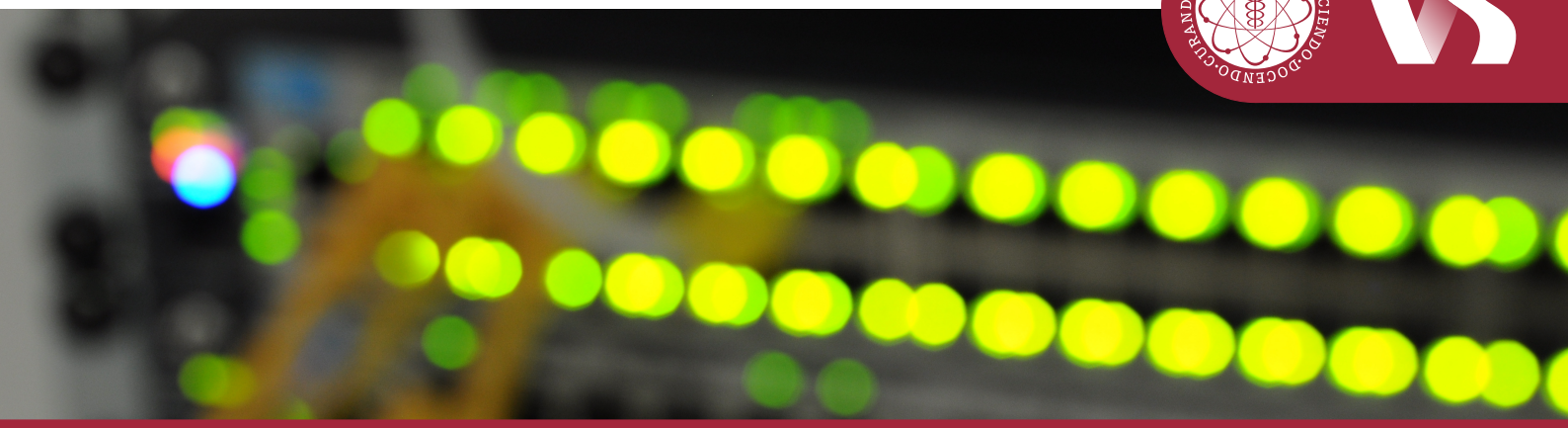

Approaches to anonymize network connections and circumvent censorship

SLR: Sourceless Routing using split network routes over Layer 3

Amjad Mashaal
Bachelor's Thesis

Examiner: Prof. Dr. rer.nat. Frank Kargl
Supervisor: Leonard Bradatsch
Submission Date: August 27, 2019

Issued: August 27, 2019

I hereby declare that this thesis titled:

Approaches to anonymize network connections and circumvent censorship
SLR: Sourceless Routing using split network routes over Layer 3

is the product of my own independent work and that I have used no sources or materials other than those specified. The passages taken from other works, either verbatim or paraphrased in the spirit of the original quote, are identified in each individual case by indicating the source.
I further declare that all my academic work was written in line with the principles of proper academic research according to the official "Satzung der Universität Ulm zur Sicherung guter wissenschaftlicher Praxis" (University Statute for the Safeguarding of Proper Academic Practice).

Ulm, August 27, 2019

Amjad Mashaal, student number 1042967

Abstract

True low-latency high-throughput privacy-maintaining systems have been always sought after by the cybersecurity community. Since its induction, those seeking privacy would mainly rely on Tor. However, Tor's latency and throughput are simply not good enough for the current era of web platforms. We introduce a new scheme, named SLR, that relies on omitting source-identifying information from the L3 packets altogether, as well as separating forward and backwards network routes. In this thesis we introduce the core concept of this scheme and we compare its privacy-maintaining capabilities to those of the state-of-the-art low-latency privacy solution, Tor. We also discuss the design of a prototype that we used to evaluate the scheme's performance. Our measurements show that our prototype of SLR can provide speeds of up to 14.5 Mbps, and that further optimisations could increase that limit. The results also point to SLR providing at least the same level of privacy as Tor, despite requiring further extensive security analysis to confirm or refute that.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Research Questions . . . 1
- 1.3 Related Work . . . 2

2 Core Concept **3**
- 2.1 Design . . . 4
- 2.2 Discovery and Support Infrastructure . . . 6

3 Implementation **7**
- 3.1 Circuit Establishment . . . 8
  - 3.1.1 First Header . . . 11
  - 3.1.2 Second Header . . . 13
- 3.2 Communication Over Established Circuit . . . 14

4 Prototype **23**
- 4.1 Circuit Establishment . . . 24
- 4.2 Communication Over Established Circuit . . . 25

5 Effects on Higher Layer Protocols **27**
- 5.1 ICMP . . . 27
  - 5.1.1 ICMP Error Message Type 1: Destination Unreachable . . . 27
  - 5.1.2 ICMP Error Message Type 2: Packet Too Big . . . 28
  - 5.1.3 ICMP Error Message Type 3: Time Exceeded . . . 28
  - 5.1.4 ICMP Error Message Type 4: Parameter Problem Message . . . 29
- 5.2 UDP . . . 29
- 5.3 TCP . . . 29
- 5.4 Summary . . . 30

6 Measurements **31**
- 6.1 Circuit Establishment . . . 33
- 6.2 Communication Over Established Circuit . . . 34
  - 6.2.1 ICMP . . . 35
  - 6.2.2 UDP . . . 35
  - 6.2.3 TCP . . . 37

6.3 Analysis 43

7 Security Evaluation **45**
7.1 SLR Security Analysis 45
7.1.1 Discovery 46
Discovery of Keys and Forward Secrecy 46
7.1.2 Circuit Establishment 47
7.1.3 Communication Over Established Circuit 48
Probability of Revealing Identities 49
7.2 Comparison With Tor 51

8 Conclusion **55**
8.1 Research Questions 55
8.1.1 Can we design a scheme that relies on omitting the source address to achieve anonymity comparable to Tor? 55
8.1.2 What consequences will this scheme have on higher level networking protocols and how can they be addressed? 55
8.2 Future Work and Final Thoughts 56
8.2.1 Forward Secrecy 56
8.2.2 Comparison With Tor 56
8.2.3 Prototype 57

Bibliography **59**

1 Introduction

The need of a low-latency, high-throughput, pseudo-anonymous network has existed since the early days of the Internet, giving rise to multiple solutions starting with David Chaum's concept of mixer networks introduced in 1981 [1]. From there-on, multiple other systems followed, including Freedom Network [2], Cebolla [3], and Tor [4]. While the other systems have died out, with Freedom being discontinued in 2001 [5], and Cebolla's last update being in 2003 [6], Tor continues to remain the state-of-the-art solution [7].

1.1 Problem Statement

Tor introduces a high level of practical anonymity by maintaining a high ratio of noise generated by the vast number of users of the Tor network. The default setting uses 3 mixer nodes that aim to obfuscate the identity of both end-hosts in a manner that requires an adversary to observe the network flows through all mixer nodes in order to deterministically reveal the identities of the end-hosts.

However, the introduction of the intermediate nodes negatively affects latency and bandwidth. As shown in [8], the mean time until receiving first response byte is between 2 to 3 seconds, and the time taken to complete a 5 MiB request is more than 100 seconds. Measurements conducted by Akamai [9] during 2009, the year in which the Tor tests were conducted, show that the average global internet speed at the time was 1.7 Mbps, meaning that downloading a 5 MiB file should have taken on average 23.5 seconds over a direct connection.

The problem can be summarised as follows: the state-of-the-art solution has an average bandwidth rate of only 25% of the global average bandwidth.

1.2 Research Questions

We break down the problem and summarise it using 2 research questions that we attempt to answer in this thesis.

1. Can we design a scheme that relies on omitting the source address to achieve anonymity comparable to Tor?

2. What consequences will this scheme have on higher level networking protocols and how can they can be addressed?

1.3 Related Work

Before Tor, multiple schemes were introduced that provide pseudo-anonymity for a large number of users.

The Freedom System [2] is one of the earliest examples of such schemes, relying on the basic idea of mixer networks introduced by David Chaum [1] in 1981. Freedom System actively omits identifying information such as the source IP address, TCP and IP checksums, and other fields. A pseudo-identity named Nym is used to identify clients anonymously. Nyms are controlled by a central server maintained by the operator of the system.

In 2002 Zach Brown introduced a scheme that resembles Tor even more closely, named Cebolla [3]. Cebolla relies on UDP tunnels between multiple links and routes active traffic by re-encapsulating the packets in a similar manner to Onion Routing [10].

All these solutions were overtaken by the current state-of-the-art, Tor. The origin of Tor is Onion Routing [10], a mechanism that allows secure and anonymous communication between two end-hosts over intermediate nodes by re-encapsulating the packets under multiple layers of encryption, each layer intended to be decrypted by only one node. Onion Routing allows intermediate nodes to route the packets without knowing the identities of both end-hosts, or the content of the data they are routing. Tor was released publicly in 2003 [11].

The previous schemes explored the idea of re-encapsulation and omission of identifying information to preserve privacy, but none explored circular routing over split network routes. This idea was discussed by Takeuchi et al. in [12], despite being for different purposes. The paper explores using node-disjoint forward and backward paths, but it focuses on that from a different perspective, performance, and within a different environment, mobile ad hoc networks. It does not discuss the effect of splitting the routes on privacy. Their findings show that splitting the routes alleviates packet collision, and thus improving the network's performance.

In this thesis we merge both these concepts to design a privacy system that maintains anonymity with less overhead than Tor by relying on circular routes, Onion Encryption, and omission of meta-data that identifies the source of the packets.

2 Core Concept

The core concept of SLR relies on two essential ideas: omitting the source's IP address, and circular routing. The idea is that the delivery of data does not rely in any way on the existence of data identifying the source. We attempt to explore this concept in networks and introduce a protocol that is able to route messages correctly without relying on plain-text information identifying the source of these messages.

The reasoning behind this approach is that the omission of information identifying the source simply hides the source: packets with untraceable sources provide privacy that increases without requiring extra mixing infrastructure. The routing infrastructure itself becomes the mixer network. Each layer of routing that a packet goes through contributes to the obfuscation of the identity of the source.

As shown in figure 2.1, a packet has multiple paths to go through the network. As the point of observation moves from the source to the destination, the set of possible sources grows, making it harder to identify the actual source. This is all done without requiring any dedicated infrastructure that acts as a mixer network. The routing infrastructure itself becomes a huge mixer network.

This concept however introduces some hurdles. Letters have a source address on them so that if the delivery infrastructure fails to deliver the letter, the failure can be reported to the sender. The same goes for the internet. There is a feedback system that the internet uses to inform the source of packets of the status of delivery and provides suggestions to guarantee delivery of following packets. Omission of any information linking the packets to the sources will prevent any feedback data from being routed correctly. The source will not be able to deterministically identify failure of delivery of packets, and it will not be able to identify the reason behind such failure.

While the internet has evolved to be able to handle a situation with no delivery feedback, this is generally recognised as a sub-optimal environment. Such setup can gravely affect the performance of the network. In chapters 5 and 6, we investigate the effects of the setup on the network and we propose alternatives for features that the setup can potentially break.

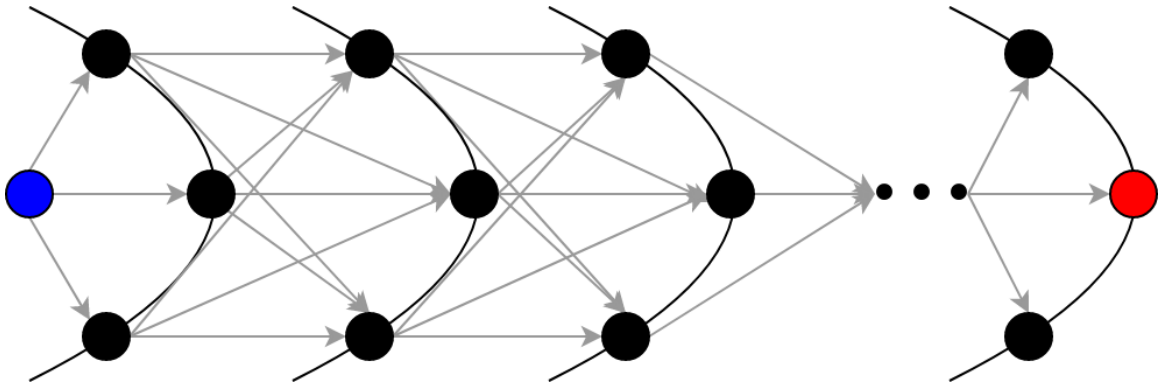

Figure 2.1: Visualization of the how the routing infrastructure contributes to the obfuscation of the identity of the source. The source is coloured in blue while the destination is coloured in red. As the point of observation progresses further towards the destination along a route between the blue and red nodes, the set of possible sources grows.

2.1 Design

In this section we propose a design for a scheme, SLR, that does not require source-identifying information to route the packets. We do this using intermediate nodes and a circular route.

As shown in figure 2.2, we construct a network path where each node forwards the packets to the node that follows it directly, where the nodes are connected in a circular path. The nodes do not have any information regarding the other nodes in the path, except for the identity of the following node. Routing of the packets is done without having

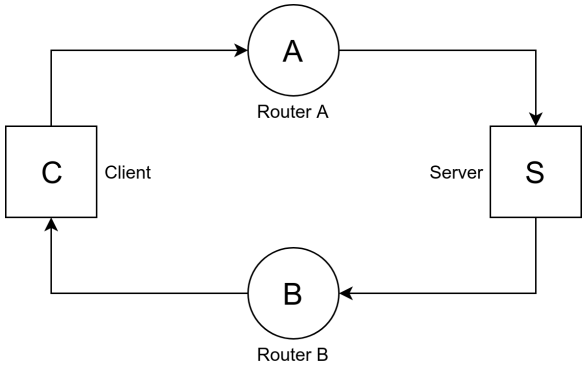

Figure 2.2: A figure showing the routes between the client and the server.

to, or being able to, identify their source.

We establish sessions that end-hosts can use to communicate over anonymously and securely. Because of its connected structure, we use ”circuit” as a synonym for SLR’s network sessions. Each circuit is linked with a specific set of keys, and a specific sequence of specific nodes.

Before starting communication, information is exchanged between nodes in a circuit establishment phase. The client, C, obtains the IPs and keys of the circuit nodes through an out-of-band communication scheme. It takes care of appending this data and encrypting it securely for each node. The resulting packet is sent to the first node, A, which takes care of forwarding it to S, then to B, then back to C.

We use Onion Encryption [10] to encrypt the IPs, re-encapsulating them under multiple layers each removed by each of the nodes throughout the circular path. C encrypts its IP using a key shared B, allowing B to route the packets back to C. It also encrypts B’s own IP using a key shared with S, instructing S where to route the response to. Same goes for A, where the client also encrypts the IP of the server S using a key shared with A, so that A can forward the packets to the server. Using this design, every node is aware of the next destination, but none of the previous hops. This allows C to communicate securely with S without revealing its IP. Each node decrypts the packet to read the data intended for it, then it omits it and forwards the rest of the data to the following node.

