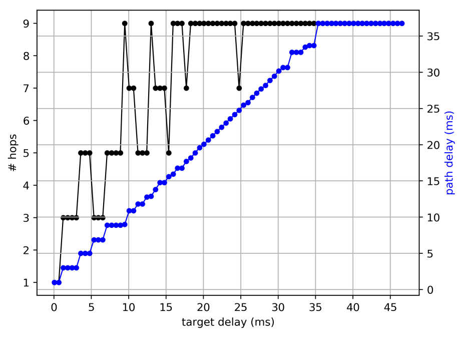
If we look at the target delay value of 12ms in Fig. 4.3, we can see that for a small increase in latency, the number of hops almost double, but increasing the delay a bit further bisects the number of hops again. Because the current setup is heavily network dependent, we will now look at a way to compare *Any path routing* among different network topologies and node pairs.

Instead of specifying the target delay by a specific number, we define it to be the percentage increase from the shortest path between two nodes. This allows us to compare the number of nodes versus the growth of the delay among different networks. To get kind of an average per network, we sample the network in the following way:

1. Randomly pick a node X in the network.



(a) Network showing the Uni-C Backbone in Denmark. The blue node pair indicates start and destination of the path.



(b) The blue line shows the actual path latency vs. the target latency. The black line indicates the number of hops needed to reach the target latency.

Figure 4.3: Accuracy of Any path routing and the influence on the number of hops.

2. Randomly pick a node Y that is n hops away from X when considering the shortest path.
3. If the X,Y pair has previously been sampled, redo step 1 and 2.
4. Calculate any path closest to target latency.
5. Repeat this 5 times and take the average number of hops per percentage increase.

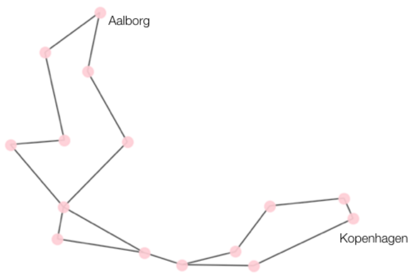
We sample the network for n = 1,3, and 5. This evaluation method is applied on three different networks that differentiate ether in size or in the average degree per node. Fig 4.4 shows the result per network.

4.4 Throughput Obfuscation

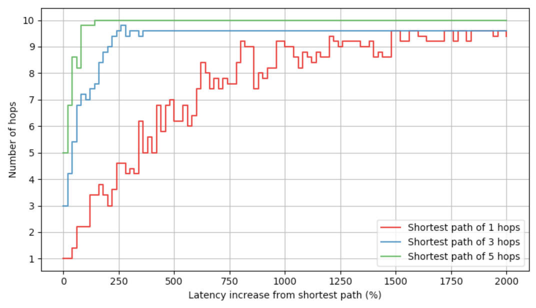
Next we evaluate the bandwidth throttling. We are interested in knowing how well the Meter regulates the traffic and if the obfuscation might be revealed. Fig. 4.5 is the result of the following setup:

- We use the two networks shown in Fig. 4.5a-4.5b, where every link has a capacity of 10Mbps. The network is completely empty to exclude interaction between measurement and other traffic.
- The Meter is set to increase the throttling over time, shows by the decreasing red line in Fig. 4.5c-4.5h.
- The bandwidth is then measured with iPerf3 using TCP traffic.
- The iPerf3 *Sender* measures the upload rate, whereas the iPerf3 *Receiver* measures the download rate. The sender also provides information about packet retransmission and TCP window size.
- Fig. 4.5e - 4.5h uses the same method, but the Meter only sets the yellow or red parameters respectively.

