TÉCNICO LISBOA

Enabling Censorship-Resistant Tor Communications through WebRTC-Based Covert Channels

Francisco Manuel Almeida Silva

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Nuno Miguel Carvalho dos Santos
Prof. Diogo Miguel Barrinha Barradas

Examination Committee

Chairperson: Prof. Paolo Romano
Supervisor: Prof. Nuno Miguel Carvalho dos Santos
Member of the Committee: Prof. Kevin Christopher Gallagher

November 2022

For those who fight every day for their freedom of expression,

Acknowledgments

The work presented in this thesis was only made possible with the help of numerous people who I crossed paths with during these last two years. I am forever grateful to all of them.

I would like to start by thanking both of my advisors, Prof. Nuno Santos and Prof. Diogo Barradas. Their immense knowledge and rational amazes me still to this day. Furthermore, their never ending motivation pushed me to move forward even when things were not going as expected. I am truly lucky to have been guided by them and thank them for all of their guidance, patience and perseverance.

I would also like to leave a huge thanks to all of my friends and colleagues, who helped me not only from a technical standpoint but also to grow as a person. I leave a special thanks to Rui Paças and Rodrigo Silva for making my journey unforgettable and supporting me when times were rough.

I want to also thank the faculty of the Distributed Systems Group (GSD), specifically the Syssec Group for their willingness to exchange ideas. A special thanks to Vítor Nunes for his patience and fruitful discussions throughout this project, and to José Brás for the always available help and constant motivation.

Most importantly, I would like to thank the unconditional support of all my long-time friends and family, who have always been there for me. To my parents and brother, for never giving up on me and motivating me to push myself harder every time. And to my girlfriend, Mafalda, for her patience to stay by my side despite the constant stress and preoccupations. Her amazing support kept me sane during the long weeks were everything went wrong.

Resumo

A rede de anonimato Tor é extensivamente usada por journalistas e denunciantes para partilhar informação sensível de forma segura. Isto levou regimes opressivos a bloquear acesso à rede Tor usando extensivas técnicas de interferência de rede. *Pluggable Transports* do Tor como o meek ou o obfs4 apontam a ajudar utilizadores do Tor a lograr censura usando uma ponte para encapsular o tráfego em protocolos de rede impercetíveis, como o HTTPS, ou outro protocolo de ofuscação alternativo. No entanto, túneis encobertos criados desta forma podem ser identificados com alta precisão recorrendo a classificadores de *machine learning*. Esta deteção é ainda mais eficaz quando os fluxos de rede estão ofuscados ou encriptados, visto que apenas precisa de calcular as propriedades estatísticas dos fluxos. Este trabalho propõem o TorCloak, um novo *pluggable transport* do Tor que oferece alta resistência contra ataques de análise de tráfego baseados em técnicas *machine learning*. Para isto, aproveitamos as recentes inovações em comunicações resistentes à censura para encapsular tráfego Tor em aplicações de vídeo baseadas na tecnologia WebRTC. Canais encobertos criados desta forma impedem um censor de distinguir tráfego TorCloak de tráfego normal WebRTC, mesmo usando técnicas como *deep packet inspection* e classificadores de *machine learning*. Neste trabalho apresentamos um completo estudo de instrumentação do protocolo WebRTC assim como o desenho e implementação do sistema TorCloak.

Palavras-chave: Rede de Anonimato Tor, Evasão de Censura na Internet, Transmissão de dados encobertos em Fluxos Multimédia, Análise de Tráfego

Abstract

The Tor anonymity network is widely used by journalists and whistleblowers to safely share sensitive information. This has led repressive regimes to block access to Tor by employing extensive network-level interference techniques. Tor pluggable transports such as meek or obfs4 aim to help Tor users in bypassing simple blocking mechanisms by leveraging a bridge to tunnel Tor traffic through either an inconspicuous carrier protocol, e.g., HTTPS, or an alternative traffic obfuscation protocol. However, covert tunnels like these can be accurately identified using machine learning classifiers. This detection approach excels even when the network flows are obfuscated or encrypted as it only needs to compute the statistical properties of flows. This work proposes TorCloak, a new Tor pluggable transport that offers strong resistance against traffic analysis attacks based on state-of-the-art machine learning techniques. To this end, we harness recent innovations in censorship-resistant communication to securely tunnel covert Tor traffic through video streaming applications based on WebRTC technology. Covert channels created this way prevent a censor from distinguishing TorCloak traffic from unmodified WebRTC streams using deep packet inspection or machine learning classifiers. In this report, we present a complete instrumentation study of the WebRTC protocol and the design and implementation of the TorCloak system.

Keywords: The Tor Anonymity Network, Internet Censorship Circumvention, Multimedia Covert Streaming, Traffic Analysis

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis addresses the problem of Internet censorship circumvention. It has the objective of bypassing state-level censorship by establishing a covert channel over WebRTC video streams, to allow the free access to the Tor Network in regions controlled by oppressive regimes. This covert channel must be established in such way that the censor cannot actively distinguish it from unmodified WebRTC video streams, even when deploying advanced traffic analysis techniques. To address these challenges, we present TorCloak, a new Tor Pluggable Transport the enables the secure transmission of Tor traffic over WebRTC video streams.

1.1 Motivation

Tor[1] is a widely used overlay network that allows a user to browse the Internet anonymously, without exposing its IP address to the destination. When a user wants to connect to a web page, an encrypted path is constructed, passing along at least three servers. The communication is forwarded through said servers and delivered to the final destination (for example, a web server). Tor implements a protocol that includes multiple layers of encryption, tunneling the network packets in such a way that no single server can know the source IP address and the final IP address of the packets. Given these properties, journalists use Tor to communicate more safely with whistleblowers and dissidents. Non-governmental organizations use Tor to allow their workers to connect to their home website while they are in a foreign country, without notifying everybody nearby that they are working with that organization.

Since Tor allows for these kinds of anonymous communication, several state-level adversaries have started targeting Tor and making strong attempts at identifying and blocking this kind of traffic. There is a growing sophistication of techniques that censor states can employ to achieve this, such as blocking IP addresses of Tor relays or using DPI (Deep Packet Inspection) to recognize traffic generated by Tor clients and blocking it. There are however some advances on Tor's side also to try to disguise Tor traffic and avoid this kind of blockage, namely by leveraging the so-called Pluggable Transports [2]. Pluggable transports such as meek [3] or obfs4 [4] aim to help Tor users in bypassing simple blocking mechanisms by leveraging a Bridge to tunnel Tor traffic through either an inconspicuous carrier protocol, e.g., HTTPS,

or an alternative traffic obfuscation protocol. However, recent advancements in traffic analysis [5] have shown that covert tunnels like these can be accurately identified using machine learning (ML) classifiers. This is effective even when the network flows are obfuscated or encrypted, as this kind of analysis only needs to consider the statistical properties of the network flows, e.g., packet lengths, to identify unusual patterns. Given today's sophistication of ML-assisted traffic analysis techniques, such attacks present a real and serious threat, enabling state-level adversaries to detect and block Tor traffic even when using existing pluggable transports.

In this work, we propose to build TorCloak: a new technology that will allow Internet users to circumvent Tor blocking and give users unrestricted access to Tor while, simultaneously, preventing these communications from being detected by government-controlled ISPs. TorCloak will consist of a new Tor pluggable transport (and surrounding ecosystem) that will establish covert channels between Tor browsers and TorCloak bridges, which act as proxies to the free Tor network. TorCloak will set up these covert channels by piggybacking Tor traffic on the video streams of widely-used web conferencing services based on WebRTC technology, e.g., Whereby.com. Thus, under the guise of a regular video call, users will be able to freely and stealthily access services like SecureDrop without being blocked or detected.

To this end, we propose to leverage Protozoa [6], a recent censorship-resistant communication tool to securely tunnel covert Tor traffic through video streaming applications based on WebRTC technology. WebRTC is used by many popular video conferencing services for making real-time video calls, e.g., Jitsi Meet. Currently, Protozoa can establish high-performing, traffic analysis-resistant covert channels over encrypted WebRTC streams to tunnel TCP/IP payload traffic to the open Internet. Our idea is then to harness this capability in our system to securely transmit Tor traffic between Tor clients and bridges. Covert channels created this way will prevent a censor from distinguishing TorCloak traffic from unmodified WebRTC streams using deep-packet-inspection (DPI) or statistical traffic analysis, a cornerstone for the successful deployment of previous pluggable transports.

Our approach also includes the design of a TorCloak bridge distribution infrastructure to accompany our solution. This will allow Tor users to easily find bridges to connect and use our software. This will help mitigating one of Protozoa's practicality issues, in that a Protozoa client was assumed to know a third-party outside the censored region with whom the client could establish a covert channel with. This bridge distribution service will allow our solution to scale to a larger amount of users.

1.2 Challenges

In order to provide a secure and inconspicuous covert channel over WebRTC in the presence of a sophisticated traffic analysis-capable adversary, we need to overcome several challenges. The first challenge of this work is how to **encode Tor data across multiple WebRTC implementations**. WebRTC has been currently adopted by numerous services that integrate real-time communication capabilities. This integration has been greatly facilitated by the simplicity of the JavaScript WebRTC API [7]. This creates a lot of opportunities to create tools similar to our proposal, but it does also introduce a lot of varia-

tion on the implementation used. Since WebRTC is an open-source project, each WebRTC-based web streaming application can implement its own variation of it. We need to encode our covert data while maintaining the overall structure of the WebRTC protocol, to avoid errors in the client side or possibly in the WebRTC gateway (explained in Section 3.2). These errors would severely impact our tool's throughput as well as make it easier for the adversary to detect out of the ordinary WebRTC video streams and block them. Secondly, we need to **balance performance and resistance against traffic analysis**, since we cannot use the full space of the WebRTC data structure, the throughput of our covert channel is reduced drastically. There is a inherent trade-off between the bandwidth of the covert channel and how resistant to errors and traffic analysis it is. It is important to find a reasonable balance so that the system still has enough bandwidth to allow users to use it for typical Internet tasks. Lastly, we need **design a practical TorCloak bridge distribution service** to guarantee a secure and stable support to our tool. We need to, once again, find the balance between having a system easy to find for any user that needs it while maintaining some difficulty for a censor to identify it and block it. This is key for the wide adoption of our system in all kinds of oppressive regimes.

1.3 Contributions

To address the above mentioned challenges, this thesis proposes a new Tor Pluggable Transport named TorCloak, that will enable the secure transmission of covert Tor communications over WebRTC. In particular, this thesis makes the following contributions:

- A characterization study of WebRTC and an instrumentation framework for the WebRTC protocol: our study delivers a detailed explanation on the insides of the WebRTC protocol and how it can be manipulated to create covert channels. We also provide a debugging toolkit which can be leveraged by practitioners to understand and develop new information-hiding applications in the infrastructure of most popular WebRTC-based web streaming applications.

- The design and implementation of TorCloak, a new covert transport mechanism for Tor traffic which resists against state-of-the-art traffic analysis attacks. TorCloak is compatible with the latest Pluggable Transport (Version 3) and fully integrated with the Tor Client, introducing a covert transport mechanism for Tor traffic, resistant to traffic analysis attacks.

- The design of a bridge distribution infrastructure, necessary to support TorCloak, including a broker to serve as the meeting point between TorCloak Clients and TorCloak Bridges.

- An extensive experimental evaluation of TorCloak. We evaluate system's network performance (throughput and latency) and compare it to other similar systems.

1.4 Thesis Outline

The remaining of the document is organized as follows. Chapter 2 presents the necessary background of TorCloak and the related work described in the literature. Chapter 3 shows the studies of the WebRTC protocol and present the framework needed to instrument it. Chapter 4 describes the complete design and implementation of our system. Lastly, Chapter 6 presents the result of the experimental evaluation of TorCloak, and Chapter 7 concludes this thesis.

Chapter 2

Background and Related Work

Over the last few years, the research community has introduced several advances related to privacy-enhancing tools (PETs), many aimed at enabling secure and anonymous censorship-resistant communication tools for the Internet. TorCloak surfaces from this line of work and aims to further improve and extend the previous tools. In this section, we provide some necessary background on Tor and discuss the related work involving pluggable transports. We then analyze all of the previous work at the genesis of this project. Lastly, we cover other kinds of attacks and defenses to the Tor network complementary to this work.

2.1 Overview of the Tor Anonymity Network

Tor [1] is a circuit-based low-latency anonymous communication network. The basic concept behind Tor is called onion routing [8], on which a client's traffic is forwarded through several servers (nodes), to improve anonymity and reduce the chance of being tracked. This is mainly because each node only knows its predecessor and successor, storing no data of the original requester. Tor relies on these nodes - called *relays* which are mostly maintained by volunteers and forward traffic along a circuit (see Figure 2.1). Circuits are paths chosen based on predefined policies [9] and are composed of three relays: an *entry*, *middle* and *exit* relay. To establish a circuit, clients obtain a list of current relays and then exchange session symmetric keys with each relay in a circuit, one at a time – referred to by *telescoping* path-build design [8]. These symmetric keys are valid during the session. Relays discard keys when the circuit is closed providing perfect forward secrecy. The relay list is made available by special relays that act as *directory authorities*. Data that flows between these relays is composed of fixed-size cells (512 bytes), encrypted by symmetric keys previously exchanged with the client. At a minimum setting, Tor cells are triple encrypted, once for each relay. Each cell is intended to a specific relay; upon decrypting it with the correct symmetric key, the relay can check if the cell is intended to be forwarded to the next relay or a control cell which contains data to be interpreted by the relay (for example, a HTTP request).

Tor is compatible with the majority of TCP applications without any modifications or kernel require-

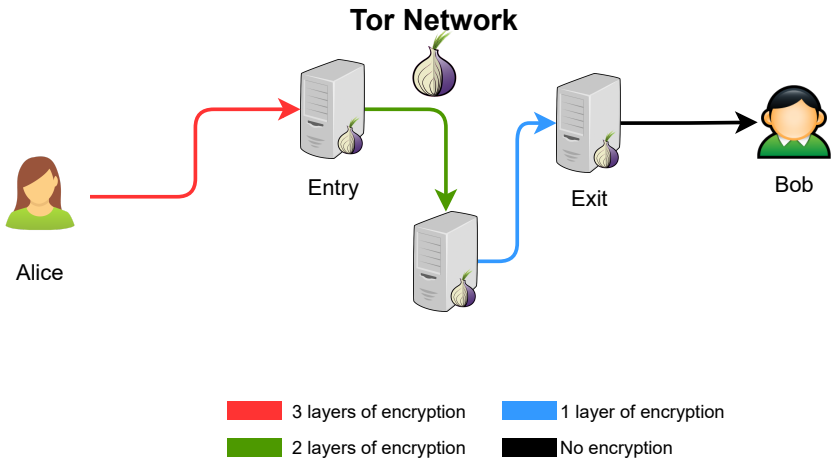


Figure 2.1: Example Tor circuit enabling Alice to connect anonymously with Bob.

ments. By providing a SOCKS interface, it allows, e.g., an email client application to be used on top of Tor without any modifications on the email client itself apart from changing a proxy configuration.

2.2 Tor Bridges and Pluggable Transports

Due to certain censorship-resistant mechanisms offered by Tor, many users living in regions controlled by oppressive regimes have adopted the system. For this reason, Tor has become a large target of blockage in such regimes. As explained above, Tor uses relays to forward the traffic, and such relay list is public. This makes it easy for ISPs to block the IPs of such servers, impeding users from connecting to them. To overcome this blockage, Tor started employing *bridges*. Tor Bridges [10] consist of unlisted proxies, outside of the censored region, whose goal is to forward the user's traffic to a Tor relay. Thus, instead of connecting to a potentially blocked relay's IP address, users connect first to such bridges and get their traffic routed to one of the available relays. Some bridges are public, deployed by volunteers of the project and others are private and only known to a few [10].

However, bridges are still vulnerable to traffic fingerprinting attacks. If a user connects to a bridge using the vanilla Tor protocol, it becomes possible to identify Tor traffic and blacklist that bridge. Matic et al. [11] show that 55% of public bridges are vulnerable to aggressive blocking. As referred in Section 2.1, Tor uses a fixed data format, with cells all of the same size. This makes it possible to identify such traffic and blacklist the bridge's IP address, just moments after it starts being used. Another way to identify Tor traffic is based on TLS, used to encrypt Tor communication. TLS uses an extension [12] called Server Name Indication (SNI). This is used to add the domain name to the TLS header to be used in the *Client Hello* handshake phase. This SNI is added before the TLS handshake, and so, is not encrypted, allowing an attacker to eavesdrop on what domain name the user is trying to reach. Tor inserts a bogus domain name in this field, following a fixed random string. This property can be uniquely identified as Tor's traffic, enabling, once again, censors to block the bridge.

As a defensive technique, Tor employs *pluggable transports* (PT) [2]. They consist of obfuscation wrappers that hide Tor traffic between a user and a bridge, to protect against traffic analysis attacks.

They transform the Tor traffic flow so that censors that monitor the traffic between client and bridge will see traffic patterns that do not conform to Tor's networking protocol. Tor Browser already employs a set of PTs, the most known being the following ones:

- *obfs4* [4]: It is the PT with the highest available number of bridges. It provides a level of obfuscation for an existing authenticated protocol, like SSH or TLS. The obfs4 protocol encrypts all traffic using a symmetric key shared between the bridge and the user derived from both client's and bridge's initial key [13]. An attacker would only see a randomly encrypted byte stream.

- *meek* [3]: It uses a technique called *Domain Fronting* [14]which is the use of different domain names at different layers. It consists in encoding a legitimate website domain in the SNI and using the Host field in the encrypted HTTP request to encode the blocked website's domain. meek does require a CDN (Content Delivery Network) because a CDN's frontend server (called an "edge server"), upon receiving a request for a resource not already cached, forwards the request to the domain found in the Host header. This means that the censor will see the legitimate allowed domain but the edge server will request and deliver the blocked website.

- *Format-Transforming Encryption (FTE)* [15]: FTE transforms Tor traffic to arbitrary formats using their language descriptions. FTE extends conventional symmetric encryption with the ability to transform the ciphertext into a format of the user's choosing, indicated by an inputted regex expression. Users can then force protocol misidentification across a broad range of Deep Packet Inspection (DPI) systems.

However, as indicated by Wang et al. [16], these obfuscation tools are still subjected to several attacks, enabling a censor to successfully detect and block them. Obfs4 and FTE are both shown to be susceptible to entropy-based tests, with a successful detection rate of nearly 100%. Entropy-based tests consist of, for a given flow, calculating the entropy of every packet payload and comparing them with the maximum/minimum/average entropies in packets in each direction (upstream or downstream), to efficiently detect if a flow has been obfuscated or not. Another tecnhique, using ML-based attacks, leveraging *timing-based features*, based on the traffic patterns of meek, the authors were also able to distinguish meek traffic from typical TLS connections and essentially detect it with a success percentage of approximately 98%. This is because meek clients will periodically send packets to a meek server, to check for data to send, creating timing patterns that differ from the typical TLS connections.

Developed with a different goal in mind than the existing PTs, TorK [17] is a new pluggable transport being developed that aims at eroding the power of traffic correlation attacks [18] that can potentially be launched by state-level adversaries.