While the keys used for encrypting the initial circuit establishment packets are obtained out-of-band, the keys used during active communication are derived from a Diffie-Hellman key exchange and are newly created per each session.

We use numerical circuit identifiers, CIs, to identify the circuits. CIs are used to allow nodes to link received packets to the set of keys and IPs communicated earlier. This allows us to communicate the keys and IPs only once. The IPs are not communicated over active circuits.

In the circuit establishment phase, C sends multiple CIs, each for a different node. The nodes receive the CI and locally link the IPs and keys extracted from the packet with their respective CI before forwarding the packet to the next node.

C and S insert CIs into the packets for active sessions. CIs are different per node, but they remain constant over a single session.

All active traffic communicated through SLR is wrapped under multiple layers of encryption. The reason behind this manner of encryption is changing the fingerprint of the payload as it traverses across the routing infrastructure between the nodes. Other-

wise, an adversary in control of the infrastructure between both endpoints and one of the active Routers will be able to track the path of the packets between C and S. However, additional padding may be required using randomly generated data in order to further spoof the fingerprint of the packets since parts of the meta-data used to identify the connection can remain identical over multiple round-trips.

2.2 Discovery and Support Infrastructure

We consider the discovery and support infrastructure to be out of the scope of the thesis. In this section we propose multiple ideas and mechanisms for discovery. However, the proposed concepts are only initial ideas that require investigation and proper analysis.

The discovery of nodes available to act as A or B can be done out-of-band. We attempt to introduce a technique for discovery, ensuring both privacy and redundancy. We propose using Chord [13], which is an implementation of Distributed Hash Tables.

Because we are not really storing external data in a distributed infrastructure, we propose altering the design so that each node is responsible only for its own public key. C initially has a list of some of the nodes that are part of the Distributed Hash Table. This list is transmitted out-of-band. The Dynamic Hash Table has a size of 2^n, where $1 \leq n \leq \infty$. The value of n is also communicated out-of-band. C picks a numeric identifier between 0 and 2^n, and requests the node responsible for this identifier through the initial Chord node it already has knowledge of. As C receives the reply, it should be able to employ this node as an active Router over one of the routes.

Another different approach would be to use directories and follow Tor's tracks [4][14]. There are multiple defences employed by Tor against compromising such directories. These defences include cross-signing the directories' messages. Another defence is requesting a large number of nodes. This can be used to disallow the directory itself to identify which nodes are chosen by a certain client.

3 Implementation

This chapter proposes methods to implement SLR using existing protocols and infrastructure. We start by explaining implementations of basic design concepts discussed in chapter 2 such as the representation of SLR's data, and then we follow by explaining in detail the two main phases of an SLR communication: circuit establishment in section 3.1, and active communication over an established circuit in section 3.2.

We use IPv6 and its currently existing features to implement SLR. One example of such features is IPv6 Extension Headers [15]. We explain in detail how we use Extension Headers in section 3.2. While implementations using other Layer 3 protocols [16] could be possible, we do not investigate this approach.

To hide the identity of the source, we set the source IP address of the IPv6 packet to 0:0:0:0:0:0:0:0, the Unspecified Address [17]. RFC 4291 [18] states that this address indicates the absence of an address.

We aim to insert any SLR-related data directly following the IPv6 header. The way we do this differs between pre and post circuit establishment due to the limitations of the IPv6 Extension Headers. Ideally we should be able to use L3 headers that have a structure designed for communication of encrypted content with specific fields for keys, nonces, and other encryption meta-data. The Extension Headers should also allow larger content. IPv6 Extension Headers are limited to only 264 bytes including the Extension Header's own header, which is not enough for SLR's circuit establishment phase. We show the amount of data we require during circuit establishment and how we solve the problem of limited header's length in section 3.1.

Following NIST's recommendations [19], we use asymmetric RSA keys that are 2048-bits long, and symmetric AES keys that are 256-bits long. According to NIST, keys with that length are expected to be sufficiently secure through the year 2030. We use RSAES-OAEP encryption scheme for RSA [20].

The choice to pick RSA keys to asymmetrically encrypt data can be debated. Elliptic Curve Cryptography [21] can be a better alternative. While we do already use Elliptic Curve Integrated Encryption Scheme (ECIES) [22] to encrypt data exchanged over an active circuit, we could also use it during circuit establishment. We suggest exploring this route in future work and comparing the performance to the currently proposed imple-

mentation.

For AES, we encrypt using Galois/Counter Mode (GCM) [23] to maintain authentication using symmetric encryption. We use randomly generated nonces during encryption. Nonces introduce random bits, protecting against attacks based on analysing multiple messages encrypted using the same key [24].

During circuit establishment, both asymmetric and symmetric encryption are used to exchange confidential information, while during active communication, only symmetric encryption using negotiated keys is used to encrypt data.

3.1 Circuit Establishment

The first phase of communication over SLR is circuit establishment. Certain information is required to be exchanged between the nodes to allow them to decrypt and forward active traffic correctly. Figure 3.1 shows the data exchanged between the nodes.

Specifically, we exchange the following data:

- **AES Keys**: C generates 3 different AES keys for each node. It encrypts the keys using the intended receiver's RSA public key. The keys are used to encrypt the multi-layered headers generated by C and sent during circuit establishment.
- **IPs**: The IPv6 addresses of the nodes are sent by C. Each IP is part of a header that is encrypted using the intended receiver's key. We show the owner of the IP using a subscript. For example, IP_C is C's IPv6 address.
- **Nonces**: AES-GCM requires a nonce for encryption. C randomly generates a nonce for each encrypted header.
- **ECDHE Public Keys**: We exchange ECC X25519 public keys [21] used to derive a symmetric key that is used to encrypt data over an active circuit. C generates 3 different sets of keys and sends the public keys to the nodes. In return, each node generates its own pair of ECC keys and appends the public key to the packet, which is routed back to C. We use the notation $ECDHE_{XY}$ for the ECC public keys, where X is the generator of the key and Y is the intended receiver.
- **Circuit Identifiers (CIs)**: C generates 4 different numerical circuit identifiers. The circuit identifiers are used to link the packets with a set of keys and IPs. We use the notation CI_X, where X is the owner of the CI.

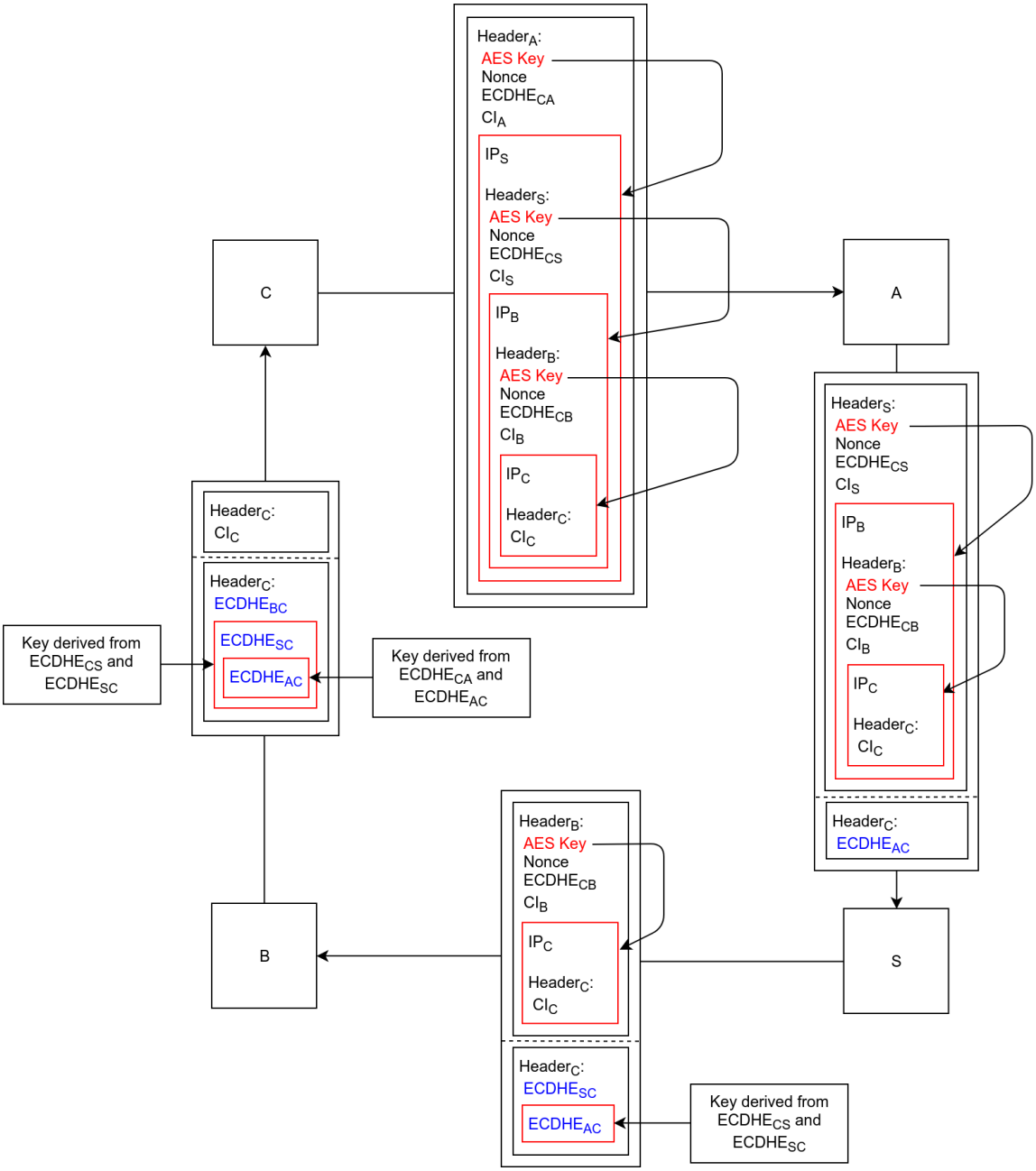

Figure 3.1: A diagram showing the incoming and outgoing data for each node during the Circuit Establishment phase. The dotted line shows the border between the first header and the second header. Red is encrypted and blue is signed, where a blue ECDHE$_{XY}$ includes both the encrypted key as well as the signature. The arrows show which keys are used to encrypt what. The AES Keys are encrypted using the receiver's RSA public key.

As the packet traverses the circuit, each node decrypts the packet, extracts the plain text information intended for its use, saves the included data, adds data concerning itself, and then forwards the packet to the next hop. The last packet sent from B back to C acts as an acknowledgement packet, verifying that the circuit establishment is successful. After receiving the packet, C can proceed to use symmetric encryption and send packets to S.

This data is exchanged in an independent phase prior to active communication mainly in order to lower bandwidth overhead. We want to avoid having to send the set of keys and IPs along with every packet, so we instead instruct the nodes to save the keys and IPs and link them with a specific numerical circuit identifier, CI. The CI is later used to allow the nodes to identify the circuit and link the packets with previously communicated keys and IPs.

Every node has knowledge of two CIs, its own CI, and the CI belonging to the node following it. We explain in section 3.2 the reasoning behind this, as well as how both CIs are used.

We use multiple sets of keys and encryption schemes during circuit establishment. The asymmetric RSA keys are communicated out-of-band. We assume C has knowledge of them. We use RSA to encrypt the symmetric AES keys coloured red in figure 3.1. The encrypted symmetric keys are used encrypt the headers generated by C for each node, indicated by red borders in the figure.

We also generate Elliptic Curve public keys that are used to derive yet another set of symmetric keys used to encrypt data transferred over a circuit, indicated by the notation ECDHE_{XY}, where X is the creator of the key, and Y is the intended receiver.

The ECDHE keys are used to derive symmetric keys used for encrypting data over an active circuit. We use HKDF [25] with SHA 512 hash algorithm [26] to generate a 32 bytes long key. The derived symmetric keys are also used to encrypt the contents of the second header as shown in the figure.

Note that the Diffie-Hellman key exchange performed during circuit establishment is important. It is used to generate a new set of symmetric keys in addition to the ones generated by C. One may argue that the AES keys generated by C can be used for encrypting information over an active circuit. We discuss the setbacks of this approach in section 7.1.1. While we cannot maintain forward secrecy during circuit establishment, we exchange a new set of randomly generated keys to maintain forward secrecy for data exchanged over an established circuit.

While encryption provides confidentiality, it does not provide authenticity. Adver-

saries can intercept the connection between C and A and inject a malicious ECDHE public key into the flow. The adversary can then craft a packet that it can send to B, allowing for a full Man in The Middle (MiTM) attack. To counter this, we sign the exchanged ECDHE public keys to ensure authenticity.

We require all ECDHE public keys to be signed by their respective sources, except for C. While the public keys for A, B, and S should prove the identity of its respective owner, a key generated by C would be randomly generated and thus signing messages by C would be redundant since linking the keys with C would not be possible, and if we were to construct a mechanism to verify that a certain key is owned by C then we would be introducing a security flaw since the aim of SLR in the first place is to hide C's identity.

During Circuit Establishment we use custom headers instead of IPv6 Extension Headers because of the limitations of this feature. IPv6 Extension Headers have a maximum length of 2^8 - 1 bytes. The length of data we transfer during Circuit Establishment is larger than that limit. We communicate 3 different 256-bits AES keys, 3 32-bits nonces, 3 256-bits ECDHE public keys, 4 32-bits CIs, and 3 128-bits IP addresses. We encrypt and sign the data under multiple layers, each with a 256-bits AES key, resulting in an additional 3 X 16-bits signatures. All adding up to 2192 bits, or 274 bytes.

We choose to use custom headers to avoid overhead and expensive processing caused by having to split the data into multiple IPv6 Extension Headers. We simply set the IPv6 Packet Next Header value to 99, and we append our data. The data always has constant length, allowing the parsing of specific fields without having to use delimiters that allow the receiver to dynamically determine the length of the field.

However, we split the data into two top-level custom headers to allow for an easier implementation. The headers do not contain any additional information causing any bandwidth overhead. The first header is data generated by C, and the second header is data generated by other nodes for C. We use a dotted line in figure 3.1 to split between the two headers.

3.1.1 First Header

The first header contains data generated by C for other nodes. It is shown in 3.1 as the header above the dotted line.

All the data is encrypted using AES keys that are randomly generated by C. The AES keys are different, each is encrypted by the intended receiver's asymmetric RSA key. These asymmetric keys are shown using the notation Asymmetric_X, where X is the owner of the key.

The following list shows how the first header is created and processed in consequential steps.

