Theodor Signebøen Midtlien

Reducing distinguishability of DTLS for usage in Snowflake

Master's thesis in Communication Technology
Supervisor: David Palma
June 2024

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Theodor Signebøen Midtlien

Reducing distinguishability of DTLS for usage in Snowflake

Master's thesis in Communication Technology
Supervisor: David Palma
June 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Title: Reducing distinguishability of DTLS for usage in Snowflake
Student: Signebøen Midtlien, Theodor

Problem description:

In the Internet today, there are diverse attempts by censors (e.g. governments, institutions, and service providers) to regulate, monitor, or entirely stifle access to the open internet. This phenomenon, known broadly as internet censorship, represents both a technical challenge and a significant societal concern, impacting human rights, freedom of expression, and the global exchange of ideas.

Snowflake is a censorship circumvention system that has had growing traction recently. Operating on the principle of volunteerism and decentralization, Snowflake employs ephemeral, short-lived proxies run by volunteers to resist blocking by censors. However, Snowflake has been successfully blocked at multiple occasions by fingerprinting the handshake of the Datagram Transport Layer Security (DTLS) protocol which is used for data transmission with WebRTC.

There are two main goals in this thesis. Firstly, a system for performing fingerprinting of different DTLS implementations will be developed. The system is going to be validated using data sets. Subsequently, the DTLS implementation used in Snowflake will be extended to support fingerprint-resistant features, inspired by *uTLS*. Validation of the DTLS implementation shall be done using the fingerprinting system.

Approved on: 2024-02-26
Main supervisor: Palma, David, NTNU

Abstract

The motivation behind this thesis is censorship circumvention. Snowflake is a technology that is used today to provide access to the free and open Internet for people located in areas that practice censorship. With users adopting the system, censors have spent an effort at trying to detect and block its traffic. We have seen that censors have been able to do so by fingerprinting the DTLS implementation that is produced by the Pion library used by Snowflake. The aim of this thesis is to reduce the distinguisability of said DTLS library. We developed a tool named, *dfind* for analyzing and finding passive field-based fingerprints of DTLS. This tool was validated using a data set with known fingerprints, and found that the extensions field was especially vulnerable for identification. To combat such fingerprints, we implemented *covertDTLS*, a Go library inspired by *uTLS*. Our module extends the Pion DTLS library with handshake hooking to offer mimicry and randomization features. To ensure that mimicking remains up-to-date, we developed a novel continuous delivery workflow for generating fresh DTLS-WebRTC handshakes from popular browsers. Using *covertDTLS* with Snowflake resulted in us not being able to find any fingerprints. We conclude that mimicking and randomization are effective countermeasures against passive, stateless, and field-based fingerprinting.

Sammendrag

Motivasjonen bak denne avhandlingen er å omgå sensur. Snowflake er en teknologi som i dag brukes av mennesker i områder som praktiserer sensur for å få tilgang til det frie og åpne Internett. Ettersom systemet har vist seg å være effektivt, har sensorer brukt ressurser på å prøve å oppdage og blokkere dens trafikk. Vi har sett at sensorer har vært i stand til å gjøre dette ved å finne og blokere fingeravtrykket av DTLS-implementeringen som produseres av Pion-biblioteket brukt av Snowflake. Målet med denne avhandlingen er å redusere identifiserbarheten til nevnte DTLS-bibliotek. Vi utviklet et verktøy, kalt *dfind*, for å analysere og finne passive feltbaserte fingeravtrykk av DTLS. Dette verktøyet ble validert ved bruk av et datasett med kjente fingeravtrykk, og fant at extensions-feltet var spesielt sårbart for identifikasjon. For å bekjempe slike fingeravtrykk implementerte vi *covertDTLS*, et Go-bibliotek inspirert av *uTLS*. Vår modul utvider Pion DTLS-biblioteket med imitasjons- og randomiseringsfunksjoner. For å sikre at imitasjonen forblir oppdatert, utviklet vi en ny kontinuerlig leveringsprosess for å generere ferske DTLS-WebRTC-håndtrykk fra populære nettlesere. Bruk av *covertDTLS* med Snowflake resulterte i at vi ikke klarte å finne noen fingeravtrykk. Vi konkluderer med at imitasjon og randomisering er effektive mottiltak mot passive, tilstandsfrie og feltbaserte fingeravtrykk.

Preface

This Master's thesis marks the completion of the 5-year Communication Technology study program at the Norwegian University of Science and Technology (NTNU).

Acknowledgments: I would like to express my gratitude for the guidance I have received from my supervisor, David Palma. I have thoroughly enjoyed the enlightening discussions over multiple years working together on various projects. Censorship circumvention is not done alone and I have been fortunate to be able to work with the anti-censorship team at the Tor Project: special thanks to Cecylia Bocovich, David Fifield, meskio, onyinyang and shelikhoo. I finally thank the maintainers at Pion for being positive of my features and reviewing my code: Sean DuBois, Atsushi Watanabe, Daenney.

Contents

List of Figures xi

List of Tables xii

List of Algorithms xiii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research questions . . . 3
- 1.3 Communities . . . 3
- 1.4 Ethics . . . 4
  - 1.4.1 Anonymity . . . 4
  - 1.4.2 Censorship circumvention . . . 6

2 Background 9
- 2.1 Preliminaries . . . 9
  - 2.1.1 Censorship and circumvention . . . 9
  - 2.1.2 Snowflake . . . 10
  - 2.1.3 Datagram Transport Layer Security (DTLS) . . . 11
  - 2.1.4 WebRTC . . . 14
  - 2.1.5 Fingerprinting . . . 16
  - 2.1.6 Network traffic obfuscation . . . 17
- 2.2 State of the art . . . 17
  - 2.2.1 Detecting Snowflake . . . 17
  - 2.2.2 Fingerprinting TLS . . . 20

3 Method 21
- 3.1 Methodology . . . 21
- 3.2 Scope and assumptions . . . 23
  - 3.2.1 Validation vs. evaluation . . . 23
  - 3.2.2 Fingerprinting scope . . . 23
  - 3.2.3 Censor's capabilities . . . 23
- 3.3 Exploratory work . . . 24

3.3.1 Fingerprinting tool . . . 24
3.3.2 Handshake generation . . . 25
3.3.3 DTLS library . . . 25

4 DTLS fingerprint discovery **27**
4.1 Architecture and implementation . . . 27
4.2 Results and discussion . . . 31
4.3 Further work . . . 33

5 DTLS handshake generation **35**
5.1 Overview and tools . . . 36
5.2 Implementation . . . 37
5.3 Results and discussion . . . 39
5.4 Further work . . . 40

6 DTLS library **41**
6.1 Feature support . . . 41
6.2 Implementation . . . 42
6.2.1 Handshake Hooks . . . 43
6.2.2 Mimicry . . . 45
6.2.3 Randomization . . . 46
6.3 Results and discussion . . . 48
6.3.1 Mimicked *ClientHello* . . . 48
6.3.2 Randomized *ClientHello* . . . 49
6.4 Further work . . . 52

7 Conclusions **53**

References **55**

Appendix

A Extensions fingerprints for Snowflake **61**
A.1 MacMillan et at. data set . . . 61
A.2 Snowbox Extension . . . 67

B SQL queries for dfind **69**
B.1 SQL query 1 . . . 69
B.2 SQL query 2 . . . 69
B.3 SQL query 3 . . . 69
B.4 SQL query 4 . . . 69
B.5 SQL query 5 . . . 69

C DTLS handshake generation workflow **71**

List of Figures

1.1 Estimated average simultaneous Snowflake users by day, from Bocovich *et al.* [11] . . . 2

2.1 Architecture of Snowflake [1] . . . 10
2.2 Message flights for the full DTLS 1.2 handshake [28]. Optional messages and flights are indicated with an asterisk. . . . 12
2.3 The WebRTC protocol stack . . . 15
2.4 Per-bit feature importance for identification of browser and application pair from MacMillan *et al.* data set, from Holland *et al.* [37] . . . 19

3.1 The design cycle for this thesis. Color-coding shows how **A1** and **A2.1** are used in later validation steps. . . . 21

4.1 *dfind* architecture overview . . . 28

5.1 Overview of the workflow architecture . . . 38

6.1 A mimicked *ClientHello* message, the highlighted bytes are replaced with values from the hooked message. . . . 45

List of Tables

4.1 Fields parsed by the dissector. The *ClientHello* and *ServerHello* columns indicate if the field is present in that message. The analysis column shows which fields are used in an automatic analysis 29
4.2 Number of parsed handshakes from Macmillan et al. data set 31
4.3 Identifying fields found during automatic analysis that can be used for fingerprinting Snowflake 32
4.4 Extensions that can be used for fingerprinting Snowflake 32

5.1 Identifiers for generated Chrome handshake 40

6.1 Identifiers for fresh Snowflake traffic from Snowbox 49

List of Algorithms

4.1 Finding unique values of given field and type of implementation . . . 29
4.2 Analyzing extensions by fuzzy string matching 30
6.1 Randomize order and length of a list 47

Chapter 1
Introduction

This chapter starts by presenting the motivation behind this thesis, which is adapted from the specialization project preceding this work [1]. Following the motivation, we define two research questions that we aim to answer. We then introduce key communities, some of which we have collaborated with, working on areas of open-source software and censorship circumvention. The chapter closes with a discussion about ethics, where some concerns are brought to light and an ethical stance is given.

1.1 Motivation

The Internet facilitates global sharing of information and ideas, such freedom of opinion and expression are protected by Article 19 of the United Nations Universal Declaration of Human Rights (UDHR) [2]. However, there are diverse attempts by censors (e.g. governments, institutions, and service providers) to violate these rights by regulating, monitoring, or, in some cases, by entirely stifling access to the open Internet [3–5]. This phenomenon, known broadly as Internet censorship, represents both a technical challenge and a significant global societal concern, impacting free speech and human rights at large. Internet is even being censored in regions often considered "*free*", such as the European Union (EU) [6] and Norway [7]. These are concerns shared by the Internet community at large, which is exemplified with the creation of the Human Rights Protocol Considerations Research Group (HRPC)[1]. The HRPC is doing ongoing research on human rights network protocol design [8] and has released Request For Comments (RFC) 8280 [9] on "Human Rights Protocol Considerations", both of which encourage censorship resistance in protocols.

The Onion Router (Tor) Project[2] is a digital rights non-profit organization that is on the forefront of researching and developing open-source software providing anonymity and privacy for users of the Internet. Their initial goal was to implement

[1] https://hrpc.io/
[2] https://www.torproject.org/

onion routing, a distributed network where traffic hops across multiple nodes, applying encryption at each step, before reaching its intended destination. This renders the original sender and receiver of the data difficult to trace, thus offering anonymity. Today, the project also develops the Tor browser and offers systems to circumvent forms of Internet censorship.

Censors see anonymity as a threat, and as they have grown tech savvier, they have begun detecting and blocking traffic that appears to be routed through the Tor network. This has led to the introduction of Pluggable Transports in Tor [10]. Pluggable Transports obfuscate Tor traffic, making it appear indistinguishable from regular Internet traffic, thereby allowing it to bypass censoring mechanisms. This disguise ensures that users can access the open Internet even from heavily censored regions without raising red flags.

One of the Pluggable Transport that has had a lot of traction lately is the Snowflake[3][11] censorship circumvention system (see Figure 1.1). Operating on the principle of volunteerism and decentralization, Snowflake employs ephemeral proxies run by volunteers using Web Real-Time Communication (WebRTC) peer-to-peer connections. So far, censors have not shown willingness to block WebRTC as a protocol [11], which allows Snowflake to blend in with the long tail of other traffic.

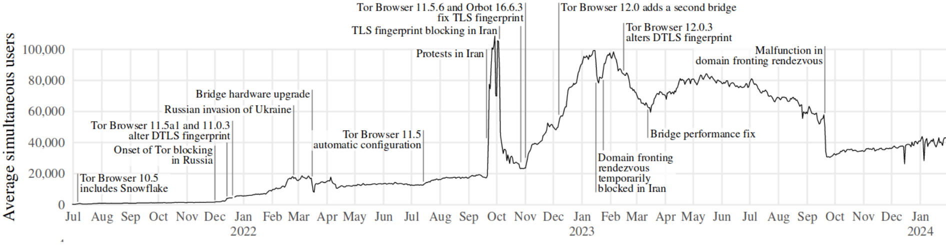


Figure 1.1: Estimated average simultaneous Snowflake users by day, from Bocovich *et al.* [11]

No censorship circumvention system is perfect, and Snowflake has been successfully blocked at multiple occasions. An example of this is Russia blocking Snowflake in May of 2022 [12]. They did this by fingerprinting unique *ClientHello* messages in the Datagram Transport Layer Security (DTLS) protocol which is used by WebRTC. This method has previously been discussed in literature and was a known attack vector [13]. Reactive measures have been deployed by the Tor project to remove the

[3]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake

distinguishing *ClientHello* fingerprint in the DTLS implementation by Pion[4], but other weaknesses may still exist [14].

Transport Layer Security (TLS), being a similar protocol to DTLS, has been studied for use in censorship circumvention. Frolov *et al.* [15] found multiple ways of fingerprinting TLS, including the *ClientHello* method used to block DTLS in Snowflake. To handle this problem, the researchers developed a library called *uTLS*[5] that aims to protect against fingerprinting. However, no such library exists for DTLS, which concerns the team behind and actively developing Snowflake:

"Owing to practical considerations, Snowflake's defenses to DTLS fingerprinting are not very robust, and are reactive rather than proactive. In the realm of TLS one may use uTLS, but there is as yet no equivalent for DTLS. The present way of altering DTLS fingerprints in Snowflake is to submit a patch to Pion when a feature used for fingerprinting is identified.", Bocovich *et al.* [11].

1.2 Research questions

Following the presented motivation, we define the following research questions for this thesis:

> **RQ1**: What kind of fingerprints can be used to identify different implementations of DTLS?

> **RQ2**: How can we create a fingerprint-resistant implementation of DTLS for usage in Snowflake?

1.3 Communities

This section shortly describes some communities and initiatives that have been invaluable during working on this thesis and for protecting the free and open Internet at large. We have directly collaborated with some of these communities, as censorship circumvention is a community-effort.

The Tor Project organization is divided into multiple teams developing different technologies for Internet freedom, where the anti-censorship team[6] is the one responsible for Snowflake and making Tor reachable anywhere in the world. During this thesis we have been part of that team, participating in many of the weekly public Internet

[4] https://github.com/pion
[5] https://github.com/refraction-networking/utls
[6] https://gitlab.torproject.org/tpo/anti-censorship

Relay Chat (IRC) meetings[7], contributing to discussions and receiving feedback on ideas.

Pion is a community-owned organization working on cross platform open-source software for real-time media and data communication. All their projects are written in Go and the main focus is implementing the WebRTC protocol stack. Snowflake adopts the Pion implementation of both DTLS and WebRTC. During this thesis, we contributed to their DTLS implementation, adding handshake hooking features.

Other important initiatives on the field are the Open Observatory of Network Interference (OONI)[8] and Censored Planet[9]. Both aim to measure and analyze internet censorship, and are valuable sources of data for researchers on the area.

Net4People BBS[10] is a multilingual open forum for people to share research, news and have discussions about Internet censorship circumvention. It is a highly valuable bulletin-board for gathering information about the needs of users affected by censorship, and getting an up-to-date view of the current landscape.

1.4 Ethics

The purpose of this subsection is to discuss some of the ethical aspects related to online anonymity and censorship circumvention. Its purpose is not to conduct a rigorous philosophical analysis, but rather highlight ethical issues, different opinions and set the stance of the thesis.

1.4.1 Anonymity

Snowflake is an enabling technology for accessing the Tor anonymity network. Even though our focus is on Snowflake as a censorship circumvention system, we have to take into consideration the ethical questions that arise with offering online anonymity. The Tor anonymity network might conjure different associations. For some, it might be a technology used to host illicit gun and malware markets, child sexual abuse material (CSAM) and a place for extremist communities. For others, it is a technology that facilitates freedom of expression without discrimination and is a critical tool for protecting the identities of activists, journalists and whistle-blowers, like Edward Snowden [16].

Nurmi *et al.* [17] have investigated CSAM availability, searches and its users on the Tor network. In their paper, they critique top computer science venues for not

[7] https://meetbot.debian.net/tor-meeting/2024/
[8] https://ooni.org/
[9] https://censoredplanet.org/
[10] https://github.com/net4people/bbs

enforcing their own ethical guidelines as a lot of research on anonymous networks are accepted without mentioning the potential harms. They crawled onion websites (services only on the Tor network with hidden IP addresses and location) over a five-year period, and analyze 176.683 domains to find that one-fifth host CSAM (by using text-based detection). Three prominent Tor search engines worked with the researchers to display a questionnaire when their users searched for CSAM keywords. Analyzing the answers from the intervention, they found that about half of the users want to stop using such content and around two-thirds of those who are seeking help have not received it. They conclude that there is an urgency to deploy public health programs for CSAM users, where anonymous online therapy hosted on Tor is actively sought and is having promising results.