Specifically, TorK prevents a network-level adversary from unambiguously deanonymizing the IP address of a Tor circuit creator. This is achieved by enabling K users to voluntarily open a *k-circuit*. A k-circuit is a new communication primitive that simultaneously i) connects K participants to an entry point in the Tor network, ii) tunnels through these connections the traffic of the real Tor circuits created by the participants, and iii) hide the real identity of the circuit creators by making each connection indis-

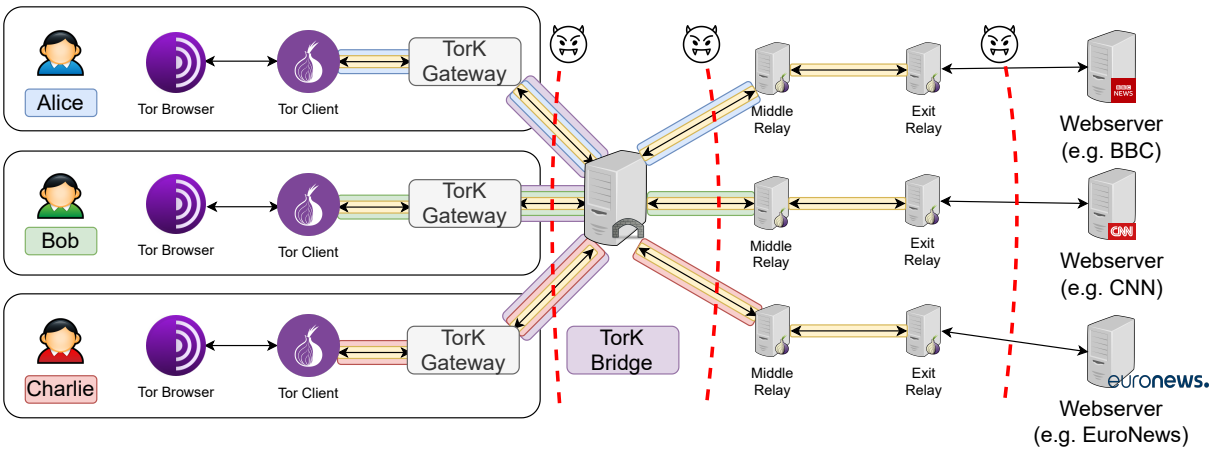


Figure 2.2: TorK architecture representing a k-circuit. All hops are observed by the attacker – dashed red line. All three members opened a Tor circuit.

tinguishable from one another. Thus, TorK offers the property of K-anonymity for all the participants of a k-circuit.

Figure 2.2 conveys this idea by depicting deployment of TorK in a simple scenario involving three users. The system itself consists of two components that interact with the Tor infrastructure: a *gateway* and a *bridge*. The gateway is software that runs on the users' local computers and acts as a client towards the bridge. The bridge, in turn, consists of a standalone server that accepts connections from one or multiple client gateway instances. Gateway and bridge implement a Tor pluggable transport and work together to exchange traffic between the Tor client software running on the users' computers and the Tor network. By leveraging the pre-existing Tor infrastructure and APIs – i.e., bridge and pluggable transports – TorK is fully compatible with Tor.

The figure illustrates how three users – Alice, Bob, and Charlie – can help each other to prevent a network-level adversary from finding out which websites each of them is individually accessing over the Tor network. They begin by running the TorK gateway software on their computers and request the establishment of a k-circuit where $k = 3$ participants. Each gateway opens a channel with the TorK bridge. We call each of these channels a *segment* of the k-circuit. Once all three segments are ready, the bridge instructs the gateways to authorize the local user to access the Tor network. Bob, for instance, can now open the Tor browser and access the BBC website. The Tor browser locally connects via a SOCKS interface to the Tor client software which in turn creates a Tor circuit through which the HTTPS request to `bbc.com` is sent. The resulting Tor circuit payload is tunneled via the k-circuit segment that connects Bob's gateway to the bridge. Similar to Bob, Alice and Charlie are also free to access the Tor network and create their circuits through which the traffic to `cnn.com` and `euronews.com`, respectively, can be tunneled.

However, TorK was designed to enhance the anonymity of Tor circuits, not Tor's censorship resistance capability, which is the goal of our project. Given that TorK exchanges standard Tor messages between client and bridge, an adversarial ISP observing the client's communication can immediately detect the traffic and block it. Unfortunately, the remaining PTs presented above offer limited protection against ISPs that employ traffic analysis techniques based on machine learning classifiers to differenti-

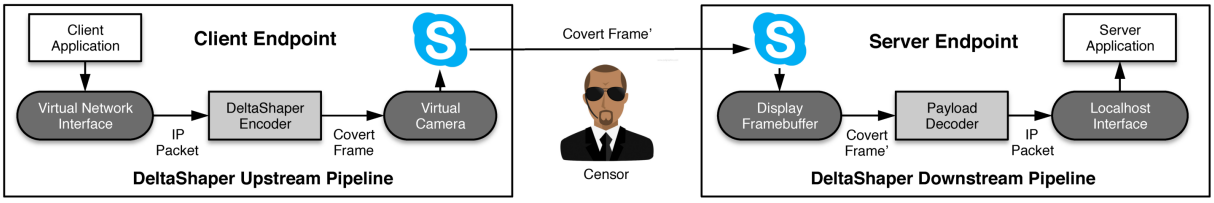


Figure 2.3: DeltaShaper architecture.

ate between regular traffic and the PT-generated traffic [5]. In contrast, TorCloak aims at being robust against these attacks by leveraging the latest generation of multimedia covert streaming techniques.

2.3 The Early Stages of Multimedia Covert Streaming

Over the last few years, several censorship-resistant systems have been proposed and prototyped by the scientific community, as a way to share and access blocked information on the Internet. In particular, an approach that has garnered considerable attention is called *multimedia covert streaming* [19]. Its general idea is to rely on covert channels to allow for the stealthy transmission of sensitive data through an apparently innocuous multimedia stream.

FreeWave [20] was one of the first tools to implement multimedia covert streaming. It allows users to encode covert data into the acoustic signal of Voice-Over-IP (VoIP) connections, using Skype. However, FreeWave is vulnerable to traffic analysis attacks since the packet length distribution of the traffic containing covert messages is nothing similar to that of a normal human speech communication through a Skype call.

The next step in the evolution of multimedia covert streaming was to use the video signal for establishing covert channels. Facet [21] enables tunneling censored videos through video calls of applications such as Skype. To prevent detection of the covert channel, this tool embeds the censored video in a portion of each video frame, filling the remaining space with a legitimate conversation video, to enable the traffic pattern to be similar to the one produced by a legitimate call. This approach provides active attack (when an attacker attempts to modify or alter the content), resistance by design, since any perturbation in the network will cause the same effect on a regular or covert video transmission. When compared with FreeWave, Facet [21] provides greater resilience to both active and passive attacks (when an attacker observes and monitors the contents).

Facet's insights have been further explored by the development of DeltaShaper [22], a tool that offers data-link interfaces and supports TCP/IP applications that tolerate low throughput and high latency links. Figure 2.3 represents the architecture of the system illustrating the covert traffic exchange between the endpoints of a networked application. On the sending side, the transmitter receives the payload and encodes it in a video stream that is fed to Skype using a virtual camera interface. Skype transmits this video to the remote Skype instance and the received stream is captured from the Skype video buffer. A decoder then extracts the payload from the video stream and delivers it to the application. The same procedure is applied at both endpoints of a Skype call, thus, effectively, supporting a bi-directional

channel in this way.

The main challenge tackled by DeltaShaper is how to encode covert TCP/IP data into the video frames of a Skype call while maintaining unobservability, which consists in the ability of a user to use a service without third parties being able to observe that the service is being used, while also maintaining reasonable throughput. To achieve high throughput, we would ideally use as many bits as possible from the video frame. However, doing this might bring some problems: almost every video streaming application does some kind of compression, using some technique to reduce the number of bits sent. This means that DeltaShaper's covert data bits embedded in the video frame may be lost due to compression. Besides that, encoding an excessive number of bits into the video frames will most likely generate a network stream pattern much different from a typical Skype call, which will break the unobservability goal.

To overcome these challenges, DeltaShaper uses a custom data encoding scheme based on the following ideas:

- *Blend carrier frames and payload frames*: to mimic the network traffic of a normal Skype call, payload frames – i.e., video frame segments that encode the covert TCP/IP data to be sent – are blended (overlapped on the top-left corner) into the carrier frame. Carrier frames can be obtained from pre-recorded, typical Skype calls or a live webcam video feed.

- *Support tunable payload frame encoding*: A payload frame is composed of one customizable payload block, which has a grid of cells. Each cell has a contiguous fixed-size area of pixels. The color code of the pixels is used to encode the bits of information of the payload block. The encoding scheme is defined by these parameters: size of the payload frame in pixels (a_p), size of the cells in pixel (a_c), number of bits for color encoding (b_c) and the payload frame rate (r_p).

The parameters mentioned above can be tuned to improve DeltaShaper's resilience to color changes introduced by Skype or to mitigate the video compression effects. The values used in DeltaShaper that strike a balance between these factors are: (a_p,a_c,b_c,r_p) = (160x120,4x4,1, 3). With these numbers, we can calculate the maximum number of bits we can send per frame (N), as well as the maximum throughput for the defined frame rate: $160x120$ pixels, which gives us a total of 1200 cells per frame. Since we have 1 bit (binary black-white image color code), the payload block is $N = 1200$ bits. With three frames per second, this gives it a throughput of 3.6Kbps. This means that the performance of DeltaShaper's covert channels is relatively low, limiting the breadth of client applications that this tool can sustain. In particular, this limitation hinders the transmission of payloads from low-latency anonymity networks such as Tor.

2.4 Encoded Media Tunneling

Prior work on DeltaShaper has shown that the lack of control over Skype's multimedia pipeline hinders the ability to create covert channels that simultaneously provide high throughput and strong resistance against traffic analysis. In fact, by using machine-learning (ML) to perform traffic analysis, censors can

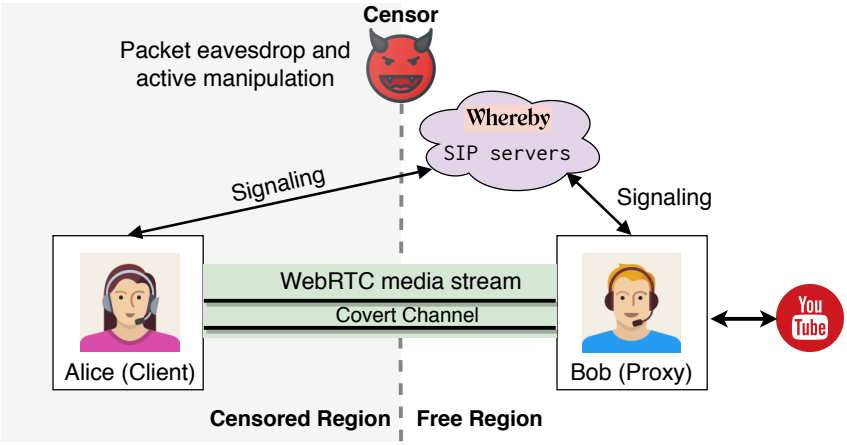


Figure 2.4: WebRTC covert tunnel over a Whereby call.

detect small deviations in the packets, such as packet size or timing, and effectively distinguish normal video streams from streams with covert data embedded. While DeltaShaper can tolerate such attacks to some degree, the channel performance ends up being highly sacrificed.

Protozoa [6] is a new tool that aims to fix both of these problems, by using web streaming application-based WebRTC, creating a *covert tunnel* between the user located on the censored region and another one (e.g. a friend) on an uncensored region. Protozoa employs a technique named *encoded media tunneling* which consists of embedding the covert data into the already encoded video frames, after, for example, a lossy compression algorithm has been applied. This allows it to highly boost the bandwidth of the covert channel, since, unlike DeltaShaper, no redundancy is required since the data will not be going through any kind of compression algorithm after.

The general operation of Protozoa can be grasped with the help of Figure 2.4 which shows two Internet users. One of them (the *client*) is located in a censored region where the access to certain content or services (e.g., Youtube) is blacklisted by a state-level censor who can inspect and control all the network communications within its perimeter of influence. The client intends to overcome these restrictions and access blocked content without the censor's awareness. This will be achieved with the help of a trusted user (e.g., a family member) located in the free Internet region who will act as a *proxy*.

Protozoa creates a high-performance covert channel ($\approx$1.4Mbps) between the client and the proxy such that the former can transparently tunnel through all the IP traffic generated between local networked applications (e.g., a web browser) and remote Internet destinations (e.g., Youtube). To create such a covert tunnel, the two users must establish a video call using a WebRTC-enabled streaming website, such as Whereby (`https://whereby.com`). As it is common in such services, one of the users must first create a chatroom, obtain a corresponding URL identifier, and share that URL with the other user via some out-of-band medium (e.g., SMS or email). Upon receiving that URL, the invited user can then join the chatroom and a video call is initiated. At this point, the WebRTC stack implementation of each of the users' browsers engages with the Whereby servers into the execution of WebRTC-specific signaling protocols that guarantee the authentication of both communicating peers and the integrity and confidentiality of the ensuing peer-to-peer encrypted video transmission between them. Given that

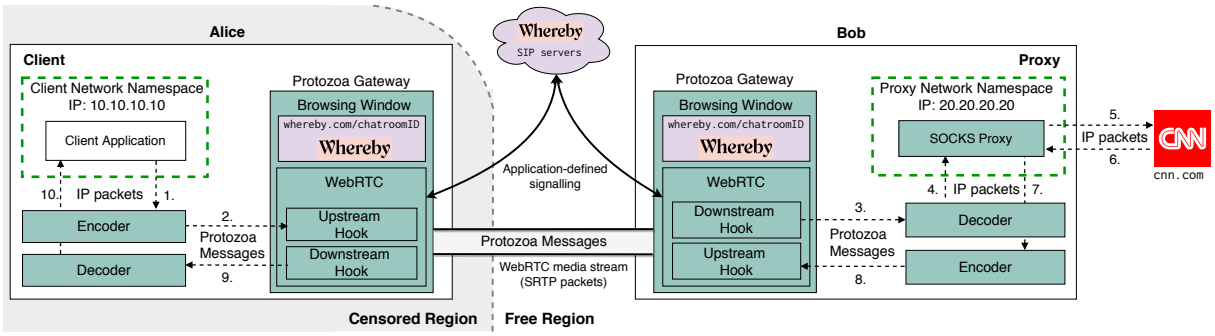


Figure 2.5: Architecture of Protozoa (components colored in a darker shade).

WebRTC [23] is a W3C standard and is built in all major browsers, Protozoa can be generally used and adapted to several applications that follow this standard for video conferencing.

Figure 2.5 depicts the architecture of Protozoa and shows Alice (client) executing a local application for accessing `cnn.com` using Bob's computer as a proxy. The carrier application consists of a web application that uses the WebRTC protocol for providing a point-to-point live streaming service between its users, e.g., Whereby. Typically, such an application consists of a backend that handles the signaling and session establishment of video calls between participants, and client-side JavaScript & HTML code that initiates video calls and manages the video transmission via the WebRTC API provided by the browser. The resulting media stream will be used by the system as the carrier for a covert channel.

Protozoa's software bundle targets the Linux platform, and it is set up differently to work as a client or proxy by two participating parties. The client will be able to execute unmodified IP applications whose traffic can seamlessly be routed through the WebRTC covert channel by the proxy to destination hosts anywhere in the free Internet region. The software bundle itself is composed of four components:

1. *Gateway server*: responsible for the setup and management of a point-to-point WebRTC covert channel between two parties. It is built out of a modified Chromium Browser [24] and uses two hooks - *upstream* and *downstream* to intercept outgoing and incoming WebRTC streams.

2. *Encoder service*: responsible for encoding streams of IP packets generated by local networked applications.

3. *Decoder service*: perform the reverse operation of the former, reading incoming messages and extracting the enclosed IP Packets and then writing them to a raw socket to be routed to its final destination on the Internet.

4. *SOCKS proxy server*: executed only on the proxy server, it exposes a network interface configured with an IP address for which all of the upstream IP Packets (sent by the client in the censored region) are routed; therefore, all of the packets are transparently delivered to the SOCKS proxy and then delivered to the final destination, e.g., a remote Internet website.

As pointed out above, Protozoa replaces the bits of the encoded video signal after the input has been compressed by the WebRTC video codec, helping to increase the capacity of the channel, as well as its resistance against traffic analysis. It does this by leveraging two hooks built into the WebRTC

stack – upstream and downstream – which can, respectively intercept outgoing frame data after being processed by the video engine, and intercept incoming data frames right after the transport layer has reconstructed an encoded frame.

Having this design allows Protozoa to have the following **advantages:** it can achieve a channel bandwidth capacity in the order of 1.4 Mbps while providing strong resistance to traffic analysis. The evaluation results showed that if a censor would like to block 80% of all Protozoa flows it would erroneously flag approximately 60% of all legitimate traffic. This shows that distinguishing between Protozoa streams and legitimate streams is close to random guessing. It also relies on the fact that blocking the WebRTC protocols altogether (blocking Protozoa usage) would lead to significant collateral damage for a censor: media streaming applications, specially these days, have a fundamental importance in a country's economy and social relations. Protozoa does however have its **limitations** as well: On its own, Protozoa cannot perform the discovery of trusted proxies. It relies on the user inside the censored region to know a contact outside of the censored region, in who he can trust and rely on to be his proxy. It does not possess any mechanism that matches users to proxies in an automated way. It is also incapable of detecting Sybil nodes, state-level attackers impersonating users of the system in order to infiltrate the system and track down legitimate users.

2.5 Enhancing WebRTC Covert Channels with Video Steganography

Using the same basis of Protozoa, Stegozoa [25] is a more recent attempt at enhancing WebRTC covert channels using video steganography. Like Protozoa, it leverages WebRTC video channels to create its covert channel. However, unlike Protozoa, Stegozoa does not operate under the assumption that the participants of a videocall exchange media in a peer-to-peer fashion. If that was the case, given that peer-to-peer connections carrying the video streams are encrypted end-to-end, not even a state-level adversary with unrestricted access to the network infrastructure would be able to observe the raw video content of these WebRTC streams. Unfortunately, a myriad of WebRTC application challenge this assumption by relying on *WebRTC Gateways* (see 3.2): specialized servers that mediate the transmission of media streams between the participants engaged in a videocall for scalability and performance purposes. The point in fact is that, other than blindly relaying media packets between participants (as in the case of TURN servers [26]), WebRTC gateways have the ability to decrypt, validate, and perform arbitrary operations on incoming media streams [27]. Evidence suggests that popular WebRTC services, like Discord, do make use of WebRTC gateways to inspect audio data exchanged by users [28].

Unfortunately, this means that Protozoa users can be trivially detected if they make use of web conferencing services that leverage WebRTC gateways controlled by an adversary, e.g., when using a domestic video-conferencing infrastructure under the control of a censor. This risk exists because Protozoa encoded video frames can be clearly identified as they carry arbitrary covert data payload that fails to adhere to the specification of media codecs.