1. C initiates the circuit by crafting a new IPv6 packet and creates a string to store the header
   a) Prepare Header_C
      i. Append CI_C to the header
   b) Prepare Header_B
      i. Generate 256-bits AES Key, encrypt it using Asymmetric_B, and append it to the header
      ii. Generate Nonce, ECDHA_{CB}, and CI_B and append them to the header
      iii. Encrypt IP_C along with Header_C using the generated AES Key. Append the result to the header
   c) Prepare Header_S
      i. Generate 256-bits AES Key, encrypt it using Asymmetric_S, and append it to the header
      ii. Generate Nonce, ECDHA_{CS}, and CI_S and append them to the header
      iii. Encrypt IP_B along with Header_B using the generated AES Key. Append the result to the header
   d) Prepare Header_A
      i. Generate 256-bits AES Key, encrypt it using Asymmetric_A, and append it to the header
      ii. Generate Nonce, ECDHA_{CA}, and CI_A and append them to the header
      iii. Encrypt IP_S along with Header_S using the generated AES Key. Append the result to the header
2. A receives the packet, modifies it, and sends it to S
   a) Parse the header. Use own private key to decrypt the AES key
   b) Use the AES key and the Nonce to decrypt IP_S and Header_S
   c) Save ECDHA_{CA} and IP_S and link them with CI_A
   d) Replace the contents of the header with Header_S
3. S receives the packet, modifies it, and sends it to B
   a) Parse the header. Use own private key to decrypt the AES key
   b) Use the AES key and the Nonce to decrypt IP_B and Header_B

c) Save ECDHA_{CS}, IP_B, and CI_B and link them with CI_S

d) Replace the contents of the header with Header_B

4. B receives the packet, modifies it, and sends it to C
   a) Parse the header. Use own private key to decrypt the AES key
   b) Use the AES key and the Nonce to decrypt IP_C and Header_C
   c) Save ECDHA_{CB} and IP_C and link them with CI_B
   d) Replace the contents of the header with Header_C
5. C receives the packet
   a) Parse the header. Use CI_C to link the received data with the earlier generated one

3.1.2 Second Header

The second header contains data generated by other nodes for C. It is shown in 3.1 as the header below the dotted line.

It contains the ECDHE public keys that are used to derive symmetric keys used for encrypting data transferred over an active circuit. The following list of steps shows how the second header is created and parsed.

1. A receives the circuit establishment packet, modifies it, and sends it to S
   a) Generate ECDHA_{AC} and sign it with own private key. Append it to a new string representing a new header, then append it to the packet
2. S receives the packet, modifies it, and sends it to B
   a) Generate ECDHA_{SC} and sign it with own private key
   b) Encrypt second Header using ECDHA_{CS} and ECDHA_{SC}
   c) Replace the second Header with the signed ECDHA_{SC} followed by the encrypted Header
3. B receives the packet, modifies it, and sends it to C
   a) Generate ECDHA_{BC} and sign it with own private key
   b) Encrypt second Header using ECDHA_{CB} and ECDHA_{BC}
   c) Replace the second Header with the signed ECDHA_{BC} followed by the encrypted Header
4. C receives the packet

a) Parse the header. Recursively read the ECDHE keys, verify their signatures using the respective node's public key, link them with CI_C, and use them to decrypt the encrypted Headers

The result is summarised in the following list showing fields known and saved by each node:

- **A**:
  - Session key derived from $ECDHA_{CA}$ and $ECDHA_{AC}$
  - CI_A
  - CI_S
  - IP_S
- **S**:
  - Session key derived from $ECDHA_{CS}$ and $ECDHA_{SC}$
  - CI_S
  - CI_B
  - IP_B
- **B**:
  - Session key derived from $ECDHA_{CB}$ and $ECDHA_{BC}$
  - CI_B
  - CI_C
  - IP_C

3.2 Communication Over Established Circuit

Following successful circuit establishment, the network is ready to route active traffic. To start, we reiterate the information possessed by every node. Each node is aware of the IP of the following node, communicated during circuit establishment. Each node has a session key derived from its own ECC key as well as an ECC key sent by C during circuit establishment. Each node is also associated with a specific CI that it uses to identify the circuit, and is aware of the CI of the following node.

A certain structure is used to append meta-data to the IPv6 packets to allow A and B to route them correctly. Figure 3.2 shows the high-level changes we apply to a packet as it is routed through SLR. The Source IP is set to 0:0:0:0:0:0:0:0, and the Next Header value is set to 60 to reflect the existence of an IPv6 Destination Options Extension Header

IPv6 Packet		IPv6 Packet
Source IP	→	0:0:0:0:0:0:0:0
Destination IP		Destination IP
Next Header	→	60
Other IPV6 headers...		Other IPV6 headers...
		Extension Header
Payload	→	**Encrypted Payload**

Figure 3.2: Diagram showing the modification of the packets. Note how we modify the original IPv6 packet's source IP to 0:0:0:0:0:0:0:0, the Next Header to 60, and how we insert the External Header between the IPv6 header and the L4 packet.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	*Next Header:* 99								*Hdr Ext Len:* Header length according to the included data								*Options and Padding:* 0x1e 0x?? | 0x?? 0x?? 0x?? 0x??...															
4	32	*0x?? 0x?? 0x?? 0x??...*																																
8	64																																
12	96																																

Figure 3.3: Structure of the IPv6 Destination Extension Header. Note how we use the vertical bar character to split between the values identifying the option's type and length, and the data representing the content of the option.

[27][15].

Figure 3.3 shows the structure we use for the Extension Header. We set the Extension Header's Next Header field to 99, the value representing a private encryption scheme as assigned by the IANA [28]. This value was picked to allow us to append the encrypted packet, which is inserted following the Extension Header as shown earlier in figure 3.2.

We fill the Extension Header with a single IP Option. The IP Option consists of two bytes followed by the content of the option. The first byte is a value identifying the type of the option, which, in our use case, is fixed to 30 (0x1E in hex). 0x1E is the value indicating experimental use [29]. The second byte consists of the length of the option in bytes. We insert our data as the IP Option's content, and we set the IP Option's length according to the size of the data.

Figure 3.4 shows how the meta-data is added and modified along the circular path in a more detailed manner. C creates two nested headers intended for A and S. A reads Header_A and omits it. It then decrypts Header_S as well as the Payload and forwards the data to S.

S reads its own header and decrypts the payload before processing it. As S crafts a response, it appends a new Header_B before forwarding the packet to B, which encrypts the packet before adding Header_C and forwarding the packet to C. C decrypts both layers and processes the response.

The following list explains the values shown in the graph and how they are used.

- **Header_X**: We use multiple headers to define information for difference nodes. X is the intended receiver. Headers are sometimes nested and encrypted. We show that in the graph using the red border and the arrow showing the key used to encrypt the header.
- **Circuit Identifiers (CIs)**: Each packet contains a Circuit Identifier, CI_X, where X is the intended receiver. The circuit identifiers are used to link the packets with a set of keys and IPs. Each node along the route replaces CI_X with CI_Y where Y is the following node. The CIs are communicated during Circuit Establishment.
- **Nonce_{header}**: We encrypt information in headers such as the nested header and the Next Header value using AES-GCM. Along the key, a nonce is required to securely encrypt the data. We insert this nonce into the header to allow the intended receiver to decrypt the data.
- **$\text{Nonce}_{payload}$**: We also encrypt the payload using a different nonce. We attach this nonce to the header as well.
- **Payload Signature**: AES-GCM encryption generates a 16-bits signature to maintain authenticity. The IPv6 Extension Header's length field is in multiples of 8 bytes. This ends up forcing us to pad the header. We choose to insert the payload's signature into the header instead of appending it to the body of the payload to avoid additional bandwidth overhead.
- **Next Header**: Original Next Header value of the IPv6 packet. We use the notation Next Header_{XY} to indicate the source, X, and the intended receiver, Y.
- **Payload**: We encrypt the payload under multiple layers of encryption. On the forward route from C to S, C encrypts the payload under two layers of encryption. A removes the first layer, and then S removes the second one before processing it. On the backwards route, S adds only one layer of encryption. B receives the packet and encrypts the payload again before forwarding the packet to C. C is then able to decrypt both layers before processing the response.

For any subscripted item A, A_{XY} and A_{YX} are interchangeable. Both indicate the existence of an operation executed using the symmetric encryption key derived from ECDHA_{XY}

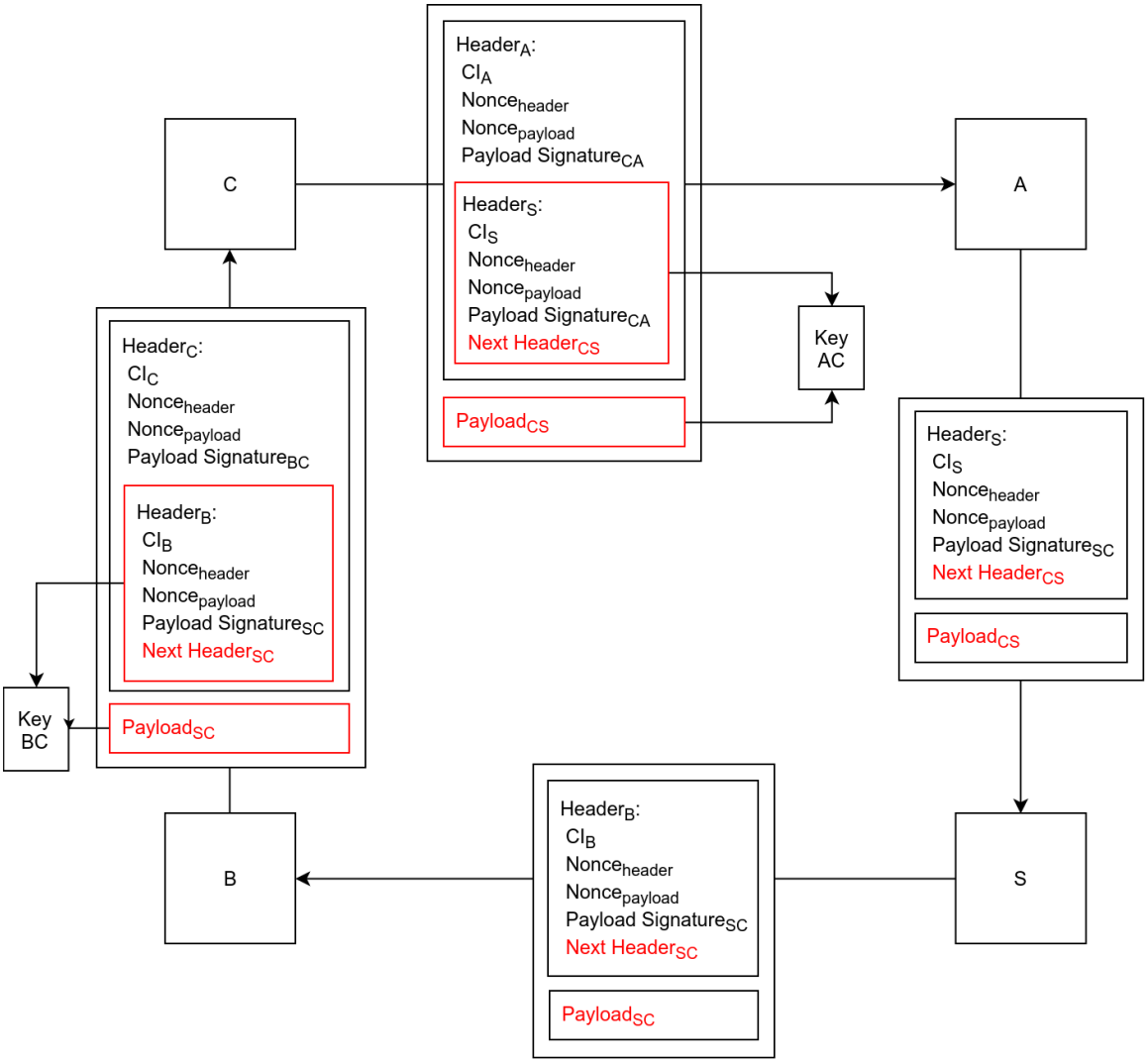

Figure 3.4: A diagram showing the incoming and outgoing data for each node during the active communication phase, following successful circuit establishment. Red is encrypted. Boxes with the content Key XY indicate data that is encrypted multiple times with the top layer encryption using Key XY.

and ECDHA$_{YX}$, whether its signing, or encryption. However, the order of X and Y still indicates the performer of the cryptographic operation on the subscripted item, X, and its intended receiver, Y.

We encrypt the payload to avoid leaking identifying information, and we wrap it in multiple layers of encryption to avoid fingerprinting the packet along a network route.

While no proper mixing is performed, we aim to change the packet's fingerprint in an unidentifiable manner at each hop using encryption, making it harder to link the ingress and egress of an anonymising node. We recommend researching additional mixing techniques. However, we currently consider this out of scope of the thesis.

We communicate the Circuit Identifiers over each transmission to allow nodes to link packets with specific circuits. For each circuit, each node is aware of two CIs. The CI gets changed by each node to counter fingerprinting by an adversary.

As C sends a packet, it inserts a numerical CI$_A$ which A uses to identify the circuit. A then goes on to replace CI$_A$ with a different CI$_S$ before forwarding the packet to S, which S then uses to identify the circuit. Each node is aware of both its own CI, and the CI belonging to the node following it.

We append randomly generated nonces to the Header. We use two nonces, Nonce$_{header}$ and Nonce$_{payload}$. Nonce$_{header}$ is used during encrypting headers, and Nonce$_{payload}$ is used during encrypting the payload.

We also split the generated payload signature from the payload encrypted using AES-GCM and append it to the Header since we already have available unused space in the Extension Header, instead of having to limit the maximum payload size even further.

As C generates a packet, we insert 2 32-bits CIs, 4 32-bits nonces, 2 128-bits payload signatures, and a 136-bits Next Header value after considering the signature, as well as an additional 128 bits generated from encrypting Header$_S$ itself. The total overhead is 712 bits, or 89 bytes.

Excluding the payload signatures the data would amount to 73 bytes. Due to the Extension Header's length value being in multiples of 8 bytes [15], the minimum size we can set the Extension Header's payload length to is 128 bytes, leading to a waste of 55 bytes, hence the reasoning for using that space for the payload signature values.

The size taken up by the Extension Header, which is 128 + 8 = 136 bytes, forces us to decrease the default MTU from 1500 bytes [30] to 1324 bytes after excluding IPv6's 40 bytes headers. In chapter 6 we analyse the effect of decreasing the MTU on the perfor-

mance of the network, independent of SLR.

We believe that the implementation can be improved by using a custom-defined header for the meta-data during communication over an established circuit. In the future, a new standardised header could also be defined to be used by SLR.

The following list explains in further detail how C crafts a packet that it wants to send to S, and the steps taken by each node to parse and route the packet to the next node, as well as the steps S performs before being able to pass the packet to the kernel to process it.