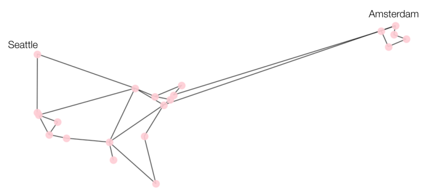
Let us look at the results in Fig. 4.5. In Fig. 4.5c and 4.5d we notice that the receiver rate never overshoots the bandwidth limit set by the meter. We now look at the retransmissions and sender rate. We can see that every time the sender wants to send more packets than allowed by the meter, all packets get dropped and need to be resend. If we look at the influence of the yellow parameters on our meter, Fig. 4.5e and 4.5f, we notice that at the start of each measurement a blow of packets burst the Meter, but then get regulated after some seconds. Fig. 4.5g and 4.5h on the contrary show that those bursts are limited. An other thing to notice is the influence of the RTT on throughput. The throughput is the average amount of data over time, so the area below the black line of the sender and receiver plots. Figures in the left column, with a short RTT, have a higher throughput than the ones on the right with a larger RTT.



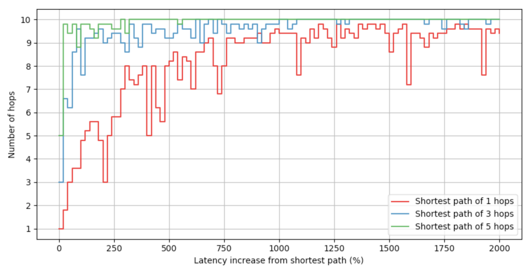
(a) Uni-C, DEN; Mean degree: 1.13; Aggregated link size: 826Km



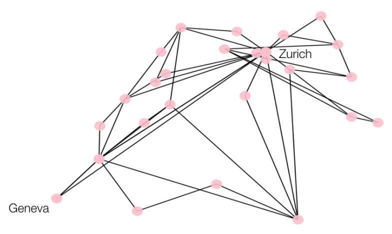
(b) Latency increase vs. #hops on network (a)



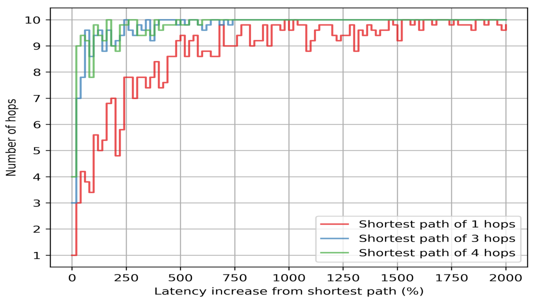
(c) Bandcon, USA/EU; Mean degree: 1.33; Aggregated link size: 34365Km



(d) Latency increase vs. #hops on network (c)



(e) SwitchL3, CH; Mean degree: 1.7; Aggregated link size: 3586Km



(f) Latency increase vs. #hops on network (e)

Figure 4.4: Latency routing overhead. Increase of latency compared to the number of hops for different networks.

4.5 Discussion

Overall, our results show that our framework successfully obfuscates the performance properties, but let us take a closer look at each individually:

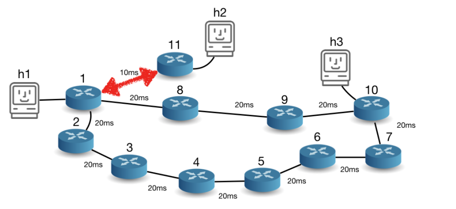
Increasing the **packet loss rate** (Chapter 4.2) by a certain percentage works as expected, meaning that we are on average within 1% to the target percentage. On average our loss rate seems to be rather below than above the actual target percentage. This might ether be due to some bad luck, or to some non uniform behaviour of the pseudo random number generator. In a real scenario with numerous traffic flows, additional losses would be measured due to possible congestion.

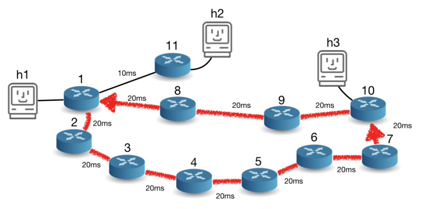
The results of the **latency obfuscation** (Chapter 4.3) show that we can increase the latency of individual links within paths. Depending on the network and the source destination pair, the latency routing algorithm can find paths with a latency really close to the target latency. From Fig. 4.3 we can conclude that a small increase in latency can mean a duplication in hops. Increasing the latency a bit further though can result in a bisection again. Therefore we can say