The Tor Project has a official 'FAQ' page, which addresses conerns of abuse[11]:

"*Tor's mission is to advance human rights with free and open-source technology, empowering users to defend against mass surveillance and internet censorship. [..] we condemn the misuse and exploitation of our technology for criminal activity. It's essential to understand that criminal intent lies with the individuals and not the tools they use. Just like other widely available technology, Tor can be used by individuals with criminal intent. And because of other options they can use it seems unlikely that taking Tor away from the world will stop them from engaging in criminal activity. [..] Our refusal to build backdoors and censorship into Tor is not because of a lack of concern. We refuse to weaken Tor because it would harm efforts to combat child abuse and human trafficking in the physical world, while removing safe spaces for victims online. Meanwhile, criminals would still have access to botnets, stolen phones, hacked hosting accounts, the postal system, couriers, corrupt officials, and whatever technology emerges to trade content.*"

Offering total anonymity can be seen as removing accountability for persons doing harmful actions online. The ethical discussion around non-accountability goes back to Plato's Republic with the Ring of Gyges [18, 19]. In the Republic, the wearer of the mythical ring gains power to become invisible at will. Plato uses the story to explore the idea that people act justly only because they fear the consequences of being caught and punished, suggesting that if they could act without being seen (anonymous), they might commit unjust acts freely.

R. Bodle gives an ethical justification for protecting online anonymity in "The ethics of online anonymity or Zuckerberg Vs. 'Moot' " [19]. He argues that focusing only on the harms of anonymity is short-sighted and obscures the benefits. The approach taken by Bodle, is a method that integrates various ethical and meta-ethical theories. This pluralistic method combines Utilitarianism and social utility with

[11] https://support.torproject.org/abuse/

Kantian principles, like the rights-based view. Bodle suggests that anonymity should be recognized as an instrumental good, essential for enabling other rights such as free speech and privacy, and thus should be protected.

Acknowledging that there are unwanted effects of offering total anonymity online, and that there is a need for discussion and research (which falls out of the scope of this thesis), we take the stance of anonymity being an attribute to strive for in the Internet. A democratic society requires anonymity to mobilize participation of all groups, including marginalized and vulnerable populations, political dissidents, whistle-blowers and citizens who wish not to be under surveillance [19].

1.4.2 Censorship circumvention

For ethical questions regarding censorship circumvention, Corrigan-Gibbs *et al.* [20] draw lessons from humanitarian aid, to help avoid repeating the same mistakes of failed humanitarian interventions. In this section we echo these lessons and relate them to Tor, Snowflake and this thesis.

- "*Impartiality is impossible. [..] Internet freedom organizations that provide tools and trainings to activists, bloggers, and civil society activists must appreciate that who they are teaching is as important as what they are teaching*". We are seeing non-profit and non-governmental organizations with experience in humanitarian aid adopting technologies from the Tor project, such as Amnesty International[12]. Our stance is to trust humanitarian organizations to teach these technolgies to people living under extreme conflict and violent oppression, but be opportunistic with teaching people in more democratic areas (and support any and all efforts at this).
- "*It's hard to consent when you don't understand*". The peer-to-peer nature of Snowflake proxies allows the user and proxy to know each other's IP addresses. So a censor could, in theory, run a campaign hosting many proxies and collect the IP addresses of people accessing Tor, trying to find users inside their networks. A censor would not know exactly what they are doing, but the act of accessing the free Internet might by itself be a punishable offense. People might not be aware of this, and users should be educated about this danger, so to not feel a false sense of security.
- "*Our tools promote our cultural norms*". There are different interpretations of the UDHR, and Internet freedom technologies reflect our own set of cultural values. Corrigan-Gibbs *et al.* conclude that it is not unethical to promote value-laden human-rights, but encourage transparency of the values of the

[12] https://securitylab.amnesty.org/amnesty-on-tor/

researchers of these technologies, we give our stance in the final paragraph of section 1.4.1.

- "*Impact is elusive*". Tor is collecting metrics on how many users from a given location using the different pluggable transports per day[13]. However, we do not know if those users actually feel safe and free to speak their mind.

Corrigan-Gibbs *et al.* state: "*Defending human rights does not automatically put us above ethical censure.*" Indeed, with this section we tried to confront and acknowledge ethical issues, and we encourage researchers in the field to do the same.

[13]https://metrics.torproject.org/

Background

In this chapter we present relevant background and split it in two main parts. Firstly, preliminary knowledge on enabling and related technologies to Snowflake and network protocol fingerprinting is provided. Secondly, state-of-the-art research is described for the field of network traffic fingerprinting, with the main focus on Snowflake and DTLS.

2.1 Preliminaries

This section presents technologies, definitions and terms central to this thesis. It gives a top-down view of how the concepts fit together, starting with censorship circumvention as a general topic, continuing to explain Snowflake, and its enabling protocols, WebRTC and DTLS. Finally, fingerprinting and fingerprint resistance approaches are described.

2.1.1 Censorship and circumvention

A censor aims to perform two actions, detection and blocking [21]. Detection involves classifying traffic to determine which should be permitted or denied. After detecting prohibited communication, the censor blocks it by, for example, injecting TCP RST packets [3]. Effective censorship requires both accurate detection and low-cost blocking - without blocking, it would only be surveillance. Errors can occur, leading to false negatives (missed blocks) or false positives (incorrect blocks). Censorship circumvention uses obfuscation techniques to complicate the censor's classification task, blending in with benign traffic, increasing its error rate and operational costs in computation, time and money.

Collateral damage happens when censorship blocks necessary or desirable resources, like critical websites or tools, causing civil discontent and reduced productivity. This forces the censor to balance blocking undesirable communications against avoiding collateral damage. The main goal of effective circumvention is to raise the

censor's error rate, making blocking too expensive and unacceptable. How sensitive a censor is to collateral damage varies based on the censor's resources, motivations, and the importance of the blocked content to the affected population.

Censors typically employ either blocklists or acceptlists approaches to control internet access. The blocklist approach blocks specific sites, addresses, or content deemed undesirable, allowing all other traffic. This method is more common [3] because it is less disruptive and easier to implement, but it requires constant updates to remain effective against new circumvention techniques. On the other hand, the acceptlist approach permits only pre-approved sites and services, blocking everything else. While this method offers tighter control and reduces the need for continuous updates, it is highly restrictive and can significantly disrupt legitimate internet use, leading to greater collateral damage.

2.1.2 Snowflake

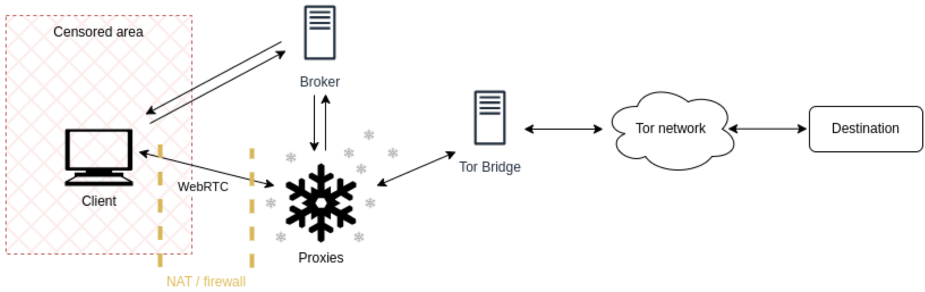


Figure 2.1: Architecture of Snowflake [1]

The Snowflake circumvention system involves three key participants: the client, proxies, and a broker. The architecture of Snowflake, depicted in Figure 2.1, highlights its main components and communication channels. The client is a user operating Snowflake within a region where a censor is blocking traffic to specific destinations (IP addresses). To bypass these IP blockages, the client routes its traffic through one of the proxies using an encrypted WebRTC data channel. These proxies are operated by volunteers with unblocked IPs that have access to the open internet. The client contacts the broker to locate an available proxy in a process called rendezvous.

To initiate contact with the broker, the client must use an indirect, unblockable channel to bootstrap into Snowflake. There are three supported methods for

rendezvous: domain fronting[1], Accelerated Mobile Pages cache and Simple Queue Service [11]. Once the indirect channel is established, the client communicates with the broker, which matches the client with an idle proxy from its pool, based on self-reported Network Address Translation (NAT) types. The broker then facilitates the exchange of Session Description Protocol (SDP) [22] offers and answers between the client and proxy, as specified by WebRTC.

Following rendezvous, the client and proxy must navigate NAT traversal during the connection establishment phase. Devices behind NATs and firewalls typically only allow outgoing connections initiated by the client. To address this, WebRTC employs the Interactive Connectivity Establishment (ICE) [23] procedure, which opens direct communication through NATs and firewalls. The ability of a client and proxy to establish a connection is determined by their NAT types, using Session Traversal Utilities for NAT (STUN) [24] and Traversal Using Relays around NAT (TURN) [25] servers within the ICE procedure. Upon successfully establishing a connection, the client and proxy can exchange traffic.

The final phase of Snowflake involves data transfer, which includes a persistent session layer and an ephemeral data channel. A persistent session is maintained using Turbo Tunnel [26], which adds sequence numbers and acknowledgments to data exchanged between the client and a bridge. This ensures that if the current proxy becomes unavailable, the data will be retransmitted through a new proxy. For the ephemeral channel, WebRTC data channels are used, allowing for the transmission of encrypted and integrity-protected data via DTLS.

2.1.3 Datagram Transport Layer Security (DTLS)

DTLS is a protocol designed by the Internet Engineering Task Force (IETF) to provide secure communication for datagram-based applications, similar to how Transport Layer Security (TLS) secures applications over TCP. While DTLS is based on TLS, their key differences are due to the nature of the underlying transport protocols. As a consequence of operating over UDP, DTLS has additional mechanisms to handle packet loss, reordering, duplication and fragmentation. The protocol is widely used in applications where low-latency and real-time communication are critical, such as Voice-over-IP, Internet of Things (IoT) and Virtual Private Networks (VPN) [27].

DTLS comprises two primary components: the handshake and the record layer. The DTLS handshake is responsible for negotiating cryptographic algorithms and keys between the client and server. Once these parameters are established, the DTLS

[1] It is worth noting that the anti-censorship team have to constantly update the content providers they use for domain fronting, as many of the providers are stopping their support of the service [11].

record layer takes over, encapsulating the data from the upper layers into encrypted records that are transmitted over UDP.

Figure 2.2: Message flights for the full DTLS 1.2 handshake [28]. Optional messages and flights are indicated with an asterisk.

DTLS has undergone several iterations to enhance security and performance. DTLS 1.0, specified in RFC 4347 [29] and released in 2006, provided the foundational security features adapted from TLS 1.1 for use over UDP, including basic handshake mechanisms and the use of cookies to prevent denial-of-service attacks. However, it lacked support for newer cryptographic algorithms and was vulnerable to some attacks discovered later.

DTLS 1.2, introduced in RFC 6347 [28] in 2012, built upon TLS 1.2. It incorporated stronger cryptographic algorithms like SHA-256, AEAD cipher suites for better security and performance, and enhanced negotiation of cipher suites and extensions, addressing some vulnerabilities of its predecessor. DTLS 1.2 is the most common version deployed on the internet [27].

The most recent version is DTLS 1.3, based on TLS 1.3. It features a streamlined handshake that reduces the number of round-trips, lowering latency. This version also provides default forward secrecy and elimination of outdated cryptographic algorithms. DTLS 1.3 became a standard in 2022 with RFC 9147 [30], but has seen little adoption.

Figure 2.2 shows the full DTLS 1.2 handshake. This process involves multiple "flights" of messages exchanged between the two parties to negotiate security parameters, authenticate each other, and establish shared secrets for encrypted communication.

- **Flight 1**: The handshake begins with the client sending a *ClientHello* message to the server, containing the protocol version, a randomly generated number (*ClientRandom*), *SessionID*, supported cipher suites, compression methods, and any relevant extensions such as Application-Layer Protocol Negotiation (ALPN) and Supported Groups.
- **Flight 2 (optional)**: To prevent denial-of-service attacks from spoofed IP addresses, the server may respond with a *HelloVerifyRequest* message. It includes a stateless cookie created as a HMAC of a secret, the client parameters and IP. The client must echo back in a subsequent *ClientHello* message. This step is optional but recommended to verify the client's reachability and mitigate resource exhaustion risks.
- **Flight 3 (optional)**: Following the receipt of the optional *HelloVerifyRequest*, the client resends the *ClientHello* message, this time including the server-provided cookie. This resubmission proves the client is reachable at the IP address from which the initial *ClientHello* was sent.
- **Flight 4**: Upon receiving the *ClientHello*, the server responds with a sequence of messages: *ServerHello*, *Certificate*, *ServerKeyExchange*, *CertificateRequest*, and *ServerHelloDone*. The *ServerHello* message includes the server's chosen protocol version, a randomly generated number (*ServerRandom*), *SessionID*, chosen cipher suite, compression method, and any relevant extensions. The *Certificate* message provides the server's certificate chain, proving its identity. If the selected cipher suite requires it, the *ServerKeyExchange* message contains the server's public key information necessary for key exchange. The *CertificateRequest* message, which is optional depending on the server's security policy, requests the client's certificate for mutual authentication and provides a list of supported signature algorithms. The *ServerHelloDone* message indicates the end of the server's initial handshake messages.
- **Flight 5**: The client then responds with its own sequence of messages: *Certificate*, *ClientKeyExchange*, *CertificateVerify*, *ChangeCipherSpec*, and *Finished*.

The *Certificate* message is sent if the server requested the client's certificate. The *ClientKeyExchange* message contains the client's public key information or pre-master secret, depending on the key exchange method. The *CertificateVerify* message provides a signature over previous handshake messages using the client's private key, proving ownership of the certificate. The *ChangeCipherSpec* message indicates that the client will switch to the newly negotiated cipher suite and keys for subsequent messages. The *Finished* message, which is a hash of the entire handshake up to this point and encrypted with the new session keys, allows the server to verify that the handshake was not tampered with.

- **Flight 6**: The handshake concludes with the server sending its own *ChangeCipherSpec* and *Finished* messages. The *ChangeCipherSpec* message signifies that the server will also switch to the newly negotiated cipher suite and keys. The *Finished* message, similar to the client's, is a hash of the entire handshake encrypted with the new session keys, enabling the client to verify the integrity of the handshake. The client and server can now send encrypted records to each other.

To provide extra flexibility, DTLS utilizes extensions in the *ClientHello* and *ServerHello* messages. The extension field can be of variable size, with a maximum size of 2 bytes, allowing different amounts of extensions to be in any order. This is a way to negotiate additional features without altering the core protocol. For example, the ALPN extension allows clients and servers to indicate the application protocol to be used over a secure connection during the handshake process. This ensures that both parties settle on a specific protocol (such as HTTP/2 or HTTP/3) before any data is transmitted. The *Supported Groups* extension provides a list of curve groups so the client and server can agree on a group for the Elliptic-curve Diffie-Hellman key exchange, ensuring compatibility and security. These extensions are specified in various RFCs, often for both TLS and DTLS.

The Encrypted Client Hello (ECH) is an extension currently in draft (draft-ietf-tls-esni-18[2]) for TLS/DTLS that aims to enhance privacy and security by encrypting the *ClientHello* message. This will protect privacy-sensitive information in other extensions.

2.1.4 WebRTC

WebRTC enables peer-to-peer audio, video, and data sharing directly between web browsers and mobile applications, making it ideal for applications like video conferencing and live streaming. The protocol is being maintained by a working

[2]https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-18

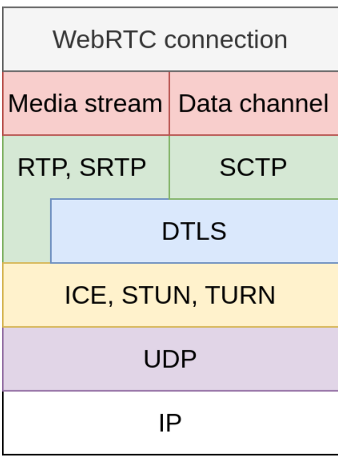


Figure 2.3: The WebRTC protocol stack

group at the Internet Engineering Task Force (IETF)[3] and is documented as a W3C recommendation[4]. WebRTC stitches together many existing single-purpose technologies to make a suite of protocols - see Figure 2.3 for how the protocol stack is layered. This subsection outlines WebRTC's core components: signaling, connecting, securing, and communicating.

- **Signaling**: WebRTC signaling involves the exchange of SDP messages, which convey information about media capabilities and network configurations needed to establish a connection. This process includes the exchange of offers and answers between peers to negotiate session parameters and the transmission of ICE candidates to facilitate network address discovery.