1. C crafts a packet that it wants to send to S. It modifies the packet by inserting a new Extension Header containing one IP Option, and certain meta-data, before forwarding it to A.
   a) Prepare Header_S
      i. Append CI_S to the Header
      ii. Generate 2 random numbers Nonce_{header} and $\text{Nonce}_{payload}$ and append them to the Header
      iii. Encrypt the payload using Key_{SC} and $\text{Nonce}_{payload}$ and split the signature from the payload. Append the signature to the Header, and replace existing payload with the encrypted payload
      iv. Extract the original Next Header value and encrypt it using Key_{SC} and Nonce_{header}, and append it to the Header
      v. Change original Next Header field of the IPv6 header to 60 (Destination Options for IPv6), and set the Next Header value of the Extension Header to 99 (Any private encryption scheme)
   b) Prepare Header_A
      i. Append CI_A to the Header
      ii. Generate another 2 random numbers Nonce_{header} and $\text{Nonce}_{payload}$ and append them to the Header
      iii. Encrypt payload again using Key_{AC} and $\text{Nonce}_{payload}$ and split the signature from the payload. Append the signature to the Header, and replace existing payload with encrypted payload
      iv. Encrypt Header_S using Key_{AC} and Nonce_{header} and append it to the Header
      v. Replace the contents of the IP Option with Header_A
   c) Set the source IP of the IPv6 packet to 0:0:0:0:0:0:0:0
   d) Send the packet to A

2. A receives the packet, modifies it, and sends it to S
    a) Parse the Header. Use CI_A to identify the circuit
    b) Append the Payload Signature to the payload. Use Key_{AC} and $Nonce_{payload}$ to decrypt the payload. Replace the payload with the decrypted payload
    c) Use Key_{AC} and $Nonce_{header}$ to decrypt $Header_S$ and replace the contents of the Header with the decrypted $Header_S$
    d) Send the packet to S
3. S receives the packet and parses it
    a) Parse the Header. Use CI_S to identify the circuit
    b) Append the Payload Signature to the payload. Use Key_{SC} and $Nonce_{payload}$ to decrypt the payload. Replace the payload with the decrypted payload
    c) Use Key_{SC} and $Nonce_{header}$ to decrypt Next Header and replace the Next Header value of the IPv6 packet with the decrypted value
    d) Remove the Extension Header
    e) Send the packet to the kernel to be processed

As S receives a packet from C, it processes it before crafting a reply. The following list shows the steps S takes to send a reply back to C, and how C should parse the packet before being able to pass it to the kernel to process it.

1. S crafts the reply packet. It modifies the packet by inserting a new Extension Header containing one IP Option, and certain meta-data, before forwarding it to B.
    a) Insert a new Extension Header into the packet, containing one IP Option that contains $Header_B$
    b) Append CI_B to $Header_B$
    c) Generate 2 random numbers $Nonce_{header}$ and $Nonce_{payload}$ and append them to $Header_B$
    d) Encrypt payload using Key_{SC} and $Nonce_{payload}$ and split the signature from the payload. Append the signature to $Header_S$, and replace existing payload with encrypted payload
    e) Extract the original Next Header value and encrypt it using Key_{SC} and $Nonce_{header}$, and append it to $Header_B$
    f) Change original Next Header of the IPv6 header to 60 (Destination Options for IPv6), and set the Next Header value of the Extension Header to 99 (Any private encryption scheme)
    g) Set the source IP of the IPv6 packet to 0:0:0:0:0:0:0:0

h) Send the packet to B

2. B receives the packet, modifies it, and sends it to C
   a) Create a new string, Header_C
   b) Append CI_C to Header_C
   c) Generate 2 random numbers Nonce_{header} and $\text{Nonce}_{payload}$ and append them to Header_C
   d) Encrypt payload again using Key_{BC} and $\text{Nonce}_{payload}$ and split the signature from the payload. Append the signature to the string, and replace existing payload with encrypted payload
   e) Encrypt Header_B using Key_{BC} and Nonce_{header} and append it to Header_C
   f) Replace the content of the IP Option with Header_C
   g) Send the packet to C
3. C receives the packet
   a) Decrypt First Layer
      i. Append the Payload Signature to the payload. Use Key_{BC} and $\text{Nonce}_{payload}$ to decrypt the payload. Replace the payload with the decrypted payload
      ii. Use Key_{BC} and Nonce_{header} to decrypt the encrypted Header_S
      iii. Replace the content of the IP Option with Header_S
   b) Decrypt Second Layer
      i. Append the Payload Signature to the payload. Use Key_{SC} and $\text{Nonce}_{payload}$ to decrypt the payload. Replace the payload the payload with the decrypted payload
      ii. Use Key_{SC} and Nonce_{header} to decrypt Next Header and replace the Next Header value of the IPv6 packet with the decrypted value
      iii. Remove the Extension Header
      iv. Send the packet to the kernel to be processed

4 Prototype

Following the design and conceptual implementation of SLR, we created a prototype. The prototype aided us with running tests and measurements, and being able to observe the performance of the network running over SLR. The prototype's code exists at [31].

The scripts created for circuit establishment and communication over an active circuit are separate. We hard-code multiple values that should be obtained externally, such as the IP of the entry node. For the active communication over established circuit code, we also hard-code the values that should be communicated over circuit establishment.

The prototype is coded for Python version 2. We use the Linux program iptables [32] to intercept the packets and push them into NetfilterQueue [33], a Linux Kernel queue that can be polled from by a user-space program. We configure iptables to intercept the packet using specific rules.

A user-space program can process the intercepted packet from NetfilterQueue. It can read and modify the packet before either accepting or dropping it. We use dpkt [34] Python library to parse the packets. To simplify the implementation, we opt to drop all intercepted packets and instead resend them after their modification.

The exact iptables rules used and versions of the packages are environment specific. We list the exact versions we used to run the prototype in chapter 6, where we also list and analyse the measurements we have obtained.

The iptables rules we use sometimes catch messages we do not want to pass over SLR. Specifically, we do not route ICMP neighbour solicitation and neighbour advertisement messages over SLR to allow routing to function properly. This is done by checking the Next Header value of the IPv6 packets, and if the value is 58, indicating an ICMPv6 packet [28], we check the ICMPv6 packet's type. If the value is either 135, corresponding to a neighbour solicitation message, or 136, corresponding to a neighbour advertisement message [35], we signal NetfilterQueue to accept the packet without any modification.

The following sections explain in further details the approaches we have taken to code the prototype for circuit establishment and communication over established circuit, each as an independent process.

4.1 Circuit Establishment

We created a prototype to test circuit establishment. The main purpose of the Python program was to run the circuit establishment process 1000 times and measure the average time taken. The measurements are shown in section 6.1.

All the RSA public keys are hard-coded. The AES keys used for encrypting the headers are also hard-coded for ease of testing. In a real-life scenario, the RSA keys should be obtained out-of-band, and the AES keys should be randomly generated.

C starts the circuit establishment process by generating the random values needed to establish the circuit. It encrypts the headers recursively, and it sends the packet to A. It then activate the NetfilterQueue listener waits for response packets.

We use dpkt to craft a packet containing the information C needs to send as shown earlier in figure 3.1. All nonces and CIs are randomly generated 32-bits integers. We use the struct library [36] to normalise the format of the generated values. We also use the socket library [37] to normalise the format of the IPv6 addresses. We send the packet using the socket library.

On the other nodes, the Python scripts should be running with NetfilterQueue passively waiting for packets. As packets are intercepted, they should be passed to the corresponding Python script which parses the packets accordingly.

As A receives the packet it extracts the AES key, nonces, CI_A, and $ECDHE_{CA}$. The offsets used to locate the fields are hard-coded. There is no variance in the size of the data generated and thus adding logic to locate the packets dynamically would be inefficient. It uses the AES key to decrypt the nested header, which contains IP_S and $Header_A$.

It then generates its own $ECDHE_{AC}$ key pair. $ECDHE_{AC}$ can be used along with $ECDHE_{CA}$ to derive a symmetric key used for communication over an established circuit between C and A. However, we skip implementing this functionality as part of the circuit establishment prototype for the sake of simplicity. The code for communication over established circuit uses a hard-coded value for the symmetric key as a placeholder.

Node A signs $ECDHE_{AC}$ public key using its private RSA key $Asymmetric_A$ and appends it to the second header. Finally, it forwards a packet containing $Header_A$, and Header 2 containing the signed $ECDHE_{AC}$ public key, to S.

This continues for S. However, to counter fingerprinting, S uses the symmetric key derived from $ECDHE_{CS}$ and $ECDHE_{SC}$ to encrypt $ECDHE_{AC}$. The same is done by B, which encrypts the entire blob containing $ECDHE_{SC}$ and $ECDHE_{AC}$ using the symmetric key de-

rived form ECDHE_{CB} and ECDHE_{BC}.

As the packet reaches C, it reads ECDHA_{BC} and uses it along with its own ECDHE_{CB} to derive a symmetric key. The symmetric key can then be used to decrypt the nested Header 2. It does this repeatedly, reading ECDHE_{SC} and using it along with ECDHE_{CS} to derive a symmetric key, until it is able to read ECDHE_{AC}.

During every step C verifies the signature of the key using the corresponding node's public key. If a failure is detected, circuit establishment stops and must be restarted.

4.2 Communication Over Established Circuit

While the prototype for circuit establishment aims to measure the delay it takes to communicate and parse the elements needed to establish a circuit, the prototype for communication over established circuit has allowed us to run network tests over SLR and measure the performance.

During this phase we solely rely on symmetric encryption using AES running in Galois/Counter mode. We hard-code the symmetric encryption keys. We also hard-code the CIs as well as the IPs of the nodes. All these values should remain constant over an active circuit.

We use the hard-coded AES keys to encrypt and decrypt both the nested headers and the payload. The manner in which this is done follows figure 3.4. From C to A, each node removes a layer of encryption, while from S to C, each node adds one.

As C intercepts an outgoing packet, it sets its source IP to 0:0:0:0:0:0:0:0, sets the Next Header value to 60, and goes on to generate and append values as shown earlier in the figure.

Like circuit establishment, the nonces are 32-bits integers that are generated randomly. The different nonces are used to encrypt different parts of the packet. Nonce_{header} is always used to encrypt header information, while $\text{Nonce}_{payload}$ is always used to encrypt the payload.

Same as circuit establishment, the fields always have static offsets, so we don't need any logic to allow locating fields dynamically. The payload's size itself can always be known by subtracting the hard-coded length of the meta-data from the IP packet's payload length value. We always update the payload length value after modifying the packet.

All nodes have scripts running and passively waiting for packets intercepted by iptables. C and S each have two scripts always running, client.py, server.py, client2.py, and server2.py.

client.py parses outgoing packets that are eventually received by server.py. server.py takes care of parsing the packet before passing it to the loopback network interface. S can also send an outgoing packet to C, which is parsed by server2.py before it's sent. As C receives the packet, client2.py parses it before passing it to the loopback.

We route incoming packets to the loopback network interface to allow them to be processed by an application. The loopback device allows us to route the packets to the kernel, which takes care of parsing it before forwarding it to the correct application.

5 Effects on Higher Layer Protocols

Since SLR relies on modification of an L3 protocol, IPv6, verifying that the changes do not interfere with the higher-level protocols is necessary. While network layers should be isolated, as we will show in the next sections that is not always the case in practice.

We focused on 3 protocols: ICMP, UDP, and TCP. We analysed the protocols and attempted to determine whether the modification of the packets interferes with the protocol's transmissions in a manner that induces failures.

We analysed the RFCs created for ICMP and UDP to theoretically evaluate compatibility with our protocol, while for TCP we relied on experiments to detect failures. We suggest alternatives for some features of the protocols where we recognise that the modification of the packets could possibly break them.

5.1 ICMP

RFC 4443 [38] specifies multiple ICMP messages. The RFC lists two types of such messages: error messages, and information messages. The information messages are 128, Echo Request, and 129, Echo Reply. Using the proposed diagram both messages are expected to be transferred successfully and we have been able to prototype this functionality working as expected.

For errors messages, RFC 4443 section 2.4 notes that ICMP error messages must not be generated as a result of receiving a packet whose source address is the IPv6 Unspecified Address [17]. As earlier noted in Section 1 2, we are using that IP to spoof the source IP of generated packets. In the following sections we propose methods to handle the absence of ICMP error messages.

5.1.1 ICMP Error Message Type 1: Destination Unreachable

According to RFC 4443, the Destination Unreachable message should be generated by a router or by the IPv6 layer in the originating node in the situation where a packet cannot be delivered to the destination address for reasons other than congestion.

The Destination Unreachable message is not required to be generated upon failure to deliver packets to a certain destination, as shown by the use of the word ”SHOULD” in the RFC and the definition of ”SHOULD” according to RFC 2119 [39]. In fact, RFC 4443 notes that for security reasons it is recommended that implementations should allow sending Destination Unreachable messages to be disabled. Higher level protocols are expected to have other methods to identify unreachable destinations, such as timeouts.

5.1.2 ICMP Error Message Type 2: Packet Too Big

RFC 4443 states that the Packet Too Big message must be generated by a router if it is not able to forward a packet because it is larger than the MTU of the outgoing link. This packet is used as part of Path MTU Discovery [40].

As per RFC 8200 [27], we are breaking IPv6 functionality by not handling this error message. According to RFC 8200, handling the messages is a requirement for IPv6 implementations to qualify as such.

However, considering that the main use of Packet Too Big messages is Path MTU Discovery, there exists multiple approaches to handle the lack of the error message during PMTUD, especially in a situation where all ICMP messages are blocked, also known as ICMP black holes.

RFC 4821 [41] discusses a solution to this problem using Packetization Layer Path MTU Discovery. The RFC proposes using higher layer protocols to probe Path MTU. In our implementation we can rely on either ICMP echo messages or on TCP to probe the Path MTU. A detailed explanation of an implementation is out of scope.

5.1.3 ICMP Error Message Type 3: Time Exceeded

RFC 4443 states that Time Exceeded message must be generated by a router if it receives a packet with Hop Limit 0, or if after decreasing the Hop Limit of a packet it becomes 0. The message is used to report too low initial hop limits, or routing loops.

To the best of our knowledge there is no proposed approach to handle a network where Time Exceeded messages are blocked. In this situation, higher level protocols must be relied on to detect a problem where the hop limit is too low. Further work is needed to determine how that can be done.

5.1.4 ICMP Error Message Type 4: Parameter Problem Message

RFC 4443 states that a node processing a packet should generate a Parameter Problem message to the packet's source if it encounters a problem with the IPv6 packet header or extension headers that prevents it from successfully processing the packet.

In the RFC the word "SHOULD" is used to describe the extent to which sending this message is important. From this we can establish that supporting this specific message is not a requirement, and should not lead to a broken connection.

5.2 UDP

User Datagram Protocol (UDP) [42] is a very simple network protocol created to deliver packets with no reliability guarantees and no duplication protection.

The RFC notes how the Checksum field in a UDP packet's header is calculated, and notes how a pseudo-header containing source and destination address defined in the lower level protocol, the IP protocol, is used to calculate that checksum.

In the prototype the approach we take to setting the IP packet's source IP to 0:0:0:0:0:0:0:0 is by intercepting the packet before it is emitted. We modify the packet's source without changing its checksum. Because of this implementation artefact, scripts running on the packet generator's sides should recalculate the checksum accordingly. We showed how this is done earlier in chapter 4.