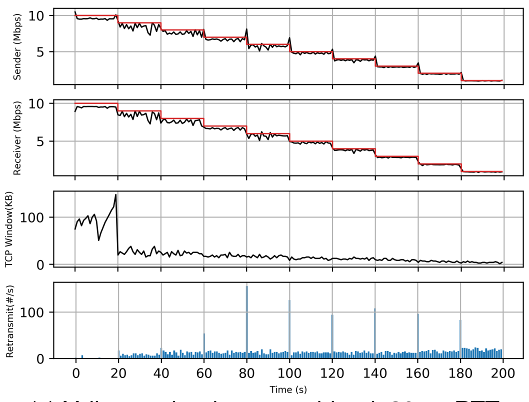
that the number of hops taken is very network dependent. Overall, an increase in latency will increase the number of hops, as the path must become longer.
If we consider latency obfuscation on different networks, Fig. 4.4, we can see that the number of hops approximately doubles when the latency is increased by 100%, independent of the underlying network. Additionally we can see that the growth rate of hops vs. the increase in latency behaves similarly for each network. In contrary, it changes significantly for further spaced nodes, already consuming many hops for the shortest path. Those used up nodes are missing to reach larger latencies, meaning that the latency increment reaches its limit faster. We see this effect in the faster flattening of the curve. The boundary of the curves at 10 hops is due to the limited number of hops our source routing header can provide.

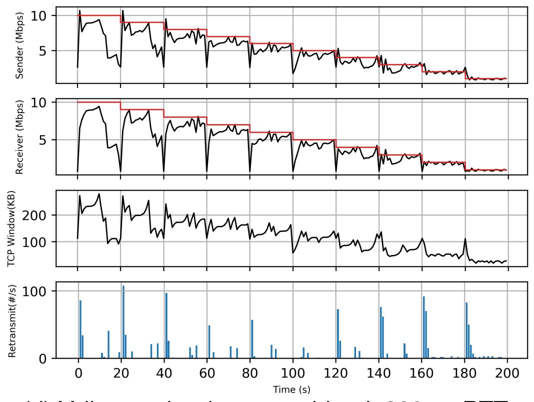
Next we discuss the **bandwidth obfuscation** (Chapter 4.4). Fig. 4.5c shows that for a short RTT the bandwidth throttling works perfect: There is no burst blowing through our rate limitation that would reveal our obfuscation. Also we notice that the sender and receiver rate stay close to the target bandwidth, meaning that we almost have the highest possible throughput. At the beginning of some new measurements, like at the 80th second, we see that TCP slow start tries to pierce the Meter, but most packets get dropped as can be seen in the retransmission section.
For large RTT, Fig. 4.5d shows that there is still no traffic burst breaking through the Meter. Therefore our obfuscation also holds here. But when looking at the throughput and the retransmissions, we see that the overall throughput is less. We notice to have a larger TCP window, which is due to the larger distance. The combination of a larger window and a slower response time leads to a more shifted and overshooting response, resulting in an overall decreased throughput.
The impact of the committed burst size and the peak burst size can be seen in Fig. 4.5e to 4.5h. Recall that the yellow rate has a burst size of one second. This means that at every new measurement there is an unlimited burst of 833 packets blowing through the Meter, before the committed information rate gets applied. We can very well see these peaks in the *only yellow rate* plots, but we also see that shortly after, the Meter starts to regulate the traffic as expected. The evaluation of the peak burst size in the *only red rate* plot indicates that the red rate is probably the one leading to our good bandwidth throttling results. It almost looks identical to the plot with the combined rates.

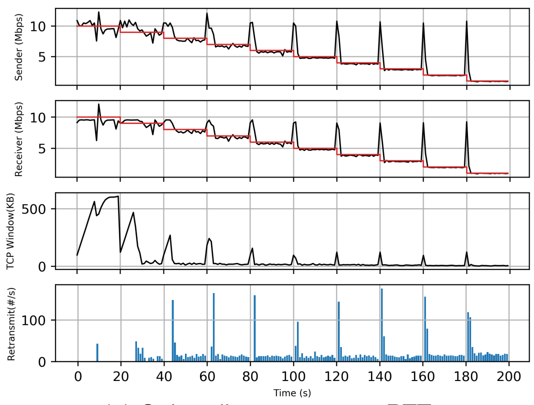
Running all the measurement on a **virtual environment** presumably creates some misbehavior by adding workload dependent noise. Almost all values of our results look as expected, except Fig. 4.5f. Each link on the test network has the maximum bandwidth set to 10Mbps, but as we can see, we get peaks of up to 20Mbps. A feasible explanation might be a particular combination of queued packets on the path, the way iPerf measures and rounds values and the threading of Mininet.