- **Connecting**: To establish a peer-to-peer connection, WebRTC employs the ICE framework for NAT traversal. STUN servers help discover public IP addresses and port mappings, while TURN servers relay traffic when direct peer-to-peer connections are not possible. This combination ensures that peers can connect despite the presence of NAT devices and firewalls.

- **Securing**: WebRTC uses DTLS to secure communications, encrypting data to prevent eavesdropping and tampering. DTLS with Secure Real-Time Transport Protocol (SRTP) is used for encrypting media streams. The SRTP keys are negotiated during the DTLS handshake. Additionally, DTLS supports certificate-based authentication, enabling peers to verify each other's identities.

[3] https://datatracker.ietf.org/wg/rtcweb/documents/
[4] https://www.w3.org/TR/webrtc/

- **Communicating**: WebRTC supports two primary types of communication: media streams and data channels. Media streams, which include audio and video, use the Real-time Transport Protocol (RTP) over SRTP to ensure secure transmission, prioritizing quality and low latency for real-time communication. Data channels, on the other hand, facilitate the transmission of arbitrary data using SCTP over DTLS, offering reliability and flexibility for applications like file sharing and text chat.

2.1.5 Fingerprinting

Network protocol fingerprinting is the process of identifying and classifying network protocols based on their unique characteristics or patterns, similar to how fingerprints uniquely identify individuals. These protocol-specific patterns enable the detection and analysis of the communication protocols used within a network.

Guoqiang Shu and David Lee introduced a formal methodology for network protocol fingerprinting, outlining a taxonomy that addresses the challenges of fingerprinting through three main components: active and passive experiments, fingerprint discovery, and fingerprint matching [31].

Active fingerprinting involves engaging with the target system by sending specific probes or queries and analyzing the responses to gather information about the protocol and its implementation. While this method can be intrusive and may cause some disruption to the target system, it is highly effective in extracting detailed protocol information. Techniques commonly used in active fingerprinting include delaying, dropping, modifying, or injecting packets into existing connections. An example of a tool that employs active fingerprinting is *nmap*[5], a widely used network scanning utility.

On the other hand, passive fingerprinting relies on observing and analyzing network traffic patterns without direct interaction with the target system. This approach is less intrusive and does not risk disrupting the target system. However, it may be less accurate than active fingerprinting. Passive fingerprinting utilizes deep packet inspection (DPI) to examine protocol fields or analyze statistical traffic patterns.

Fingerprint discovery involves systematically uncovering a fingerprint for an unknown implementation. This process gathers comprehensive information to create a unique identifier for the protocol. Discovery can be conducted with explicit guidance from the protocol specification or implementation source code (white-box) or without such guidance (black-box).

[5]https://nmap.org/

Fingerprint matching is the process of comparing collected fingerprints to determine if they originate from the same protocol implementation. This can be done through exact one-to-one mapping or probabilistically, assessing the likelihood that two fingerprints correspond to the same implementation.

2.1.6 Network traffic obfuscation

There are two main obfuscation techniques used in practice to combat fingerprinting: mimicking and randomization [21].

Mimicking, also known as mimicry or steganography, aims to replicate the behavior of a protocol. The goal is to make it challenging to distinguish between the genuine protocol and the obfuscating protocol. Houmansar et al. [32] argue that mimicking application layer protocols is particularly challenging and fundamentally flawed, a criticism summarized by the phrase “The parrot is dead”. Perfect mimicry, using a protocol as intended, is often referred to tunneling.

Randomization, often referred to as polymorphism, involves implementing random protocol features to make the traffic appear dissimilar to any protocol or pattern that a censor might block. The objective is to eliminate all statistical characteristics, causing the traffic to resemble “junk” data. However, this approach can be ineffective if the censor employs whitelist blocking, as the traffic would not match any approved protocols. *obfs4*[6] is an example of a pluggable transport that employs randomization for censorship circumvention.

2.2 State of the art

This section presents the state of the art on fingerprinting Snowflake and related network protocols.

2.2.1 Detecting Snowflake

David Fifield and Mia Gil Epner [33] are the first publicly to explore ways of fingerprinting parts of the Snowflake system in their paper “Fingerprintability of WebRTC”. The authors conducted a manual analysis of different WebRTC applications to identify features that could be used to fingerprint them. They examined the traffic of Google Hangouts, Facebook Messenger, OpenTokRTC, Sharefest, and Snowflake. Their findings revealed significant fingerprinting potentials in the DTLS and STUN/TURN protocols used by WebRTC, including differences in cipher suites, extensions, and certificate details. To answer how much WebRTC traffic exists in the wild, they deployed a fingerprinting script for DTLS on a day’s worth of network

[6] https://gitlab.com/yawning/obfs4

traffic from Lawrence Berkeley National Laboratory. Running this experiment, they found only a handful of fingerprints and concluded that WebRTC traffic is relatively scarce in real-world traces. Future work suggested enhancing their fingerprinting script, expanding traffic analysis to more datasets, and developing automated tools to analyze STUN and TURN protocols further.

The work of MacMillan *et al.* [13] at Princeton University is the most prominent work on evaluating the indistinguishability of Snowflake by fingerprinting DTLS handshakes. They collected the largest data set to date (which is publicly available[7]) with 6,500 handshakes of different WebRTC based applications. From the *ClientHello* and *ServerHello* handshake messages, they extracted the length, message sequence, fragment offset, DTLS version, SID length, cookie length, cipher suite length, cipher suites, extension length, extensions and chosen cipher fields. To find fingerprints they used one-hot-encoding to transform non-numeric header fields into binary features and performed classification with the random forest machine learning algorithm. They found multiple ways of fingerprinting Snowflake: sending the optional *HelloVerifyRequest* message, offering the `supported_groups` extension in the *ServerHello* message, and not offering the `renegotiation_info` in the *ServerHello* message. Although their data set is publicly available, their classification software is not. How they collected the data set is unclear and is never explained in their paper.

Chen *et al.* [34] extend the work from MacMillan *et al.* by using more machine learning algorithms to perform fingerprinting of DTLS handshakes. They extracted similar features as the previous work. For their data set, they generated traffic from automated scripts and combined it with data set from MacMillan *et al.* After evaluating the different algorithms they claim an average accuracy of 99.8%, only requiring a few hundred handshakes. The random forest algorithm performed best again, and the chosen cipher suite, fragment length and cipher suites fields ranked the highest in order of feature importance. They also performed pre-identification by Domain Name Server (DNS) requests to known STUN and domain-fronting servers. Their artifact, "F-ACCUMUL", and data set are not publicly available.

Wang *et al.* [35] use statistical properties of the traffic pattern of handshakes to perform fingerprinting of DTLS traffic in Snowflake. This is a different approach than field features in the handshake as seen in previous research. Their two main features of importance, with a clear margin, are total time between packets and packet length mean. A Docker image of their setup is available in dockerhub as `xinbigworld/ubuntu:1.2`[8]

Xie *et al.* [36] also detected Snowflake using statistical properties of the traffic

[7] https://github.com/kyle-macmillan/snowflake_fingerprintability

[8] https : / / hub.docker.com / layers / xinbigworld / ubuntu / 1.2 / images / sha256 - 3213fded0606c0e59b4b31845910a020d2a340c9dc4e810c2e7381ed02d3b22e

patterns, but of HTTPS during the rendezvous phase. With this approach they were able to block Snowflake traffic before being assigned an ephemeral proxy. They trained a decision tree model using packet sizes, direction, time and network speed feaures with a data set they created. Their code is available publicly at GitHub[9], and their data set is hosted on Baidu Netdisk[10].

Holland *et al.* [37] in "New Directions in Automated Traffic Analysis" use automated machine learning to do general traffic analysis tasks. One of their use cases was applying their tool to identify DTLS applications with the data set from MacMillan *et al.* They claim that nPrintML can almost perfectly identify the browser and application pair that generated the handshake, without the need of manual feature engineering. Figure 2.4 shows a heatmap of automatic detected features, with total length being the most important. Their tool is publicly available on GitHub[11].

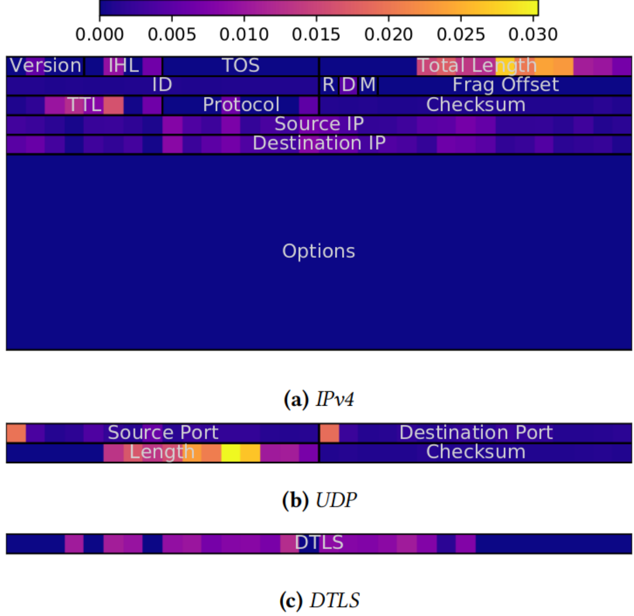


Figure 2.4: Per-bit feature importance for identification of browser and application pair from MacMillan *et al.* data set, from Holland *et al.* [37]

[9] https://github.com/Xanole/SnowDT
[10] https://pan.baidu.com
[11] https://github.com/nprint/nprintml

2.2.2 Fingerprinting TLS

Although Snowflake uses DTLS and not TLS, the protocols are so similar that it is worth exploring the realm of TLS fingerprinting, and existing mitigation techniques.

Sergey Frolov and Eric Wustrow developed the fingerprint-resitant uTLS library [15]. As part of the process they collected and analyzed TLS traffic to create fingerprints. They extracted fields from the *ClientHello* and *ServerHello* messages, combing and hashing them to create an unique fingerprint. In addition to the hashes, they used the Levenshtein distance between the extracted fields to group fingerprints. Their library employs multiple techinques for obfuscating traffic: low-level access to the handshake, randomized *ClientHello* fingerprint, mimicking *ClientHello* messages of other implementations and use of multiple fingerprints.

In the Master thesis of Erwin Janssen [38], different TLS implementations were fingerprinted using model learning. Janssen employed a three-step process: *build*, *learn*, and *identify*. In the *build* step, different DTLS implementations were compiled, packaged and published in an automated pipeline. The *learn* step used `LearnLib`[12] to infer state machines that describe the behavior of the built DTLS implementations. The *identify* step applies these learned models to match fingerprints as a state identification problem. While the model learning approach could distinguish between different TLS implementations, the number of unique models produced was relatively small, leading to overlaps where different implementations exhibited similar behavior. Janssen suggests that expanding the learning alphabet, to include more message types and variations, incorporating invalid message and employing various fuzzing techniques could improve the uniqueness of the models. All of Janssen's source code is publicly available on GitHub under the `tlsprint`[13] organization.

[12] https://github.com/LearnLib/learnlib
[13] https://github.com/tlsprint

Chapter 3 Method

This chapter outlines the overall methodology of the thesis. Some assumptions are established, along with defining what is in scope. We then explain the artifacts to be developed, how they should be validated and discuss different approaches.

3.1 Methodology

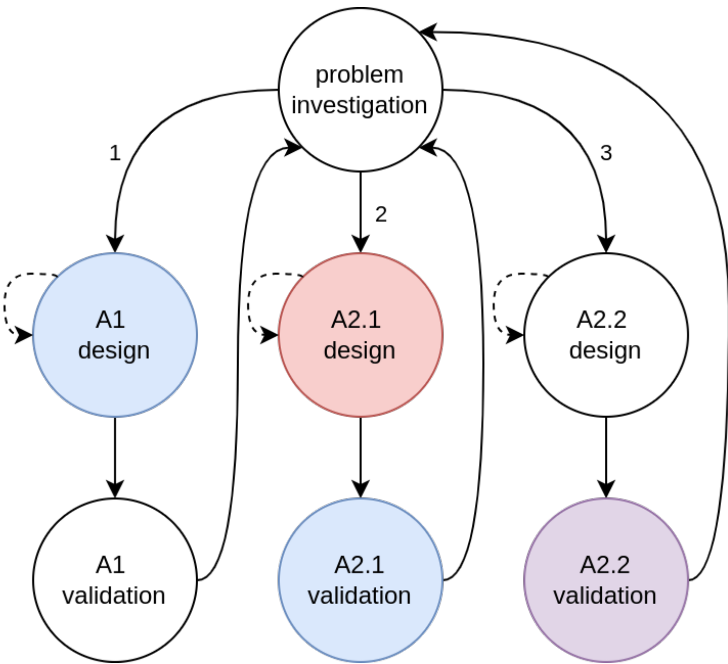


Figure 3.1: The design cycle for this thesis. Color-coding shows how **A1** and **A2.1** are used in later validation steps.

We adopt the design science methodology [39], which is a structured approach in the form of the design cycle, to research information systems and software engineering. The design cycle begins with the problem investigation, where we analyze and understand the problem, including its context and underlying issues. This is followed by treatment design, where potential software solutions, called artifacts, are developed based on the insights gained from the investigation. Finally, treatment validation involves testing and investigating the effect of the artifact prototype in a model of the problem context to ensure it effectively addresses the problem. These tasks form a cycle, because we iterate back to the problem investigation, starting the process again, with new knowledge gained from the previous treatment design and validation.

Figure 3.1 shows the design cycle for this thesis, outlining the main cycles and how they interconnect. The inital problem investigation was done in the previous two chapters, where we looked at problems (motivation and ethics) and context (communities and background). To answer the research questions for this thesis, we design and validate three artifacts:

A1: DTLS fingerprint discovery tool
The aim of this artifact is to simulate a censor discovering distinguishing DTLS fingerprints of Snowflake. Validation is done by using the tool to find fingerprints in the MacMillan *et al.* [13] data set. The results from this cycle will answer **RQ1**.

A2.1: DTLS handshake generation setup
This artifact will automatically generate fresh DTLS handshakes, to be used for mimicking DTLS connections and as a data set for further research. Validation includes comparing the handshakes to manually generating handshakes, testing for consistency and using **A1** to compare the automatically generated handshakes against the MacMillan *et al.* data set. This artifact will be combined with **A2.2** to answer **RQ2**.

A2.2: Fingerprint-resistant DTLS library
The final artifact is a Go module that extends the Pion DTLS library to offer fingerprint resistance features (inspired by *uTLS*), and aims to answer **RQ2**. It will use the fingerprints generated by **A2.1** for mimicking purposes. **A1** will be used to validate that the library does not produce any distinguishable fingerprints, comparing it to a baseline of fresh Snowflake traffic.

To benefit other researchers working on the area, ensure reproducibility and scientific transparency, we will release all software artifacts built in this thesis publicly under open-source licenses. However, some of the manual packet captures are withheld to make sure there are no privacy compromises.

3.2 Scope and assumptions

We use this section to further scope and define the context.

3.2.1 Validation vs. evaluation

In this thesis, we will only do validation. The goal is to predict how artifacts will interact with our context, simulated in a constructed environment. Evaluation, on the other hand, is to deploy the artifacts in a real-world context, with real users, running over time. In our case, this would require deploying the fingerprint-resistant library with Snowflake in a real-world censored area and do empirical analysis of deployment. We consider such an attempt to be beyond our scope, as it requires a careful planning (protecting privacy of users, infrastructure) to yield proper results and would be a whole thesis on its own.

The aim of this thesis is not to validate how cost-effective discovered fingerprints are, only that fingerprints exist. Therefore using the tool (**A1**) to do fingerprint matching and setting up an environment to block real-time traffic as a censor also falls outside the scope. We assume that the simple regex fingerprints it produces will be the most efficient and preferred way for a censor to do blocking. However, demonstrating and evaluating it is not a goal.

3.2.2 Fingerprinting scope

For fingerprinting, we consider only the DTLS handshake to be in scope, not the encrypted record layer. We will also not explore traffic pattern (flow) analysis (e.g. timings, packet size, speed) such as Wang *et al.* [40] and Xie *et al.* [36] (we suspect they did not use field features as they analyzed HTTPS which already used *uTLS*). Even though they claim promising results, we believe it is difficult to know if the statistical properties are of the DTLS implementation or the network itself. Such approaches require highly-controlled environments to not fingerprint the underlying network. Bocovich *et al.* [11] also warn against this, as traffic analysis attacks have historically been overestimated due to un-realistic base rates [41].

3.2.3 Censor's capabilities

Tschantz et al. [3] did a study in 2016 to ground the evaluation of circumvention approaches in empirical observations of real censors. They found that censors

prefer simple cost-effective solutions, with mostly passive monitoring (e.g. DPI) and some active probing. They suggest that censorship circumventors should concern themselves more with low-cost exploits. We assume that a censor will prefer simple, stateless and deterministic solutions to perform detection and blocking. This will effect how we design our artifacts, and we will focus on removing low-hanging fruits before considering more advanced attacks.