5.3 TCP

As stated earlier, we have not been able to analyse the protocol's RFCs. However, we ran tests and measurements to determine whether the protocol can run in a stable manner over SLR.

We faced the same issue with the checksum due to the existing implementation. The scripts running on the packet generator's side should recalculate the checksum after changing the source IP address.

Otherwise, we were not able to observe any breakages. The protocol was able to run successfully over SLR. We documented the results of these measurements in section 6.2.3.

5.4 Summary

From our observations and analysis we can confidently claim that with the exception of ICMP's Time Exceeded message, ICMP, UDP, and TCP can run optimally over SLR.

While some modifications are required, the modifications do not impair the performance of the protocols.

We also note that other protocols will need meticulous analysis to establish their compatibility with SLR. We have not observed a general pattern of dependencies of Layer 4 protocols on IPv6. We conclude that each protocol requires case-by-case analysis.

6 Measurements

As explained earlier in chapter 4, we used Python and dpkt to be able to measure the effect of our prototype on the performance of higher level protocols. The chapter also analyses the code we used to obtain the measurements.

To obtain the measurements, we used a set of programs and libraries running on Ubuntu operating system with specific hardware.

We used 4 PCs using Intel Pentium Processors E5200 [43] and connected to an Allied Telesyn GS924i network hub [44] to run the tests. The network links had speeds of 1 Gbps. The PCs were running Ubuntu version 19.04 Server Edition [45]. They were running Linux Kernel version 5.0.

The versions of programs and libraries we have used are as follows:

- **Python**: 2.7.16
- **cryptography**: 2.3
- **dpkt**: 1.9.2
- **NetfilterQueue**: 0.8.1
- **pycrypto**: 2.6.1
- **iptables**: 1.6.1
- **iperf3**: 3.6

The netfilterqueue size was adjusted to 8192 packets. As shown in figure 6.2, the average RTT was 0.144 milliseconds.

To run the prototype, we have to configure iptables before running the scripts. We configure iptables using the following commands.

```
ip6tables -A OUTPUT -d 2100::102/128 -j NFQUEUE --queue-num 1
ip6tables -A INPUT -s 0:0:0:0:0:0:0:0/128 -d 2100::101/128 -j NFQUEUE
    --queue-num 2
```

Code 6.1: ip6tables rules for C

```
ip6tables -A INPUT -s 0:0:0:0:0:0:0:0/128 -d 2100::103/128 -j NFQUEUE
    --queue-num 1
```

Code 6.2: ip6tables rules for A

```
ip6tables -A INPUT -s 0:0:0:0:0:0:0:0/128 -d 2100::104/128 -j NFQUEUE
    --queue-num 1
```

Code 6.3: ip6tables rules for B

```
ip6tables -A INPUT -s 0:0:0:0:0:0:0:0/128 -d 2100::102/128 -j NFQUEUE
    --queue-num 1
ip6tables -A OUTPUT ! -s 0:0:0:0:0:0:0:0/128 -d 2100::104/128 -j NFQUEUE
    --queue-num 2
```

Code 6.4: ip6tables rules for S

Note the IPv6 IPs used. The following list shows the IPs we have set for our nodes:

- **C**: 2100::101
- **A**: 2100::103
- **B**: 2100::104
- **S**: 2100::102

After configuring iptables, the Python scripts should be started. The scripts do not require any arguments to be executed.

In general, the order in which they are started is not critical. The order is irrelevant because in general the scripts react passively. They wait for intercepted packets before parsing them.

However, because in circuit establishment client.py starts sending packets once the script is executed, running it should be deferred to the end.

The following sections show and analyse the measurements for both phases of SLR, Circuit Establishment, and Communication Over Established Circuit.

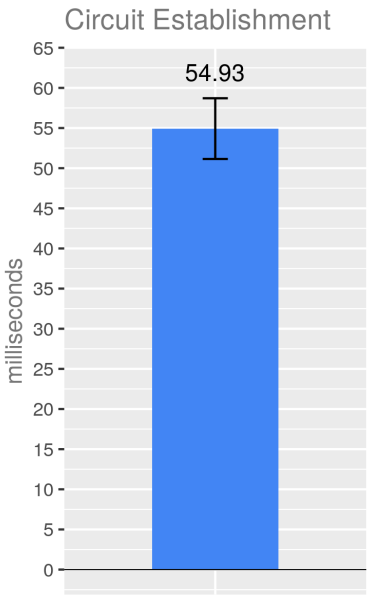

Figure 6.1: A diagram showing the average time SLR takes to complete the Circuit Establishment process in a testing environment. Vertical axis is latency in milliseconds. The error-bar shows a standard deviation of 3.78 milliseconds.

6.1 Circuit Establishment

We measured the time SLR takes to establish the circuit. We ran the prototype for circuit establishment. The code executes the circuit establishment process 1000 times and outputs the time taken per each round.

We wanted to determine whether the circuit establishment process can be established in a reasonable amount of time, below 1000 ms. We set the bar so high since this process can be done in the background, and is done only once per circuit.

Figure 6.1 shows the average value measured being 54.93 milliseconds, with a standard deviation of 3.78 milliseconds.

The values observed are very acceptable. Even though our prototype can be optimised further, the observed values are so low that pre-emptively establishing the circuits in the background would be wasteful, since they can be established on the run within tens of milliseconds.

6.2 Communication Over Established Circuit

We measured the effect of SLR on ICMP, UDP, and TCP. We evaluated the protocols on 3 different setups: direct, benchmark, and SLR. Direct measurements measured the performance of the protocol without any modification to the flow. This acts as a base to compare with.

We also ran tests on a setup where packets on C's side are redirected to netfilterqueue before being sent, and then polled by a Python script that directs netfilterqueue to accept the packet without any modification or parsing. This was done to be able to measure the effects of parsing the packet independent of the performance hit caused by the kernel redirecting the packet to Python. We call this setup "Benchmark".

The following snippet shows the code we used to run the benchmark setup.

```
import dpkt

from netfilterqueue import NetfilterQueue

def modify(packet):
    packet.accept()

nfqueue = NetfilterQueue()
nfqueue.bind(1, modify)
nfqueue.run()
```

Code 6.5: The code used on the Client to measure the impact of passing packets to Python through netfilterqueue on the performance of the network

The Benchmark setup uses the same iptables rules. Because the received packets have real IPs, they are not intercepted by iptables and thus the setup works.

For tests that measure the bandwidth, we had to lower the maximum datagram size. SLR appends meta-data required to route the packets correctly to the packet, taking up space out of the IPv6 packet. We explained the impact of this earlier in section 3.2.

We added tests that compare different values of the maximum L4 packet size to be able to measure the effect of SLR on the performance of TCP independent of the lowered MSS.

The following sections show and analyse the measurements we obtained for each of the protocols we have tested.

6.2.1 ICMP

To evaluate ICMP over SLR, we ran tests comparing the latency over our different setups and compared them.

We used the following command to run the tests.

```
ping 2100::102 -c 3000
```

Code 6.6: The command executed to run ICMP tests

As shown, we used ping to measure the effects of using SLR on ICMP's performance. The test consisted of 3000 rounds of sending ICMP echo request packets and keeping track of the time it takes for the ICMP echo reply messages to arrive.

The results showing average latency and standard deviation are shown in 6.2. The average latency for a direct connection between C and S is 0.144 ms, with a standard deviation of 0.013. The average latency for the Benchmark setup is 0.183 ms, with a standard deviation of 0.019. Finally, the average latency during running ICMP on our prototype of SLR is 3.459 ms, with a standard deviation of 0.096.

The results show a performance hit of 2400% compared to Direct setup, and 1890% compared to Benchmark setup. While the absolute value itself is very small, being 3.459 milliseconds, the percentage hit is huge and a more optimised implementation is needed to lower the gap.

6.2.2 UDP

We used iperf3 to evaluate and measure the effects of using SLR on UDP's performance. We measured the average values over a timespan of 3000 seconds. We limit the UDP maximum datagram size to 1316 kbytes. We executed the program using the following command.

```
iperf3 -c 2100::102 -t 3000 --udp -b 1G -l 1316
```

Code 6.7: The command executed to run UDP tests

In our tests we compare direct connections, Benchmark connections, and SLR by comparing the values of bandwidth, packet loss, and jitter.

Figure 6.3 shows the results for the bandwidth tests, figure 6.4 shows the results for the packet loss tests, and figure 6.5 shows the results for the jitter tests.

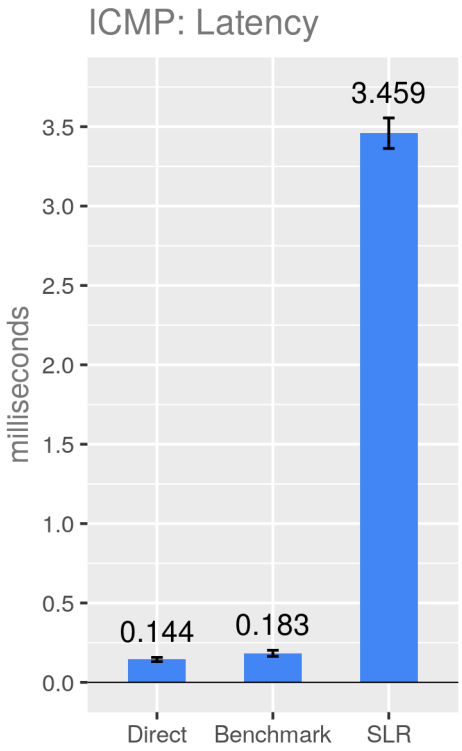

Figure 6.2: A diagram showing the effect of using SLR on the performance of ICMP by comparing the latency to direct connections and Benchmark. Vertical axis is latency in milliseconds.

We measure the bandwidth on both the Client and the Server. This is because in SLR's case the forward and backward paths can possibly have different bandwidth limitations, or different rates of packet loss affecting the amount of data delivered, and thus the bandwidth. This can be seen in figures 6.3, where the Server sends data at only 45% of the Client's rate, and in 6.4 where the packet loss measured on the Server's side is 55%.

The average bandwidth for a direct connection is 933 Mbits/sec. Average bandwidth for a Benchmark connection is 763 Mbits/sec. With reduced maximum datagram size, the average bandwidth for a direct connection falls to 864 Mbits/sec, and the average bandwidth for a Benchmark connections falls to 698 Mbits/sec. Average bandwidth for a connection running on SLR is 36 Mbits/sec on the Client's side, and 15.7 Mbits/sec on the Server's side.

The average packet loss rate for a direct and benchmark connections stayed at 0% independent of the maximum datagram size. For SLR, the average packet loss rate is 55%.

Figure 6.5 shows that the average jitter for a direct connection is 0.005 ms. Average jitter for a Benchmark connection is 0.008 ms. With reduced maximum datagram size, the average jitter for a direct connection becomes 0.003 ms, and the average jitter for a Benchmark connections becomes 0.01 ms. Average jitter for a connection running on SLR is 0.121 ms.

6.2.3 TCP

We used iperf3 to measure the effects of using SLR on TCP's performance. We measured the average values over a timespan of 3000 seconds. We limit the TCP MSS to 1304 kbytes. We executed the program using the following command.

```
iperf3 -c 2100::102 -t 3000 -M 1304
```

Code 6.8: The command executed to run TCP tests

We compared average bandwidth and congestion window size (cwnd) to direct and Benchmark connections. We tested direct and Benchmark connections with lowered MSS. Figure 6.6 shows the results for the bandwidth tests, while figure 6.7 shows the results for the cwnd size tests.

The average bandwidth for a direct connection is 924 Mbits/sec. Average bandwidth for a Benchmark connection is a very close 923 Mbits/sec. With reduced MSS, the average bandwidth for a direct connection falls to 916 Mbits/sec, and the average bandwidth for a Benchmark connections falls to 898 Mbits/sec. Average bandwidth for a connection

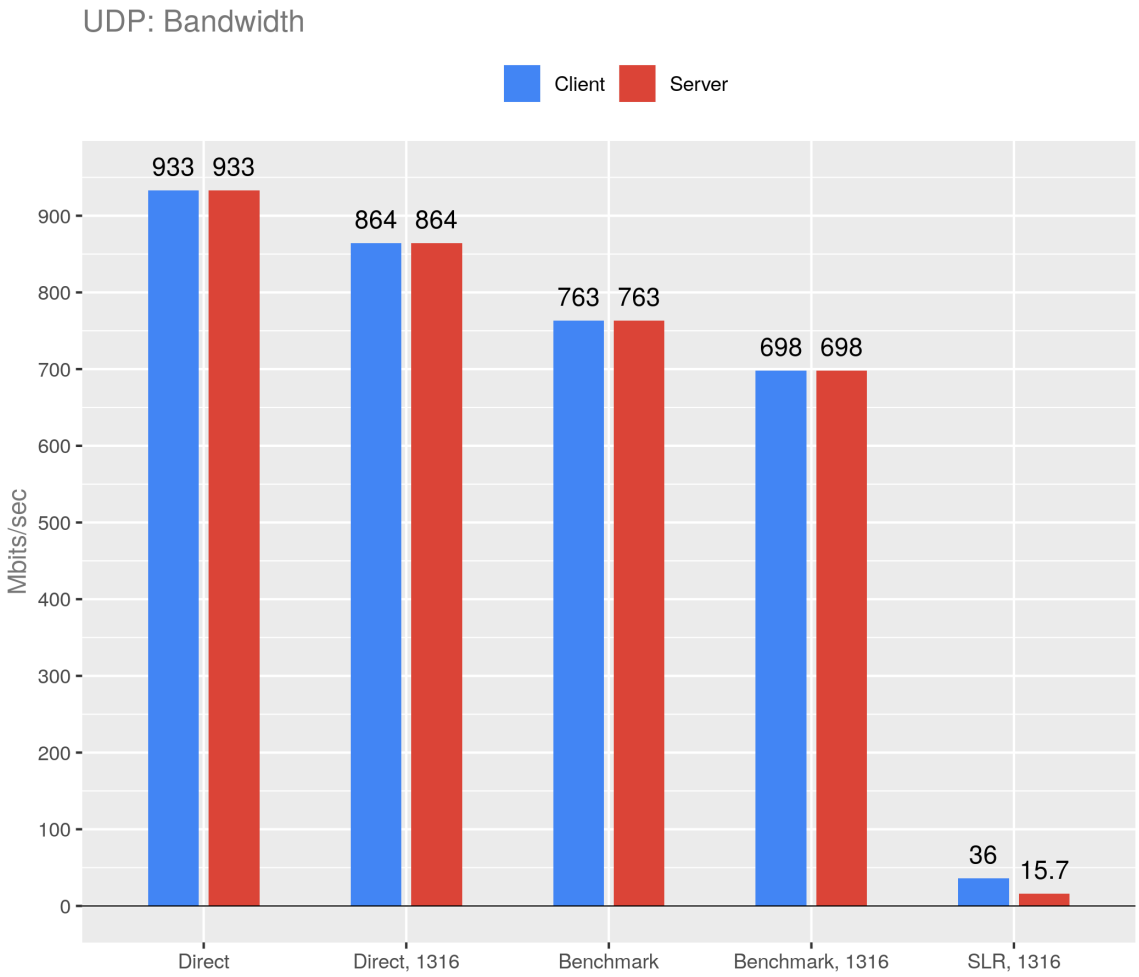

Figure 6.3: A diagram showing the effect of using SLR on the performance of UDP by comparing the bandwidth to direct connections and Benchmark. Vertical axis is bandwidth in Mbits/sec.