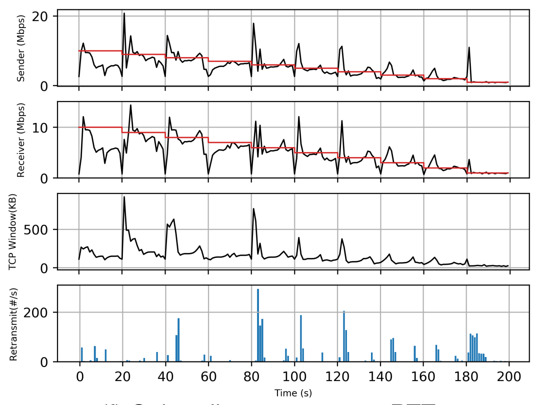

(a) Path with 20ms RTT.

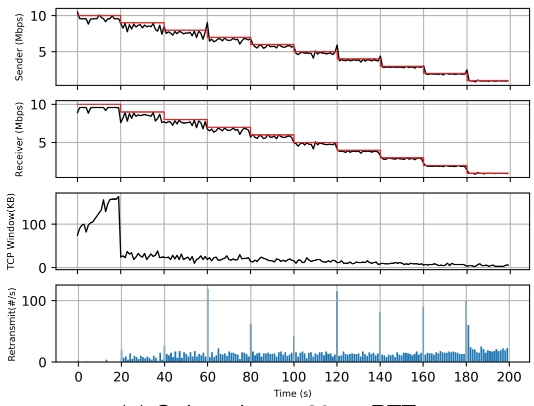

(b) Path with 200ms RTT.


(c) Yellow and red rate combined, 20ms RTT


(d) Yellow and red rate combined, 200ms RTT


(e) Only yellow rate, 20ms RTT


(f) Only yellow rate, 200ms RTT


(g) Only red rate, 20ms RTT

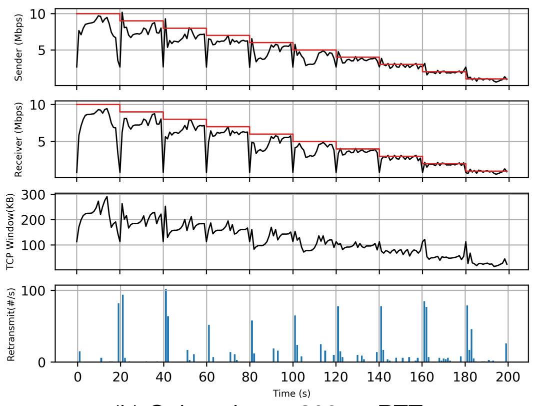

(h) Only red rate, 200ms RTT

Figure 4.5: Evaluation of bandwidth throttling via Meter. (a) and (b) show the path taken by the iPerf flow in the plots beneath. (c) and (d) evaluate the Meter in normal mode, used to throttle the bandwidth. To see the influence of each color, plot (e) and (f) evaluate the Meter only using the yellow settings and (g) and (h) by only using the red settings.

Chapter 5

Conclusion

This chapter concludes the thesis. We summarize our contribution, draw a conclusion and finally give an outlook on possible future work.

5.1 Summary

In this thesis, we designed and implemented a framework to obfuscate network performance properties. The properties considered are latency, packet loss and bandwidth. By using P4, we were able to worsen all properties directly on the data plane. Our obfuscation controller calculates and sets all obfuscation rules on the responsible edge switch. Each edge switch contains all rules for IP sources that use it as an entry point. A rule matches on the source and destination IP range and traffic type. A packet entering the network will first be classified, then matched with existing obfuscation rules and obfuscated respectively. To increase the packet loss rate, we use a pseudo random number generator and drop the packet if the number drawn is below our rate. The latency is increased by source routing the packet, based on a precalculated path. Bandwidth throttling is done by using a Meter with the obfuscated rate as Information Rate, a Peak Burst Size of 44 packets and a Committed Burst Size of 44 Packet. The evaluation of our framework shows that performance properties can be worsened to reach specified values.