We further assume that a censor prefers passive fingerprinting over active probing. The Great Firewall of China have been deploying active probing for *Shadowsocks* [42, 43] and *obfs3* [44]. This is probably due to it not being possible to perform passive field based fingerprinting, because of the randomized nature of the protocols. Active fingerprints are hard to discover, because you have to find a bug or side-channel of the protocol. A censor would also have to deploy infrastructure to send the fingerprint payloads. There have not been any active attacks on Snowflake, as far as we know.

3.3 Exploratory work

This section presents different design approaches that we explored, but did not implement, for each of the artifacts described in Section 3.1. These are represented by the dotted paths in Figure 3.1. We link the approaches to our scope, assumptions and the feasibility of implementing them.

3.3.1 Fingerprinting tool

Active fingerprinting

Janssen [38] and Rasoamanana *et al.* [45] have both shown that state machine inference (model learning) can be used to fingerprint TLS implementations. We could imagine a censor running an active model learning algorithm like Angluin's L^* [46] offline against various DTLS implementations. Using the inferred state machines from the learning, a censor could isolate the differentiating state as a probe. Fitera *et al.* [47] used protocol state fuzzing and state machine learning for finding security vulnerabilities in various DTLS implementations (and open-sourcing their tool, *dtls-fuzzer*[1]). Interestingly, they found that from *pion/dtls* they could infer a state machine model of 66 states, the most of any of the other implementations. This suggests that there might be possibilities of finding a probable state and develop it as an active fingerprint if they are available. Even though this approach is very promising, we chose not continue to explore it as we assume a censor would prefer passive fingerprints. We prioritized our work towards exploring cheaper attacks and mitigating them.

[1] https://github.com/assist-project/dtls-fuzzer

3.3.2 Handshake generation

Capturing real traffic

The most obvious way of making a data set of DTLS traffic would be to capture real-world traffic, perhaps on the campus network. However, there are major drawbacks to this approach. It is first and foremost a complex undertaking to do in a privacy preserving manner [41]. Secondly, captured traffic will at some point become stale. A collected data set is only valid for mimicking purposes shortly after the period it is captured. Capturing real traffic will offer more diversity, but implementations change and we need fresh handshakes. The trade-off is between accuracy (real handshakes) and being future-proof (synthetically generated handshake). We decided to not perform captures of real-world traffic.

3.3.3 DTLS library

Tunneling in browsers

Snowflake uses protocol tunneling, that is, using WebRTC as intended. To hinder fingerprinting, we could use a browser implementation of the WebRTC and DTLS stack, and it would be implementation-dependent tunneling. This approach is tempting as Snowflake would produce a real browser fingerprint, an idea discussed by the anti-censorship team [2]. There are two ways one could use a browser implementation, the first is using the Tor browser itself, the second is by bringing your own browser (BYOB).

There are two main challenges with using the Tor browser approach. The most prominent is the lack of support of for WebRTC. It used to have it disabled, since UDP was not supported by Tor[3]. The browser dev team is working on it, but the issues for WebRTC bugs are still open on GitLab[4]. With our own testing, we did not find WebRTC to be working. Even if Tor's WebRTC implementation would work, experience with the *meek* pluggable transport[5] has shown that this approach requires constant maintenance. Even though the browser-team aims to rebase to the lastest versions of Firefox ESR quickly, their Tor browser patches might break, take time to fix and be behind the Firefox user-base.

With BYOB, Snowflake would look for a browser on your system and use that for WebRTC. We believe this is implementable, but would require a very high amount of work and maintenance. To migrate to such an approach, Snowflake looses the

[2] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/40014#note_2823772

[3] https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/41021

[4] https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/41486#note_2886170

[5] Issue #13442, #15512, #18927, #22515, https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/meek/-/issues/

cross-platform benefits of running pure Go. The team would have to support different ways of hooking into the WebRTC libraries of browsers on desktop, Android and iOS separately.

Chapter 4
DTLS fingerprint discovery

In this chapter the first artifact (**A1**) for this thesis is described. The artifact is a setup for simulating a censor discovering fingerprints of DTLS traffic to identify Snowflake usage.

4.1 Architecture and implementation

To address RQ1, we want to analyze captured DTLS traffic of different implementations and find fingerprints. We recreated a setup similar to the one presented in the work of Frolov and Wustrow [15]. Their tool was not released, and even if it had been, adaptation from TLS to DTLS would have been necessary. We developed a tool called *dfind*[1], a Go program designed to discover passive fingerprints by identifying fields specific to different DTLS implementations. These fingerprints could be represented as a regular expression pattern and used to block Snowflake in real-time with DPI.

Figure 4.1 provides an overview of the fingerprinting setup we have implemented. Captured DTLS traffic is read from *pcap* files, handshakes are parsed and relevant features (fields) are extracted. The traffic is from known implementations and the extracted features are stored in a database along with a type tag indicating the specific implementation. Finally, the database is queried to find unique fields that form a fingerprint, and fuzzy matching techniques are used to find extensions to manually analyze for fingerprints.

To parse the *pcap* files we explored a few options. Fifield et al. wrote a *bro/zeek* script[2] to parse captured traffic and output a fingerprint for research on "Fingerprintability of WebRTC" [33]. The initial exploration of the 8-year-old script was not successful, as the script was written for *bro* and would have to be migrated to the newer version of the system called *zeek*. Doing this migration introduced too much overhead for using this approach. *tshark*, the command line version of Wireshark,

[1] https://github.com/theodorsm/dfind
[2] https://github.com/miagilepner/dtls-fingerprint

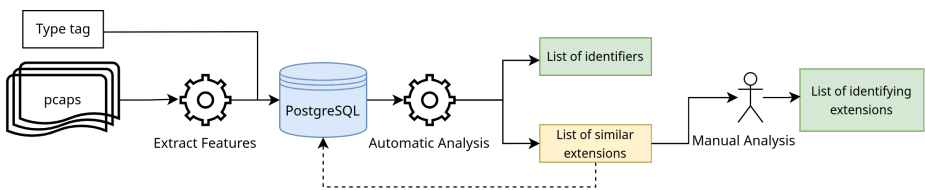


Figure 4.1: *dfind* architecture overview

has a DTLS dissector. Unfortunately, its JSON-formatted output is not mature yet. When we exported DTLS traffic as JSON, lists were not parsed correctly and the ciphersuites entry only contained a single cipher of 8 in total. There is an active issue on the project's GitLab with various related JSON-problems[3]. For a dissector written in Go, we could use the Pion DTLS library, but we do not want to use a library that we are testing with the tool itself.

We chose to use the `gopacket/gopacket`[4] library to read *pcap* files and wrote a simple DTLS handshake dissector, which was helpful to better understand the DTLS protocol structure. Table 4.1 shows the fields supported by our dissector.

For the DTLS handshake messages, only the *ClientHello* and *ServerHello* are parsed. These messages are also the most commonly used in state-of-the-art research. The fields in the Hello messages are most relevant for fingerprints, as the messages contain lists or extensions. Both lists and extensions can be of any order or length, and is implementation specific depending on what features are supported.

In *dfind*, parsed fields and its corresponding implementation type is stored as a fingerprint table in a PostgreSQL database. To discover fingerprints, the database is queried as part of an automatic analysis step with two routines. One routine finds unique values of fields which are identifying for a certain implementation type, the other finds similar hex-strings in the extensions of each type, so that they can be further manually analyzed. The SQL-queries used in the routines can be found in Appendix B.

The unique field routine consists of two SQL-queries and is described in Algorithm 4.1. This routine is used with every field marked in the 'Analysis' column in Table 4.1. It is worth noting that only the complete list extensions as a hex-string is considered during analysis, not individual extensions.

[3]https://gitlab.com/wireshark/wireshark/-/issues/17125
[4]https://github.com/gopacket/gopacket

Fields	ClientHello	ServerHello	Analysis
Handshake type	✓	✓	
Length	✓	✓	✓
Fragment offset	✓	✓	
Major version	✓	✓	
Minor version	✓	✓	
Cookie length	✓	✓	✓
Cipher length	✓		✓
Ciphers	✓		✓
Chosen cipher		✓	✓
Extension length	✓	✓	✓
Extensions	✓	✓	✓

Table 4.1: Fields parsed by the dissector. The *ClientHello* and *ServerHello* columns indicate if the field is present in that message. The analysis column shows which fields are used in an automatic analysis

Algorithm 4.1 Finding unique values of given field and type of implementation

We let t denote the type to find the fingerprint of, for example: *snowflake*. f is a field to be checked for uniqueness, for example: *ciphers*.

1. Initialize the array of identifiers $I \leftarrow \emptyset$
2. Set array $V \leftarrow$ all values of f where type $= t$ ▷ SQL query 1
3. **for** each $v \in V$ **do**
    (a) Set array $T \leftarrow$ types where $f = v$ ▷ SQL query 2
    (b) **if** T only contains t, **then** $I \leftarrow I \cup v$

   end for
4. Return I

To analyze the similarity of extensions encoded in hex-strings, we used the built-in fuzzy string matching of PostgreSQL[5]. Most of the supported methods of string proximity are based on phonetics and are not useful when we are analyzing similarity of hex-strings. Levenshtein is a much used edit distance for determining how dissimilar two strings are by insertions, deletions or substitutions [48]. Two strings with a Levenshtein distance of 0 are identical, and a higher number corresponds to how dissimilar the strings are.

Algorithm 4.2 shows the steps for analyzing and finding fingerprints in extensions. Steps 1 -2 are done as an automatic analysis routine, while step 3 is done manually. Step 3 consisted of opening every highlighted pair in Wireshark and looking for differences between them. A regular expression was created for the difference, and tested on the rest of the data set. If the regular expression was only found for Snowflake traffic, then we had found a fingerprint.

Algorithm 4.2 Analyzing extensions by fuzzy string matching

We let t denote the type to find the fingerprint of, for example: *snowflake*.

1. Set array $E \leftarrow$ all extensions where type $= t$ ▷ SQL query 3
2. **for** each $e \in E$ **do**
   (a) Set array $S \leftarrow$ all extensions with Levenshtein(extension, e) between 1 and 32 ▷ SQL query 4
   (b) **for** each $s \in S$, insert (s, e) in the database. ▷ SQL query 5

   end for
3. **for** each extension pair in the database, manually compare them in Wireshark.

The Damerau–Levenshtein distance (which is not supported by PostgreSQL) also considers transpositions of single characters. Implementing more advanced edit distances like Damerau-Levenshtein and groupings with locality-sensitive hashing are out of scope for this thesis, and we only utilized the built-in Levenshtein distance.

[5]https://www.postgresql.org/docs/current/fuzzystrmatch.html

4.2 Results and discussion

To validate the *dfind* artifact, we used the data set from MacMillan et al. [13]. The data set consists of 6555 DTLS handshakes from WebRTC traffic of Snowflake, Facebook Messenger, Google Hangouts and Discord running on Firefox and Chrome.

Table 4.2 shows how many *ClientHello*s and *ServerHello*s were parsed by our tool. We can see that there are more re-transmissions of *ClientHello*s for Snowflake, than for the other implementations.

Type	# handshakes	# parsed CH	# parsed SH
Snowflake	991	4366	990
Discord	1989	3119	1987
Google	1995	1657	1246
Facebook	1580	1674	1866

Table 4.2: Number of parsed handshakes from Macmillan et al. data set

Table 4.3 contains identifying values of fields found in the automatic analysis routine for Snowflake. The fields are sorted by occurrence (how many handshakes included the fingerprint), from most common on the first row, to the least common on the last. None of the identifying values presented in Table 4.3 are discussed in Macmillan et al., thus all of them are new fingerprints from the data set.

44% of Snowflake handshakes had a message with an identifying length. This corresponds well with the most important features found by Wang *et al.* [35] and Holland *et al.* [37]. The high number of unique extensions combined with the varying list of ciphers can be theorized to be what causes the total length to be the most identifying.

Unique extensions was the most common fingerprint for a DTLS handshake, with 63% of handshakes having a unique value for the entire extensions field. We conclude that extensions have a high potential for being used as fingerprints. The reason for this, is that extensions is not a fixed size field, it can be in any order, of varying length or contain many different extensions. To pinpoint more closely what the implementation difference is, we analyzed the extensions using the fuzzy matching routine.

The automatic analysis gave us 13 pairs of extensions to further look into manually. Table 4.4 shows the results from this manual work. It is worth noting that the regular

expression fingerprint is for DPI of the extension header field, not the entire handshake message bytes.

Fields	Value	Occurrence
Extensions	41 extensions, see Appendix A.1	63%
Length	67, 99, 103, 119, 123, 134, 150, 154, 156, 160, 174, 176, 180, 223, 269	44%
Extensions Length	27, 49, 68, 98, 102, 161, 187	20%
Ciphers	`c02bc02fcca9cca8c00ac009c014`, `c02bc02fc00ac014`, `c02bc02fc00ac014c0acc0ae`, `13011303c02bc02fcca9cca8c00ac009c013c014`, `cca9cca8c02bc02fc009c013c00ac014009c002f0035000a`	19%
Cipher length	8, 12, 14, 20	16%

Table 4.3: Identifying fields found during automatic analysis that can be used for fingerprinting Snowflake

Extension	Message	Fingerprint	Occurrence
a) `use_srtp`	CH	`*000e000900060001000800070 0*`	1%
b) `use_srtp`	SH	`000e*`	2%
c) `renegotiation_info`	SH	`*ff01000100*` missing	100%
d) `supported_groups`	SH	`*000a*`	99%

Table 4.4: Extensions that can be used for fingerprinting Snowflake

Fingerprint a) in Table 4.4 was still found in freshly captured snowflake traffic, running the Snowflake web browser extension[6]. The fingerprint was found in Snowflake captures from ValdikSS in 2021[7], and was not found in the non-snowflake capture[8] from the same time. The fingerprint was also found by joining a voice channel in Discord on Chromium [9]. This suggests that the fingerprint is not valid anymore, without collateral blocking. `use_srtp` is not discussed in MacMillan *et al.* [13]

Fingerprint b) in Table 4.4 is a new fingerprint that has not been discussed in Macmillan et al., or seen any real world usage. The `use_srtp` extension is required

[6]version 0.7.3 in Chromium (122.0.6261.39 Official Build Arch Linux 64-bit), go version 1.22.0, Kernel 6.7.5-arch1-1

[7]https://ntc.party/t/ooni-reports-of-tor-blocking-in-certain-isps-since-2021-12-01/1477/19

[8]https://ntc.party/t/ooni-reports-of-tor-blocking-in-certain-isps-since-2021-12-01/1477/24

[9]version Chromium (122.0.6261.39 Official Build Arch Linux 64-bit), kernel 6.7.5-arch1-1

for WebRTC, but this fingerprint is having it as the first extension in the list of extensions in the *ServerHello*. We were not able to find this fingerprint in fresh captures of Snowflake, so it is uncertain if it can still be used to identify Snowflake.

Fingerprint c) in Table 4.4 is not offering the `renegotiation_info` extension in the *ServerHello*. This fingerprint was also found by Macmillan et al. Fresh Snowflake traffic did not contain this fingerprint, as the Pion library always adds `renegotiation_info` to the *ServerHello* since February 2021[10].

Fingerprint d) in Table 4.4 was also found by Macmillan et at. and they recommended not to use this optional extension in the *ServerHello*. This fingerprint has also been used in the real world by Russia in 2022 [11]. On another occasion, Russia also blocked traffic containing specific bytes at a offset that corresponds to the list of `supported_groups` in *ClientHello*s [12]. Both of these fingerprints based on `supported_groups` have been fixed. This fingerprint did also occur once for Facebook traffic, thus it did not show up as a unique extension in the automatic analysis of fields.

Having found both new and known fingerprints, validates the capabilities of the *dfind* artifact. This also shows how we are able to find cheap regular expression fingerprints of fields, without the need for applying machine learning. As far as we know, censors have not deployed machine learning to block Snowflake, only DPI regex patterns.

4.3 Further work

We prioritized fuzzy matching and manual analysis of extensions. To further automate this routine, we could parse each extension as its own field. Then we could use the same routine as for the regular fields, trying to find unique values. If we parse each extension defined by the Internet Assigned Numbers Authority[12], we could also check for custom extensions that are not defined in the specification.

A short-coming of our tool, is that it does not parse and reassemble fragmented messages. However, there were only 11 fragmented messages in the data set of 991 Snowflake handshakes.

The cookie field in DTLS consists of a time (GMT) and pseudo-random bytes. It might be the case that one could find cryptographic differences in the implementation of the pseudo-randomness.