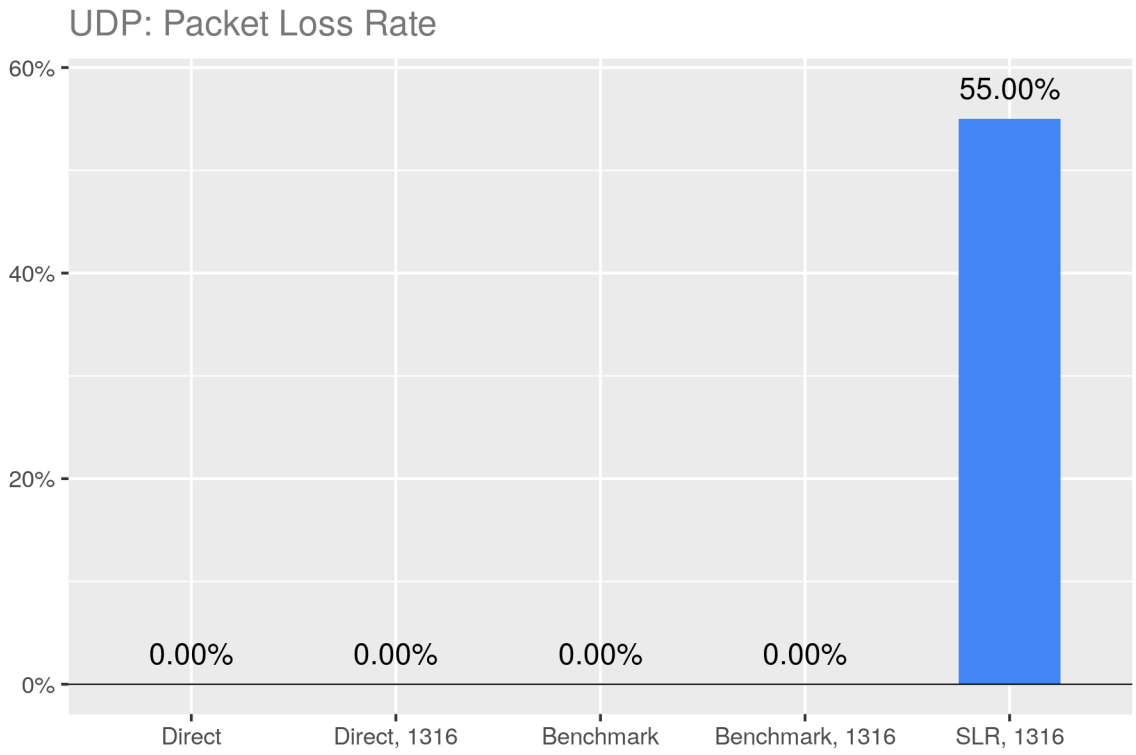

Figure 6.4: A diagram showing the effect of using SLR on the performance of UDP by comparing the percentage of packet loss to direct connections and Benchmark.

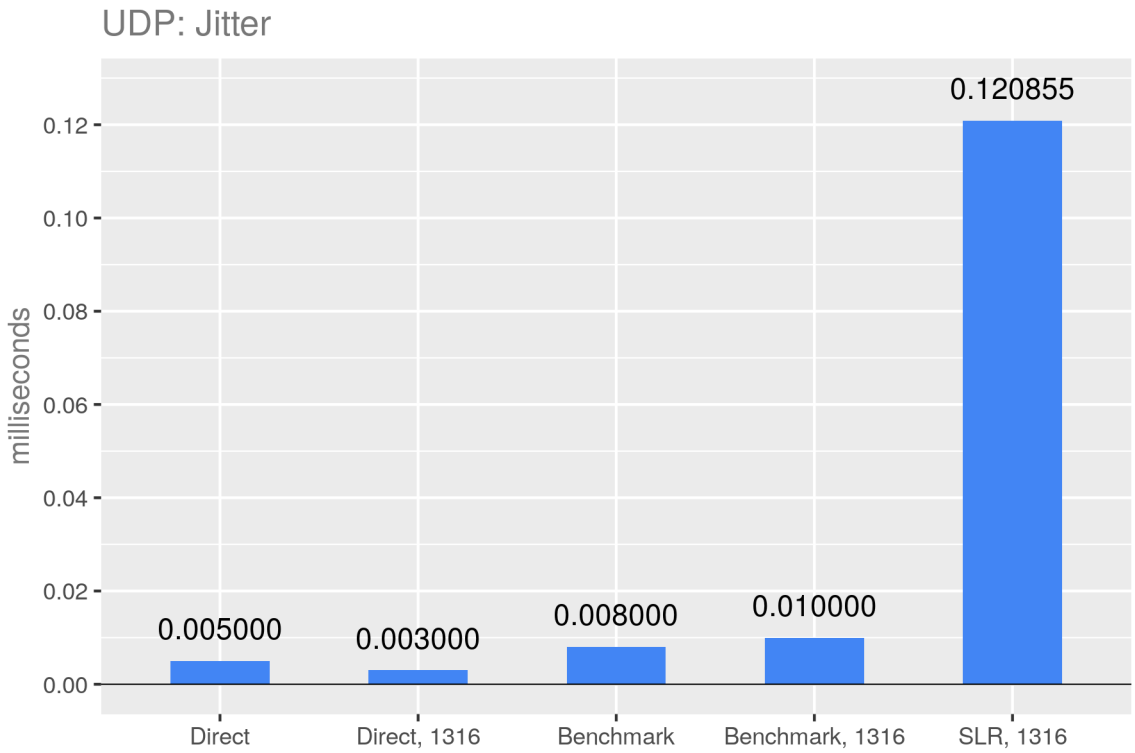

Figure 6.5: A diagram showing the effect of using SLR on the performance of UDP by comparing the average jitter to direct connections and Benchmark. Vertical axis is average jitter in milliseconds.

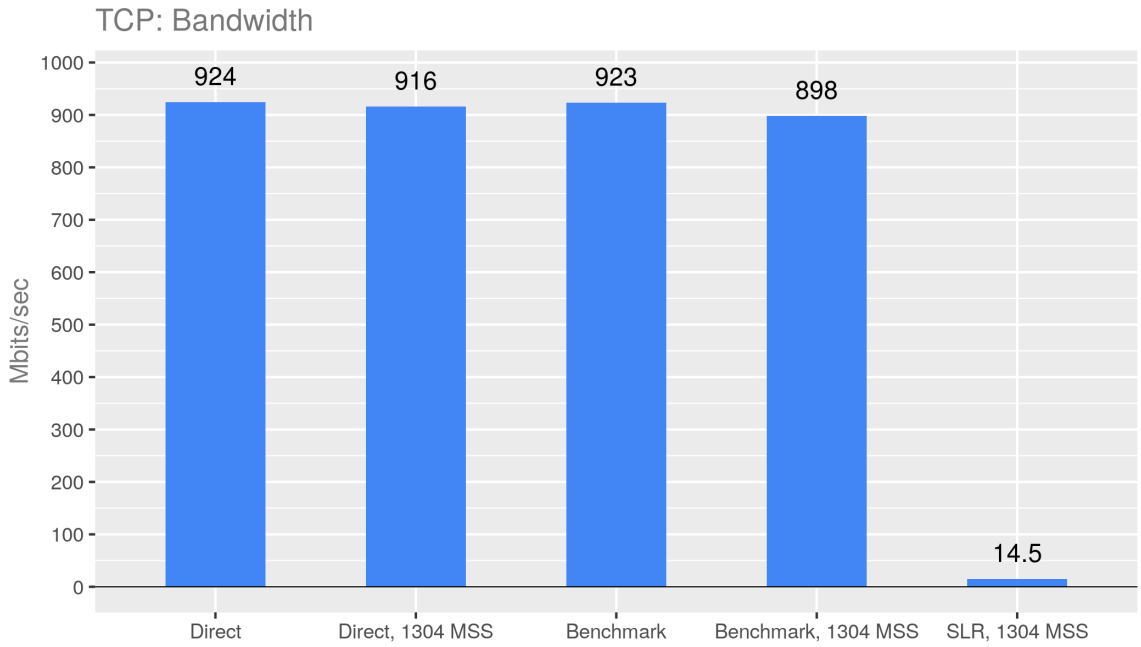

Figure 6.6: A diagram showing the effect of using SLR on the performance of TCP by comparing the bandwidth to direct connections and Benchmark. Vertical axis is bandwidth in Mbits/sec.

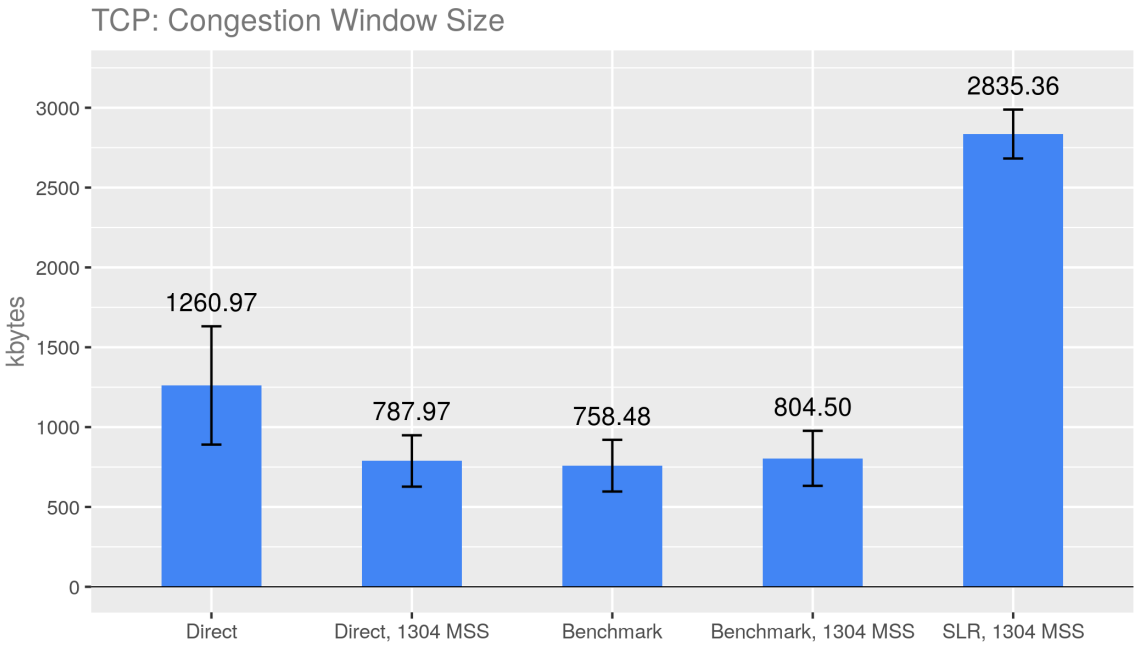

Figure 6.7: A diagram showing the effect of using SLR on the performance of TCP by comparing the congestion window size values to direct connections and Benchmark. Vertical axis is congestion window size in kbytes.

running on SLR is 14.5 Mbits/sec.

The cwnd values were measured to determine whether TCP congestion control was working properly. Changes in cwnd were detected, and while the value observed while running TCP on SLR is conspicuously large, the congestion control algorithm itself runs successfully.

As figure 6.7 shows, the average cwnd value measured for TCP running over a direct connection is 1260.97 KBs with a standard deviation of 370.43, while the average value measured for TCP running over Benchmark setup is 758.48 KBs with a standard deviation of 161.99.

With reduced MSS, the average cwnd size for a direct connection is 787.97 KBs with a standard deviation of 160.91, while the average value measured for TCP running over Benchmark setup is 804.5 KBs with a standard deviation of 172.4.

Average cwnd size for a connection running over SLR is 2835.36 KBs with a standard deviation of 153.24 KBs.

6.3 Analysis

By comparing the measurements for the Benchmark setup with the measurements of SLR, we can observe a severe hit to the network's performance. We suggest analysing the bottlenecks of the prototype, and determining whether a more optimised solution is possible.

The measurements of Direct and Benchmark setups show that redirecting the packets to the Python script is not the bottleneck, hence the conclusion that the script itself needs to be optimised. A more efficient algorithm that parses the packet, as well as more efficient encryption and decryption, should alleviate that performance hit.

7 Security Evaluation

We define the attack model's adversary as one that can compromise or operate a percentage of the network on which the anonymising system is operated, or a partial combination of the nodes themselves, specifically nodes A, B, and S.

We define compromising the connection as follows: in a scenario where C is sending and receiving data to and from S, the connection is compromised if an adversary can reveal the identity of C and S and link them to the network stream. This is assuming that the content of the stream itself does not reveal C's identity.

We define compromising a node as getting access to the node's private keys, being able to observe the traffic flowing through the node, and being able to observe the logic followed to route the packets.

In the following sections we analyse our defences against a partially compromised network and we try to prove that the proposed design is as secure as Tor with respect to the narrowly scoped attack model. Extensive formal security analysis of the system is needed to conclusively measure and prove the level of anonymity and the effectiveness of the defences against adversaries that the design suggests. This is considered out of scope in the context of this document, and can be done as part of future work.

We start by analysing simple attacks that target specific devices across the network in order to compromise the identity of C and S. An example of such attack would be having A, B, S, or any combination of them controlled by an adversary. We mainly focus on statistical traffic analysis attacks, such as flow correlation attacks [46].

This analysis does not consider nor propose any security countermeasures to an attack on C itself. We consider this scenario to be out of scope.

7.1 SLR Security Analysis

From the security perspective there are three different phases to analyse. The first phase is pre-circuit establishment, or the discovery phase. We analyse the security implications of using third-party sources to discover nodes and keys.

7.1.1 Discovery

Pre-circuit establishment, C needs to obtain the identities of nodes A and B acting as part of the SLR anonymising infrastructure. We discuss this in details in section 2.2, where we propose two different approaches to discovering nodes: using an implementation of distributed hash tables named Chord, and using directories.

Using Chord we can distribute a list of SLR node identities over multiple nodes. To retrieve a specific node, C generates a random number k between 0 and 2^n, where 2^n is the size of the DHT. This random number points to a specific entry in the DHT.

There is a problem with this approach. The randomly generated identifier later used to request a node's identity has to be shared with some of the other nodes within the Chord infrastructure. On average, log(n) out of n nodes know which identifier is generated. This can be mitigated by using recursive requests, but it will still cause security problems concerning the first few nodes engaged in the recursive call.