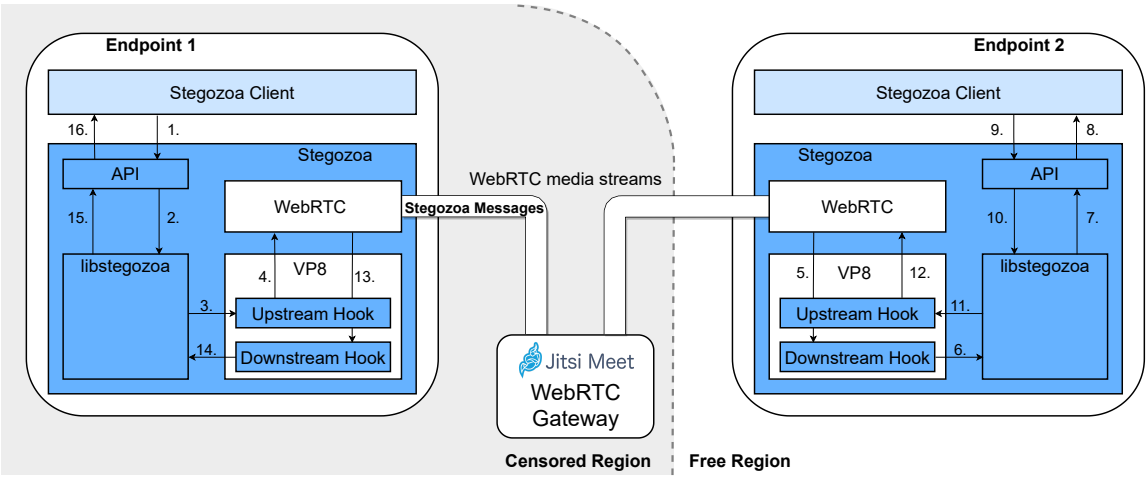


Figure 2.6: Architecture of Stegozoa (components of the system highlighted in blue).

Stegozoa is a novel tool for circumventing Internet censorship that provides a messaging exchange service offering reasonable throughput for the surreptitious transmission of covert data over WebRTC video calls. It can withstand traffic analysis and media steganalysis attacks launched by state-level adversaries in control of WebRTC gateways. To meet this goal, Stegozoa creates covert channels by augmenting the WebRTC encoding pipeline with the ability to steganographically embed covert data into the compressed representation of WebRTC video frames.

Figure 2.6 depicts the architecture of Stegozoa. The system itself consists of a gateway service composed of three main elements: an API, the `libstegozoa` library, and a set of media interception hooks located inside a modified Chromium Web browser (similar to Protozoa). The coordinated interaction of these elements, acting at different abstraction layers, allows Stegozoa to establish a covert channel between two users – Alice and Bob – and stealthily exchange messages through this channel. A client application running executed at each communication endpoint offers a user-friendly interface for sending and receiving messages through the API exposed by the Stegozoa gateway.

Like Protozoa, for establishing covert channels, Stegozoa instruments the media pipeline of WebRTC video conferencing applications. These applications, for instance Jitsi Meet, rely on JavaScript code that executes in users' browsers as part of the WebRTC framework for performing peer discovery, signalling, and calling establishment operations. Users' browsers are also responsible for encoding, transmitting, and decoding media data relying on WebRTC's native C++ implementation, enabling call participants to exchange media streams. In the context of Stegozoa, these media streams (video streams in particular) will be forwarded between call participants via a WebRTC gateway while acting as a carrier for steganographically-marked covert data.

Figure 2.7 illustrates how two users, Alice and Bob, can establish a covert communication channel using Stegozoa to exchange some sensitive content, e.g., news articles. First, Alice accesses a given WebRTC video-conferencing service, like `meet.jit.si`, creates a chatroom (A1), and shares the chatroom's URL (A2) with Bob (B1), using some out-of-band channel, e.g., e-mail. Likewise, Alice and

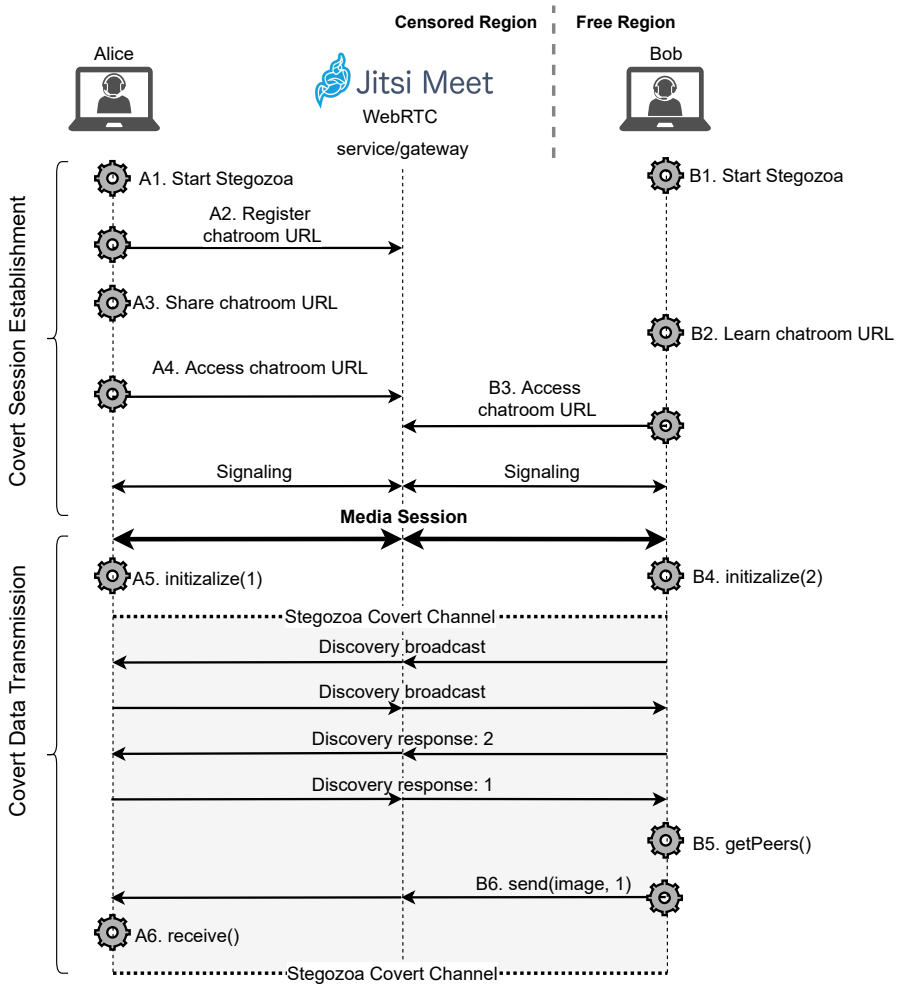


Figure 2.7: Stegozoa's covert channel establishment.

Bob also exchange a shared secret – this secret will be used to bootstrap Stegozoa's steganographic encoder. Then, both Alice and Bob join the chatroom (A3 and B2) and start transmitting video content. Note that while access to the chatroom may also be protected by a password, the adversary, has described before, is already assumed to possess the ability to visually and programmatically inspect the media exchanged by Alice and Bob. Once the media session is in place, Alice and Bob invoke Stegozoa's `initialize` API call (A4 and B3) to bootstrap their local Stegozoa endpoints. As part of Stegozoa's bootstrapping process, `libstegozoa` will generate and broadcast *discovery packets*, special control packets that will be steganographically marked in video frames and which will act as a participant's presence beacon. Other Stegozoa participants who listen to this beacon will then reply with their participant ID in a response packet. Then, the sender of the discovery packet will collect the observed replies and register the current Stegozoa peers' IDs. In our example, after peer discovery is finished, Bob can list his current peers' IDs by issuing the `getPeers` API call (B4). Armed with the information that Alice's ID is equal to 1, Bob can now send a message to Alice by invoking the send API call (B5). Finally, Alice will be able to receive Bob's message after automatically invoking Stegozoa's `receive` API call (A5).

The described mechanisms allow Stegozoa to achieve an average throughput of 8.2 Kbps, sacrificing

a higher throughput for a stronger threat model. It also allows it to highly resist traffic analysis attacks, obtaining a maximum AUC of only 0.6, meaning a censor willing to block 50% of all Stegozoa streams (TPR = 0.5) would erroneously block about 40% of legitimate WebRTC streams (FPR = 0.4). Furthermore, it also provides resistance against video steganalysis: with proper configuration, the classifier was only able to obtain a value of 0.6 for the AUC.

2.6 When WebRTC meets Pluggable Transports

As referred in Section 2.2, a continuous effort is being made by the community to further increase the number of Tor Pluggable Transports, to enable free internet access in censored regions. Seeing the potential in WebRTC to perform covert data transfer, as explained in Section 2.4, a new Tor pluggable transport, developed by the same author as meek [3], appeared, under the name of Snowflake [29, 30]. Snowflake is based on peer-to-peer connection through proxies that run in web browsers, that anyone can easily deploy.

Snowflake's main design is composed of three main components:

- many *snowflake proxies*, which perform the communication over WebRTC between with the client and forward their traffic to the bridge.
- a *broker*, which consists of a database that matches clients and proxies.
- a *Tor bridge*, consisting of a full-featured proxy capable of connecting the user's traffic to its final destination.

The basic functioning of Snowflake is the following: a client i) sends the broker a message containing their IP address and other metadata necessary to establish a WebRTC connection. The broker then ii) finds an available snowflake proxy (previously registered) and forwards the client's message to it, which iii) sends back its answer containing the necessary information as well. The broker then iv) forwards the proxy's response to the client. After this, a WebRTC connection can be established between client and proxy. The proxy then connects to a bridge to forward the client's traffic. Snowflake's blocking resistance relies on having a large number of deployed proxies. Since it allows for a client to easily change from one proxy to another, if a censor manages to block the IP address of one proxy, another proxy will quickly take its place. However, a problem still arises from this design. The original request, from client to a single broker, to request the attribution of a proxy, is still subjected to blockage since the broker is a single server. Well, Snowflake solves this issue by making this initial request use a technique called domain fronting (explained in Section 2.2), which consists of encoding a legitimate website domain in the SNI and using the Host field in the encrypted HTTP request to encoded the blocked website's domain, in this case, the broker's domain. This is the basis of meek pluggable transport. The difference here is, while meek uses this technique exclusively for all of the client's traffic, incurring in high monetary costs necessary to deploy it, Snowflake offloads most of the data transfer to WebRTC, significantly reducing the costs.

While tackling the problem of detection by address, Snowflake is however still susceptible to fingerprint-based attacks, by using machine-learning to perform traffic analysis. Similar to Protozoa, Snowflake's final goal is to create a data flow similar to other WebRTC applications, making it very hard to detect. But, unlike Protozoa, which uses WebRTC media channels to encode its covert data, Snowflake encodes its data into the data channels of WebRTC. This means that, despite both being encrypted, an observer can easily distinguish a media channel from a data channel. This can lead to blockage when an unusual amount of traffic from a data channel is detected since, in the usual uses of WebRTC (e.g. video calls), most of the data is transferred via media channels.

Furthermore, as acknowledged by the author and the Tor Team, Snowflake contains some features that allow it to be potentially identified [31] by a state-level adversary. The added setup costs brought by the use of domain fronting, while reduced, still also represent a clear disadvantage to the system. In contrast, TorCloak aims at further reducing the chances of detection by a traffic analysis attack while still maintaining an easily deployed infrastructure and software.

2.7 Emerging Mobile Privacy-Enhancing Technologies

Since the introduction of the first smartphone in 2007, millions of people now run around daily with these devices, counting on them for their everyday tasks and more. Following this uprising, a plethora of useful services are delivered to these users via mobile applications. These applications can be obtained easily through what are called app stores, which contain thousands of applications ready to be installed. In many ways, these technological advances greatly facilitate our lives and improve our day-to-day activities. However, another side of the coin must be made aware. All of this ease of use comes with some security and privacy risks. Firstly, smartphones these days, to facilitate their users' lives, collect and store a lot of personal data, which is sometimes not properly stored and intentionally shared for profits [32, 33]. Secondly, with such a large market, app developers and companies constantly, intentionally or not, put users' privacy at risk, since Android's open design allow for a zero-permission app to easily infer the presence of specific apps, or even collect the full list of installed apps on the phone through standard APIs [34].

Due to all of these growing problems, the research community has started directing a great amount of effort to develop Privacy-Enhancing Technologies for mobile platforms, to offer solutions to safely and anonymously collect user's data, as well as to be able to use your mobile device in a more safe and private setting. One of those tools, Opaak [35] uses a cryptographic mechanism called *anonymous credentials* [36–38] to define a series of protocols that allow users to demonstrate possession of a credential granted by a trusted authority without revealing anything other than the fact that they own such a credential. This can be particularly helpful in websites that count on votes, ratings, reviews, and recommendations, such as IMDB, Yelp, and Amazon.

Specifically, Opaak supports *anonymous single sign-on (SSO) authentication* and *anonymous message boards*. A user can use their phone number to register with an anonymous identity provider (AIP) and thus acquire an anonymous credential. The anonymous credential forms the basis for users to cre-

ate cryptographic proofs that enable them to anonymously authenticate to relying parties so they can use the services they offer, being of login or making any kind of comment, post, or others.

On this topic, two other notable digital identity management (DIM) frameworks are OpenID [39] and Microsoft's CardSpace [40]. They primarily focus on allowing users to consolidate and manage their digital identities, not offering anonymity from relying parties like Opaak.

Another framework that uses *anonymous credentials* is Privacy and Identity Management for Europe (PRIME) [41], an ongoing effort to develop a comprehensive framework for privacy for the web and its applications, although it's far more complex and contains a lot more components to reach its wide scope of design goals. PRIME also focuses on managing the identity attributes (e.g., age, sex, marital status) of users and allowing them to selectively disclose these attributes to relying parties.

VeryIDX [42] is a similar system to PRIME, also focusing on providing identity attributes to relying parties. They achieve this by using their aggregate zero-knowledge proof protocol based on an aggregate signature scheme by Boneh et. al [43]. Similar to Opaak, they also store the secrets on the user's phone.

However, all of these tools, while striving to maintain privacy and anonymity, fail to protect the user against network-based attacks on anonymity such as correlating the IP addresses or even blocking the users' access to such anonymity-driven services. TorCloak aims to help surpass this issue by further strengthening users' privacy and anonymity online by providing strong resistance against traffic analysis attacks. Hopefully, combined with some of the previously mentioned work, it will allow for a more private internet use.

Summary

In this chapter, we started by introducing the notion of the Tor anonymity network and their main entities. We then introduced Tor Bridges and Pluggable Transports as the main techniques to fight Internet censorship. Next, we discussed some multimedia covert streaming systems, with focus on Protozoa, giving a brief rundown of the WebRTC protocols used by Protozoa for establishing the covert channel. Lastly, we presented some of the emerging mobile PETs in the literature. In the next chapter, we will discuss the WebRTC protocol in more details and give an overview of our instrumentation framework for the protocol.

Chapter 3

Instrumentation Framework for WebRTC and VP8/VP9

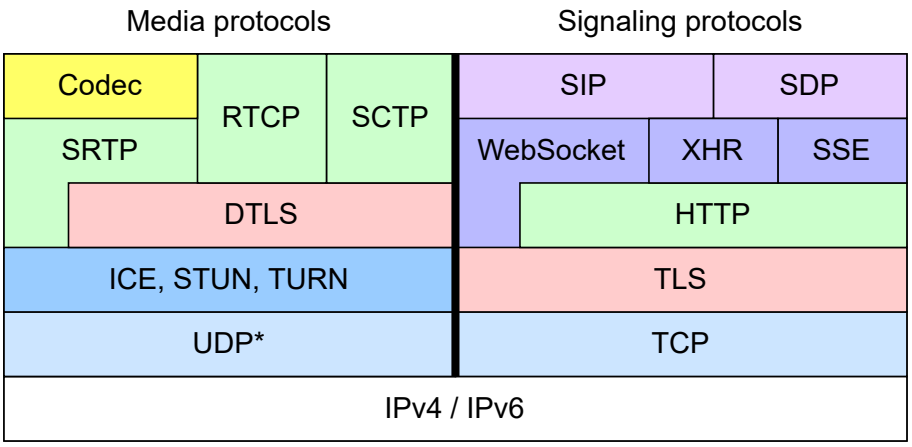
In this chapter, we present a detailed explanation our for instrumentation framework for the WebRTC Code Stack. It begins by taking a deep dive into the WebRTC protocols (Section 3.1), followed by the introducing of the concept of WebRTC Gateways (Section 3.2). Then, in Section 3.3 we detail WebRTC's video encoding and decoding flows, understanding the difference between several data structures and processing methods. Lastly, Section 3.4 describes our implementation of the instrumentation of these video flows.

3.1 A Deep Dive into WebRTC Protocols

WebRTC [23], short for Web Real-Time Communication, is an open-source web-based framework, which allows exchanging real-time media directly between browsers. WebRTC is integrated directly into the user's browser, requiring no additional plugins or browser extensions. Most browsers have built-in support for WebRTC, allowing for the use of this technology by any web application.

WebRTC is basically a set of protocols joined together. We will now explain in more detail the internals of WebRTC, concentrating on four main phases of the protocol: Signaling, Connecting, Securing and Communication. In Figure 3.1 we can see an overview of the WebRTC protocol stack.

Signaling: At the start of the connection, a peer (i.e., the initiator of a video call) does not know where to find the other peer nor what kind of information they will exchange (e.g., the media codecs used to exchange video), so a discovery procedure must be made. The signaling protocol is not standardized in WebRTC, granting developers the freedom to use a signaling protocol of their own choice in WebRTC applications. An example of a popular signaling protocol is SIP [44]. SIP uses an existing protocol called SDP (Session Description Protocol) [45], a key-value protocol. SDP is constructed in a text-based format and is used to announce and manage session invitations by sending information like the media capabilities and supported codecs of the browser, IP address, port number, and media exchange



*in some scenarios, for example because of firewall restrictions, TCP may be used

Figure 3.1: WebRTC protocol stack.

protocols, bandwidth usable for the communication and authenticity data. The signaling phase works in an Offer/Answer manner, with one of the peers making an offer (initiating the call) and the other providing an answer (for example, by rejecting some codecs or limiting the bandwidth) – starting the negotiation of the session characteristics. SDP also provides the necessary information (authenticity and integrity data) for the establishment of a direct connection between the peers, needed for the next phase.

The signaling phase requires the existence of dedicated third-party infrastructure (for example, a SIP server), accessible to both peers, for the exchange of metadata to occur. As shown in Figure 3.2, Alice's and Bob's browsers first contact the signaling server to establish a direct connection. The infrastructure needed for the signaling phase is the responsibility of the WebRTC application.

Connecting: This phase deals with the establishment of a direct bidirectional connection between the two peers. Establishing such a connection is far from trivial: the peers are usually in different networks, behind a NAT (Network Address Translation), with no direct connectivity; negotiation of IP versions, as many devices still do not support IPv6. WebRTC makes use of another existing protocol – ICE [46] (Interactive Connectivity Establishment) – in order to solve the above challenges. Peers start by querying STUN servers for their public IP addresses (unless the direct connection is possible right away, e.g., the peers are in the same network), programmatically create NAT mappings, and send the mapping information to other peers. If the direct connection is still not possible (e.g., the NAT device creates a different mapping for each destination address, making it impossible to reuse an existing one), ICE falls back to using a TURN server. The TURN server has the responsibility of relaying the traffic between the peers and overcoming the limitations of NAT devices. When a remote peer gets traffic, they see it coming from the TURN server, although the server can not inspect its content; as we will see in the next phase, the traffic is encrypted. The WebRTC application usually provides the TURN servers.