[10] https://github.com/pion/dtls/commit/d18b8c0d2dc829d684fd08cfd56773436ffcd51f

[11] https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/issues/40014#note_2765074

[12] https://www.iana.org/assignments/tls-extensiontype-values/tls-extensiontype-values.xhtml

We only analyze the hello messages, further work should also consider other messages (as they are mostly not analyzed in state of the art yet). We believe the *Certificate Request* message might be a good candidate to find identifiers. It contains the signature hash algorithm field, that can be in any order and be specific to what the implementation supports.

The authors of nPrintML claim that it can successfully find fingerprint features and perform identification in a fully automated fashion [37]. We do have a heatmap of their feaures (see Figure 2.4), but is hard to read the features as they cut off the DTLS payload. It would be interesting to compare our specific and deterministic approach to a more general machine learning tool.

Chapter 5
DTLS handshake generation

This chapter describes a continuous deployment (CD) setup with GitHub Actions workflows for generating fresh DTLS-WebRTC handshakes with the most recent version of common browsers. The purpose of this setup is two-fold. Firstly, a pipeline to keep the mimicking functionality up-to-date with popular browsers (that usually silently update "themselves"). Secondly, a public corpus of DTLS handshakes that will grow over time. A censor would have access and the ability to do large scale collection of DTLS traffic, so a publicly available data set is valuable for researchers to keep up.

The much used public DTLS traffic data set by MacMillan *et al.* from Princeton is over 4 years old. In such a time, the implementations for the browsers included in the data set have undergone changes and users updated to newer versions. We therefore form the hypothesis: "current browser versions have different fingerprints than what is available in publicly available data sets". This was demonstrated in the previous chapter were previously discovered fingerprints in the data set cannot be reproduced by updated browsers. Stale handshakes are not useful for mimicking purposes, as they are not representative of today's traffic. We explored the Docker image of Wang *et al.* [35], and did not find any DTLS handshakes captures. Xie *et al.* [36] did provide a GitHub repository with their artifacts. However, they only gave a link to `pan.baidu.com` for their data set, which requires a registered user with a chinese phone number. With one stale data set, and another we could not obtain, we saw the need to create a setup for collecting new DTLS handshakes.

uTLS relies on volunteers to produce fingerprints to mimic. They collect fingerprints from users visiting their website[1], which can be used for auto-generating mimicking configurations. We had trouble using their service, as it has been slow, or even unreachable (timeouts) at times. Albeit a highly realistic way of collecting fingerprints, there have been concerns from the anti-censorship team that the fin-

[1] https://tlsfingerprint.io/

gerprints are not up to date[2]. The maintainers and contributors are slow to add new fingerprints to the library code. In the time of writing, the latest versions of browsers supported by uTLS for mimicking are: Chrome 120, Firefox 120 and Safari 16.0. These versions were released in December 2023, November 2023 and September 2022 respectively. Our novel CD approach does not need human interaction and should keep itself up to date to the latest versions without maintenance.

5.1 Overview and tools

To automate browser interactions, we considered two popular open-source tools: Selenium[3] and Playwright[4]. Selenium, the older of the two, scripts browsers natively using webdrivers[5], which is a W3C standard. This ensures that we use the same browser binaries as an actual user, unlike Playwright, which ships with its own binaries. Thus, Selenium is preferred for our workflow.

We needed to decide which WebRTC application to use in the workflow. While real-world applications like Discord, Slack, or Jitsi generate realistic traffic patterns, they are quite resource-intensive. For instance, Jitsi, an open-source video conferencing project, requires four Docker containers to run. We captured handshakes from these applications using the same Chromium version[6], which resulted in identical *ClientHello* fingerprints. This indicates that different WebRTC applications produce mostly the same fingerprint with the same browser version. Multiple synthetic WebRTC applications were also explored. By synthetic, we mean applications that are not meant to be used by users in a production environment, but only for testing.

One such synthetic application is 'wpt' (web-platform-tests), a project to create "Test suites for Web platform specs". This test suite supports most W3C standards and contains a large test suite for WebRTC. The project has seen wide adoption, with Chromium using it to conduct interoperability testing between browsers[7]. However, the source code of 'wpt' is 1.1GiB, containing a lot of functionality and tests for other web specifications we do not need for our workflow. Looking for an even more minimal application, we found `webrtc/samples`[8], a repository containing sample applications written by the WebRTC project. Although 'wpt' is closer to what can be considered an industry standard for testing different WebRTC implementations, both synthetic applications produced the same fingerprint as the real applications

[2]http://meetbot.debian.net/tor-meeting/2022/tor-meeting.2022-02-17-15.59.log.html#l-96
[3]https://www.selenium.dev/
[4]https://playwright.dev/
[5]https://www.w3.org/TR/webdriver1/
[6]Version 122.0.6261.94 (Official Build) Arch Linux (64-bit)
[7]https://chromium.googlesource.com/chromium/src/+/HEAD/docs/testing/web_platform_tests.md
[8]https://github.com/webrtc/samples

mentioned above. `webrtc/samples` also contains a GitHub workflow for testing the establishment of a mediastream channel between two browsers using Selenium. We selected this workflow as the starting point for our fingerprint generation, as it contains a script for our preferred browser automation tool and produced the same handshakes as the more commonly used applications.

As Snowflake uses data-channels and not mediastreams, we checked if there was a difference in the handshakes for the different channels. Mediastreams use SRTP to deliver audio and video over UDP, while data-channels do not. However, while analyzing up-to-date captures we see that both channels add the *use_srtp* extension in the handshake. This happens because WebRTC does not want to redo the DTLS handshake if the other type of channel is required in a offer later on. Thus, handshakes are ambiguous to which type of channel is used, which is ideal for our purposes.

5.2 Implementation

The workflow (see Appendix C) runs `ubuntu-latest` on a GitHub-hosted runner and is split into two main jobs. The first job, named 'handshake-capture', is for generating fresh DTLS handshakes of browsers and uploads them as GitHub artifacts. The second job, called 'commit-fingerprints', adds the *pcap* artifacts to the repository and parses them into a Go file. Figure 5.1 shows an overview of the workflow architecture and its steps. To keep fingerprints fresh, the workflow runs as a cron job once a day.

Since the 'handshake-capture' job shall be used for both Firefox and Chrome, we utilize a matrix for the job. We define the browser as a variable, and the matrix allows us to create multiple jobs from one definition. For each variable, a job is created and all the matrix jobs runs in parallel. This matrix can potentially be expanded to add other browsers in the future.

The 'handshake-capture' job starts with a series of setup steps. It checks-out the latest branch of the repository, installs *tshark* and *Node.js*[9]. After the setup, the latest version of Chrome or Firefox is downloaded using `@puppeteer/browsers`. This is the recommended way of getting browser binaries according to the Chrome developers[10]. Next, a packet capture starts on the loopback interface with *tshark*. While the capture is running, the WebRTC application is scripted using Selenium. After the application has successfully established a mediastream, the *tshark* capture is stopped. Since all traffic is captured on the loopback interface, we filter the capture only for DTLS handshake traffic using *tshark* filters. The last step in the job is to upload the filtered *pcap* as a GitHub artifact.

[9]https://nodejs.org/en
[10]https://developer.chrome.com/blog/chrome-for-testing/

Figure 5.1: Overview of the workflow architecture

When both of the previous matrix jobs have been completed without failure, the 'commit-fingerprints' job is triggered. First, all *pcap* artifacts are downloaded from both 'handshake-capture' jobs. A checkout of the main repository is performed and a Go environment is set up. The handshake captures from both Firefox and Chrome are parsed by a Go script that extracts the *ClientHello* fingerprint and adds it as a hex-string to `fingerprints.go` (see Appendix D). This file can be imported and used for mimicking later. Since we have generated code in `fingerprints.go`, we run *gofmt*[11] and a linter to make sure that the code is formatted and its syntax correct. If we generate a fingerprint of a browser version not present in the repository already, both the hex-string fingerprint and the handshake *pcap* are committed to the repository.

5.3 Results and discussion

To make sure that the workflow creates the same fingerprints consistently, we ran the workflow every 30 minutes for 12 hours. This generated handshake messages of Chrome version `124_0_6367_91` and Firefox version `125_0_2`. In the 145 recorded handshakes (containing 87 CH and 58 SH), all the fingerprints were exactly the same. This validates that the fingerprint generation tool is consistent, and suggests that there is little variation in the handshake a specific browser version will generate.

We also analyzed the generated traffic using *dfind* and the data set from MacMillan *et al.* No identifiers were found for the Firefox traffic. This suggests that the fingerprint generation tool generates realistic traffic, and that the Firefox implementation has not changed significantly or noticeably. During manual analysis we also found that Firefox adds the `record_size_limit` extension in the *ClientHello*. This extension is not supported by Pion now, and is a certain way of differentiating DTLS implementations by use of whitelist. For Chrome traffic, multiple identifiers were found (see Table 5.1). The results show that the recent Chrome implementation has changed the order of supported ciphers and the order for the SRTP protection profiles, compared to the browser versions used by MacMillan *et al.* We created this workflow because we suspected that such changes are present in newer versions of browsers, and confirms our hypothesis. Manually capturing traffic on Chromium[12] with Discord resulted in the same handshake as the generation tool. Demonstrating that the workflow generates the same traffic as a user interacting with a real application using an updated browser.

[11] https://pkg.go.dev/cmd/gofmt
[12] version 122.0.6261.39 Official Build Arch Linux 64-bit, kernel 6.7.5-arch1-1

Fields	Value
Length	132
Extensions Length	68
Extensions	`use_srtp`: 00010008000700
Cipher length	22
Ciphers	c02bc02fcca9cca8c009c013c00ac014009c002f0035

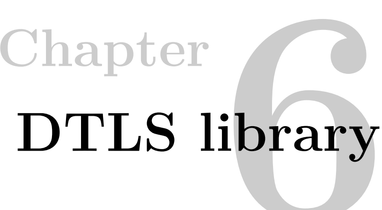
Table 5.1: Identifiers for generated Chrome handshake

5.4 Further work

We already capture the entire DTLS handshake, so exporting the hex-string fingerprint of other messages than the *ClientHello* could be added in the workflow. This would be useful for mimicking purposes.

For the setup to be more representative of the general internet users, it would be relevant to run the fingerprint workflow on Windows and MacOS, which are both supported by GitHub-hosted runners[13]. Using MacOS we could test Safari/Webkit natively, which we cannot do with the current setup. Having a setup with virtual mobile phones running iOS and Android would also be important, as this is how many people access the internet today. We did a quick analysis of a single DTLS handshake captured by using Jitsi with Fennec (`v125.3.0`) on Android and found that it produced the exact same *ClientHello* as in the generated Firefox handshake. This is not too surprising as Fennec is based on Firefox. We are not aware of any available data sets for DTLS traffic produced by mobile devices.

[13]https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources



DTLS library

In this chapter a fingerprint-resistant DTLS Go module (**A2.2**) that extends *pion/dtls* with mimicked and randomized *ClientHello* features. The first section starts with an examination of DTLS features supported by the *pion/dtls* library compared to Firefox and Chromium. Then, we provide implementation details for our module, and finish with a validation and analysis of the developed features.

6.1 Feature support

This section gives a short overview of the features supported or missing from the Pion library.

pion/dtls aims at cohering to the DTLS1.2 specification only. It currently supports 8 ECDHE cipher suites, with `chacha20poly1305` being planned. The library supports the following extensions:

- `ServerName`
- `SupportedEllipticCurves`: `P-256`, `P-384` and `X25519`
- `SupportedPointFormats`: `Uncompressed`
- `SupportedSignatureAlgorithms`[1]: `SHA256ECDSA`, `SHA384ECDSA`, `SHA512ECDSA`, `SHA256RSA`, `SHA284RSA`, `SHA512RSA` and `ED25519`
- `UseSRTP`: `AES128_CM_HMAC_SHA1_80`, `AES128_CM_HMAC_SHA1_32` , `AEAD_AES_128_GCM` and `AEAD_AES_256_GCM`
- `ALPN`
- `UseExtendedMasterSecret`

[1]Observed in various captures

- `ConnectionID`
- `RenegotiationInfo`

We know from the discussion in Section 5.3 that Pion does not implement the `record_size_limit` extension. Both the missing extension and *chacha* cipher can be used to block the Pion library with an allowlist.

The Pion library exposes a `PaddingLengthGenerator` through the config API which adds padding bytes to inflate ciphertext size, in order to obscure content size from observers. This can most likely be used to counteract flow statistics fingerprinting based on packet length, such as done by Wang *et al.* [35]. However, Xue *et al.* suggests in their pre-publication paper on 'Fingerprinting Obfuscated Proxy Traffic with Encapsulated TLS Handshakes' that multiplexing is a better approach than padding [49].

Doing a manual analysis of the code of browsers, we found that only Firefox partly supports DTLS 1.3 using their Network Security Services (NSS[2]) library, while Chromium with the boringSSL[3] library does not. Version 1.2 is the preferred version used in Firefox and version 1.3 can only be enabled by actively changing the `media.peerconnection.dtls.version.min`[4] flag to 772 (DTLS 1.3). We also saw from our generated handshakes in the previous chapter, that DTLS 1.2 was theonly version available, confirming that its not yet deployed by browsers.

6.2 Implementation

Here we present the implementation of *covertDTLS*[5], a Go module that extends *pion/dtls* to provide mimicry and randomization features.

The initial pull request (PR) to the Pion DTLS library contained both the fingerprint generation workflow (**A2.1**) from Chapter 5 and mimicking features (while planning to add randomization as a separate PR later). Adding the features to up-stream was done to minimize friction of deployment, as applications using the Pion library could just update to the newest version to our features. Two of the Pion maintainers, Daenney and Atsushi Watanabe, commented that keeping fingerprints up to date with the workflow was too much of a maintenance cost, and beyond the scope of implementing features from the RFC (the entire discussion can be found in the PR[6]). A refactoring was therefore done, with the features being replaced with

[2] https://firefox-source-docs.mozilla.org/security/nss/index.html
[3] https://boringssl.googlesource.com/boringssl
[4] https://searchfox.org/mozilla-central/source/modules/libpref/init/all.js#347
[5] https://github.com/theodorsm/covert-dtls
[6] https://github.com/pion/dtls/pull/631

general hooks in the configuration API that exposes handshake messages. Both the workflow and the mimicking was moved to its own repository called *covertDTLS*[7], that uses the added hooks for manipulating handshakes. Randomization features were added to the module later.

6.2.1 Handshake Hooks

The handshake hooks we added to *pion/dtls* are shown in Listing 6.1. The `Config` struct is the exposed API for configuring a DTLS client or server. Each hook is defined as a function that takes the hooked message type (e.g. `MessageClientHello`) as the only parameter and returns a type that implements the general `Message` interface (see Listing 6.2). The general `Message` interface is used for assembling a message struct into raw bytes to be sent over the wire, and disassemble raw bytes into a struct. We implemented hooks for the *ClientHello*, *ServerHello* and *CertificateRequest* handshake messages, as they are the most likely messages to be fingerprinted (as discussed in Chapter 4).

Listing 6.1: Message hooks added in the configuration API of *pion/dtls*

```
1 type Config struct {
2   // other configuration options...
3
4   ClientHelloMessageHook func(handshake.MessageClientHello) handshake.Message
5   ServerHelloMessageHook func(handshake.MessageServerHello) handshake.Message
6   CertificateRequestMessageHook func(handshake.MessageCertificateRequest) handshake.
        ↪ Message
7 }
```

Listing 6.2: Message interface

```
1 // Message is the body of a Handshake datagram
2 type Message interface {
3         Marshal() ([]byte, error)
4         Unmarshal(data []byte) error
5         Type() Type
6 }
```

The pion library provides a handler for each handshake flight, consisting of a *parse* function and a *generate* function. Listing 6.3 shows how we inject our hook in the `flight1Generate` function. The handshake message is constructed from the internal state and configuration (lines 6-14), and passed to a returning packet struct (lines 25-34). If the hook is *nil* (Go's zero value), the original constructed message is passed on (line 22). If the hook is not *nil*, indicating that the hook has been set in the configuration, then the constructed message is passed as input to the hook (line

[7] https://github.com/theodorsm/covert-dtls

20), and the returned message from the hook is passed to the packet struct. When the `Marshal` method is called later to build the raw bytes, the function injected by our hook will run. The `ClientHelloMessageHook` was injected in `flight1` and `flight3`, `ServerHelloMessageHook` in `flight4` and `flight4b` (an optional flight for ressuming a previous session), and `CertificateRequestMessageHook` was injected in `flight4`.