If the initial DHT node is compromised, the identifier can be linked with C and thus C can be linked with the requested A or B node, and since the Chord nodes are distributed and independent, there exists a non-negligible probability of that happening. In addition to this, compromising the initial node will allow an adversary to return malicious A and B nodes to C, contributing to further compromising its identity.

The other approach is following Tor's tracks and using directories. This approach is also not perfect. The directories themselves can be compromised, sending malicious responses to C.

Extensive research needs to be put into discovery to devise a technique to obtain identities of the nodes securely in a zero-trust environment.

Discovery of Keys and Forward Secrecy

Section 2.2 explores multiple approaches to discover anonymising nodes. The infrastructure used for discovery can be used to discover cryptographic public keys. However, relying on static keys to encrypt the first packet sent by C breaks Forward Secrecy [47]. Compromising the private key will allow an adversary to decrypt the identities of all end-hosts that have communicated with the compromised node. This security vulnerability severely undermines our system's security.

The main complexity behind ensuring Forward Secrecy in Circuit Establishment can be explained by focusing on a key exchange between only two nodes: C and B. Diffie-Hellman key exchange relies on generation of random keys and the exchange of the

public components. However, we establish the circuit by sending C's IP encrypted using a key shared between C and B. Before decrypting this message, B wouldn't be able to respond to C in the first place and thus an exchange of public keys would not be possible.

One approach would be to regularly send randomly generated keys to the directory, where each Client can pull a key and use it for establishing a circuit. The keys are one-time use only. Once a key is pulled, the directory should remove it from the list of available keys for B. This approach requires a method for C to request a key without the directory being able to link the requester with the owner of the key. It also requires a method for B to update the directory in an authenticated manner, without relying on long-term keys.

Another approach would be relying on a mathematical method to generate new secret keys derived from older ones, in a manner that would be consistent with the derivation of public keys on the other end. To maintain Forward Secrecy, the new keys cannot be used to derive the old ones. Multiple papers discuss methods to do this, such as Forward Secure Non-Interactive Key Exchange [48] that relies on generic levelled multilinear maps, and Forward Secure Non-Interactive Key Exchange from Indistinguishability Obfuscation [49]. Further research into these technologies is needed to determine their practicality and compatibility with our protocol. However, the current state of the protocol does not address this problem.

7.1.2 Circuit Establishment

During circuit establishment, we exchange information between the nodes as shown in section 3.1. We divide this information into two groups: randomly generated data, and confidential data containing identity information.

The confidential identity data is always encrypted, while the randomly generated data is exchanged in plain text. There is, however, one exception: AES keys. While they are randomly generated per circuit, they are also encrypted.

When analysing circuit establishment we only concern ourselves with revealing identifying information. To reveal any such information during circuit establishment the keys used to encrypt the AES keys must be compromised. The attacks would thus focus on exploiting the manner in which the private component of the node's key pair itself is stored, which is out of the scope of this thesis. Other attacks focusing on the generation of the keys themselves, but we consider this to be out of scope as well.

The AES keys are encrypted using the receiver's asymmetric public key. This key is

obtained out-of-band pre-circuit establishment. We encrypt the AES keys because they are used to encrypt the nested headers. If an adversary would be able able to reveal the AES key, they would also be able to reveal critical information. Compromising A's AES key would reveal S's identity, compromising S's would reveal B's, and compromising B's would reveal C's.

As shown in chapter 3, the length of keys we use are based on the recommendations of NIST. According to NIST, keys with the configured length should be secure through the year 2030.

We use the different AES keys to encrypt the headers multiple times. We wrap the headers under multiple layers of encryption to counter fingerprinting.

7.1.3 Communication Over Established Circuit

We define two possible scenarios where two different models of an adversary attempt to attack the SLR network. We call the first Attacker$_{Individual}$, an attacker that aims to compromise the network by compromising individual nodes. The second is Attacker$_{ISP}$, an attacker that controls a percentage of the routing infrastructure and can observe the flow of packets between the nodes.

We start by analysing attacks that can be performed by Attacker$_{Individual}$.

To deterministically reveal the identity of S, A needs to be compromised, while to reveal the identity of C, B needs to be compromised. However, compromising a single Router does not constitute an attack due to the limited knowledge held by each of the Routers. To reveal both identities at the same time, both Routers need to be compromised.

This in itself cannot be used to link the forwards and backwards flow of packets together. The meta-data received by A and B are encrypted, and this data changes as a packet passes through S. To link the packets, S would need to collude revealing the existence of a link between CI_S and CI_B. Another way to link the forward and backwards flows would be by carefully crafting packets of specific sizes and statistically analysing the flows.

Generalising this further, to reveal the identity of both hosts and be able to link them to a certain network stream, both Routers forwarding traffic to end-hosts along both of the routes needs to be compromised, in addition to either S's collusion, or statistical attacks that allow an adversary to link the flows with high confidence.

Attacker$_{ISP}$ can attack the network if it can observe packet flows between the set of nodes forming an SLR circuit. If the entire infrastructure routing the packets between the nodes A, B, C, and S, is observable, an adversary can reveal the identity of the communicating hosts using statistical methods, and without actively compromising any of the nodes. An example is flow correlation attacks [46], used to correlate a flow over an input link of a node with a flow over an output link of the same node.

If the adversary can locate A, they will be able to trace back the packets to an end-host. Conversely, if they can locate B, they will be able to trace back the packets to S. An adversary that only has access to infrastructure over a single route will be able to reveal information about only one of the end-hosts, assuming ability to identify the router on the path, being either A or B. Such an attack can be combined with other forms of attacks to reveal the identities of both hosts.

Countering this attack is possible by requiring A and B each to be in a different network that is not common with C or S. By placing A and B across different networks an adversary is required to have access to trace packets across these paths, which is practically much more difficult than if the routers exist in the same Autonomous System (AS). This technique is used by Tor [14], where nodes are required to be in different subnets.

Probability of Revealing Identities

We analyse the probability of compromising the identities of both end-hosts by an adversary who has control over a certain percentage of the network. We only consider A and B being compromised, since compromising other nodes will not contribute to revealing the identities of both ends. We assume a flat structured network topology, and we assume that each node is equally likely to be compromised.

Let N be the set containing all the nodes of the network, and let K be the set containing the compromised nodes. n is the number of nodes in N, and k is the number of nodes in K. k = n/m where m is a constant and $1 \leq m \leq \infty$.

$\Pr(\{A, B\} \subseteq K) = \Pr(A \subseteq K) * \Pr(B \subseteq K \setminus \{A\}) = \frac{k}{n} * \frac{k-1}{n-1} = \frac{1}{m} * \frac{\frac{n}{m}-1}{n-1} \approx \frac{1}{m^2}$ for a large n.

In other words, the probability of compromising the identity of both end-hosts is proportional to the square of the percentage of the compromised network. Figure 7.1 shows this relation. For the probability of having both A and B compromised to be 50%, $\frac{1}{\sqrt{2}}$% of the network, or 70.7% of the network, needs to be compromised.

Note how we do not include the complexity of tracing the source of the packets. This is because this does not add complexity to a correlation attack. An observer of A and B will be able to observe the destinations of the packets, and by correlating the flows, the

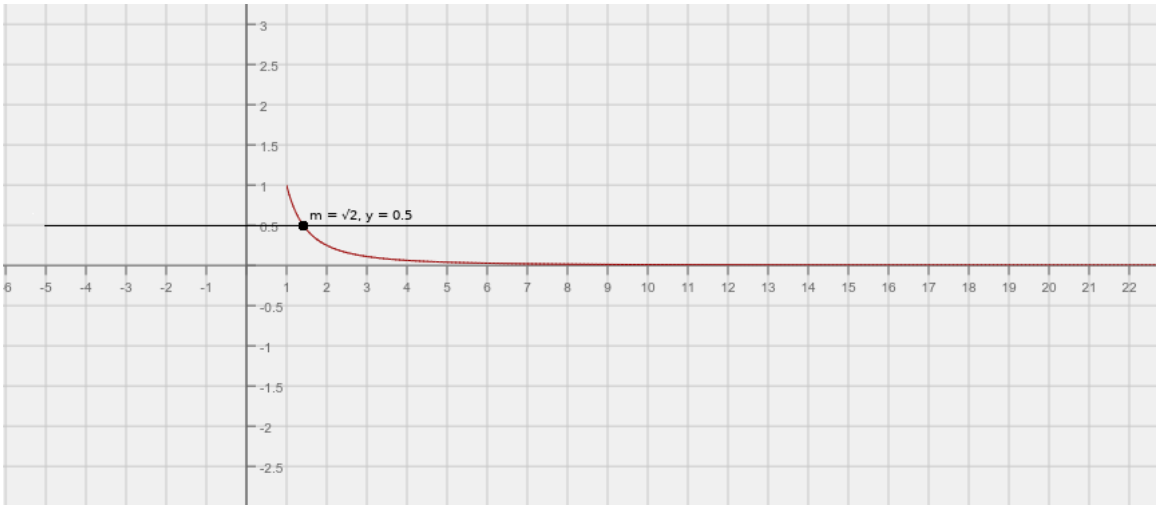

Figure 7.1: Relation between the percentage of the compromised nodes and the chance both A and B are compromised. X axis is the value for m, where $\frac{n}{m}$ is the percentage of the compromised nodes out of the network, and the Y-axis is the probability that both A and B are compromised. The horizontal line is drawn at m = $\sqrt{2}$, where the probability of both A and B being compromised is 50%.

identities of C and S are revealed.

However, the complexity of tracing the source of the packets communicated over SLR gives rise to a security strength point. In a situation where an adversary does not directly control A and B, but rather observes the network ingress and egress of A and B, the observation point affects the information obtained by an adversary. This is explained earlier in chapter 2 and figure 2.1.

For example, in a scenario where an adversary observes A's ingress traffic, if there exists a large number of nodes between C and A, the closer the observation point is to A, the more difficult it will be to correlate the flow with C itself.

On the other hand, for an adversary observing the traffic between A and S, the closer the observation point is to S, the harder it will be to correlate the flow with A, and so on for an adversary observing traffic between S and B, and B and C.

The statistical effect of this on the probability of compromising the network depends on the number of the intermediate routing nodes, the probability of compromising them, and the amount of traffic generated over different circuits that are using the same nodes. A simplification of the routing infrastructure would counter the purpose of this analysis, and a deeper analysis is beyond the scope of the thesis, so we leave it for future work.

7.2 Comparison With Tor

We attempt to compare our protocol's design with Tor's. We focus on the communication over active establishment phase. We start with comparing the details of the design itself and the differences in how both protocols operate after establishing a circuit. We compare the role and responsibilities of each node, and then we compare how much information is each node required to have to assume these responsibilities.

We should note that the differences between our protocol and Tor's are considered. Tor is an overlay network running over Layer 4, while our protocol runs on Layer 3 [16]. A comparison between systems with such fundamental differences is tricky, but we attempt this comparison nevertheless since the foundational goal of both systems is the same: anonymous communication. To achieve this, we abstract many concepts to make them comparable.

A Tor circuit consists of 3 main nodes, Entry, Relay, and Exit [4]. This is excluding the Client and the Server themselves. In Tor's design, both end-hosts, the client and the server, are excluded from the anonymising network and are considered external nodes. This is different to the design we propose in this thesis. However, our design can be easily altered to exclude S from the anonymising network by using S as a gateway to external networks.

In Tor's terms, our design has no Relay nodes because there are no multiple consecutive mix nodes in our design. We can abstract SLR in terms of Tor's Entry and Exit nodes by stating that both A and B each act as partial Entry and Exit nodes, each partially acting so.

While Tor establishes privacy by routing the packets through multiple nodes, we do that by splitting the forwards and backwards traffic and omitting source-identifying information from the packet headers.

Tor's design dictates that C sends packets to the Entry node that then routes it to a Relay node and then to an Exit node and then to a server. A response would pass through the same route from the Exit node, to the Relay node, to the Entry node, and finally to the client. However, in SLR, the request and response packets would take separate routes where A and B would each assume partial responsibilities of the Entry and Exit nodes.

More specifically, while the Entry and Exit nodes process packets emitted by both end-hosts, and both nodes exist consecutively along the same routes, A and B process packets emitted only by one end-hosts, and both nodes are not connected nor are they aware of the identities of each other.

Table 7.1: SLR's knowledge matrix

	C	A	S	B
C	X	X	X	X
A		X	X	
S			X	X
B	X			X

Table 7.2: Tor's knowledge matrix

	C	Entry	Relay	Exit	S
C	X	X	X	X	X
Entry	X	X	X		
Relay		X	X	X	
Exit			X	X	X
S				X	X

The result of this is that SLR's and Tor's nodes each are aware of a different set of identities, both aiming to minimise the identities one node is aware of, and to maintain that one malicious node cannot reveal the identities of the end-hosts. We show the knowledge possessed by nodes in both systems in tables 7.1 and 7.2.

The tables are knowledge matrices comparing SLR to Tor. An X on row A and column B means that node A has enough information to identify node B. The tables show that no one node is aware of both C and S, and thus a single node acting in a malicious manner would not risk compromising the identities of the end-hosts.

Attacks on Tor can be divided into two categories: attacks on mix networks, and attacks on the specific design of Tor. Attacks on mixed networks rely on analysing the ingress and egress traffic of the anonymisation layer between end-hosts and attempting to correlate the traffic with specific users [50][51]. These methods can be reliably used by a omnipresent adversary to identify the end-hosts by analysing meta-data such as the patterns of the exchanged packets, or latency.

Tor's attack model excludes such adversaries: their security model only concerns adversaries able to compromise or observe a percentage of the network [52]. Tor assumes that the noise generated by the large number of users will form a protection against correlation attacks.

The Tor design specification states that no mixing, padding, or traffic shaping is per-

formed. We follow the same path guaranteeing at least the same level of security. In a situation where the number of SLR's users is as large as Tor's, the generated traffic should act as noise in itself, countering flow correlation attacks and other attacks relying on statistical analysis.

8 Conclusion

In this chapter we summarise our findings. We state to what extent we answered the research questions, and we state the answers themselves. We also state and explain improvements that could be done as part of future work.

8.1 Research Questions

In section 1.2 we listed the research questions we tried to answer in this thesis. In the following sections we explore the answers we could find for each of them.

8.1.1 Can we design a scheme that relies on omitting the source address to achieve anonymity comparable to Tor?

We have designed a scheme, SLR, that relies on omitting the source address as shown in chapter 2. We have proposed an implementation as shown in chapter 3 and we have designed and implemented a prototype as shown in chapter 4.

However, we have not been able to properly compare our protocol to Tor's, and thus we are not able to confidently claim that SLR is as secure.

While the initial results shown in chapter 7 show that SLR provide at least the same level of security provided by Tor, especially considering Tor's lack of defences against correlation attacks, more work is required to be able to prove or refute this idea in a more solid manner.