Securing: WebRTC uses DTLS (Datagram Transport Layer Security) [47] in order to establish a secure channel. DTLS is a variant of TLS, functioning over UDP instead of TCP. The secure channel is created

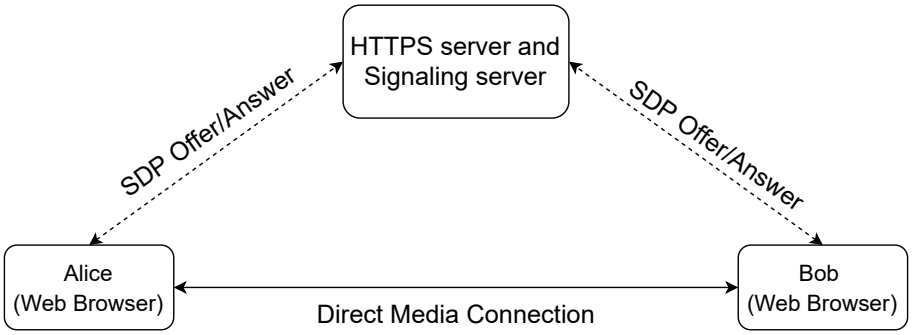


Figure 3.2: Signaling phase.

by exchanging session keys with DTLS, which are then used by another protocol – SRTP (Secure Real-time Transport Protocol) [48]. SRTP is responsible for encrypting RTP packets (explained in the next phase), providing confidentiality, integrity, authenticity, and resistance against replay attacks to the media packets exchanged between the two peers.

Communicating: This phase concerns itself with the exchange of media content. It also handles problems or dynamic changes to the call's conditions, like bandwidth decrease or packet loss. Two existing protocols are used, RTP (Real-Time Transport Protocol) and RTCP (RTP Control Protocol) [49]. RTP has the job of transporting media content (audio and video streams). It allows one or multiple streams, guarantees reliable delivery and adapts dynamically to the network conditions. This is achieved thanks to the assistance of RTCP, which exchanges control information and metadata about the call [50, 51]. RTCP informs the peer of possible problems in transmission, delays, packet loss and other conditions so that the exchange of packets can be adapted and optimized. RTCP is also useful for debugging purposes, as any error in the sending of packets generates a specific RTCP report.

Now that we explained the WebRTC protocol and seen how a secure session between two peers can be established, we will discuss in more detail the WebRTC architecture and how it handles multiple clients in heavy media sessions.

3.2 WebRTC Gateways

As mentioned, WebRTC calls are, by default, peer-to-peer: that is, every peer sends its video and audio stream to every other peer, in a mesh-like configuration. However, this type of solution is not scalable beyond a few participants [52, 53]. Another major problem of WebRTC peer-to-peer calls was IP address leakage [54, 55]. As it obvious, for the basis of WebRTC to work each peer must know the other peers' IP address in order to establish a connection. In the early days of the Internet this might not have been a problem, but now, with all of the user's concerns with privacy, it's a huge deal. You might not want to share your IP address with your video call peer or peers. In order to solve these problems, most popular WebRTC services use media servers, called gateways, to protect and scale video calls for many participants [56].

These are usually called **WebRTC gateways:** media servers that become a central management

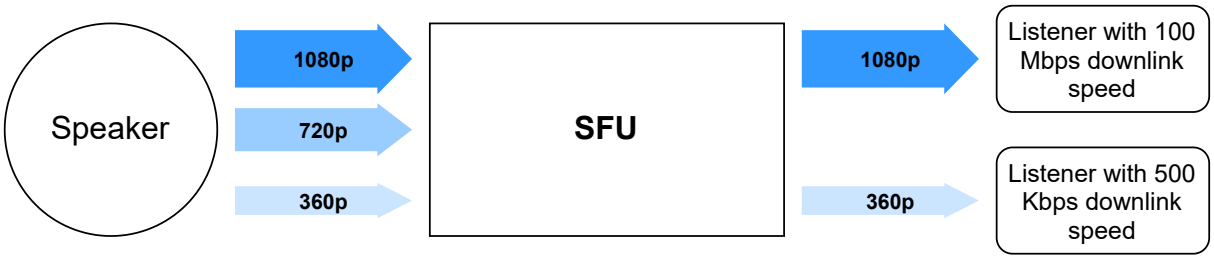


Figure 3.3: Simulcast: the SFU sends the appropriate stream to each participant.

machine for the video call, communicating with each peer individually. This configuration allows each peer to send their stream to the central server instead of sending to every other peer. It highly reduces resource consumption and avoids a direct connection between each peer, avoiding possible IP address leakage. Basically, each participant has a connection with the gateway.

There are two main configurations of gateways used by popular WebRTC services:

- *MCU (Multipoint Control Unit)* [57]: a server which mixes all of the incoming stream of the multiple peers connected to them and sends back a single stream to the participants, using a topology called Point-to-multipoint. Each participant only needs to send it's audio and video to one place and receives the mixed audio and video of the remaining participants. This mixed stream can be different and configured differently to each one of the participants - this results in heavy server-side computing, making it not so popular for most WebRTC services [58].

- *SFU (Selective Forwarding Unit)* [59]: the most used approach, based on a technique known as simulcast [60]. SFUs implementations vary but most can implement different techniques to manage the audio and video streams of each peer according to their connectivity and hardware. This solves one of the biggest issues of modern days videoconferencing, that is adapting the video stream to everyone's network and computer capabilities. SFUs allow different peers to have different speeds/resolutions of video streams according to their network's capabilities, as shown in Figure 3.3. Simulcast uses a technique called Scalable Video Coding (SVC) [61], available in several video codecs. It basically consists of a video stream encoded in the form of embedded bitstream structured in layers, allowing to select and/or decode more or less of this layers. Depending on the number of layers, a different quality video is generated.

It is however important to note that WebRTC gateways implementations are not always exactly like the described implementation - different WebRTC services adapt their implementation according to their needs and infrastructure, not following any specific implementation standard, as we noted during our testing, and will now describe. Most WebRTC services do not exactly publish how their services work internally and do not share implementations of their gateways. This makes it hard for developers and testers to make use of their structure, since it basically becomes a black box. In the next section we will describe our process to instrument WebRTC.

3.3 Understanding WebRTC Video Frame Flows

WebRTC is built on top of an open standard. It's a collaboration between several people and entities. It is also constantly growing and being developed in several different programming languages. For this reason, specially for someone who is not familiarized with WebRTC, it's particularly hard to find specific documentation about the code and it's flow, without actually following the code function by function. For this particular reason, we had the need to create an instrumentation tool that would allow us to better understand and debug the whole WebRTC C++ Code Stack. This tool will serve as a complement to the already existing WebRTC statistical and logging capabilities [62].

One of the first important steps to master WebRTC is to figure out the actual flow the program takes during a videocall. This allows us to visualize each function and better understand where certain errors may be occurring. To do this, we placed several variables across the WebRTC function following its function calling. This allowed us to reconstruct two of the main flow paths of WebRTC, as shown in Figures 3.4 and 3.5: **receiving** and **sending** of video frames and packets. These two flows are crucial in our context and in the development of similar multimedia covert streaming tools based on WebRTC. They allow us to know when to intercept the already encoded video frames and replace it with covert data, without inducing errors due to verification mechanisms that could result in the discarding of frames.

It also important to note two key definitions to understand the flows. There are two types of data structures used in WebRTC sending and receiving:

- **RTP Frames:** actual encoded video frames, with a larger size, already processed by the video engine and codec and ready to be sent
- **RTP Packets:** pieces of video frames, with a reduced size. They are sent through the network and reconstructed in the receiver end to create a video frame again.

The tool is composed of several simple incremented counters, as shown below, placed in different locations of the WebRTC code stack, and incremented each time that piece of code runs.

```
// Frames
std::atomic<uint32_t> global_frame_receive_counter = 0;
std::atomic<uint32_t> global_frame_send_counter = 0;
std::atomic<uint32_t> global_dropped_frames_congestion_window = 0;
std::atomic<uint32_t> global_dropped_frames_media_optimization = 0;
std::atomic<uint32_t> global_dropped_frames_by_encoder = 0;
std::atomic<uint32_t> global_dropped_frames_by_source = 0;
std::atomic<uint32_t> global_dropped_frames_size = 0;
std::atomic<uint32_t> global_dropped_frames_by_encoder_queue = 0;
std::atomic<uint32_t> global_dropped_frames_by_decoder = 0;

// Packets
std::atomic<uint32_t>  global_packets_sent = 0;
```

```
std::atomic<uint32_t>  global_packets_received = 0;

std::map<uint32_t, int>  ssrc_counter;
std::map<uint32_t, SsrcChange> ssrc_change_tracker;
uint32_t last_ssrc= 0;
```

This variable allow not only to confirm the path of execution of the WebRTC program but also to check for possible errors in frame or packet processing, why these errors are occurring and detect whether they are locally generated or possible due to the presence of a WebRTC gateway.

3.4 Instrumenting the WebRTC Receiving and Sending Flows

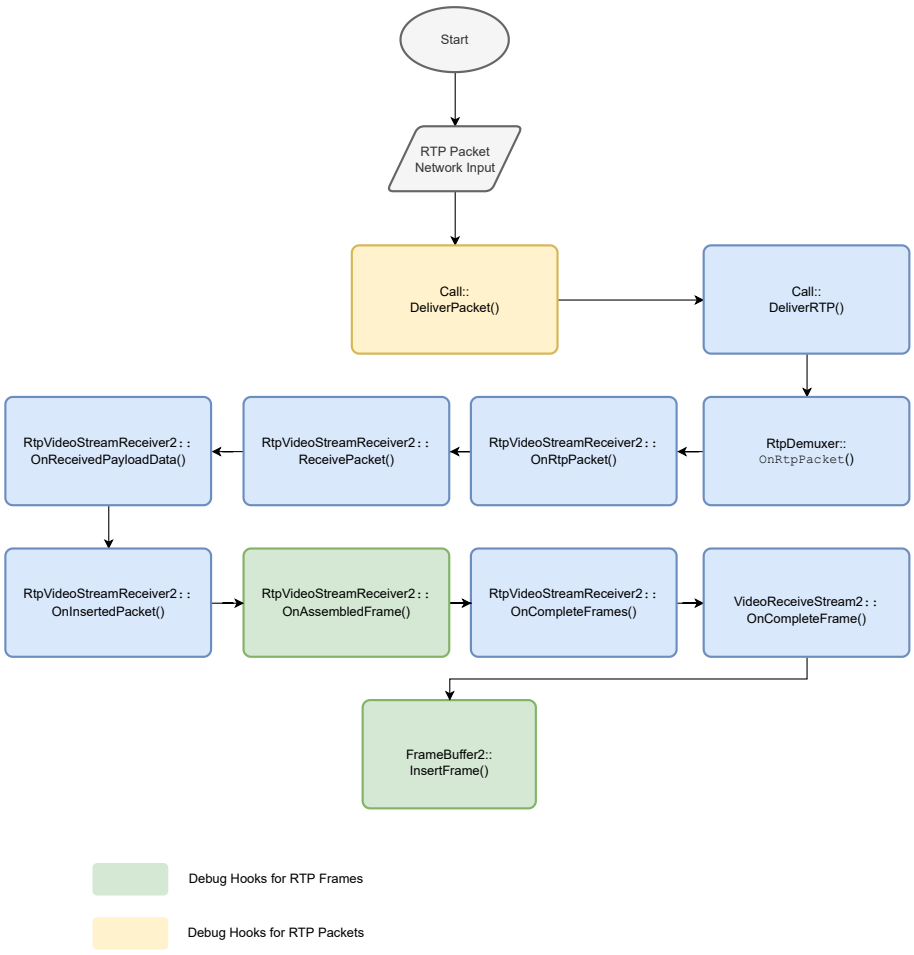


Figure 3.4: WebRTC Receive Flow.

In Figure 3.4 we can take a deeper dive into the receiving of a frame in the WebRTC Code Stack:

1. The first step is to retrieve the actual packets from the host's network and construct or RTP Packets.
2. The packets are then routed through several functions for processing until they are actual reconstructed into frames and fed to *RtpVideoStreamReceiver2::OnAssembledFrame()*

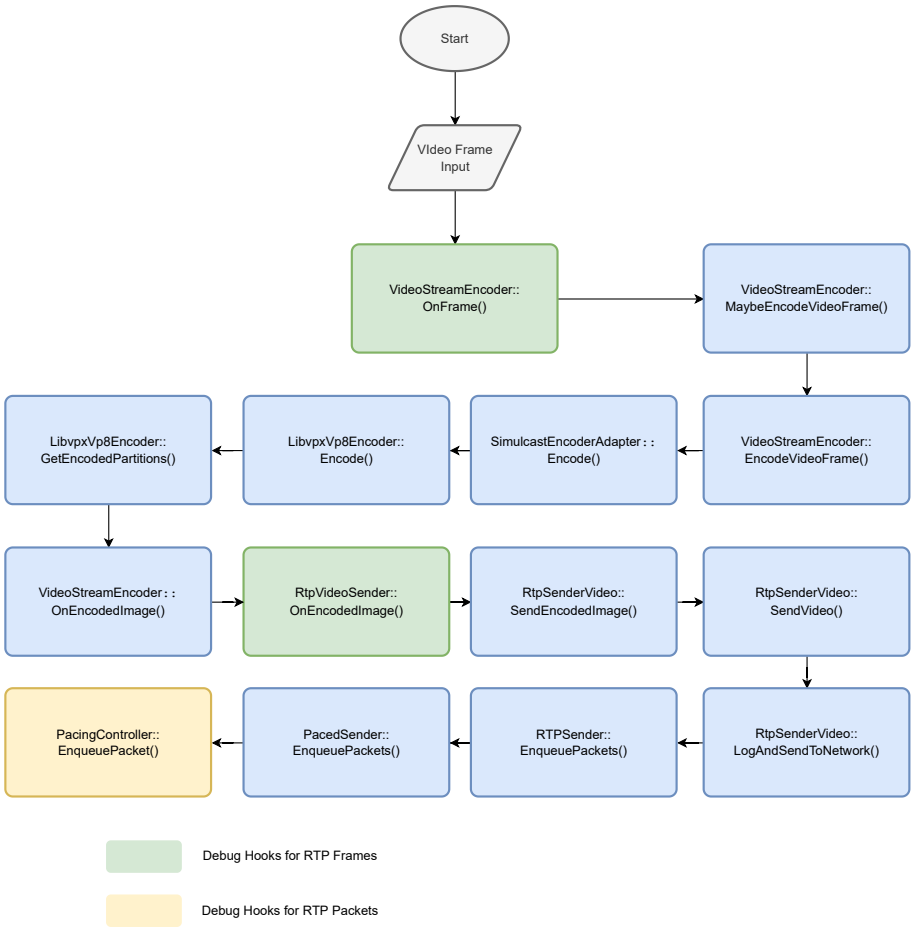


Figure 3.5: WebRTC Send Flow.

3. Finally, right before being routed outside of WebRTC's control and onto the actual screen of the user, the frames are inserted into a buffer to then be routed to the user's scren

The key point to retrieve here is that we want to make sure that our counters are placed right on each edge of the WebRTC path. What we mean by this is we want to make sure that we can account for a packet right as it enter the WebRTC scope and account for a frame right before it leaves the scope of WebRTC. This way we can make sure that any possible errors or packet loss that occurs inside of our host's WebRTC scope we can detect and understand why. It also makes sure that if the packet loss or any other error is due to some outside entity not related to our host, we can be aware of that. For example, if we see that in our sender end we are sending 10 packets to the network but only receiving 5 in the receiver end of the WebRTC scope, we are sure that any possible issue that is occurring is unrelated to WebRTC itself.

In Figure 3.5 we can analyze the sending flow of a frame in the WebRTC program flow:

1. The first function will receive a video frame from the input device, which could be a webcam or some virtual emulator.

2. WebRTC will then process the frame and encode it according to the configured video codec, in this example, VP9.

3. After the encoded process has been done, the frame is ready to be sent through the network.
4. The frame is then transformed into multiple packets that can then be sent through the network onto the receiving end.

We can also see that we have placed counters in three functions:

- *VideoStreamEncoder::OnFrame*: this is the first function of the WebRTC program scope to interact with a video frame, received from the video input, and so plays a major role in identifying if any errors on the frames could be due to the video input and not the actual WebRTC program.
- *RtpVideoSender::OnEncodedImage*: this is the function called after the encoding process of the video frame is complete, and so, from this point forward any error that could occur is not related to the video encoder in any way.
- *PacingController::EnqueuePacket*: lastly, this is the last function that will deal with the RTP packets generated from the encoded video frame, and so, any errors that occur after this point are due to external entities not related to the WebRTC program.

In summary, the goal of instrumenting WebRTC in this form is to allow developers and researchers to easily understand the basis of the WebRTC infrastructure and design without the need to read through pages and pages of technical documentation. Specifically, we intend to shed light into the most important and critical areas of code for the development of a multimedia covert streaming tools based on WebRTC. For this, it is important to be able to insert covert data without disrupting previous video encoding or other processes that can lead to errors and discarding of frame, worsening the tool's throughput.

Summary

This chapter has detailed the fundamentals of the WebRTC protocols and it's accompanying infrastructure. We explained the different stages of establishing a connection for the WebRTC protocol and introduced gateways as a solution to privacy issues and larger WebRTC video calls, allowing for better performance and scalability in larger video calls. Lastly, we explained our process for instrumenting the protocol. This chapter servers as the basis for the next chapter, as understanding WebRTC fully is key to developing our solution - TorCloak, which we will present in the next chapter.

Chapter 4

TorCloak

In this chapter, we present the design and implementation of TorCloak, our proposed new Tor Pluggable Transport that leverages WebRTC services to deploy covert channels to bypass censorship. We begin by describing the design goals (Section 4.1) followed by the threat model (Section 4.2). In Section 4.3, we present the architecture of our system and its components. In the following sections, we provide a more detailed description of the most key aspects and challenges of TorCloak, namely: managing the Bridge Infrastructure (Section 4.4), tunneling Tor Traffic through WebRTC (Section 4.5), and the extension to mobile platforms (Section 4.6).

4.1 Design Goals

The overall goal of TorCloak is to securely bypass Tor traffic censorship imposed by an adversary using advanced traffic analysis techniques. Such an adversary is assumed to actively attempt to detect streams that make use of TorCloak to bypass its censorship mechanisms and actively disrupt and tear down those streams. The design of TorCloak is driven by the following sub-goals:

1. **Unobservability:** A censor must not be able to distinguish regular WebRTC videocall streams from streams carrying covert data.
2. **Unblockability:** There must be significant collateral damage to a countries' social and economic status if a censor attempts to block the carrier WebRTC application upon which TorCloak sits.
3. **Video-carrier independence:** TorCloak's encoding strategy should be able to allow it to work under any WebRTC-based carrier application.
4. **Reasonable performance:** TorCloak must achieve sufficient performance for allowing most typical Internet tasks (e.g. exchange e-mails, upload files, watch standard to high-resolution videos).
5. **Uphold Tor's anonymity properties:** Since we are building our system to use Tor traffic, it is crucial that it maintains the many security and privacy features of the Tor Network and does not pose a threat to the Tor infrastructure.