Listing 6.3: Injected hook in the `flight1Generate` function of *pion/dtls*

```
// ...
// Setting state and applying configurations.
// ...

// Constructing the original ClientHello message
clientHello := &handshake.MessageClientHello{
  Version: protocol.Version1_2,
  SessionID: state.SessionID,
  Cookie: state.cookie,
  Random: state.localRandom,
  CipherSuiteIDs: cipherSuiteIDs(cfg.localCipherSuites),
  CompressionMethods: defaultCompressionMethods(),
  Extensions: extensions,
}

var content handshake.Handshake

// Injecting hook
if cfg.clientHelloMessageHook != nil {
  hs = handshake.Handshake{Message: cfg.clientHelloMessageHook(*clientHello)}
} else {
  hs = handshake.Handshake{Message: clientHello}
}

return []*packet{
  {
    record: &recordlayer.RecordLayer{
      Header: recordlayer.Header{
        Version: protocol.Version1_2,
      },
      Content: &hs,
    },
  },
}, nil, nil
```

Pion requires test coverage of their codebase which is checked by a CI/CD pipeline at PRs. We expanded their end-to-end (E2E) tests for the *ClientHello* and *ServerHello* hooks, which only succeed if a entire handshake is completed. The test for the `ClientHelloMessageHook` also checked if the cipher list were changed successfully, while for the `ServerHelloMessageHook` we checked if we could inject

an ALPN extension. As we could not verify if fields have been changed for the *CertificateRequest* message, we wrote a unit test for `flight4` to test the corresponding hook. The test checked if we could remove all the signature hash algorithms.

After we added tests that covered our changes, the maintainers merged our features into the master branch of *pion/dtls*[8].

6.2.2 Mimicry

We refactored the workflow to be included in *covertDTLS* and the generated *ClientHellos* are added to `fingerprints.go` (see an example in Appendix D) in their own `pkg/fingerprints` package. To mimic (replay) these fingerprints, we developed the `pkg/mimicry` package.

Rather than creating a Go *struct* for mimicked fingerprints like they do in uTLS, we adopt a lower level approach, manipulating raw bytes. This reduces the chance of accidentally changing a fingerprint when marshaled from a *struct*. Figure 6.1 show how we only replace the bytes for the *ClientRandom*, *SessionID* and *Cookie* fields when mimicking a message.

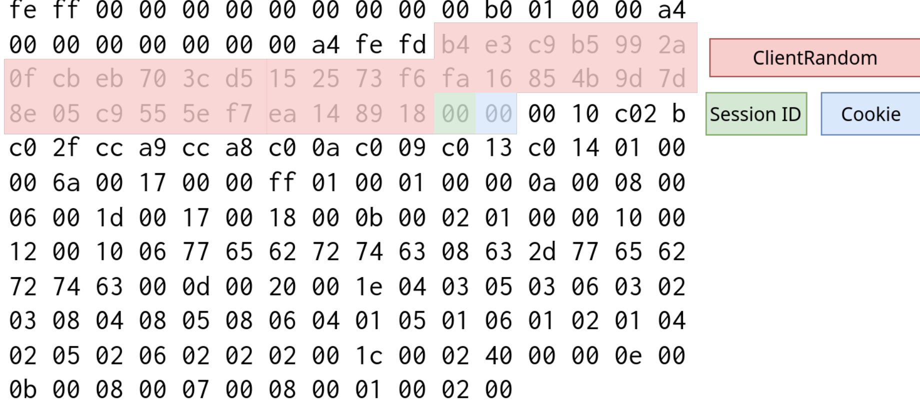


Figure 6.1: A mimicked *ClientHello* message, the highlighted bytes are replaced with values from the hooked message.

Mimicking is provided by the `MimickedClientHello` *struct*, which implements the `Message` interface and a hook method for `MessageClientHello`. When the

[8]https://github.com/pion/dtls/commit/8738ce19f77e598a194c2d1d87dc9d72e5c2e948

hook is triggered it will copy the original *ClientRandom*, *SessionID* and *Cookie* values into its own fields and return itself as a `Message`. The Pion library calls the `Marshal` function to assemble the message into bytes. `MimickedClientHello` will decode a hex-string fingerprint from `fingerprints.go` into bytes and replaces the *ClientRandom*, *SessionID* and *Cookie* with the stored original values. This will ensure that the handshake will function correctly, but the rest of the message bytes are exactly as the fingerprint.

Listing 6.4 shows an example of using `MimickedClientHello`. We provide a `LoadFingerprint` method, so that a user of the module can select a specific fingerprint, or use their own hex-string fingerprint. Notice that we also have to specify a list of `SRTPProtectionProfiles` at line 17, as our provided fingerprints are from WebRTC traffic, requiring SRTP.

Listing 6.4: An example using a mimicked *ClientHello* message with *covertDTLS*

```
1  import (
2    "github.com/pion/dtls/v2"
3    "github.com/theodorsm/covert-dtls/pkg/fingerprints"
4    "github.com/theodorsm/covert-dtls/pkg/mimicry"
5  )
6
7  // Get a specific fingerprint
8  fingerprint := fingerprints.Mozilla_Firefox_125_0_1
9
10 clientHello := mimicry.MimickedClientHello{}
11
12 // If no specific fingerprint is loaded, the most recent one will be used
13 clientHello.LoadFingerprint(fingerprint)
14
15 cfg := &dtls.Config{
16     // SRTP needs to be enabled with fingerprints
17     SRTPProtectionProfiles: []dtls.SRTPProtectionProfile{dtls.
         ↪ SRTP_AES128_CM_HMAC_SHA1_80, dtls.SRTP_AES128_CM_HMAC_SHA1_32, dtls.
         ↪ SRTP_AEAD_AES_128_GCM, dtls.SRTP_AEAD_AES_256_GCM},
18     ClientHelloMessageHook: clientHello.Hook,
19 }
20
21 // Use config with connection...
```

6.2.3 Randomization

The last package in *covertDTLS* is `pkg/randomize` (see Listing 6.5 for an example of usage), which implements a hook for randomization of an intercepted *ClientHello* message.

The goal is to create so many different fingerprints that it is unfeasible to maintain a blocklist for a censor. We focus mainly on randomizing fields consisting

of lists, and introduce Algorithm 6.1 as the way of re-ordering such fields. The ShuffleRandomLength algorithm will shuffle a list, or create a sub-set of the original list with a shuffled order.

Algorithm 6.1 Randomize order and length of a list

Let S denote a non-empty set of elements to be randomized. *rlength* is a boolean for deciding if the output set should be of random length.

ShuffleRandomLength(S,*rlength*)

1: $Out \leftarrow \emptyset$
2: $Tmp \leftarrow S$
3: **if** $rlength = true$ **then**
4: $n \leftarrow_R [1, len(Tmp)]$ ▷ pick random value in range
5: **else if** $rlength = false$ **then**
6: $n \leftarrow len(Tmp)$
7: **end if**
8: **while** $len(Out) < n$ **do**
9: $pick \leftarrow_R Tmp$ ▷ pick random element in set
10: $Out.append(pick)$
11: $Tmp.remove(pick)$
12: **end while**
13: **return** Out

The algorithm was implemented using Go generics, so we could use the procedure on any type of slice (Go's array). We randomized the following fields: ciphers, extensions, `supported_groups`, `signature_algorithms`, `use_srtp` and `ALPN`. The extensions were only shuffled, while we randomized the length and order of the other fields. `ALPN` might contain a list of protocols, but we only add one of 15 common protocol ids (defined by IANA[9]). This is also the only extension that will be injected if it is not already contained in the original message, but we provide a flag for disabling this feature. We do otherwise not inject any values that are not in the hooked message as to minimize the potential of announcing unsupported features. This also makes the randomization application agnostic, as it will use the needed features for a given application, not just WebRTC as the mimicked fingerprints do.

[9]https : / / www.iana.org / assignments / tls - extensiontype - values / tls - extensiontype - values.xhtml#alpn-protocol-ids

Listing 6.5: An example using a randomized *ClientHello* message with *covertDTLS*

```
import (
  "github.com/pion/dtls/v2"
  "github.com/theodorsm/covert-dtls/pkg/randomize"
)

randClientHello := randomize.RandomizedMessageClientHello{RandomALPN: true}

cfg := &dtls.Config{
    ClientHelloMessageHook: randClientHello.Hook,
}

// Use config with connection...
```

6.3 Results and discussion

The preceding two subsections aim at validating and discussing the mimicry and randomization features implemented in **A2.2**.

6.3.1 Mimicked *ClientHello*

To validate the mimicry capabilities of *covertDTLS* we integrated the module into a forked version of Snowflake, and compared its fingerprint to fresh Snowflake traffic. The version we forked (commit 22a94)[10] used *pion/webrtc* version `v3.2.29` and *pion/dtls* version `v2.2.7`. The libraries were quite far behind the current master branch, so we had to fork both of the old versions and manually backport our hooking features. *snowbox*[11] was used simulate a Snowflake environment, building a local deployment consisting of a client, server, proxy and broker.

We analyzed traffic (using *dfind*) from the non-modified version of Snowflake and our forked version with mimicking enabled, comparing both to the MacMillan *et al.* data set, also adding our generated handshakes of recent browsers. Table 6.1 shows the identifying features found for fresh Snowflake traffic using *snowbox*. This Snowflake fingerprint was not found in Chapter 4, indicating that the implementation has changed. Running *snowbox* using *covertDTLS* and mimicking recent Firefox and Chrome fingerprints gave **no identifiers**, showing that our module successfully mimicked recent browsers. This validates the mimicking features provides fingerprint-resistance using our fingerprint tool (**A1**).

[10]https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/tree/22a945971d1580682f2fe6e2e8ca6585e451436f

[11]https://github.com/cohosh/snowbox

Fields	Value
Length	64, 114
Extensions Length	60
Extensions	Appendix A.2
Cipher length	12
Ciphers	`c02bc02fc00ac014c02cc030`

Table 6.1: Identifiers for fresh Snowflake traffic from Snowbox

With the mimicked Firefox *ClientHello*, the unsupported extension `record_size_limit` was added. Even though Pion does not explicitly support it, the server echos the extension back in the *ServerHello*. We suspect that the server adds the unsupported extension because of GREASE [50] support, a mechanism to prevent extensibility failures.

The fingerprints we provide in `pkg/fingerprints` are only suitable in a WebRTC scenario. However, we provide the `LoadFingerprint` method to make t possible for any type of DTLS traffic to be mimicked using *covertDTLS*.

6.3.2 Randomized *ClientHello*

This section finds the theoretical amount of possible fingerprints we can generate using our ShuffleRandomLength algorithm, before presenting the results of using *dfind* (**A1**) to analyze the results of capturing 1000 handshakes using randomized *ClientHello*s with `covertDTLS`. A discussion comparing the theoretical amount and our empirical results is given.

We can consider finding all possible fingerprints as a distinguishable permutations problem. The equation for the number of permutations where r elements are selected from a set of n distinct elements is given by:

$$P(n,r) = n(n-1)(n-2)\ldots(n-r+1) = \frac{n!}{(n-r)!} \quad \text{where} \quad n,r \in \mathbb{N} \tag{6.1}$$

We let $P_\alpha(n)$ be the number of permutations of a set with n elements using Algorithm 6.1 with $rlength = false$. This shuffles a fixed length set, thus we can use Equation 6.1 with $r = n$:

$$P_\alpha(n) = P(n,n) = \frac{n!}{(n-n)!} = \frac{n!}{(0)!} = n! \tag{6.2}$$

$P_\beta(n)$ denotes the number of permutations of a set with n elements using Algorithm 6.1 with $rlength = true$. There will be a set of r-permutations for each length $r \in [1, n]$. Each of theses sets will be independent of each other. This leads us to the following equation for the total amount of permutations with shuffling and randomizing length:

$$P_\beta(n) = \sum_{r=1}^{n} P(n, r) \tag{6.3}$$

We know from the feature section (6.1) that Pion supports 8 ciphers. The number of possible values for the shuffled ciphers field are therefore:

$$P_{ciphers} = P_\beta(8) = \sum_{r=1}^{8} P(8, r) = \sum_{r=1}^{8} \frac{8!}{(8-r)!} = 109,600 \tag{6.4}$$

To find the number of permutations of the extensions field, we need the number of possible extensions and supported values for a given extension. We found that our various DTLS handshakes captures usually contained up to 7 extensions with `APLN` (where we add one of 15 protocols).

Our implementation also shuffles `supported_groups`, `use_srtp` and `signature_algorithms`, with 3, 4, and 7 possible values as identified in Section 6.1. Since each permutation of a extension affects the fingerprint of the entire extensions field, we consider the permutations dependent of each other, leading us to Equation 6.5:

$$\begin{aligned} P_{extensions} &= P_\alpha(7) \times P_\beta(3) \times P_\beta(4) \times P_\beta(7) \times 15 \\ &= 7! \times \sum_{r=1}^{3} P(3, r) \sum_{r=1}^{4} P(4, r) \sum_{r=1}^{7} P(7, r) \times 15 \\ &= 7! \times \sum_{r=1}^{3} \frac{3!}{(3-r)!} \sum_{r=1}^{4} \frac{4!}{(4-r)!} \sum_{r=1}^{7} \frac{7!}{(7-r)!} \times 15 \\ &= 994,218,624,000 \end{aligned} \tag{6.5}$$

We believe the theoretical possible amount of identifying fields, $P_{ciphers}$ and $P_{extensions}$, are sufficiently large for a censor to be unable to use a blocklist. To check if the implementation actually reflects this, handshakes would need to be collected. We ran an E2E test with *pion/dtls* 1000 times, and used *covertDTLS* to randomize *ClientHello*s to check if the module produced the same fingerprint at different times.

It would be even more realistic to use *snowbox*, but the setup requires a long startup time (of multiple minutes) to initialize the handshake, as the client waits for a proxy.

We used *dfind* to parse and analyze the capture from the E2E runs. The capture contained 1867 *ClientHello*s and 867 *ServerHello*s. Similarly as in Table 4.2, we see that there are about two *ClientHello*s for each *ServerHello*. Assuming that the randomized *ClientHello* is only accepted by the server if there is a *ServerHello* reply, we can estimate that $\frac{2000}{2} - 867 = 133$ handshakes failed of 1000, which is 13% of handshakes. This indicated that randomization is somewhat unstable, however when the client re-tries a handshake, a new randomized *ClientHello* will be made and this can be repeated until the handshake completes.

Considering finding the probability of having overlapping fingerprints as a birthday problem, we can use the following equation for finding the expected number of unique fingerprints taking n samples and d possible fingerprints:

$$E(n, d) = n \left(\frac{d-1}{d} \right)^{(n-1)} \tag{6.6}$$

Using Equation 6.6, we would expect there to be $E(1867, 109600) \approx 1835$ unique ciphers. However, we found only 530 out of 1867 were distinct ciphers, being lower than expected. We analyzed the capture and found that the maximum amount of ciphers were 6. Rather than defaulting to all supported ciphers, *pion/dtls* only add 6 of 8. The number of possible permutations are therefore $P_\beta(6) = 1956$, giving an expected value of unique fingerprints of $E(1867, 1956) \approx 718$. This corresponds better to our findings.

A problem with randomization of the cipher suite list is that the first cipher in the list is preferred, thus having a random first cipher might lead to the client and server negotiating a less secure or worse performing cipher to be used. For IoT use-cases this might be too big of a performance hit, if no careful configuration is done.

All of the 1867 *ClientHello* messages had unique extensions. This corresponds well with our estimate of $P_{extensions}$, where the total number of possible fingerprints is very high, and therefore a low chance of producing the same extensions bytes twice.

The analysis above validates that randomization of the extensions is especially effective technique for fingerprint-resistance, while the randomization of ciphers has potential, if implemented carefully.

6.4 Further work

The final section of this chapter suggest some future directions for improving our *covertDTLS* tool.

The anti-censorship team is positive about adopting the *covertDTLS* module, however there are work to be done to integrate it properly into Snowflake. *pion/dtls* is not used directly in Snowflake, but indirectly through the use of *pion/webrtc*. There are currently dependency problems that did not seem fixable without forking and backporting most of the Pion modules (*pion/dtls, pion/ice, pion/stun and pion/stun*) or waiting for a new major release of WebRTC. The Pion team just released a beta for version 4 of WebRTC, and we hope that the hooking features are added to the next minor beta release.

In Section 6.3.2 we discussed that randomization made the handshakes somewhat unstable. Further testing of the stability of both the mimcry and randomization features should be done. The users of the module might accept sacrificing some stability and performance (by trying the handshake multiple times before succeeding) to have fingerprint-resistance.

Adding support for mimicking of the *ServerHello* and *CertificateRequest* messages should be possible to do, as we already implemented the hooks for those messages in *pion/dtls*. To diversify which of the fingerprints are mimicked, we could also pick a random fingerprint to mimic, not just the freshest fingerprint as we do now.

RFC8447 [51] defines 11 (non-pre-shared-key) ciphers for DTLS1.2. To make randomization of cipher lists more effective, we could use all of the specified ciphers for randomization, as it would yield $P_\beta(11) = 108,505,111$ number of possible fingerprints. This might make the feature even more unstable, as we annonce more ciphers that might not be supported by Pion.