8.1.2 What consequences will this scheme have on higher level networking protocols and how can they be addressed?

For the second research question, we have been able to meticulously analyse ICMP and UDP's features as shown in chapter 5 and we can confidently answer the question with respect to those protocols. Both have been observed to be working correctly over SLR, except for ICMP's Time Exceeded message.

However, while we were not able to analyse TCP with the same level of detail due to the protocol's complexity, we have run sufficient measurements as shown in chapter 6 to allow us to claim that TCP can run optimally over SLR.

Due to the entwining of protocols from different layers, each different protocols requires its own set of analysis and measurements, so we cannot claim that all other protocols, such as FTP, SCTP, or SSH, do work over SLR without problems.

8.2 Future Work and Final Thoughts

While we have been able to design and test a fundamentally new approach to network anonymity, we were faced by a number of hurdles and technical difficulties. In this section we propose approaches to overcome these hurdles and to achieve a theoretical design of SLR that is more secure, and more optimised.

8.2.1 Forward Secrecy

As shown in section 7.1.1, we have not been able to design a mechanism that maintains forward secrecy during circuit establishment. The theoretical solutions are methods to establish Forward Secure Non-Interactive Key Exchange. We have not been able to construct a practical implementation of these methods.

The lack of Forward Secrecy during Circuit Establishment constitutes a major security deficiency in comparison to Tor. This problem can be approached from another angle by constructing a practically suitable solution, without requiring the solution to be theoretically unbreakable. An example is generating multiple single-use keys as proposed in section 7.1.1. The suitability of this approach requires extensive research and measurements that we have not had the chance to perform.

8.2.2 Comparison With Tor

Another major hurdle we encountered was the comparison with Tor. We have not been able to compare the protocols properly. We have not been able to conclusively claim whether or not SLR provides privacy equivalent to Tor's.

Among the problems we had were that Tor's design specification is outdated and does not match the current implementation. While we compared SLR to the published specification, this does not constitute a proper comparison if Tor's implementation is simply different.

Another difficulty was analysing the design attacks. The design of Tor is too complex to analyse and compare to our proposed protocol, if we only concern ourselves with the published documents.

The implementation's documentation is based not only on the design specification, but also on tens of other proposals [53]. A more careful review that includes the extremely wide range of documents spanning Tor's design and internal technicalities is needed to establish a fair and accurate comparison between SLR and Tor.

8.2.3 Prototype

The measurements we have obtained in chapter 6 show that SLR's performance is within practical limits that allow for browsing and streaming. As an example, YouTube System Requirement [54] show that the recommended sustained speed for streaming an HD 1080p video is 5 Mbps, while our TCP tests shown in section 6.2.3 show that SLR can provide a speed of up to 14.5 Mbps.

However, as a fraction of the measured bandwidth over a direct connection within our test setup, this speed is merely 1.569% of the available bandwidth.

While we have not been able to pinpoint the cause of the performance hit, we were able to run tests that show that the performance hit is not caused by the lowered maximum segment size, nor by the interception of packets using iptables and routing them to NetfilterQueue, and then to the Python scripts.

We believe that the Python scripts we have created can be optimised further. Better algorithms for parsing and modifying the packets should be created. We suggest looking into creating a kernel-space implementation, possibly using a programming language that can provide better performance than Python.

Bibliography

[1] D. L. Chaum. “Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms”. In: *Commun. ACM* 24.2 (Feb. 1981), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/358549.358563.

[2] P. Boucher, A. Shostack, and I. Goldberg. “Freedom System 2.0 Architecture”. In: (Dec. 2000). URL: https://adam.shostack.org/zeroknowledgewhitepapers/Freedom_System_2_Architecture.pdf.

[3] Z. Brown. “Cebolla: Pragmatic IP Anonymity”. In: *Proceedings of the Ottawa Linux Symposium*. Vol. 2137. June 2002, pp. 55–64. URL: https://www.kernel.org/doc/mirror/ols2002.pdf.

[4] R. Dingledine, N. Mathewson, and P. Syverson. “Tor: The Second-Generation Onion Router”. URL: https://svn.torproject.org/svn/projects/design-paper/tor-design.pdf.

[5] *ZeroKnowledge to Discontinue Anonymity Service*. Slashdot, Oct. 2001. URL: https://slashdot.org/comments.pl?cid=2388977&sid=22261&tid=158.

[6] *Cebolla Source Code*. Cypherspace Internet Security, Nov. 2003. URL: http://www.cypherspace.org/cebolla/source/.

[7] *The Anonymous Internet*. Tech. rep. Oxford Internet Institute, June 2014. URL: http://geography.oii.ox.ac.uk/the-anonymous-internet/.

[8] K. Loesing. *Performance of Requests over the Tor Network*. Tech. rep. Tor, Sept. 2009. URL: https://research.torproject.org/techreports/torperf-2009-09-22.pdf.

[9] *The State of the Internet*. Tech. rep. Akamai, 2009. URL: https://research.torproject.org/techreports/torperf-2009-09-22.pdf.

[10] D. Goldschlag, M. Reedy, and P. Syversony. *Onion Routing for Anonymous and Private Internet Connections*. Jan. 1999. URL: https://www.onion-router.net/Publications/CACM-1999.pdf.

[11] R. Dingledine. *Tor is free*. Oct. 2003. URL: https://lists.torproject.org/pipermail/tor-dev/2003-October/002185.html.

[12] M. Takeuchi, E. Kohno, T. Ohta, and Y. Kakuda. “Improvement of Throughput Using Partially Node-disjoint Forward and Backward Paths for Mobile Ad Hoc Networks”. In: *Proc. second international workshop on dependable network computing and mobile systems (DNCMS 2009)*. 2009, pp. 55–62.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishna. “Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications”. In: *ACM SIGCOMM’01* (Aug. 2001).

[14] R. Dingledine and N. Mathewson. *Tor Path Specification*. [viewed on 10/04/2019]. The Tor Project. URL: https://gitweb.torproject.org/torspec.git/tree/path-spec.txt?id=54ebbd5643600268b6af16c8603089898b72f9cb.

[15] *IPv6 Extension Headers Review and Considerations*. [viewed on 08/04/2019]. Cisco Systems. Oct. 2006. URL: https://www.cisco.com/en/US/technologies/tk648/tk872/technologies_white_paper0900aecd8054d37d.html.

[16] *ISO/IEC 7498-4:1989 – Information technology – Open Systems Interconnection – Basic Reference Model: Naming and addressing*. [viewed on 08/04/2019]. International Organization for Standardization. Nov. 1989. URL: http://standards.iso.org/ittf/PubliclyAvailableStandards/s014258_ISO_IEC_7498-4_1989(E).zip.

[17] *IANA IPv6 Special-Purpose Address Registry*. [viewed on 04/04/2019]. URL: https://www.iana.org/assignments/iana-ipv6-special-registry/iana-ipv6-special-registry.xhtml.

[18] R. Hinden and S. Deering. *RFC 4291 - IP Version 6 Addressing Architecture*. Internet Engineering Task Force (IETF). Feb. 2006. URL: https://tools.ietf.org/html/rfc4291.

[19] E. Barker. “Recommendation for Key Management, Part 1: General”. In: *NIST Special Publications* 800-57 Part 1 Rev. 4 (Jan. 2016). DOI: 10.6028/NIST.SP.800-57pt1r4.

[20] E. K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *RFC 8017 - PKCS #1: RSA Cryptography Specifications Version 2.2*. Internet Engineering Task Force (IETF). Nov. 2016. URL: https://tools.ietf.org/html/rfc8017.

[21] D. J. Bernstein. “Curve25519: New Diffie-Hellman Speed Records”. In: *Public Key Cryptography - PKC 2006*. Ed. by M. Yung, Y. Dodis, A. Kiayias, and T. Malkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. ISBN: 978-3-540-33852-9. DOI: 10.1007/11745853_14.

[22] V. Shoup. *A Proposal for an ISO Standard for Public Key Encryption (version 2.1)*. IBM Zurich Research Lab, Dec. 2001. URL: https://www.shoup.net/papers/iso-2_1.pdf.

[23] D. A. McGrew and J. Viega. “The Security and Performance of the Galois/Counter Mode (GCM) of Operation”. In: *Progress in Cryptology - INDOCRYPT 2004*. Ed. by A. Canteaut and K. Viswanathan. Berlin, Heidelberg: Springer Berlin Heidelberg, Dec. 2005, pp. 343–355. ISBN: 978-3-540-30556-9.

[24] P. Rogaway. “Nonce-Based Symmetric Encryption”. In: *Fast Software Encryption*. Ed. by B. Roy and W. Meier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 348–358. ISBN: 978-3-540-25937-4.

[25] H. Krawczyk and P. Eronen. *RFC 5869 - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*. Internet Engineering Task Force (IETF). May 2010. URL: https://tools.ietf.org/html/rfc5869.

[26] Wouter Penard and Tim van Werkhoven. “On the Secure Hash Algorithm family”. Archived on 30/03/2016. URL: https://web.archive.org/web/20160330153520/http://www.staff.science.uu.nl/~werkh108/docs/study/Y5_07_08/infocry/project/Cryp08.pdf.

[27] S. Deering and R. Hinden. *RFC 8200 - Internet Protocol, Version 6 (IPv6) Specification*. Internet Engineering Task Force (IETF). July 2017. URL: https://tools.ietf.org/html/rfc8200.

[28] *Assigned Internet Protocol Numbers*. Internet Assigned Numbers Authority (IANA). Oct. 2017. URL: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml%5C#protocol-numbers-1.

[29] B. Fenner, AT&T Labs. *RFC 4727 - Experimental Values in IPv4, IPv6, ICMPv4, ICMPv6, UDP, and TCP Headers*. Internet Engineering Task Force (IETF). Nov. 2006. URL: https://tools.ietf.org/html/rfc4727.

[30] Charles Hornig. *RFC 894 - A Standard for the Transmission of IP Datagrams over Ethernet Networks*. Symbolics Cambridge Research Center. Apr. 1984. URL: https://tools.ietf.org/html/rfc894.

[31] A. Mashaal. *SLR: Sourceless Routing using split network routes over Layer 3*. URL: https://github.com/TheNavigat/slr.

[32] P. N. Ayuso. *The netfilter.org ”iptables” project*. [viewed on 21/08/2019]. netfilter.org. URL: https://www.netfilter.org/projects/iptables/index.html.

[33] P. N. Ayuso. *The netfilter.org ”libnetfilter_queue” project*. [viewed on 21/08/2019]. netfilter.org. URL: https://www.netfilter.org/projects/libnetfilter_queue/.

[34] D. Song. *dpkt*. [viewed on 21/08/2019]. URL: https://github.com/kbandla/dpkt.

[35] T. Narten, E. Nordmark, and W. Simpson. *RFC 2461 - Neighbor Discovery for IP Version 6 (IPv6)*. Network Working Group. Dec. 1998. URL: https://tools.ietf.org/html/rfc2461.

[36] *struct — Interpret strings as packed binary data*. [viewed on 22/08/2019]. Python Software Foundation. URL: https://docs.python.org/2/library/struct.html.

[37] *socket — Low-level networking interface*. [viewed on 22/08/2019]. Python Software Foundation. URL: https://docs.python.org/2/library/socket.html.

[38] A. Conta, S. Deering, and M. Gupta. *RFC 4443 - Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. Internet Engineering Task Force (IETF). Mar. 2006. URL: https://tools.ietf.org/html/rfc4443.

[39] S. Bradner. *RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels*. Internet Engineering Task Force (IETF). Mar. 1997. URL: https://tools.ietf.org/html/rfc2119.

[40] J. McCann, S. Deering, J. Mogul, and R. Hiden. *RFC 8201 - Path MTU Discovery for IP version 6*. Internet Engineering Task Force (IETF). July 2017. URL: https://tools.ietf.org/html/rfc8201.

[41] Mathis, M. and Heffner, J. *RFC 4821 - Packetization Layer Path MTU Discovery*. Internet Engineering Task Force (IETF). Mar. 2007. URL: https://tools.ietf.org/html/rfc4821.

[42] J. Postel. *RFC 768 - User Datagram Protocol*. Internet Engineering Task Force (IETF). Aug. 1980. URL: https://tools.ietf.org/html/rfc768.

[43] *Intel® Pentium® Processor E5200*. Intel Corporation, 2008. URL: https://ark.intel.com/content/www/us/en/ark/products/37212/intel-pentium-processor-e5200-2m-cache-2-50-ghz-800-mhz-fsb.html.

[44] *Installation Guide: AT-GS916 - GS924*. Allied Telesyn, 2004. URL: https://www.alliedtelesis.com/sites/default/files/documents/installation-guides/gs916-24_ig_a.pdf.

[45] *Ubuntu 19.04 LTS (Disco Dingo)*. Canonical Ltd., Apr. 2019. URL: https://wiki.ubuntu.com/DiscoDingo/ReleaseNotes.

[46] Y. Zhu, X. Fu, B. Graham, R. Bettati, and W. Zhao. "On Flow Correlation Attacks and Countermeasures in Mix Networks". In: *Privacy Enhancing Technologies*. Ed. by D. Martin and A. Serjantov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 207–225. ISBN: 978-3-540-31960-3.

[47] W. Diffie, P. C. van Oorschot, and M. J. Wiener. "Authentication and Authenticated Key Exchanges." In: *Des. Codes Cryptography* 2 (June 1992), pp. 107–125. DOI: 10.1007/BF00124891.

[48] D. Catalano, M. Di Raimondo, D. Fiore, R. Gennaro, and O. Puglisi. "Fully non-interactive onion routing with forward secrecy". In: *International Journal of Information Security* 12.1 (Feb. 2013), pp. 33–47. ISSN: 1615-5270. DOI: 10.1007/s10207-012-0185-2.

[49] Young Kyung Lee and Dong Hoon Lee. “Forward Secure Non-Interactive Key Exchange from Indistinguishability Obfuscation”. In: *2015 5th International Conference on IT Convergence and Security (ICITCS)*. 800-57 Part 1 Rev. 4. Aug. 2015. DOI: 10.1109/ICITCS.2015.7292984.

[50] A. Back, U. Möller, and A. Stiglic. “Traffic Analysis Attacks and Trade-Offs in Anonymity Providing Systems”. In: vol. 2137. Apr. 2001, pp. 245–257. DOI: 10.1007/3-540-45496-9_18.

[51] B. N. Levine, M. Reiter, C. Wang, and M. Wright. “Timing Attacks in Low-Latency Mix Systems”. In: Feb. 2004. DOI: 10.1007/978-3-540-27809-2_25.

[52] S. J. Murdoch and G. Danezis. “Low-cost traffic analysis of Tor”. In: *2005 IEEE Symposium on Security and Privacy (S P’05)*. May 2005, pp. 183–195. DOI: 10.1109/SP.2005.12.

[53] *Tor’s protocol specifications - Proposals*. [viewed on 25/08/2019]. The Tor Project. URL: https://gitweb.torproject.org/torspec.git/tree/proposals.

[54] *YouTube - System Requirements*. [viewed on 25/08/2019]. Google. URL: https://support.google.com/youtube/answer/78358?hl=en.