5.2 Conclusion

In conclusion, our proposed framework is able to deceive selected hosts by providing them with individually obfuscated performance properties. Our evaluation shows that we can worsen the latency, bandwidth and packet loss to specific target values. Furthermore, we do not reveal any obfuscation when the network is examined with iPerf3, Ping or Traceroute. However it is likely that any other performance measuring protocol discovers our deception, ether by avoiding our rules or by fooling our obfuscation tools.

Normal production traffic is usually routed to reach maximal performance. This makes improving any parameters difficult as most packets need to be answered by their intended hosts. Overall we can say that worsening the parameters is much easier than increasing them.

An other shortcoming of our system is the lack of robustness for packet classification. The exact classification of packets into production and measurement traffic is quite challenging. Any normally routed packet already reveals information, but most often multiple packets are needed to detect their intention.

5.3 Outlook

With the advances in programmable switches and the implicated increase in network control, we can think of many possible ways to improve upon this thesis. We will only discuss some of them:

Instead of increasing the **latency**, we could try to **decrease** it. Traffic with an intended destination we control could be replied by an earlier device also in our control. A good use case here would be decreasing the RTT's from Traceroute and Ping packets probing our network. More generally though an early reply could work for any connection-less communication protocol where we know the answer for. This would allow us to catch ping packets and impersonate the intended destination. A different approach to decrease the latency could be to use priority queuing at every switch and thereby remove the queuing delay.

Increasing the bandwidth basically means to strengthen the weakest link. The idea here is to reroute all traffic to free up the weakest link. Assuming that mostly each link is shared by more than one flow, an attacker might suppose an unused link to be unlikely. Therefore she concludes that the link must be quite big because she measures a higher bandwidth. An other way is to use priority queuing where we drop all other packets before the packets of interest. This results in a smaller loss rate and therefore increases the bandwidth.

Manipulating the TCP window could possibly be used to **increase the throughput**. TCP sends at most the amount of data specified in the TCP receiver window. The maximum window size is determined between the sender and receiver, with the goal to not overload the receiver. This negotiation excludes the network. The only way the network can communicate with the sender is by dropping packets. When the window is larger than the smallest available bandwidth on the path, TCP will have the typical saw blade behaviour because it will repeatedly lose packets. The saw blade behaviour though is not the optimal throughput that could possibly be reached. If the network would manipulate the TCP receiver window, the network could possibly avoid the losses due to congestion avoidance and receive a higher throughput. How much this would increase the throughput remains a question, as there might be unexpected behaviour with other traffic still introducing packet loss.

Bibliography

[1] Behavioral model version 2. https://github.com/p4lang/behavioral-model. (visited on 01/06/2019).

[2] The bmv2 simple switch target. https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md. (visited on 01/06/2019).

[3] The internet topology zoo. http://www.topology-zoo.org/. (visited on 02/06/2019).

[4] Mininet. http://mininet.org/. (visited on 01/06/2019).

[5] Networkx: All simple paths. https://networkx.github.io/documentation/latest/reference/algorithms/generated/networkx.algorithms.simple_paths.all_simple_paths.html. (visited on 03/06/2019).

[6] Protocol numbers. https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml. (visited on 03/06/2019).

[7] Wireshark. https://www.wireshark.org/. (visited on 03/06/2019).

[8] S. Ali, G. Wang, R. L. Cottrell, and T. Anwar. Detecting anomalies from end-to-end internet performance measurements (pinger) using cluster based local outlier factor. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 982–989, Dec 2017.

[9] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44, 12 2013.

[10] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[11] D. Constantinescu and A. Popescu. Modeling of one-way transit time in ip routers. In *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, pages 16–16, Feb 2006.

[12] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. iperf3: iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. (visited on 01/06/2019).