Chapter 7 Conclusions

We conclude this thesis by revisiting the original research questions, suggesting a direction for exploratory work and reflecting on reducing the distinguishability of DTLS.

RQ1: *What kind of fingerprints can be used to identify different implementations of DTLS?*

Assuming that a censor prefers deterministic and passive fingerprints, we developed *dfind* to identify such fingerprints. The tool is implemented to parse the *ClientHello* and *ServerHello* messages from the DTLS handshake, focusing on fields that can uniquely identify different implementations. It performs an automatic analysis of extracted fields by querying a PostgreSQL database to find unique values, and employs fuzzy matching techniques to detect similarities in extensions. Using the MacMillan *et al.* dataset, *dfind* revealed that 63% of Snowflake handshakes exhibit unique extension values, suggesting that DTLS extensions are highly effective for fingerprinting. Additionally, during a manual analysis, we discovered that the *record_size_limit* extension is absent in the *pion/dtls* implementation but present in common browsers, being a fingerprint for blocking Snowflake using an allowlist approach.

RQ2: *How can we create a fingerprint-resistant implementation of DTLS for usage in Snowflake?*

We developed *covertDTLS*, a Go module inspired by *uTLS*. Instead of forking *pion/dtls* and maintaining a complete DTLS implementation as uTLS does with the standard TLS library, we contributed handshake message hooks into the upstream code. *covertDTLS* uses these hooks to inject mimicked and randomized *ClientHello* messaged to evade detection. The library includes a continuous delivery workflow for generating fresh DTLS-WebRTC handshakes from popular browsers for mimicking purposes, ensuring that the mimicking remains up-to-date and resembles common

WebRTC traffic on the Internet. The randomization feature, manipulates the ciphers and extensions lists of a *ClientHello* message and can be applied to any DTLS application. Using *dfind*, we identified fingerprints of fresh Snowflake traffic, but with mimicking a browser fingerprint with *covertDTLS* no allowlist identifiers were found. We also demonstrated that the randomization feature inflates the possible number of fingerprints, making a blocklist unfeasible. We conclude that mimicking and randomization are effective countermeasures against passive, stateless, and field-based fingerprinting.

With censorship circumventing systems adopting DTLS libraries such as *uTLS* and *covertDTLS* to prevent passive fingerprinting, censors might see the need to start exploring more expensive, stateful and active fingerprints (such as model learning, as we describe in our exploratory work, Section 3.3.1). For these new attacks, we echo the message of 'The Parrot is Dead' [32], with mimicking being almost impossible to do. We also believe that randomization would not counter-act a probable active fingerprint. The tunneling in browser approach (discussed in Section 3.3.3) might be the only possible solution to mitigate this. We suggest work should be done bridging the gap between protocol fuzzing and active fingerprinting. These are both fields that essentially try to do the same thing: find discrepancies in states. Although fuzzing is usually done for security testing to find flaws in the implementation, we believe this is the way of exploring states of protocols.

For a final reflection we wonder if the DTLS protocol could be inherently fingerprint-resistant. During this thesis, we have identified the extensions list as the field in the DTLS protocol with the highest potential for use as a fingerprint. Concealing the extensions from a passive observer would greatly reduce the likelihood of distinguishing one implementation from another. Fortunately, this feature is already in draft as the ECH extension, which encrypts the extensions list. However, for ECH to be effective to prevent fingerprinting, it must be widely adopted by popular DTLS implementations; if only one implementation supports it, it can be easily identified. Pion primarily focuses on DTLS 1.2, whereas ECH is designed for DTLS 1.3. Firefox has a low-priority, low-severity bug report for ECH support[1], and Chrome has a similar ticket[2]. While there is ongoing work in this area, we do not see widespread adoption of ECH in the near future, motivating the importance of our *covertDTLS* library in the meantime.

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=1869753
[2] https://chromestatus.com/feature/6196703843581952

References

[1] T. S. Midtlien, “Reducing distinguishability of DTLS for usage in Snowflake”, Department of Information Security and Communication NTNU – Norwegian University of Science and Technology, Project report in TTM4502, Dec. 2023.

[2] U. Nations. “Universal Declaration of Human Rights”, United Nations. (1948), [Online]. Available: https://www.un.org/en/about-us/universal-declaration-of-human-rights (last visited: Nov. 8, 2023).

[3] M. C. Tschantz, S. Afroz, Name Withheld On Request, and V. Paxson, “SoK: Towards Grounding Censorship Circumvention in Empiricism”, in *2016 IEEE Symposium on Security and Privacy (SP)*, San Jose, CA, USA: IEEE, May 2016, pp. 914–933. [Online]. Available: https://ieeexplore.ieee.org/document/7546542/ (last visited: Nov. 10, 2023).

[4] R. S. Raman, A. Stoll, J. Dalek, R. Ramesh, W. Scott, and R. Ensafi, “Measuring the Deployment of Network Censorship Filters at Global Scale”, in *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/23099.pdf (last visited: Jun. 15, 2024).

[5] R. Ramesh, R. S. Raman, A. Virkud, A. Dirksen, A. Huremagic, D. Fifield, D. Rodenburg, R. Hynes, D. Madory, and R. Ensafi, “Network Responses to Russia’s Invasion of Ukraine in 2022: A Cautionary Tale for Internet Freedom”, *32nd USENIX Security Symposium (USENIX Security 23)*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/ramesh-network-responses.

[6] V. Ververis, L. Lasota, T. Ermakova, and B. Fabian, “Website blocking in the European Union: Network interference from the perspective of Open Internet”, *Policy & Internet*, vol. 16, Sep. 19, 2023.

[7] R. Sundara Raman, P. Shenoy, K. Kohls, and R. Ensafi, “Censored Planet: An Internet-wide, Longitudinal Censorship Observatory”, in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Virtual Event USA: ACM, Oct. 30, 2020, pp. 49–66. [Online]. Available: https://dl.acm.org/doi/10.1145/3372297.3417883 (last visited: Jun. 15, 2024).

[8] G. Grover and N. ten Oever, “Guidelines for Human Rights Protocol and Architecture Considerations”, Internet Engineering Task Force, Internet Draft draft-irtf-hrpc-guidelines-20, Oct. 4, 2023, 36 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-irtf-hrpc-guidelines (last visited: Oct. 25, 2023).

[9] N. ten Oever and C. Cath, “Research into Human Rights Protocol Considerations”, Internet Engineering Task Force, Request for Comments RFC 8280, Oct. 2017, 81 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8280 (last visited: Nov. 1, 2023).

[10] “Pluggable Transport Specification (Version 1) - Tor Specifications”. (2023), [Online]. Available: https://spec.torproject.org/pt-spec/index.html (last visited: Oct. 30, 2023).

[11] C. Bocovich, A. Breault, D. Fifield, Serene, and Xiaokang Wang, “Snowflake, a censorship circumvention system using temporary WebRTC proxies”, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/bocovich.

[12] “IRC Tip about Signature used to block Snowflake in Russia, 2022-May-16 (#40030) · Issues · The Tor Project / Anti-censorship / censorship-analysis · GitLab”, GitLab. (May 16, 2022), [Online]. Available: https://gitlab.torproject.org/tpo/anti-censorship/censorship-analysis/-/issues/40030 (last visited: Nov. 1, 2023).

[13] K. MacMillan, J. Holland, and P. Mittal. “Evaluating Snowflake as an Indistinguishable Censorship Circumvention Tool”. (Oct. 14, 2020), [Online]. Available: http://arxiv.org/abs/2008.03254 (last visited: Oct. 24, 2023), preprint.

[14] “Apply Snowflake Remove HelloVerify Countermeasure”, GitLab. (Jan. 18, 2023), [Online]. Available: https://gitlab.torproject.org/tpo/applications/tor-browser-build/-/merge_requests/637 (last visited: Nov. 2, 2023).

[15] S. Frolov and E. Wustrow, “The use of TLS in Censorship Circumvention”, in *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2019/02/ndss2019_03B-2-1_Frolov_paper.pdf (last visited: Oct. 30, 2023).

[16] Edward Snowden, *Permanent Record*. Metropolitan Books, 2019.

[17] J. Nurmi, A. Paju, B. Brumley, T. Insoll, A. K. Ovaska, V. Soloveva, N. T. A. Vaaranen-Valkonen, M. Aaltonen, and D. Arroyo, “Investigating child sexual abuse material availability, searches, and users on the anonymous Tor network for a public health intervention strategy”, *Scientific Reports*, vol. 14, Apr. 3, 2024.

[18] K. E. Himma and H. T. Tavani, *Handbook of Information and Computer Ethics*. John Wiley & Sons, Inc., 2008.

[19] R. Bodle, “The ethics of online anonymity or Zuckerberg vs. "Moot"”, *ACM SIGCAS Computers and Society*, vol. 43, no. 1, pp. 22–35, May 1, 2013. [Online]. Available: https://dl.acm.org/doi/10.1145/2505414.2505417 (last visited: Jan. 4, 2024).

[20] H. Corrigan-Gibbs and B. Ford, "Welcome to the world of human rights: Please make yourself uncomfortable", in *2013 IEEE Security and Privacy Workshops*, May 2013, pp. 1–4. [Online]. Available: https://ieeexplore.ieee.org/document/6915054 (last visited: Jan. 4, 2024).

[21] D. Fifield, "Threat modeling and circumvention of Internet censorship", Ph.D. dissertation, University of California, Berkeley, Berkeley, CA, 2017.

[22] A. C. Begen, P. Kyzivat, C. Perkins, and M. J. Handley, "SDP: Session Description Protocol", Internet Engineering Task Force, Request for Comments RFC 8866, Jan. 2021, 57 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8866 (last visited: Jun. 20, 2024).

[23] A. Keränen, C. Holmberg, and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", Internet Engineering Task Force, Request for Comments RFC 8445, Jul. 2018, 100 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8445 (last visited: Jun. 20, 2024).

[24] M. Petit-Huguenin, G. Salgueiro, J. Rosenberg, D. Wing, R. Mahy, and P. Matthews, "Session Traversal Utilities for NAT (STUN)", Internet Engineering Task Force, Request for Comments RFC 8489, Feb. 2020, 67 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8489 (last visited: Jun. 20, 2024).

[25] T. Reddy.K, A. Johnston, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", Internet Engineering Task Force, Request for Comments RFC 8656, Feb. 2020, 79 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8656 (last visited: Jun. 20, 2024).

[26] D. Fifield, "Turbo Tunnel, a good way to design censorship circumvention protocols", presented at the 10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20), 2020. [Online]. Available: https://www.usenix.org/conference/foci20/presentation/fifield (last visited: Nov. 9, 2023).

[27] N. Erinola, M. Maehren, R. Merget, J. Somorovsky, and J. Schwenk, "Exploring the Unknown DTLS Universe: Analysis of the DTLS Server Ecosystem on the Internet", *32nd USENIX Security Symposium (USENIX Security 23)*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/erinola.

[28] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2", Internet Engineering Task Force, Request for Comments RFC 6347, Jan. 2012, 32 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc6347 (last visited: Oct. 26, 2023).

[29] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security", Internet Engineering Task Force, Request for Comments RFC 4347, Apr. 2006, 25 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc4347 (last visited: Jun. 12, 2024).

[30] E. Rescorla, H. Tschofenig, and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Internet Engineering Task Force, Request for Comments RFC 9147, Apr. 2022, 61 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc9147 (last visited: Jun. 12, 2024).

[31] G. Shu and D. Lee, “A Formal Methodology for Network Protocol Fingerprinting”, *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, pp. 1813–1825, Dec. 1, 2011.

[32] A. Houmansadr, C. Brubaker, and V. Shmatikov, “The Parrot Is Dead: Observing Unobservable Network Communications”, in *2013 IEEE Symposium on Security and Privacy*, Berkeley, CA: IEEE, May 2013, pp. 65–79. [Online]. Available: http://ieeexplore.ieee.org/document/6547102/ (last visited: Oct. 25, 2023).

[33] D. Fifield and M. G. Epner. “Fingerprintability of WebRTC”. (May 27, 2016), [Online]. Available: http://arxiv.org/abs/1605.08805 (last visited: Nov. 8, 2023), preprint.

[34] J. Chen, G. Cheng, and H. Mei, “F-ACCUMUL: A Protocol Fingerprint and Accumulative Payload Length Sample-Based Tor-Snowflake Traffic-Identifying Framework”, *Applied Sciences*, vol. 13, no. 1, p. 622, 1 Jan. 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/1/622 (last visited: Oct. 23, 2023).

[35] Y. Wang, G. Yang, D. Xu, C. Dai, T. Chen, and Y. Yang, “Snowflake Anonymous Network Traffic Identification”, in *Proceedings of the 13th International Conference on Computer Engineering and Networks*, Y. Zhang, L. Qi, Q. Liu, G. Yin, and X. Liu, Eds., vol. 1127, Singapore: Springer Nature Singapore, 2024, pp. 402–412. [Online]. Available: https://link.springer.com/10.1007/978-981-99-9247-8_40 (last visited: Apr. 18, 2024).

[36] Y. Xie, G. Gou, G. Xiong, Z. Li, and M. Cui, “Covertness Analysis of Snowflake Proxy Request”, in *2023 26th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, Rio de Janeiro, Brazil: IEEE, May 24, 2023, pp. 1802–1807. [Online]. Available: https://ieeexplore.ieee.org/document/10152736/ (last visited: Oct. 23, 2023).

[37] J. Holland, P. Schmitt, N. Feamster, and P. Mittal, “New Directions in Automated Traffic Analysis”, in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’21, New York, NY, USA: Association for Computing Machinery, Nov. 13, 2021, pp. 3366–3383. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3484758 (last visited: Jan. 11, 2024).

[38] E. Janssen, “Fingerprinting TLS Implementations Using Model Learning”, M.S. thesis, Radboud University Nijmegen, 2021.

[39] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. [Online]. Available: https://link.springer.com/10.1007/978-3-662-43839-8 (last visited: Nov. 14, 2023).

[40] H. Wang, B. Cui, W. Yang, J. Cui, L. Su, and L. Sun, “An Automated Vulnerability Detection Method for the 5G RRC Protocol Based on Fuzzing”, in *2022 4th International Conference on Advances in Computer Technology, Information Science and Communications (CTISC)*, Apr. 2022, pp. 1–7.

[41] R. Wails, G. A. Sullivan, M. Sherr, and R. Jansen, “On Precisely Detecting Censorship Circumvention in Real-World Networks”, *Network and Distributed System Security Symposium (NDSS)*, 2024.

[42] M. Wu, J. Sippe, and D. Sivakumar, "How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic",

[43] Alice, Bob, Carol, J. Beznazwy, and A. Houmansadr, "How China Detects and Blocks Shadowsocks", in *Proceedings of the ACM Internet Measurement Conference*, ser. IMC '20, New York, NY, USA: Association for Computing Machinery, Oct. 27, 2020, pp. 111–124. [Online]. Available: https://doi.org/10.1145/3419394.3423644 (last visited: Jan. 23, 2024).

[44] R. Ensafi, D. Fifield, P. Winter, N. Feamster, N. Weaver, and V. Paxson, "Examining How the Great Firewall Discovers Hidden Circumvention Servers", in *Proceedings of the 2015 Internet Measurement Conference*, Oct. 28, 2015, pp. 445–458.

[45] A. T. Rasoamanana, O. Levillain, and H. Debar, "Towards a systematic and automatic use of state machine inference to uncover security flaws and fingerprint TLS stacks", in *27th European Symposium on Research in Computer Security (ESORICS)*, ser. Lecture Notes in Computer Science, vol. 13556, Copenhague, France: Springer Nature Switzerland, Sep. 2022, pp. 637–657. [Online]. Available: https://hal.science/hal-03997060 (last visited: Mar. 6, 2024).

[46] D. Angluin, "Learning regular sets from queries and counterexamples", *Information and Computation*, vol. 75, no. 2, pp. 87–106, Nov. 1, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90052-6 (last visited: Nov. 13, 2023).

[47] P. Fitera, B. Jonsson, K. Sagonas, R. Merget, J. Somorovsky, and J. de Ruiter, "Analysis of DTLS Implementations Using Protocol State Fuzzing", *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2523–2540, 2020.

[48] NavarroGonzalo, "A guided tour to approximate string matching", *ACM Computing Surveys (CSUR)*, Mar. 1, 2001. [Online]. Available: https://dl.acm.org/doi/10.1145/375360.375365 (last visited: Jun. 3, 2024).

[49] D. Xue, M. Kallitsis, A. Houmansadr, and R. Ensafi, "Fingerprinting Obfuscated Proxy Traffic with Encapsulated TLS Handshakes",

[50] D. Benjamin, "Applying Generate Random Extensions And Sustain Extensibility (GREASE) to TLS Extensibility", Internet Engineering Task Force, Request for Comments RFC 8701, Jan. 2020, 12 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8701 (last visited: Jun. 23, 2024).