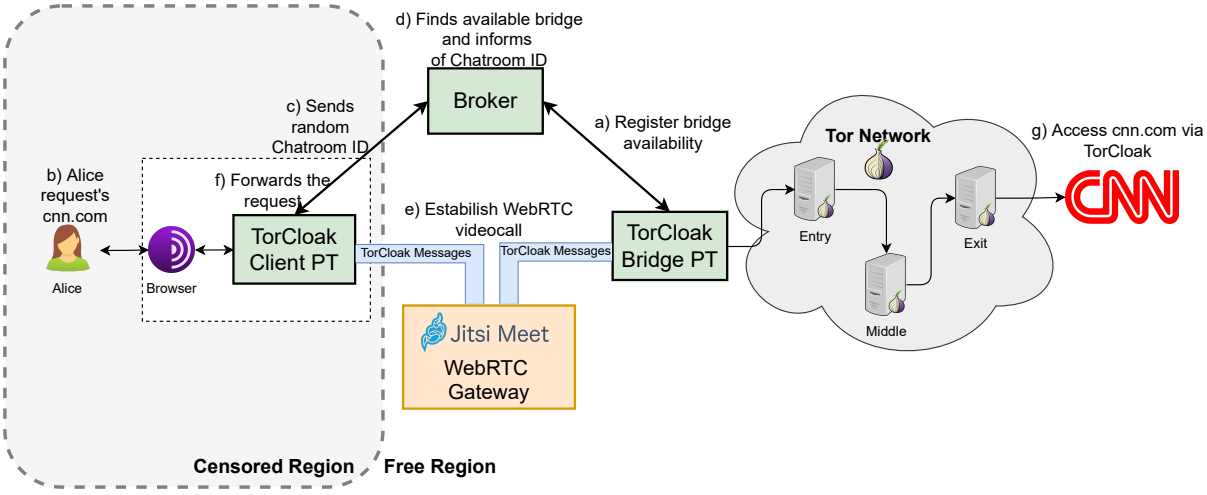


Figure 4.1: TorCloak Architecture.

4.2 Threat Model

In the context of TorCloak, the goal of the adversary is to detect and block the usage of the system, without jeopardizing legitimate WebRTC connection that can be vital to the country's economy. We assume that the adversary is a state-level censor, able to observe, store, interfere with, and analyze all the network traffic of the Internet infrastructure originated from TorCloak endpoints, if within the censor's jurisdiction. The adversary is also able to block generalized access to remote Internet services it deems sensitive, such as the Tor Network. The censor is considered to have advanced tools based on Deep Packet Inspection (DPI) and statistical traffic analysis to detect and block these services.

However, several attacks are out of scope. We assume that the censor does not have control and access over the used WebRTC gateways. In other words, the censor cannot observe the video streams and evaluate whether the video streams contain covert data or not. Secondly, we deem the adversary to be computationally bounded and unable to decrypt any encrypted traffic for services it does not control, such as Tor Traffic. The adversary's control is also limited to the network: it has no control over the software installed on end-user computers and does not have the power to deploy rogue software on these machines, with the purpose of monitoring systems on network edges. Thus, TorCloak's users and bridges are assumed to be executing trusted software. Also, as mentioned earlier, the adversary will only seek to rapidly disrupt and tear down traffic which is suspected of carrying covert channels, and it will refrain from blocking the carrier application all together, avoiding the blockage of an important and highly used service by the population and damaging its economy. Lastly, we also assume the censor has no control over the Tor network and its infrastructure, and so, cannot easily control or observe any of the traffic after it enters or exits the Tor Network.

4.3 Architecture

Figure 4.1 depicts the general architecture of TorCloak. TorCloak makes use of the WebRTC framework, which handles the signaling and establishment of video calls between the client and bridge. The system itself consists of three components (highlighted in green shade): *client*, *bridge* and *broker*. The client runs a Tor proxy that exposes a SOCKS interface to local applications, e.g., the Tor Browser or Tor's command line interface. The client then tunnels away local Tor traffic by embedding it into video frames using the Protozoa encoders and sending them through the video stream of a carrier WebRTC-based application (e.g. Jitsi Meet). At the other end of this tunnel, a bridge routes the then already decoded Tor traffic to a Tor relay so it can then be forward to its final destination. The broker coordinates all of the bridges composing TorCloak's bridge infrastructure and runs a directory service that allows users to locate available bridges.

In addition to users, there are two additional main actors in TorCloak's ecosystem: *bridge providers* and a *broker provider*. Bridge providers contribute to the TorCloak infrastructure by deploying the server hardware and running TorCloak's bridge software. These can be volunteers that wish to freely contribute to the TorCloak Project or private or public non-profit institutions willing to share spare resources for promoting freedom of speech on the Internet. The broker provider is a trusted authority whose purpose is to coordinate bridge membership. It is responsible for managing the new registers from bridge providers and making sure that community guidelines are met. These can include making sure the proper deployment of TorCloak is followed, the proper propagation of the bridge is achieved and, of course, the safety and anonymity procedures are followed. To deploy a TorCloak bridge, a bridge provider must: i) set up the server hardware, ii) install TorCloak's software, and iii) register the bridge in the broker service. The bridge can then listen for *rendezvous ID* (RID) sent by the broker, which corresponds to the address a particular TorCloak user client will join to establish a videocall session with the bridge. The rendezvous ID depends on the chosen WebRTC-based carrier application. For instance, in Jitsi Meet, it can take the form of a chatroom URL such as `https://meet.jit.si/torcloak_client1`. TorCloak users can then send the broker a randomly generated chatroom URL and join a video call on Jitsi Meet at the that chatroom URL. The broker will then request a bridge to join the same chatroom URL and being the the establishment of a TorCloak covert session.

To illustrate this process, Figure 4.1 shows Alice, a user located in a censored region controlled by a state-level adversary, using a TorCloak bridge located in a free Internet region. To access `www.cnn.com` through the Tor network in a censorship-resistant fashion, Alice must get access to the TorCloak PT client software (e.g., through some out-of-band channel) for her local platform, which can be desktop or mobile. She must then configure her Tor client to use TorCloak as the designated Pluggable Transport. Alice can then initialize her browser, which will cause the TorCloak client to generate a random RID and password on a chosen WebRTC platform (in our case Jitsi Meet) and send it to the broker. The broker will then pick an available bridge and send it the RID and password. The bridge and client can now join the chosen chatroom. It is important that the chatroom is password-protected so that only the user and the bridge can access the same room. The covert channel is then established between client

and bridge and the pluggable transport is ready to covertly tunnel Tor traffic through this channel. By employing Protozoa video encoding techniques, TorCloak will ensure that the WebRTC stream being transmitted over the network between the client and the bridge will preserve the same properties as a regular video call, therefore, offering strong resistance to traffic analysis attacks.

4.4 Management of Bridge Addresses and Membership

We begin our discussion of the main technical challenges that we had to tackle by focusing on the management of bridges' rendezvous addresses and membership. This management is essential to create a sustainable, easily scalable and censorship-resistant infrastructure.

1. *Looking up rendezvous addresses:* As described above, to establish a covert channel, the user must know in advance the chatroom's RID and password made available by the bridge for its connection. However, by constantly eavesdropping on the users' network requests, the adversary may try to intercept this information, join the same chatroom, and start snooping into the transmitted (altered) video frames. By intercepting covert Tor frames embedded in the WebRTC frames, an adversary could determine the presence of suspicious content and block the transmission. There is also the chance that, by performing data analysis during a sufficiently large period, a state-level adversary would be able to detect the transmission of such RIDs and correlate Bridges' IP addresses. The adversary could then blacklist such addresses and stop the user from being able to establish connections towards TorCloak bridges. So, there needs to exist a secure way for bridges to share their RIDs.

2. *Rotating rendezvous addresses:* Another problem related to chatroom RIDs is how to manage them and rotate them. Should RID addresses be rotated per user? Per session? Per bridge? This is important because reusing or maintaining the same RID for a long period or for multiple users increases the chance of the RID getting leaked or the allow the adversary to, via random guessing, discover a RID and monitor that specific chatroom indefinitely. Despite the video call rooms being password protected, so even if an adversary gets hold of a RID we cannot join the room, we can attempt to bruteforce the password. With enough computer power (available to most powerful censors) we can possibly bruteforce the password and join the room, giving him access to the covert session and breaking anonymity.

3. *Handling bridge churn:* TorCloak must also be able to deal with the addition or removal of bridges, and efficiently manage their distribution and workload. Bridge providers might decide to leave at any time, even during a covert data transmission session. The bridge attribution can also be done with the user's region in mind, to reduce latency. The broker must also assure that its bridges are available and functioning properly.

4. *Bridge authentication:* Since the bridge providers can be untrusted entities unless they are properly authenticated, there needs to be a mechanism in place that allows users to validate the identity

of the bridge provider before they decide to rely on their services for establishing covert channels. Otherwise, an adversary may attempt to deploy a malicious TorCloak bridge in the hopes of eavesdropping and exposing the user's transmissions and data.

We will now present our proposed solution to solve the challenges exposed above. Firstly, one way of securely and anonymously sharing the rendezvous chatroom IDs with the individual users is to use an out-of-band channel transmitting the RID using an application that is not based on Tor (otherwise it would be blocked by the adversary). Similar to the way we transmit covert data, we can pick a carrier application that i) does not depend on an ID to establish a connection, ii) is secure and encrypted end-to-end, and iii) is widely used thus offering strong resilience against blockage attempts. An example would be to use an encrypted email service (e.g. ProtonMail), chat application (e.g. Signal) or even Tor's Telegram and Email bridge sharing service [63] where the user could send a predefined standard format message to request to be issued a rendezvous chatroom ID. The user could then receive the ID and establish a WebRTC-based application video call to that chatroom ID and use TorCloak's software to start to exchange covert data.

To provide such an out-of-band channel, our initial approach was to consider Tor's pre-existing directory service for managing bridges, namely BridgeDB[1]. For TorCloak's purpose, BridgeDB would function similarly to the way it is currently implemented, but instead of helping users lookup the IP addresses of bridges available to be contacted, BridgeDB would provide them with RIDs from a host application (i.e. Jitsi Meet). This would allow the client's TorCloak to establish a WebRTC-based video call which would then serve as the covert channel for Tor traffic. Upon discussion with Tor's anti-censorship team, we realized that using BridgeDB would involve a large amount of changes to BridgeDB, which would make this approach cumbersome. Instead, we agreed to follow a structure similar to Snowflake's[30] proxy dissemination mechanism. Specifically, we deploy a broker, whose job is to connect TorCloak Clients to TorCloak bridges. This consists of a simple API which clients will use to share a RID they are intending to join, alongside with the RID's corresponding password. The broker will find an available bridge and forward that Chatroom ID so they can both join and initiate a video call. To avoid the possible blockage of the broker by a censor, in the near future, we intend to deploy a mechanism similar to Domain Fronting[14]. By offloading most of the heavy computing to the client side we achieve two important goals: i) decrease the network and computing load on the bridges, making it so that more individuals can be motivated to deploy TorCloak's bridges since it will have a very low impact on their infrastructure and accompanying costs, and ii) by compacting the information shared between client and broker, we highly reduce the cost of deploying a solution with Domain Fronting, which, when used to transfer high amounts of data, can amount to a large monetary cost [14].

The rotation of the chatroom RIDs must also be managed and controlled to ensure maximum resistance again censorship. The main problem here is to a) make the bridges available for many different clients simultaneously, while b) making it hard to block the connection towards TorCloak bridges. Leaving RIDs open for long periods of time or available to many different users will naturally make RIDs more prone to be spotted by the censors. On spin up, the TorCloak Client generates a RID for that session.

[1] `https://bridges.torproject.org/`

The client can then publish this RID on the broker, which will then transmit it to an available bridge. The bridge's only work is to join the chatroom corresponding to that RID and establish the covert session. The RID is only valid in a per session per user basis. This mean that after that specific videocall session is terminated, a new RID must be generated and shared to initiate a new session. Bridges themselves also advertise their public key certificates to the broker. The TorCloak client can then download such certificate from the broker and verify the bridge's identity using it. Upon startup, the client will request the bridge to send an authenticated message, which can then be used to authenticate the chosen bridge (hence solving the fourth challenge listed above).

Ideally, TorCloak should accrue a reasonably-sized pool of bridges. In addition, the more geographically disperse they are, the better will be the availability and performance of TorCloak. To this end, we plan on collaborating with Tor core developers and non-profit foundations that embrace freedom of information, such as the Freedom of the Press Foundation[2].

4.5 Tunneling Covert Tor Traffic through the Bridge

Tunneling Tor traffic through WebRTC covert channels while preserving the compatibility with the Tor pluggable transport API requires a non-trivial integration of complex and heterogeneous pieces of software. Figure 4.2 sheds light on how we design the internals of TorCloak's client and bridge, representing also how their subcomponents interact with each other during Alice's visit to *cnn.com* using our system. These subcomponents perform various functions, most notably: i) SOCKS [64] proxy interfacing with external applications (i.e. local Tor client), ii) client and bridge controller to orchestrate all of the components, iii) Tor pluggable transport specification implementation, and iv) WebRTC-covert channel management and data transmission. The Tor proxy is configured to use the TorCloak pluggable transport. Upon starting, it instantiates a TorCloak gateway, which is based on the Protozoa architecture (see Figure 2.5). The gateway exposes an internal SOCKS proxy to receive the locally generated Tor traffic. Tor cells can then be encoded as TorCloak messages into WebRTC frames and sent through a WebRTC-based carrier application video call tunnel. The upstream and downstream hooks intercept, respectively, outgoing and incoming WebRTC frames to be processed by TorCloak, and to be encoded or decoded accordingly. The client/bridge controller is responsible for the coordination of all these components, processing local Tor events and exceptions, and performing TorCloak-specific protocols to synchronize the client and the bridge.

In particular, the main integration challenges we faced are as follows:

1. *Retrofit Protozoa with the existing Tor pluggable transport software:* To allow TorCloak to be readily used by Tor users, the pluggable transport has to be able to interact with the whole already existing Tor infrastructure and software while leveraging the covert WebRTC channels. This required us to adapt the protocols previously implemented by Protozoa to suit the specific specifications and restrictions of Tor's pluggable transport API.

[2] `https://freedom.press/`

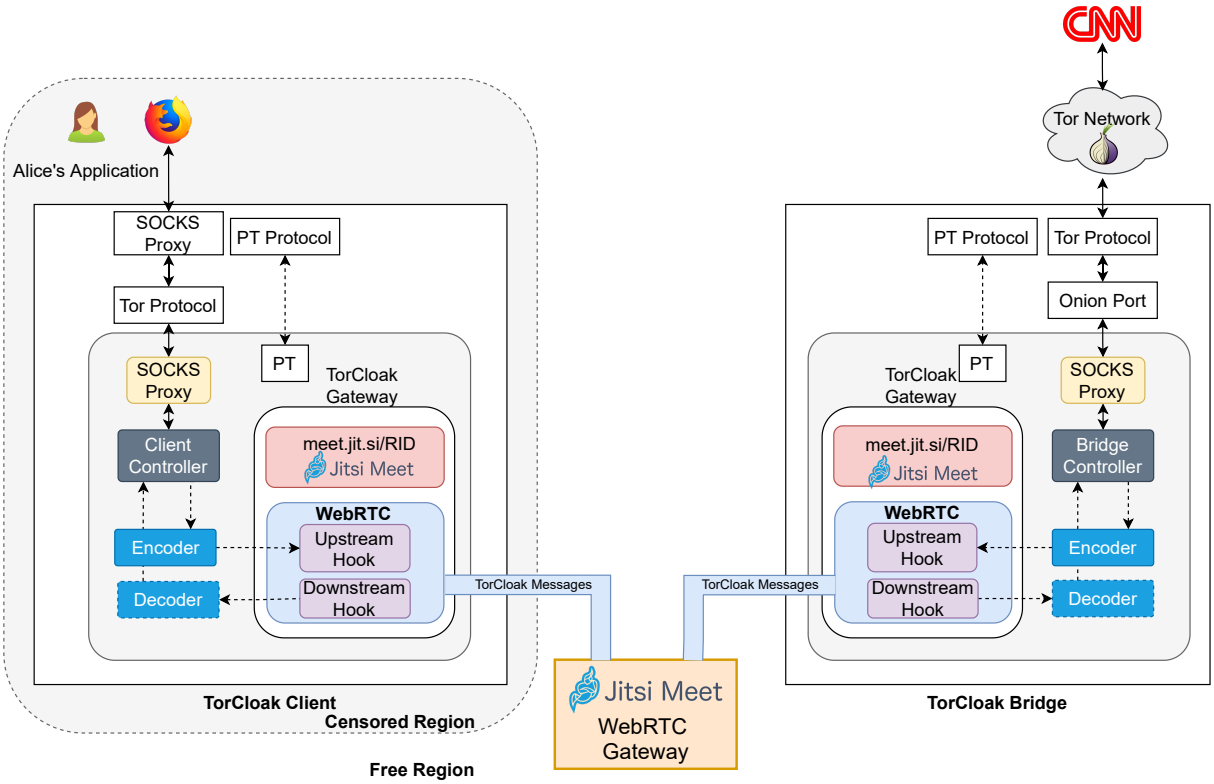


Figure 4.2: TorCloak's Main Components.

2. *Traffic analysis resistance:* We needed to consider that transmitting Tor traffic through the covert channel might create traffic patterns susceptible to traffic analysis attacks, since the frame size and delivering timings might change. We needed to assure we can produce indistinguishable patterns in the transmitted packets to preserve unobservability.

3. *Performance degradation:* It is also possible that Tor's network performance will decrease when tunneled through Protozoa channels. Despite Protozoa's improvements in the covert channel throughput, there exists a bottleneck when streaming traffic through WebRTC-based covert channels, which is bounded by the throughput achieved by the carrier video call data streams. It stands to reason that the Tor circuits tunneled through these channels will also be throttled. Assessing how Protozoa's transmission mechanism can be adapted to Tor's specific data transmission scheme was key to developing an efficient implementation of TorCloak.

Faced with the aforementioned challenges, we now present our solution focusing on two main aspects: VP9 codec adaptation and covert frame encoding and decoding.