[13] E. Haleplidis, K. Pentikousis, S. Denazis, J. H. Salim, and D. M. O. Koufopavlou. Software-defined networking (sdn): Layers and architecture terminology. https://tools.ietf.org/html/rfc7426#, 2015. (visited on 14/05/2019).

[14] J. Heinanen and R. Guerin. A two rate three color marker. https://tools.ietf.org/html/rfc2698, 1999. (visited on 01/06/2019).

[15] C. Hornig. A standard for the transmission of ip datagrams over ethernet networks. https://tools.ietf.org/html/rfc894. (visited on 03/06/2019).

[16] Iana. Ieee 802 numbers. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1. (visited on 03/06/2019).

[17] Information Sciences Institute, University of Southern California. Transmission control protocol. `https://tools.ietf.org/html/rfc793`, 1981. (visited on 01/06/2019).

[18] V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 18(4):314–329, Aug. 1988.

[19] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. `https://tools.ietf.org/html/rfc1323#`, 1992. (visited on 01/06/2019).

[20] T. Keary. Throughput vs bandwidth: Understanding the difference plus tools. `https://www.comparitech.com/net-admin/throughput-vs-bandwidth/`, 2018. (visited on 15/05/2019).

[21] LiveAction. Propagation delay. `https://www.savvius.com/networking-glossary/ethernet/propagation_delay`. (visited on 15/05/2019).

[22] G. Malkin. Traceroute using an ip option. `https://tools.ietf.org/html/rfc1393`, 1993. (visited on 01/06/2019).

[23] S. Mastorakis, J. Gibson, I. Moiseenko, R. Droms, and D. Oran. Icn ping protocol specification. `https://tools.ietf.org/id/draft-mastorakis-icnrg-icnping-03.txt`, 2018. (visited on 01/06/2019).

[24] R. Meier, D. Gugelmann, and L. Vanbever. iTAP: In-network traffic analysis prevention using software-defined networks. In *Proceedings of the Symposium on SDN Research*. ACM, 2017.

[25] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev. NetHide: Secure and practical network topology obfuscation. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[26] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for internet hosts. *ACM SIGCOMM Computer Communication Review*, 31, 08 2001.

[27] P. Pandey, M. A. Bender, R. Johnson, and R. Patro. A general-purpose counting filter: Making every bit count. pages 775–787, 2017.

[28] J. Postel. Internet control message protocol. `https://tools.ietf.org/html/rfc792`, 1981. (visited on 01/06/2019).

[29] J. Postel. Internet protocol specification. `https://tools.ietf.org/html/rfc791.html`, 1981. (visited on 01/06/2019).

[30] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, chapter Depth-limited search, pages 87–88. Prentice Hall Press, 3rd edition, 2009.

[31] A. Studer and A. Perrig. The coremelt attack. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 37–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] C. A. Sunshine. Source routing in computer networks. *SIGCOMM Comput. Commun. Rev.*, 7(1):29–33, Jan. 1977.

[33] The P4 Language Consortium. $p4_{16}$ language specification. `https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.pdf`, 2018. (visited on 01/06/2019).

[34] The P4.org Architecture Working Group. P4 pseudo random number generator. `https://p4.org/p4-spec/docs/PSA.html#sec-random`, 2019. (visited on 01/06/2019).

[35] L. Vanbever. Course on advanced topics in communication networks. `https://adv-net.ethz.ch/pdfs/03_stateful.pdf`, 2018. (visited on 01/06/2019).

[36] Wikipedia. Processing delay. `https://en.wikipedia.org/wiki/Processing_delay`, 2016. (visited on 15/05/2019).

[37] Wikipedia. Queueing delay. `https://en.wikipedia.org/wiki/Queuing_delay`, 2018. (visited on 15/05/2019).

[38] Wikipedia. Network delay. `https://en.wikipedia.org/wiki/Network_delay`, 2019. (visited on 15/05/2019).

[39] Wikipedia. Transmission delay. `https://en.wikipedia.org/wiki/Transmission_delay`, 2019. (visited on 15/05/2019).

[40] H. Zimmermann. Osi reference model - the iso model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, April 1980.