[51] J. A. Salowey and S. Turner, "IANA Registry Updates for TLS and DTLS", Internet Engineering Task Force, Request for Comments RFC 8447, Aug. 2018, 20 pp. [Online]. Available: https://datatracker.ietf.org/doc/rfc8447 (last visited: Jun. 23, 2024).

Appendix A

Extensions fingerprints for Snowflake

A.1 MacMillan et at. data set

00170000ff01000100000a000c000a001d00170018010001010000b000201000010001
200100677656272746308632d77656272746300330026002401d0020c64da2c5258f
6c154e16a56e0aff7d090cc49ff2e7c732590faec2cb912d0830002b0007067f22fef
d0303000d0020001e040305030603020308040805080604010501060102010402050 2
06020202002c00160014a83586d9a18f15504841901513120fe19ffd5d41001c00024
001000e000900060007000800010 0

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630 8632d77656272746300330026002401d00202018b6ba9
107eb5789b27fd60d1ca9f0d1d9da372792d480fa76e98697c4f142002b0007067f22
fefd0303000d0020001e04030503060302030804080508060401050106010201040 20
50206020202002c00160014f07fdaf3746a8b4ba9ff92c04167413f5624e44a001c00
024001000e000900060007000800010 0

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630 8632d77656272746300330026002401d00208325112cf
d343ab74b7802bee2c22e25be14d0c392c6d8566967f2833a433878002b0007067f22
fefd0303000d0020001e04030503060302030804080508060401050106010201040 20
50206020202002c00160014b52dd008fd09c962754e0ae3a3847852269d2e00001c00
024001000e000900060007000800010 0

00170000ff01000100000a00080006001d00170018000b00020100002300000000d
00140012040308040401050308050501080606010201000e00090006000100080007 0
0

000e00050002000100000a00080006001d00170018000b00020100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630 8632d77656272746300330026002401d0020deb0fb9ae

6f7576b084d1589b9ed516493398145bca309cb86a90588e4118d0a002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d002094ecb7ef3
54aad7fd3535c76f7e3446cec91507782cf179ff76dce1a771d8832002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c00160014cdb6f4fbca214ded7dcdc3bd206854cdfa5dfaaa001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d0020e3b845979
6ce6ae5175f40ce295b238e4a8df011c2bd3889ee34e9a06d726152002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c00160014927bde84a3a6dd69f5ed404ae26ff1fcdbc5c895001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d0020606e4788e
c0f3cc57bb1a15a9457ba05baf6664c0a23ee92e7c6081d0d250d1e002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c0016001418fbe6c7a034b94489039b4032f82f6fc872f6da001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d00202018b6ba9
107eb5789b27fd60d1ca9f0d1d9da372792d480fa76e98697c4f142002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d00201dafe15b6
639147a1cb0d70cbf070aea4b339d9ba8d5b5ced42b0fef9758e85a002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010
001200100677656272746308632d776562727463003300260024001d00200545f7ed8
a3037f5a9d143db57b5344cdc3e023ca9a31a4d02893d71715c2a01002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d00202cbfbb935
bfaed4d4b52ec4d9081c34981bf351b38ed218e33c8b6d40e96371d002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202002c001600140222f98957c41036c636ba4800404359d6c58c91001c00
024001000e0009000600070008000100

ff01000100000a00080006001d00170018000b0002010000100012001006776562
72746308632d776562727463000d0020001e0403050306030203080408050806040 10
5010601020104020502060202020001c00024000000e000b00080007000800010002 00

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d00208325112cf
d343ab74b7802bee2c22e25be14d0c392c6d8566967f2833a433878002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d002094ecb7ef3
54aad7fd3535c76f7e3446cec91507782cf179ff76dce1a771d8832002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d0020576638d1c
5104e0c28d268e02aac014d5fda08c8fbebe5a815c78a8c6a5c1869002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d0020638ff43b5
cf3d3a18d6c52484953f4d4dc307c31006e61614d3ae6a070c8ca48002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202002c00160014874561164c8e902f833bf3a64fa3fca549392b7f001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
0012001006776562727463308632d7765627274630033002600240001d002087665c290
978e7485a8481bae25665fe60ac3fe926e3de8257d412103e43ff63002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
5020602020202001c00024001000e0009000600070008000100

ff010001000017000000230000000d001400120403080404010503080505010806
06010201000e00050002000100000b00020100000a00080006001d00170018

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d00203ebeab5ea
4eb7aafc86bf5afc1f4e616eb9555529132f2fc85cab9474086487c002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c001600145b063144f7f97d5bbbe79e8e5b97e84e14972b62001c00
024001000e0009000600070008000100

00170000ff01000100000a00080006001d00170018000b00020100001000120010
0677656272746308632d776562727463000d001800160403050306030203080408050
8060401050106010201001c00024000000e000b0008000700080001000200

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d002037dcca62a
014df3e6de296370ed7205a871d7327521d57f28496d556a4c1a637002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c0016001440fe64678972aebe765a3aca104539a15604efa4001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d002087665c290
978e7485a8481bae25665fe60ac3fe926e3de8257d412103e43ff63002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202002c0016001489c912d6df08cc003bffabcda95529f42c8888d4001c00
024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d002037dcca62a
014df3e6de296370ed7205a871d7327521d57f28496d556a4c1a637002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d00203a9b832fe
3e8e83f0d021c7fc6d52b340b4ffc7ed5bcf079a38e5ed07c27977b002b0007067f22
fefd0303000d0020001e0403050306030203080408050806040105010601020104020
50206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
001200100677656272746308632d7765627274630033002600240001d0020e3b845979
6ce6ae5175f40ce295b238e4a8df011c2bd3889ee34e9a06d726152002b0007067f22

fefd0303000d0020001e040305030603020308040805080604010501060102010402050206020202001c00024001000e000900060007000800010

000d000e000c040305030603040105010601000a00080006001d00170018000b00020100000e000500020001000017000

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d0020deb0fb9ae6f7576b084d1589b9ed516493398145bca309cb86a90588e4118d0a002b0007067f22fefd0303000d0020001e04030503060302030804080508060401050106010201040205020602020202002c001600148610c16c7868478b67a2d1107617a1e5cf4e67e0001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d0020606e4788ec0f3cc57bb1a15a9457ba05baf6664c0a23ee92e7c6081d0d250d1e002b0007067f22fefd0303000d0020001e040305030603020308040805080604010501060102010402050206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d00207fccf37ffd7c2a2d71ad4e051c513b2746f9b402146bdd8947e1509c62ff0103002b0007067f22fefd0303000d0020001e04030503060302030804080508060401050106010201040205020602020202002c001600140b219a81e43c21dd6f9ff00c1ad5f8f23932c0c4001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d0020638ff43b5cf3d3a18d6c52484953f4d4dc307c31006e61614d3ae6a070c8ca48002b0007067f22fefd0303000d0020001e040305030603020308040805080604010501060102010402050206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d00207fccf37ffd7c2a2d71ad4e051c513b2746f9b402146bdd8947e1509c62ff0103002b0007067f22fefd0303000d0020001e040305030603020308040805080604010501060102010402050206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d0017001801000101000b000201000010001200100677656272746308632d7765627274630033002600240001d0020c64da2c5258f6c154e16a56e0aff7d090cc49ff2e7c732590faec2cb912d0830002b0007067f22fefd0303000d0020001e040305030603020308040805080604010501060102010402050206020202001c00024001000e0009000600070008000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d00203a9b832fe
3e8e83f0d021c7fc6d52b340b4ffc7ed5bcf079a38e5ed07c27977b002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202002c00160014b14c5a1d9cc36764ca83017f7b6e16afa787f2e0001c00
024001000e000900060007000 8000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d00200545f7ed8
a3037f5a9d143db57b5344cdc3e023ca9a31a4d02893d71715c2a01002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202002c001600142c90f9aa937e7b8dbee74f8a12468bab16220855001c00
024001000e000900060007000 8000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d00202cbfbb935
bfaed4d4b52ec4d9081c34981bf351b38ed218e33c8b6d40e96371d002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202001c00024001000e000900060007000 8000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d00201dafe15b6
639147a1cb0d70cbf070aea4b339d9ba8d5b5ced42b0fef9758e85a002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202002c0016001455d8610fe9d34ffe37e2651185d2e45c81653c03001c00
024001000e000900060007000 8000100

00170000ff01000100000a00080006001d00170018000b0002010000230000000 0d
00140012040308040401050308050501080606010201000e000900060008000700010
0

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d0020576638d1c
5104e0c28d268e02aac014d5fda08c8fbebe5a815c78a8c6a5c1869002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202002c00160014bdee1a04cc39034b7597e595092a01bfb339bf56001c00
024001000e000900060007000 8000100

00170000ff01000100000a000c000a001d00170018010001010000b000201000010
00120010067765627274630863 2d776562727463003300260024001d00203ebeab5ea
4eb7aafc86bf5afc1f4e616eb9555529132f2fc85cab9474086487c002b0007067f22
fefd0303000d0020001e040305030603020308040805080604010501060102010402 0
50206020202001c00024001000e000900060007000 8000100

A.2 Snowbox Extension

000d0010000e040305030603040105010601080 7ff01000100000a00080006001d001
70018000b00020100000e000900060008000700010000170000

Appendix B
SQL queries for dfind

B.1 SQL query 1

```
db.Query(context.Background(), fmt.Sprintf("SELECT %s FROM
    ↪ fingerprint where type = $1 group by %s", field, field),
    ↪ fpType)
```

B.2 SQL query 2

```
db.Query(context.Background(), fmt.Sprintf("SELECT type FROM
    ↪ fingerprint where %s = $1 group by type", field), cl)
```

B.3 SQL query 3

```
db.Query(context.Background(), fmt.Sprintf("SELECT max(id),
    ↪ extensions FROM fingerprint WHERE type = $1 group by
    ↪ extensions"), fpType)
```

B.4 SQL query 4

```
db.Query(context.Background(), fmt.Sprintf("SELECT count(id),
    ↪ extensions, levenshtein(extensions, $1) FROM fingerprint WHERE
    ↪  type != $2 AND levenshtein(extensions, $3) BETWEEN 1 AND 32
    ↪ GROUP BY extensions"), se.Extensions, fpType, se.Extensions)
```

B.5 SQL query 5

```
db.QueryRow(context.Background(), "INSERT INTO fuzzy_extensions (
    ↪ type_id, levenshtein, extensions) VALUES ($1, $2 , $3)
    ↪ RETURNING id", se.Id, fc.Distance, fc.CmpExtensions).Scan(&
    ↪ result)
```

Appendix C
DTLS handshake generation workflow

```
name: Fingerprinting
on:
  push:
    branches:
      - main
  schedule:
    - cron: "0 1 * * *"


jobs:
  handshake-capture:
    runs-on: ubuntu-latest
    timeout-minutes: 5
    strategy:
      fail-fast: false
      matrix:
        browser: [firefox, chrome]
        bver: [stable]
    steps:
    - uses: actions/checkout@v3

    - name: Install tshark
      run: sudo apt install -y tshark

    - uses: actions/setup-node@v4

    - run: npm install
      working-directory: .github/workflows/browser-test/
```

```
- name: Remove preinstalled github chromedriver/geckodriver from
    ↪ $PATH
  run: sudo rm /usr/bin/chromedriver /usr/bin/geckodriver

- run: Xvfb :99 &

- name: Install browser version
  run: BROWSER_A=${{matrix.browser}} BROWSER_B=${{matrix.browser}}
      ↪  BVER=${{matrix.bver}} DISPLAY=:99.0 node download-
      ↪ browsers.js
  working-directory: .github/workflows/browser-test/

- name: Get browser version
  id: "browser"
  run: echo "version=$(ls ./browsers/${{matrix.browser}} | sed -e
      ↪  's/ /_/g' -e 's/\./_/g' -e 's/\-/_/g')" >>
      ↪ $GITHUB_OUTPUT
  working-directory: .github/workflows/browser-test/

- name: Create directory for pcaps
  run: |
    mkdir ./captures/
    touch ./captures/full-capture-${{matrix.browser}}_${{steps.
        ↪ browser.outputs.version}}.pcap
    sudo chown -R root:root ./captures
    ls -lga ./captures
- name: Start tshark capture
  run: sudo tshark -i any -w ./captures/full-capture-${{matrix.
      ↪ browser}}_${{steps.browser.outputs.version}}.pcap -f "udp
      ↪ " &

- name: Run webrtc applications with jest/selenium
  run: BROWSER_A=${{matrix.browser}} BROWSER_B=${{matrix.browser}}
      ↪  BVER=${{matrix.bver}} DISPLAY=:99.0 node_modules/.bin/
      ↪ jest --retries=3 interop
  working-directory: .github/workflows/browser-test/

- name: Kill tshark capture
  run: sudo killall tshark 1> /dev/null 2> /dev/null
  continue-on-error: true
```

```
  - name: Filter DTLS handshake in pcap
    run: sudo tshark -r ./captures/full-capture-${{matrix.browser}}
        ↪ _${{steps.browser.outputs.version}}.pcap -Y "dtls.
        ↪ handshake" -w ./captures/capture-${{matrix.browser}}_${{
        ↪ steps.browser.outputs.version}}.pcap

  - name: Archive pcap
    uses: actions/upload-artifact@v4
    with:
      name: fingerprint-pcap-${{matrix.browser}}_${{steps.browser.
          ↪ outputs.version}}.pcap
      path: ./captures/capture-${{matrix.browser}}_${{steps.browser.
          ↪ outputs.version}}.pcap

commit-fingerprints:
  needs: handshake-capture
  runs-on: ubuntu-latest

  steps:
    - uses: actions/checkout@v3
      with:
        ref: ${{ github.event.pull_request.head.ref }}

    - name: Create fingerprint directory
      run: |
        mkdir -p ./fingerprints-captures
        mkdir -p ${{ runner.temp }}/fingerprints-captures

    - name: Download all artifacts
      uses: actions/download-artifact@v4
      with:
        path: ${{ runner.temp }}/fingerprints-captures
        pattern: fingerprint-pcap-*
        merge-multiple: true

    - name: Install libpcap
      run: sudo apt install libpcap-dev

    - name: Setup go
      uses: actions/setup-go@v5
      with:
```

```
        go-version: 'stable'

    - name: Run pcap fingerprint parser
      run: |
        go get .
        go run main.go ${{ runner.temp }}/fingerprints-captures

    - name: Run gofmt on fingerprints.go
      run: gofmt -s -w ./pkg/fingerprints/fingerprints.go

    - name: golangci-lint
      uses: golangci/golangci-lint-action@v4
      with:
        version: v1.56.2
        skip-pkg-cache: true
        skip-build-cache: true
        args: $GOLANGCI_LINT_EXRA_ARGS

    - name: Commit fingerprints
      run: |
        git config user.name github-actions
        git config user.email github-actions@github.com
        git add ./pkg/fingerprints/fingerprints.go
        ls -R ${{ runner.temp }}/fingerprints-captures
        ls -R ./fingerprints-captures
        fingerprints=""
        for file in ${{ runner.temp }}/fingerprints-captures/*; do
          if ! [[ -f ./fingerprints-captures/"${file##*/}" ]]; then
            mv ${{ runner.temp }}/fingerprints-captures/"${file
                ↪ ##*/}" ./fingerprints-captures/
            git add ./fingerprints-captures/"${file##*/}"
            fingerprint=$(echo "${file##*/}" | sed -e 's/.pcap//g' -
                ↪ e 's/capture-//g' -e 's/./\u&/')
            fingerprints="${fingerprints} ${fingerprint}"
          fi
        done
        git commit -m "Add fresh fingerprints" -m "$fingerprints"
        git push
```

Appendix D
Example of fingerprints.go

```
package fingerprints

//nolint:revive,unused
type ClientHelloFingerprint string

// These fingerprints are added automatically generated and added by
    ↪ the 'fingerprint' workflow
// The first byte should correspond to the DTLS version in a
    ↪ handshake message
const (
        chrome_linux_125_0_6422_141 ClientHelloFingerprint = "
            ↪ fefd46d25ef57649ecfd0fc17d5d933462d70770ea629a4d74 ..."
            ↪  //nolint:revive,stylecheck
        firefox_linux_stable_126_0_1 ClientHelloFingerprint = "
            ↪ fefd456d7850288d8a38e422000b4d6b94d96af38ae8292610 ..."
            ↪  //nolint:revive,stylecheck
)

//nolint:unused
func GetClientHelloFingerprints() []ClientHelloFingerprint {
        return []ClientHelloFingerprint{
                chrome_linux_125_0_6422_141, //nolint:revive,
                    ↪ stylecheck
                firefox_linux_stable_126_0_1, //nolint:revive,
                    ↪ stylecheck
        }
}
```

NTNU
Norwegian University of
Science and Technology