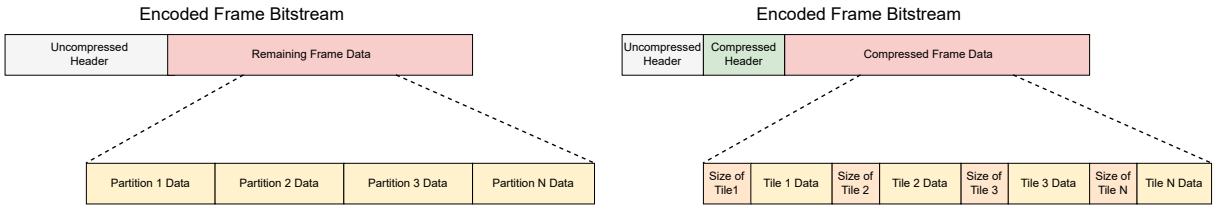
Covert frame encoding and decoding: Similarly to Protozoa, TorCloak's encoding mechanism replaces the bits of the encoded video signal, after it has been processed by the video encoding engine. This is done by modifying the WebRTC stack bundled into the Chromium Browser. The WebRTC stack contains several built-in codecs (VP8, VP9, etc) which process the video signal of local applications that use the WebRTC Javascript API [7]. To access the video frames generated by the WebRTC application and implement encoded media tunneling, the WebRTC Protozoa stack includes two included hooks that

can intercept the processing of the media streaming different directions, i.e., upstream or downstream. The upstream hook intercepts outgoing frame data, i.e., from a local camera device to the network. It is placed after the raw video signal has been processed by the video engine, and right before the frame data is passed over to the transport layer where SRTP packets are created, and sent to the network. The downstream hook intercepts incoming frame data, i.e., from the network to the local screen. It is placed right after the transport layer has finished reconstructing an encoded frame sent in multiple network packets, and right before handing it over to the video engine to be decoded and rendered on screen. We make use of a data structure called *encoded frame bitstream* (EFB) (represented in figure 4.3). This is the frame format that Protozoa also uses and it naturally separates the frame's zones where we can encoded data and those where encoding data will most likely harm the functioning of the video stream, leading to losses. We chose this data structure because, besides its header, it contains partitions that only store the encoded video bytes, and nothing else. Furthermore, this data, after being generated by the video encoding engine, is no longer modified, and is only encrypted and protected with authentication markers before being assembled into packets. It makes it a suitable place to encode our covert data, ensuring the covert data is not modified by the application and still is encrypted and authenticated. The only thing we need to keep in mind is, while it is possible to fully replace the content of the EFB field, the undisciplined corruption of a frame bitstream can prevent the video decoder in the WebRTC downstream pipeline from correctly decoding video frame data at the receiver's endpoint. We verified that in such situations, WebRTC triggers congestion control mechanisms in the downstream pipeline for ensuring the reception of video. In particular, it advertises a Picture Loss Indication (PLI) in the accompanying RTCP control channel [65], aimed at requesting the re-transmission of a key frame upon being unable to decode the corrupted frame data. By advertising PLIs and requesting retranmissions, the resulting channel bandwidth would be severely reduced and the resulting traffic patterns produced by the WebRTC application would make TorCloak vulnerable to traffic analysis. To overcome this problem, the downstream hook feeds the WebRTC video decoder with a pre-recorded sequence of valid encoded frames instead of the corrupted frames received over the network. This allows us to establish a covert channel without triggering any frame corruption control mechanisms.

In conclusion, since we are assembling our encoding and decoding strategy fully detached from any codec or special data structure, we can theoretically encode any type of data into the video frames. We are, in a simple way, replacing bytes that, in a normal setting, would represent video frame data, but in our solution will represent Tor traffic data.

VP9 codec adaptation: When adapting our prototype to newer versions and services, we discovered that several WebRTC-based videocall applications, such as Jitsi Meet, had moved away from older video codecs like VP8 and adopted newer and improved ones such as VP9. This meant that, as mentioned above, we needed to to adapt Protozoa's EFBs to the new data format of VP9. In Figure 4.3 we can see the main difference between the structures of both codecs: the addition of one extra compressed header, joining the already pre-existing uncompressed header.

So, in order to maintain the codec's headers, and avoid corrupting the frame and triggering control



(a) VP8 (b) VP9

Figure 4.3: Format of VP8 and VP9 Encoded Bitstream.

mechanisms, we need to adjust our offsets to account for the new header. Using the VP9 Decoding Specification [66] together with WebRTC's VP9 decoding structure[3], we were able to determine the correct offset calculation to be able to replace the bytes corresponding to pixel data without disrupting the headers, maintaining the frame's integrity. The general structure of a VP9 TorCloak bitstream is represented on Figure 4.4. Here we can see that TorCloak replaces the EFBP payload containing carrier video bits with covert data, while maintaing the header structure intact for the decoding process.Furthermore, we modified WebRTC's `encoded_frame.h` structure to include information be able to retrieve information about the codec used to encode the frame. This allows us to determine the codec being used in runtime and adapt our encoding strategy on the fly, enabling TorCloak to support multiple video encoding codecs. Lastly, and as mentioned above, we empirically verified that the VP9 encoded frames highly differed from the VP8 ones. This meant that, if we feeded the video decoder our previously recorded valid encoded frames corresponding to the VP8 codec, WebRTC frame corruption control mechanisms would be triggered and the frame discarded, hindering our performance and traffic analysis resistance. To overcome this issue, we re-recorded the sequence of valid encoded frames, for the VP9 codec. Additionally, by using our information about the codec being used to encode the frames for a specific session, we are able to, in runtime, switch the pre-recorded frames we are feeding the video decoder, to adapt to different video codecs.

4.6 Extension to Mobile Platforms

Although we have only implemented a desktop version of TorCloak by the time of the writing of this document, we intend to develop a mobile version in the near future. For the desktop setting, we leveraged Protozoa's mechanisms for tunneling arbitrary IP-traffic, which make use of Linux *namespaces* to create virtual network environments. However, in a mobile operating system such as Android, we cannot adopt this approach since user-level applications do not have the privileges to access kernel-level operations. As we want TorCloak to be installed and used by common Internet users, we require a user-friendly alternative, compatible with the existing Android API [67].

For this setting, we intend to follow a similar approach to already mobile-friendly PTs, following a similar approach to the one used to create ORBot[4]. ORBot can open a HTTP or SOCKS proxy to

[3] `https://webrtc.googlesource.com/src/+/ffd9187dc0d9211ad52173bf0daa5001ca7d45ee/modules/video_coding/codecs/vp9/libvpx_vp9_decoder.cc`

[4] `https://guardianproject.info/apps/org.torproject.android/`

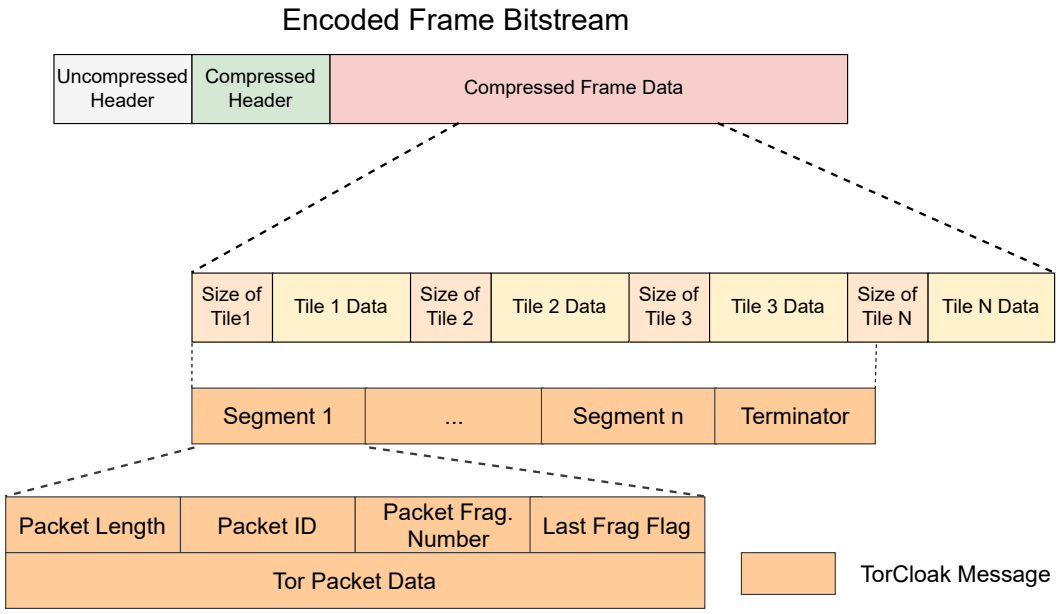


Figure 4.4: Format of VP9 encoded bitstream with replaced TorCloak data.

the local mobile application and route their traffic to the Tor Network. Recent versions also include the support for Pluggable Transports like Snowflake. We intend to use an available Android Pluggable Transports library also developed by the GuardianProject[5]. This library includes code used to develop ORBot and allows developers to add features like VPN-mode to their pluggable transport. The VPN mode allows mobiles users to route all of their device's traffic through the application, similarly to how a VPN works. It also includes pre-compiled libraries that can be used to perform regular Pluggable Transports functions in an android setting.

The main challenge we face for the Android platform is tied to the implementation of the WebRTC hooks and video feed, required for TorCloak's functioning. Currently, in our desktop setting, for TorCloak to work there needs to be a i) modified Chromium browser with our hooks and ii) a video feed, either captured directly from a webcam or replayed from a previous video recording. However, Android is composed of multiple security features that highly restrict what system resources and other applications' data have access to in order to prevent malicious applications from fully controlling the device. This highly restricts the usage of a modified Chromium Browser, complicates the automation of the process through which TorCloak users would join a chatroom, and creates obstacles in the usage of a camera emulator that can feed pre-recorded video to the WebRTC software stack. For example, by default, Android does not allow the installation of any application from a source other than the official Google Play Store. So installing a modified Chromium Browser without the permission for it would already hinder our process. Lastly, we also need to adapt most of our C++ codebase to the Android platform and making sure the libraries we use are compatible and can generate binaries suitable for a mobile setting. For this we can also make use of Android's Native Development Kit (NDK) [68], which consists of a set of tools that allows for the use of C and C++ code with Android, and provides platform libraries that can be used to manage native activities and access physical device components, such as sensors and touch input. While the above list of implementation challenges is rather substantial, it is likely we

[5] `https://github.com/guardianproject/AndroidPluggableTransports`

will be able to take advantage of the codebases of existing PTs that are specifically geared to the mobile environment (e.g., Snowflake Mobile Proxy [69]) to develop a mobile version of TorCloak.

Summary

This chapter has detailed the design of TorCloak, an Internet censorship circumvention system that leverages the video channel of existing WebRTC applications to tunnel covert Tor data. The design of TorCloak and its accompanying infrastructure allows Tor users to easily find and connect to a bridge and establish a covert channel that will allow them to use the Tor network undetectable. One of the primary concerns of the system's design is to remain undetectable, allowing the safe circumvention if censorship, while still being reasonably accessible for those who wish to use. Another concern is to make the system as abstract from the carrier application as possible, allowing it to be deployed in multiple ways and with multiple WebRTC service providers easily. We also highly detailed the security concerns and safety measures took to make sure our code is as safe as possible. The next chapter will present a comprehensive description of biggest challenges in the implementation process.

Chapter 5

Implementation

This chapter addresses the implementation details of our TorCloak prototype. Section 5.1 gives an overview of the implementation. Then, we provide several details about the implementation of TorCloak's main components, in particular the Pluggable Transport (Section 5.2), the SOCKS Proxy (Section 5.3) and the upstream/downstream pipes (Section 5.4). Next, we clarify the adaptions made to the WebRTC framework (Section 5.5) and the debugging process involved (Section 5.6). Lastly, Section 5.7 describes the full auditing process our prototype was subjected to and the vulnerabilities found.

5.1 Implementation Overview

We developed a TorCloak in about 4000 lines of C++ code. This includes the whole Pluggable Transport related code, that handles Tor events, as well as the instrumentation of the native WebRTC codebase of the Chromium browser v100.0.4896.127, a stable release from April 2022. TorCloak requires the proper establishment of a WebRTC video call session for embedding data into encoded frames sent over the network. For this purpose, WebRTC must be able to access a video feed that can be directly obtained from the physical camera available in the system. Alternatively, it is possible to setup a camera emulator by using the v4l2loopback kernel module [70] and feed recorded video with the help of the ffmpeg video library [71]. We will now describe in more detail about the most critical aspects of the implementation of TorCloak.

5.2 Pluggable Transport

In order to avoid changing Tor protocols we designed TorCloak as a Tor pluggable transport (introduced in Section 2.2). Indeed, Tor developers have published a PT specification [72],written in an RFC-like document, which details how PTs can be integrated into Tor and how the Tor binary should interact with PTs (e.g., starting and shutting down the activity of a PT).

The activation of TorCloak following the PT API is depicted in generally in Figure 5.1 : each Tor instance loads a configuration file – denoted torrc – containing TorCloak executable path, command line

arguments and Tor mode operation (e.g. bridge or client). When launched with this configuration file, Tor starts TorCloak as a subprocess. Several command line variables are set by Tor to the TorCloak executable, exposing Tor configurations, e.g. the bridge remote address. TorCloak reports back to Tor by writing success or error messages into the stdout in a standardized format, which Tor, the parent process, is able to capture and read, e.g. local port where TorCloak accepts connections from local Tor (sketched as the red/green arrows in Figure 5.1). During shutdown, Tor also takes care of a performing a controlled shutdown of TorCloak, making sure the process's memory is freed properly. This process is accomplished by closing the stdin of TorCloak. TorCloak, upon detecting a closed stdin, begins to cleanly shutdown all of its components. Apart from the initial configuration, is up to TorCloak to establish a remote WebRTC video call session connection with the TorCloak bridge to send and receive covert Tor data. In this fashion, TorCloak can be fully managed by the local Tor client, configuring all of its options and preferences automatically, with no need of any user interaction.

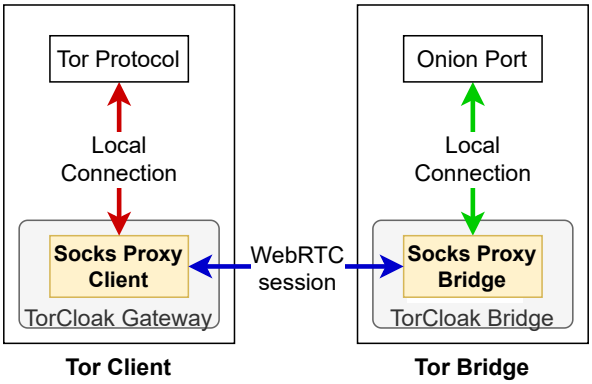


Figure 5.1: Pluggable Transport configuration and TorCloak Socks Proxy.

5.3 SOCKS Proxy

As pointed out in Figure 5.1 and detailed in Section 2.2, Tor plugabble transports make use of SOCKS proxies to communicate with the Tor Client and exchange data between them. Specifically, the TorCloak Client gateway exposes a SOCKS [64] as an endpoint for receiving Tor traffic and the TorCloak bridge exposes a reverse SOCKS proxy that accepts local Tor connections. Thus, Tor itself exposes a SOCKS proxy that allows external application to route their traffic on top of the Tor Network. When a pluggable transport is in use, Tor then routes it's traffic to the pluggable transport's SOCKS proxy also.

Internally, the proxy (see Figure 5.1) was modified to initialize the connection, following every step defined in the standard, and create a local file descriptor corresponding to the local Tor connection. This can then be passed to the WebRTC portion of the program to encode the data received on the proxy into the encoded frames of the WebRTC video call. The SOCKS proxy servers as a middle-man between the local Tor application and the WebRTC encoding/decoding hooks.

5.4 Upstream/Downstream pipes

One of the key aspects of TorCloak is the communication between the actual TorCloak process and the modified Chromium browser process. These need to exchange data in order to encode locally generated Tor traffic into the already encoded video frames of WebRTC. For this purpose, we employ two pipes for receiving upstream and downstream messages from the WebRTC layers. These pipes consist of FIFO (first in first out) queues, that i) send the frame data received over the WebRTC session running in the Chromium browser process, to then be decoded by the TorCloak process and ii) and receive the frame data with the encoded data from the TorCloak process and send it over the WebRTC session running in the Chromium browser process.

This implementation allows the two separate process to run separately while still being able to share data between them in a sequential fashion. This allows TorCloak's upstream hook to intercept outgoing frame data, i.e., from a local camera device to the network and the downstream hook to intercept incoming frame data, i.e., from the network to the local screen.

5.5 Adapting WebRTC codebase versions

The next step in the development of TorCloak was the adaptation of the original Protozoa code to newer versions of WebRTC. Originally, Protozoa was developed to operate with the WebRTC version present in Chromium browser version v79.0.3945.117, a version from January 2020. We chose Chromium version 100.0.4896.127, a stable version from April 2022, around the time we began the development. We then proceeded to adapt the preexisting decoding and encoding hooks to this newer version.

Despite the version being more than two years apart, the actual places of code that we require to perform our covert data encoding and decoding suffered minor changes. This may be due to the fact that these are core functions of the WebRTC multimedia pipeline and so cannot be highly modified without requiring a lot of changes to current software that depends on older versions. This suggests that TorCloak will remain compatible (albeit with potential minor tweaks) with upcoming WebRTC releases.

We can see in Figure 5.2 two different versions of the same `VideoReceiveStream::OnCompleteFrame` function: on the left the version corresponding to the WebRTC version in Chromium version 79 and on the right the one present in WebRTC of Chromium version 100. This function is called once WebRTC is finished assembling a complete frame from the received network packets. It is the first function to process a fully assembled frame, and so it is the optimal place to place our decoding hooks (explained in greater detailed in Section 3.3). Furthermore, the data structure received on this function, `EncodedFrame`, does not suffered any major changes between the versions. This is also key in our decoding process, allowing us to extract the same information from this data structure whatever the WebRTC version is.

The same happens in the encoding hooks of TorCloak, injected in function `RtpVideoSender::OnEncodedImage` of the `rtp_video_sender.cc`[1] file. As explained in detail in Section 3.3, this is the first function to receive a fully encoded video image from the video encoder engine. This makes it the ideal place to intercept

[1] `https://webrtc.googlesource.com/src/+/ffd9187dc0d9211ad52173bf0daa5001ca7d45ee/call/rtp_video_sender.cc`

```
void VideoReceiveStream::OnCompleteFrame(
    std::unique_ptr<video_coding::EncodedFrame> frame) {

    RTC_DCHECK_RUN_ON(&network_sequence_checker_);

    int64_t time_now_ms = rtc::TimeMillis();
    if (last_complete_frame_time_ms_ > 0 &&
        time_now_ms - last_complete_frame_time_ms_ >
kInactiveStreamThresholdMs)
    {
        frame_buffer_->Clear();
    }
    last_complete_frame_time_ms_ = time_now_ms;
    ...
}
```

```
void VideoReceiveStream2::OnCompleteFrame(
    std::unique_ptr<EncodedFrame> frame) {

    RTC_DCHECK_RUN_ON(&worker_sequence_checker_);

    int64_t time_now_ms = clock_->TimeInMilliseconds();
    if (last_complete_frame_time_ms_ > 0 &&
        time_now_ms - last_complete_frame_time_ms_ >
kInactiveStreamThresholdMs)    {
        frame_buffer_->Clear();
    }
    last_complete_frame_time_ms_ = time_now_ms;
    ...
}
```

Figure 5.2: Comparison of the same decoding function in different WebRTC versions.

this image to replace its pixel data with our covert data. Once again, both version of WebRTC contain very similar implementation of this function, allowing us to mostly reuse the code for the encoding hook.

In summary, by placing the encoding and decoding hooks in the main function of the WebRTC flow we can not only have access to all of the information we need to replace the frame's data, but also can further future proof TorCloak since it is unlikely that this function will be the target of large changes in future versions of WebRTC.

5.6 Debugging WebRTC gateways

After updating TorCloak hooks to newer WebRTC versions, we faced some problems when deploying our solution in Whereby[2], the same media platform used during Protozoa's original evaluation. Specifically, prominent WebRTC video call service providers have started to deploy WebRTC gateways to improve their system's performance and scalability, while also addressing some security concerns (see Section 3.2). Unfortunately, Whereby is not open-source and exposes very little documentation on its infrastructure, hiding details on how the gateway is implemented and what function it serves. For this reason, we sought to find more concrete details about the behavior of WhereBy's WebRTC gateway.

Debugging WebRTC sessions and gateways revealed to be an arduous and time consuming task. However, with the help of the instrumentation framework we described in Section 3.4, we were able to deconstruct each portion of the WebRTC encoding and decoding flow. Upon that, we noticed that, when using the Whereby platform, after a few minutes of use, only approximately less than half of the frames and packets sent by one of our endpoints were actually being received in the other end. By analysing the endpoints in our WebRTC call, we were able to understand it was not actually a peer-to-peer call but instead there was a third-party entity to which our endpoints were sending their data, corresponding to Whereby's WebRTC gateway.

With this information in mind, we concluded that the WebRTC gateway implementation used by Whereby employs some traffic control to ensure the video calls attain a high quality and performance. While this generally increases call quality for usual clients, it complicates the process of building a fast

[2] `https://whereby.com/`

Identifier	Description	Risk
RPT-01-001	World-read/writable encoder/decoder pipe files	Low
RPT-01-002	Potential DoS via absent last fragments	Medium
RPT-01-003	Out-of-bounds read in gatherPayload	Medium
RPT-01-004	Out-of-bounds read in retrieveDataFromFrame	Medium
RPT-01-005	Outdated and vulnerable WebRTC in Google Chrome	Medium
RPT-01-006	Lack of return value checks in several functions	Info
RPT-01-007	Discouraged usage of NF_STOLEN verdicts	Info
RPT-01-008	Unbounded Size of Stack Buffer may result in DoS	Medium
RPT-01-009	Out-of-bounds read in gatherPayload by non-termination	Medium
RPT-01-010	Injection of Fragments into Packets	Info
RPT-01-011	Memory Leak in PacketHandlerThread leads to potential DoS	Info

Table 5.1: Security vulnerabilities found during audits.

and reliable covert channel over WebRTC. Overcoming this issue would require a dedicated mechanism to track frame loss and request re-encoding of information lost in the discarded frames. To circumvent this necessity when building our TorCloak prototype, we opted to focus on the usage of Jitsi Meet[3], an alternative WebRTC video chat application which, to the extent of our knowledge and experimentation, does not apply any sort of traffic control mechanism at its WebRTC gateways. Jitsi Meet allowed us to ensure that the majority of frames sent by an endpoint would be received at the other call's endpoint.

5.7 Security Audits

This section describes the process behind the two security audits we subjected our code to, to ensure it our code is solid and has no security vulnerabilities. The audits were completed by the Cure53[4] team in weeks 25 and 36 of 2022. The code was shared on a common repository, also containing extensive documentation explaining the scope and threat model of the project. It also included the description of each file and what functions it contained. We kept in touch the whole week with the team via Element[5], a matrix-based chat, to assist with any questions in the code or program flow.

Table 5.1 contains a listing of all of the 11 security vulnerabilities found during both audits. In total, the team discovered the following vulnerabilities in four different categories: 0 high risk, 6 medium risk, 1 low risk and 4 information. This were all thoroughly identified and described in two complete reports of the audits made available by the Cure53 team, which also included the mitigation suggestions for each vulnerability. The team also conclude in their report that our code "already exhibits a first-rate security foundation, with only a few minor improvements required to achieve an exemplary security posture" [73].

Next we generally describe each vulnerability and the steps taken to mitigate it:

RPT-01-001: Refers to the encoding and decoding pipe file having open file system permissions. Fixed by allowing them to be read, modified and executed only by their owner.

[3] https://meet.jit.si/
[4] https://cure53.de/
[5] https://element.io/

RPT-01-002: This vulnerability described a possible memory issue. In the impossibility of sending an entire Tor packet in one go, TorCloak fragments the data and sends it though the channel, where it will be reassembled in the receiving end. In the receiving side, the fragments are tracked and stored in memory until all of them arrive and the packet can be reassembled. However, if all of the fragments never fully arrive and the packet cannot be fully assembled, the fragments will sit in memory indefinitely. This can possibly render the application out of memory and induce a Denial of Service. To this end, we introduced a memory management system that tracks the fragments sitting in memory and, if stored for longer than one minute, deletes them. Even with large packets and larges amount of fragments, it should only take TorCloak a few seconds to fully send and assemble all fragments, so a minute is quite sufficient. The basic implementation is the following:

```
void ServerThread::checkUnusedPackets(){
  [...]
  auto iter = _dataStore->begin();
  while (iter != _dataStore->end()) {
      std::chrono::duration<double > duration =
        std::chrono::high_resolution_clock::now() - iter->second.getLastRecvTimestamp();
      if(duration > sixty_seconds)
          _dataStore->erase(iter->first);
      ++iter;
  }
}
```

RPT-01-003: Exposes an out-of-bound reads in our pipeline. When reading fragments from the decoding pipe, we first read the length of the data in the pipe. Subsequently, we read the first two bytes of the fragment, that encode the fragment length. The issue here is there was no verification that the fragment length was not bigger than the length of the data in the pipe. By reading this size, we could be reading out-of-bounds since the fragment length could be bigger than the actual size of the pipe data. This was fixed by adding the appropriate verification.

RPT-01-004: Similar to RPT-01-003, in our Chromium Hooks, for debugging purposes, we were logging the frame buffer size of the received frames. However, by doing this, we were calculating our packet size by accessing indices of the frame buffer, but failing to validate if those indices are smaller than the total frame buffer length. This could be mitigated by adding the proper verification, but since it was only for debugging purposes, in future iterations the code was removed as it no longer served any purpose.

RPT-01-005: This issue refers to the fact that, since the beginning our development of TorCloak, newer and more secure versions of Chromium have been released, including some that mitigate some existing vulnerabilities. By using a fixed version of Chromium we introduce the possibility of having some vulnerabilities in the Chromium source code, that were not known when the version was released. To this

end, we intend to developed a mechanism to keep our Chromium and WebRTC versions as updated as possible. At the time being, however, for development and testing purposes, we are freezing the version we use.

RPT-01-006: In a more general note, the Cure53 team identified some general issues when not properly handling return values in some functions. To this end, we reviewed the code entirely and properly handled and verified each return value to make sure all possible paths of the program's flow of execution are being handled.

RPT-01-007: In earlier version of TorCloak, and similarly to Protozoa, we were using *libnetfilter* [74] queue to receive locally generated Tor traffic. One of the arguments when configuring the queue is called the verdict, referring to the destiny of the packet after it has been intercepted. To this end we were using the *NF_STOLEN* argument to discard the packet after it's intercepted. As pointed out by Cure53, the *libnetfilter* documentation indicates that *NF_STOLEN* should not be used[6] as it is specially treated in the kernel and may lead to memory leakage in older Linux Kernel versions. As suggested in the documentation, we replaced it with the *NF_DROP* argument.

RPT-01-008: When reading frame payloads from the decoding pipe, we first read the first integer from the data to determine the payload length, and then proceed to allocate an array and set its memory to zero according to this length. However, if an attacker attempts to send a large value of payload length, this will result in a segmentation fault, since there is no way we can fill that much memory, and correspond to a denial-of-service attack. As suggested by the audit team, we set an upper bound when reading this size. If the payload length is too large, it will just be ignored and the frame will not be read.

RPT-01-009: This issue is directly related to vulnerability RPT-01-003. Since a single frame can contain several data fragments, our code contains a while loop that keeps reading the fragments until the size read from the first indices of the frame buffer is 0. However, if the read size is never 0, the loop will never stop and continue to read out-of-bounds data once again. This was fixed by checking if the amount of data read is below the frame length. Once it reaches the frame length, the reading of data stops, avoiding reading outside of the frame's bounds.

RPT-01-010: As mentioned before, akin to Protozoa, an earlier version of TorCloak made use of Linux's network namespaces for transparent interception and manipulation of locally generated Tor packets. The packets were sent through the WebRTC session and, after arriving at the bridge side of TorCloak, were forwarded to their final destination using a SOCKS proxy. The issue here is that: if an attacker modifies the destination IP address to a local IP address, it can effectively have access to local hosts on the network, which is undesirable. However, since TorCloak does not rely on this mechanism in future version, the isssue is mitigated (see Section 7.2).

[6] `https://manpages.debian.org/testing/libnetfilter-queue-doc/nfq_create_queue.3.en.html`

RPT-01-011: On the sending side, when creating a data structure composed of the covert data to be sent and some necessary metadata, we have the following code:

```
PacketData* packetData = new PacketData(pktID, packet_data_copy,
                                        static_cast<uint16_t>(packet_length), 1, 1, 0);
```

This `packetData` variable is allocated in the heap and is filled with all of the necessary metadata as well as the actual Tor packet data. In the end of the function, it is written to the encoder pipe, but its memory is never freed. After a lot of minutes of runtime, this leads to a memory leak which can result in a possible Denial of Service. To fix this, we free the memory after we write its value to the pipe, and it is no longer needed.

Summary

This chapter has described the implementation details of the TorCloak prototype. The pluggable transport components, together with the SOCKS proxy and the encoder/decoder pipes allow TorCloak to setup a covert channel inconspicuous to the user. Furthermore, our better understanding and debugging of the WebRTC protocol and VP9 video codec allows us to better prepare TorCloak for the future. Finally, the two security audits have allowed us to improve our code's strength and security, making sure users will remain safe and anonymous when using our tool. The next chapter will present TorCloak's full experimental evaluation and results.

Chapter 6

Evaluation

This chapter describes the experiments that were carried out in order to evaluate TorCloak. Section 6.1 starts by detailing the evaluation methodology. We then expose our evaluation goals and approach, while also describing the metrics used to perform such evaluation (Subsection 6.1.1). Next, we present our experimental testbed and video dataset used (Subsection 6.1.2). We follow it up by presenting and analyzing our experimental performance results in Section 6.2. Lastly, Section 6.3 compares TorCloak with related work.

6.1 Evaluation Methodology

This section describes our evaluation methodology for assessing the quality and performance of TorCloak. First we will describe the goals and approach of our evaluation. Then, we present the experimental testbed we designed for performing our experiments and the metrics we used to assess the quality of our solution.

6.1.1 Evaluation Goals and Approach

Our main evaluation goals are the following: evaluate the performance of TorCloak's covert channel when in comparison to the normal Tor Network, and ii) compare our system's performance with other similar systems and assess its ability to be used for typical internet tasks.

To perform these experiments, we adopt the following set of performance metrics: we leverage *throughput* and *latency* as the metric of performance of the covert channel. This allows us to have a good view on the system's capability to be used to perform regular internet tasks, that mostly depend on the channel's bandwidth and latency

To measure the throughput and latency of our covert channels tunneled through WebRTC video calls channel, we are leveraging *iPerf* [75] and *HTTPing* [76]. This enables us to stress the covert's channel capacity and latency.

6.1.2 Experimental Testbed and Datasets

Our laboratory testbed, illustrated in Figure 6.1, is composed of four 64-bit Ubuntu 18.04.5 LTS virtual machines (VMs) provisioned with one Intel(R) Xeon(R) CPU E5506 at 2.13GHz with 8 Cores and 8GB of RAM. VM1 and VM3 execute an instance of our prototype, operating as a TorCloak client and bridge, respectively. VM2 acts as the gateway and router for the client TorCloak VM. Finally, VM4 is used to pose as a server in the open Internet which receives requests from the TorCloak bridge in VM3 acting on behalf of the client in VM1.

WebRTC application: We tested our system using Jitsi Meet, operating with the VP9 video codec and using a WebRTC gateway. This allowed us to better imitate a real life scenario, since most modern browser already deploy the VP9 codec. Furthermore, we empirically verified that Jitsi Meet employs a WebRTC gateway for its video calls.

Video dataset: To conduct our experiments, we used 500 videos from Protozoa's dataset "Chat" category. These videos were collected from YouTube and generally represent the usage of video calls for "chatting".

Tor circuit: To ensure we would keep our tests as consistent as possible, our client was configured to use a specific Tor circuit throughout our experiments. We manually selected all relays comprising each Tor circuit based on Tor's *TopRelays* list [77], and chose different nodes within Europe. We specifically chose nodes which had an advertised bandwidth of over 50Mbits and an uptime of at least 50 days. This way we can assure we are using high-performance and consistent nodes, making sure our experiments are not bottlenecked by the Tor circuit.

Network configuration: We conducted experiments in a controlled, isolated deployment to benchmark results and avoid outside interference (e.g., network impairments outside of our control). These tests are executed with an artificial network delay of 50 ms to simulate a realistic network delay for connections established within the same continent [78]. We piggybacked our performance measurements over these tests.

6.2 Performance Evaluation

This section describes the methodology used to perform our performance evaluation. We will describe the setup used to measure TorCloak's throughput, latency and resource utilization. Next, we present and analyze our obtained results obtained.

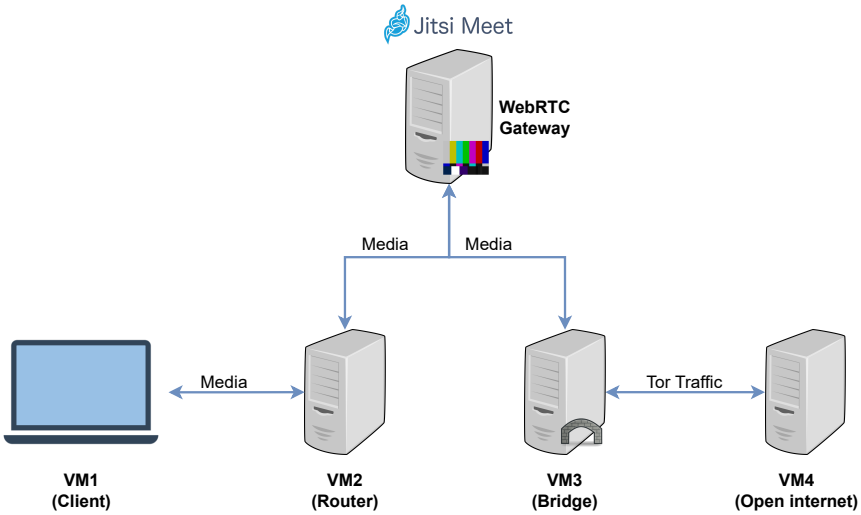


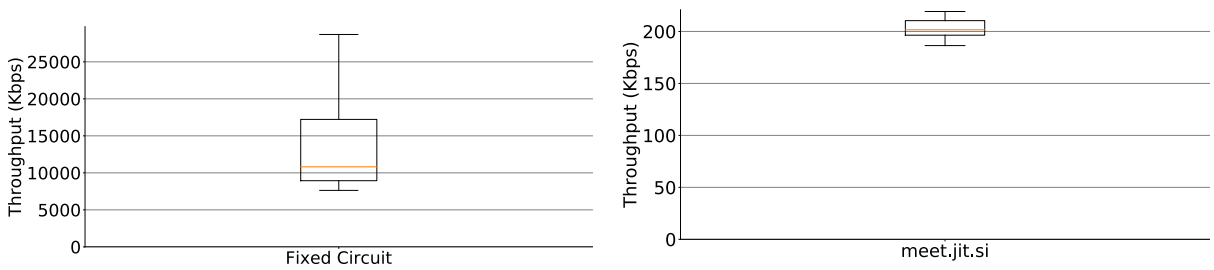
Figure 6.1: Laboratory setup.

6.2.1 Throughput

We conducted an experiment to quantify the throughput of TorCloak's covert channel. The experiment was conducted with a total of 250 runs, while mirroring the video transmitted on each side of the connection. This is to make sure we mostly eliminate any throughput variance related to the different dataset videos used. Different videos have different video frame sizes, which means they can encode more or less information. A video with rapidly changing frames will need to send more and bigger frames between peers, while a video that has little change from frame to frame, i.e. a person sitting still while talking, will need to send much less frames. By increasing the number of videos we use, we can get a better sense of the average throughput we can achieve.

We also used the same fixed Tor circuit in all of the runs to ensure the most consistency (explained in detail in 6.1.2). Even though relays advertise bandwidth as part of the Tor consensus, relays may also impose bandwidth restrictions per user/relay connection using the `TORRC` options: `PERCONNBWRATE`, `PERCONNBWBURST`, `RELAYBANDWIDTHRATE` and `RELAYBANDWIDTHBURST` which use token buckets to rate the bandwidth of client or relay data evenly to every connected client/relay discouraging greedy users. Unlike advertised bandwidth, per connection rates are not public, hence we conducted an throughput experiment (in 50 runs) without TorCloak under the same relays configuration to measure the actual per client throughput of the testing relay.

Figure 6.2 shows the comparisons between the throughput obtained by using the vanilla Tor Circuit vs. using TorCloak. Our numbers show that TorCloak achieves an average throughput of 203 Kbps, while the 90th percentile sits at 212 Kbps, and the 75th percentile at 210 Kbps. We can also see that the throughput has a slight variability, which can be explained by still some existing heterogeneity of the video source. This variability can also be attributed to throughput variations in the Tor circuit, which despite being composed of the highest throughput nodes, can still have a lot of variance through out the day as its usage and number of users increase or decrease.

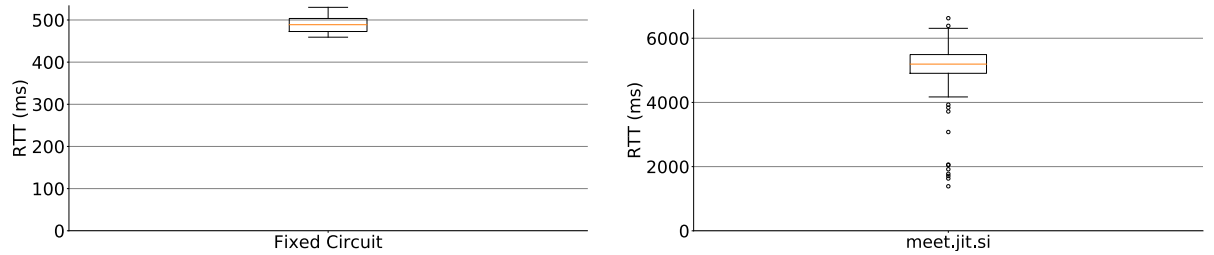


(a) Regular Tor (b) TorCloak

Figure 6.2: Regular Tor and TorCloak throughput comparison.

6.2.2 Latency

To measure the latency of the channel, we conducted a Round-trip time (RTT) measurement over TorCloak's covert channel. Since ICMP is not operational through Tor, we resorted to HTTPing [76] to conduct RTT measurements. This tool performs a HTTP request and measures the time it takes to receive the first byte of the header. Figure 6.3 depicts the comparison of the average latency between the regular Tor Network (left) and the one using TorCloak (right). We can see the average latency of the normal Tor circuit is around 500ms while the average latency using TorCloak is around 5000ms. This is accounting for the added 50ms artificial latency to emulate connections established in the same continent.



(a) Regular Tor (b) TorCloak

Figure 6.3: Regular Tor and TorCloak latency comparison.

6.2.3 Resource Utilization

We also analyzed the overall resource utilization on each instance of TorCloak, using Linux's *htop* [79] tool. This important to make sure our prototype does not consume a lot of resources, making it impractical to deploy or use in a day to day basis. We verified that both client and bridge VMs peaked at around 25% CPU usage and at around 600MB of memory usage. This numbers suggest that TorCloak can be executed on various commodity hardware platforms.

6.3 Comparison with Related Work

In this subsection, we compare the performance results of TorCloak's evaluation with those of Protozoa [6], Stegozoa [25] and DeltaShaper [22], shown in Table 6.1.

System	WebRTC Service	Throughput
Protozoa	Whereby	1.4 Mbps
TorCloak	Jitsi	203 kpbs
Stegozoa	Jitsi	8.2 kbps
DeltaShaper	Skype	3.6 kbps

Table 6.1: Performance comparison with related work.

Stegozoa: Stegozoa uses steganography to encode covert data into WebRTC video frames, so it only uses a fraction of the video payload to establish the covert channel – this fraction must be minimal so that an attacker with access to the video payload cannot detect the presence of a covert channel. Therefore, the reduced throughput of 8.2 kbps is expected and composes a compromise to a stronger threat model.

DeltaShaper: DeltaShaper encodes it's covert data before the video engine performs any kind of compression. To this end, it must encode its covert data into the video image bytes itself, and not the already encoded video bytes. To avoid loss to compression algorithms and other compression techniques, it must encode fewer bits to ensure the data is not corrupted. Consequently, it can only provide a throughput of approximately 3.6 kbps, significantly lower of that of Protozoa or TorCloak, that encode their data after the video has been encoded.

Protozoa: We also compare our values against Protozoa, as it has a similar threat model and composes the basis of TorCloak's WebRTC mechanism. We can see that, despite having similar basis, encoding its information after the video has been encoded by the video engine and after any compression algorithm was been applied, there is a significant difference in throughput. This can be due to multiple factors:

1. **Service Provider:** Whereby and Jitsi may allow their users to use different video bandwidths, which limits the amount of covert data we can transfer.
2. **WebRTC Infrastructure Changes:** At the time of designing Protozoa, Whereby still used a P2P model for most of its video calls. This means that the WebRTC traffic was not controlled by a WebRTC gateway. The WebRTC gateway can impose some kind of bandwidth control on the WebRTC traffic it forwards.
3. **Video Codec:** Protozoa was designed to operate under the VP8 video codec, unlike TorCloak which uses the newer VP9 codec. We empirically verified that VP9 frames are usually much smaller than VP8 frames: a frame with the same resolution, 1280x720, has a size of around 1585 bytes using the VP8 codec and only 88 bytes while using the VP9 codec. We can see that there

is a significant different between the two values. This shows that VP9 is better optimized and can highly reduce it's size even in the same resolution. This could seriously harm our throughput as well since they is not so much space as in the VP8 frames.

Despite this, we believe that with further optimizations in the encoding and decoding stages we can achieve a similar throughput of Protozoa. Nontheless, the current throughput is sufficient for low to medium bandwidth Internet tasks.

Summary

This chapter presented our experimental evaluation of TorCloak. It described the experiments that we performed in order to assess the performance of the covert channel. TorCloak has shown to be able to provide a reasonable throughput and latency need for typical medium-bandwidth internet tasks. The next chapter concludes this document by reviewing the key points of this thesis and introducing the future work.

Chapter 7

Conclusions

Oppressive regimes around the world have made use of increasingly restrictive Internet censorship techniques, in order to prevent their citizens from freely accessing or publishing content from and to the Internet. To tackle this issue, Protozoa leverages the video streams of WebRTC for tunnelling covert traffic. However, on its own, cannot perform the discovery of trusted proxies. It does not possess any mechanism does matches users to proxies in an automated way .It relies on the user inside the censored region to know a contact outside of the censored region, in whom he can trust and rely on to be his proxy.

In this thesis, we have presented: i) a full study and instrumentation framework for WebRTC, designed to help developers more easily develop new multimedia covert streaming systems/tools and ii) the design and implementation of TorCloak, a new Tor pluggable transport that leverages WebRTC video streams to build a covert channel for Tor traffic. TorCloak is completely compatible with Tor's Pluggable Transport Specification be fully integrated into the Tor browser. It offers users an automated and safe way to discover bridges (proxies), so that Tor users can covertly forward and receive Tor data.

The experimental evaluation conducted over TorCloak shows that a careful adaptation to newer WebRTC versions and video codecs enables the system to maintain a reasonable amount of throughput, with an average of about 203 kbps.

7.1 Achievements

The major achievements of the present work is the design and implementation of TorCloak, a new Tor pluggable transport that encodes Tor traffic on WebRTC video streams, creating a covert channel with reasonable network performance for accomplishing common Internet tasks. Other main accomplishments of this thesis encompass: i) a detailed study and instrumentation framework for the WebRTC; ii) an experimental evaluation to analyze TorCloak's performance and resource consumption.

7.2 Future Work

Our efforts in building a working prototype of TorCloak have led us to understand that more work is required to close a number of gaps in our implementation and to extend TorCloak to mobile platforms. We identify four specific directions for future work: first, we would like to further explore the optimizations on the encoding/decoding of data in the VP9 frame. Despite being usually of smaller size, you believe that we can encode a lot more data into the VP9 frame. This will require us to verify if any more frame quality control mechanism are placed, that could lead to some frames sent being dropped and consequently dropping our overall throughput. Furthermore, we can further explore the size calculation of each frame's header size to understand if there is any more places we can fit data into. We can empirically verify our channel's maximum throughput buy counting how much available space we have in each frame sent in a period of time and calculating the maximum theoretically throughput we can achieve using the VP9 codec a Jitsi Meet.

Second, TorCloak's mobile version must be developed and thoroughly tested. This will require a thorough research and debugging in Android platforms, which are not as well documented for Pluggable Transports as their desktop version counterparts. Nonetheless, as we pointed out, we already have a defined plan for the development of this version of TorCloak.

Thirdly, we will complete an implementation of TorCloak's desktop version that does not use Linux network namespaces, but that instead uses sockets to transmit Tor cell data. At the moment of writing, this version is not fully functional as some implementation difficulties must be resolved. This will enable us to better integrate TorCloak with the remaining of the Tor infrastructure and code. It will also facilitate the development of future versions of TorCloak, making sure it is compatible with most platforms.

Lastly, our whole bridge and broker infrastructure must be deployed and tested. We need to assess the system's scalability and how it can handle intense workloads from multiple users. This is key to understand whether TorCloak can be deployed as a viable solution for the majority of Tor users located within censored Internet regions.

Bibliography

[1] Tor Project. Tor faq. https://2019.www.torproject.org/about/overview.html.en, 2019. Accessed: 2022-10-31.

[2] Tor Project. Tor pluggable transports. https://2019.www.torproject.org/docs/pluggable-transports, 2019. Accessed: 2022-10-31.

[3] Arlo Breault. meek. https://trac.torproject.org/projects/tor/wiki/doc/meek, May 2019. Accessed: 2022-10-31.

[4] Y. Angel. obfs4 (the obfourscator). https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt, Jan. 2019. Accessed: 2022-10-31.

[5] D. Barradas, N. Santos, and L. Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security 18)*, pages 169–185, Baltimore, MD, Aug. 2018.

[6] D. Barradas, N. Santos, L. Rodrigues, and V. Nunes. Poking a hole in the wall: Efficient censorship-resistant internet communications by parasitizing on webrtc. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 35–48, 2020.

[7] G. DEVELOPERS. Webrtc. getting started with webrtc. https://webrtc.org/getting-started/overview, 2019. Accessed: 2022-10-31.

[8] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[9] A. Mogage and E. Simion. Statistical analysis and anonymity of tor's path selection. *IACR Cryptol. ePrint Arch.*, 2019:592, 2019.

[10] T. Project. Tor project — bridge. https://community.torproject.org/relay/types-of-relays/. Accessed: 2022-10-31.

[11] S. Matic, C. Troncoso, and J. Caballero. Dissecting tor bridges: A security evaluation of their private and public infrastructures. In *Proceedings of the 24th Network and Distributed System Security Symposium*, San Diego, CA, USA, Feb 2017.

[12] D. Eastlake et al. Transport layer security (tls) extensions: Extension definitions. Technical report, RFC 6066, January, 2011.

[13] Tor Project. A child garden of pluggable transports. `https://trac.torproject.org/projects/tor/wiki/doc/AChildsGardenOfPluggableTransports`, May 2016. Accessed: 2022-10-31.

[14] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. *Proc. Priv. Enhancing Technol.*, 2015(2):46–64, 2015.

[15] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 61–72, 2013.

[16] L. Wang, K. P. Dyer, A. Akella, T. Ristenpart, and T. Shrimpton. Seeing through network-protocol obfuscation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 57–69, 2015.

[17] V. Nunes, J. Brás, D. Barradas, K. Gallagher, and N. Santos. Tork: Hardening tor against traffic correlation attacks with k-anonymity. In *Submission to USENIX Security '23*.

[18] M. Nasr, A. Bahramali, and A. Houmansadr. Deepcorr: Strong flow correlation attacks on tor using deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1962–1976, 2018.

[19] D. Fifield. *Threat modeling and circumvention of Internet censorship*. University of California, Berkeley, 2017.

[20] A. Houmansadr, T. J. Riedl, N. Borisov, and A. C. Singer. I want my voice to be heard: Ip over voice-over-ip for unobservable censorship circumvention. In *NDSS*, 2013.

[21] S. Li, M. Schliep, and N. Hopper. Facet: Streaming over videoconferencing for censorship circumvention. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 163–172, 2014.

[22] D. Barradas, N. Santos, and L. E. Rodrigues. Deltashaper: Enabling unobservable censorship-resistant tcp tunneling over videoconferencing streams. *Proc. Priv. Enhancing Technol.*, 2017(4):5–22, 2017.

[23] S. Loreto and S. P. Romano. Real-time communications in the web: Issues, achievements, and ongoing standardization efforts. *IEEE Internet Computing*, 16(5):68–73, 2012.

[24] The Chromium Projects. The chromium projects. `https://www.chromium.org/`. Accessed: 2022-10-31.

[25] G. Figueira, D. Barradas, and N. Santos. Stegozoa: Enhancing webrtc covert channels with video steganography for internet censorship circumvention. In *Proceedings of the 2022 ACM on Asia*

Conference on Computer and Communications Security, page 1154–1167, New York, NY, USA, 2022.

[26] P. Matthews, J. Rosenberg, and R. Mahy. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766, Apr. 2010. URL https://www.rfc-editor.org/info/rfc5766.

[27] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano. Janus: a general purpose webrtc gateway. In *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*, pages 1–8, 2014.

[28] D. Wells and J. Bingham. Let's reverse engineer discord, 2020. URL https://medium.com/tenable-techblog/lets-reverse-engineer-discord-1976773f4626. Accessed: 2022-10-31.

[29] The Tor Project. Snowflake. https://snowflake.torproject.org/, 2017. Accessed: 2022-10-31.

[30] D. Fifield. *Threat modeling and circumvention of Internet censorship*. University of California, Berkeley, 2017.

[31] The Tor Project. Fingerprinting · wiki · the tor project / anti-censorship / pluggable transports / snowflake. https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake/-/wikis/Fingerprinting, 2020. Accessed: 2022-10-31.

[32] ZDNet. Despite privacy outrage, accuweather still shares precise location data with ad firms. https://www.zdnet.com/article/accuweather-still-shares-precise-location-with-advertisers-tests-reveal/, 2017. Accessed: 2022-10-31.

[33] The Verge. Facebook has been collecting call history and sms data from android devices. https://www.theverge.com/2018/3/25/17160944/facebook-call-history-sms-data-collection-android, 2018. Accessed: 2022-10-31.

[34] X. Zhou, S. Demetriou, D. He, M. Naveed, X. Pan, X. Wang, C. A. Gunter, and K. Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028, 2013.

[35] G. Maganis, E. Shi, H. Chen, and D. Song. Opaak: Using mobile phones to limit anonymous identities online. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, page 295–308, New York, NY, USA, 2012. Association for Computing Machinery.

[36] J. Camenisch, S. Hohenberger, M. Kohlweiss, A. Lysyanskaya, and M. Meyerovich. How to win the clonewars: efficient periodic n-times anonymous authentication. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 201–210, 2006.

[37] J. Camenisch and A. Lysyanskaya. An efficient system for non-transferable anonymous credentials with optional anonymity revocation. In *Proceedings of the International conference on the theory and applications of cryptographic techniques*, pages 93–118. Springer, 2001.

[38] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In *Proceedings of the International Conference on Security in Communication Networks*, pages 268–289. Springer, 2002.

[39] OpenID Foundation. Openid foundation websites. `https://openid.net/`. Accessed: 2022-10-31.

[40] V. Bertocci, G. Serack, and C. Baker. *Understanding windows cardspace: an introduction to the concepts and challenges of digital identities*. Pearson Education, 2007.

[41] J. Camenisch, R. Leenes, and D. Sommer. *Digital Privacy: PRIME-Privacy and Identity Management for Europe*, volume 6545. Springer, 2011.

[42] F. Paci, E. Bertino, S. Kerr, A. C. Squicciarini, and J. Woo. An overview of veryidx-a privacy-preserving digital identity management system for mobile devices. *J. Softw.*, 4(7): 696–706, 2009.

[43] K.-A. Shim. An id-based aggregate signature scheme with constant pairing computations. *Journal of Systems and Software*, 83(10):1873–1880, 2010.

[44] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, 2002. URL `https://tools.ietf.org/html/rfc3261`.

[45] J. Rosenberg and H. Schulzrinne. RFC3264: An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, 2002. URL `https://tools.ietf.org/html/rfc3264`.

[46] J. Rosenberg et al. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245, 2010. URL `https://tools.ietf.org/html/rfc5245`.

[47] D. McGrew, E. Rescorla, et al. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764, 2010. URL `https://tools.ietf.org/html/rfc5764`.

[48] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711, 2004. URL `https://tools.ietf.org/html/rfc3711`.

[49] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC3550: RTP: A Transport Protocol for Real-Time Applications. RFC 3550, 2003. URL `https://tools.ietf.org/html/rfc3550`.

[50] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. Analysis and Design of the Google Congestion Control for Web Real-Time Communication (WebRTC). In *Proceedings of the 7th International Conference on Multimedia Systems*, 2016.

[51] B. Jansen, T. Goodwin, V. Gupta, F. Kuipers, and G. Zussman. Performance Evaluation of WebRTC-Based Video Conferencing. *SIGMETRICS Perform. Eval. Rev.*, 2018.

[52] T. Levent-Levi. What is WebRTC P2P mesh and why it can't scale? `https://bloggeek.me/webrtc-p2p-mesh/`, 2020. Accessed: 2022-10-31.

[53] V. Pascual and G. Garcia. WebRTC beyond one-to-one communication. `https://webrtchacks.com/webrtc-beyond-one-one/`, 2014. Accessed: 2022-10-31.

[54] A. Fakis, G. Karopoulos, and G. Kambourakis. Neither denied nor exposed: Fixing webrtc privacy leaks. *Future Internet*, 12(5):92, 2020.

[55] C. Castro. Webrtc leaks: What they are and how to prevent them, Mar 2022. URL `https://www.techradar.com/vpn/webrtc-leaks`. Accessed: 2022-10-31.

[56] T. Levent-Levi. Seven Reasons for WebRTC Server-Side Media Processing. Technical report, 2015.

[57] M. Westerlund and S. Wenger. RTP Topologies. RFC 7667, 2015. URL `https://tools.ietf.org/html/rfc7667`.

[58] T. Levent-Levi. WebRTC Multiparty Video Alternatives, and Why SFU is the Winning Model. `https://bloggeek.me/webrtc-multiparty-video-alternatives/`, 2016. Accessed: 2022-10-31.

[59] B. Grozev, L. Marinov, V. Singh, and E. Ivov. Last N: Relevance-Based Selectivity for Forwarding Video in Multimedia Conferences. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2015.

[60] C. Hart and O. Divorra. Optimizing video quality using Simulcast. `https://webrtchacks.com/sfu-simulcast/`, 2016. Accessed: 2022-10-31.

[61] G. Bakar, R. A. Kirmizioglu, and A. M. Tekalp. Motion-based adaptive streaming in webrtc using spatio-temporal scalable vp9 video coding. In *Proceedings of the GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–6. IEEE, 2017.

[62] M. Foundation. Rtcstatsreport - web apis: Mdn. URL `https://developer.mozilla.org/en-US/docs/Web/API/RTCStatsReport`. Accessed: 2022-10-31.

[63] Bridges — Tor Project — Tor Browser Manual. `https://tb-manual.torproject.org/bridges/`. Accessed: 2022-10-31.

[64] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. Rfc1928: Socks protocol version 5, 1996.

[65] Rtp control protocol (rtcp) extensions for single-source multicast sessions with unicast feedback. https://datatracker.ietf.org/doc/html/rfc5760. Accessed: 2022-10-31.

[66] Rtp control protocol (rtcp) extensions for single-source multicast sessions with unicast feedback. https://datatracker.ietf.org/doc/html/rfc5760. Accessed: 2022-10-31.

[67] Documentation — android developers. https://developer.android.com/docs, . Accessed: 2022-10-31.

[68] Get started with the ndk, . URL https://developer.android.com/ndk/guides. Accessed: 2022-10-31.

[69] The tor project / anti-censorship / pluggable transports / snowflake mobile · gitlab, 2020. URL https://gitlab.torproject.org/tpo/anti-censorship/pluggable-transports/snowflake-mobile. Accessed: 2022-10-31.

[70] V4L2LOOPBACK. https://github.com/umlaeute/v4l2loopback, 2005. Accessed: 2022-10-31.

[71] FFMPEG. https://ffmpeg.org, 2000. Accessed: 2022-10-31.

[72] Pluggable Transports. Pluggable transport specification v3.0. https://github.com/Pluggable-Transports/Pluggable-Transports-spec/tree/main/releases/PTSpecV3.0, note = Accessed: 2022-10-31.

[73] Cure53. Pentest-report raptor inesc-id 06.-07.2022, 2022.

[74] Netfilter Framework. http://www.netfilter.org/, 1998. Accessed: 2022-10-31.

[75] V. GUEANT. Iperf - the ultimate speed test tool for tcp, udp and sctptest the limits of your network + internet neutrality test. URL https://iperf.fr/. Accessed: 2022-10-31.

[76] Pjperez. Pjperez/httping: Httping - a tool to measure rtt on http/s requests. URL https://github.com/pjperez/httping.

[77] Tor Project. Tor metrics - relay search. https://metrics.torproject.org/rs.html#toprelays, 2018. Accessed: 2022-10-31.

[78] Monthly ip latency data, 2022. URL https://www.verizon.com/business/terms/latency/. Accessed: 2022-10-31.

[79] htop - an interactive process viewer. URL https://htop.dev/. Accessed: 2022-10-31.