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

(extended abstract of the MSc dissertation)

Francisco Manuel Almeida Silva
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisors: Professor Nuno Santos and Professor Diogo Barradas

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. Through extensive experimental evaluation, we also show TorCloak's resistance to traffic analysis while delivering good performance.**

I. Introduction

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).

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. However, recent advancements in traffic analysis [5] have shown that covert tunnels like these can be accurately identified using machine learning (ML) classifiers.

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., meet.jit.si.

To this end, we propose to leverage Protozoa [6]. 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.

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 variation 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. These errors would severely impact our tool's throughput as well as make it easier for the adversary to

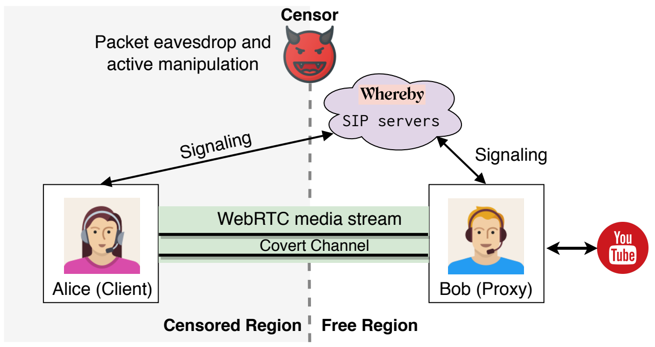


Figure 1. WebRTC covert tunnel over a Whereby call.

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 to **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.

II. Background and Related Work

In this section we provide an overview of the existing system, which TorCloak is based on: Protozoa, the existing approach for creating WebRTC-enabled covert tunnels.

A. *Encoded Media Tunneling*

Protozoa [6] is a new tool that, by using web streaming application-based WebRTC, creates 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 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 1 which shows two Internet users. One of them (the *client*) is located in a censored region where the access to certain content or services is blacklisted by a state-level censor who can inspect and control all the network communications. 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 (≈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.
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. 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 does matches users to proxies in an automated way.

III. Instrumentation Framework for WebRTC and VP8/VP9

In this section, we present a detailed explanation of our instrumentation framework for the WebRTC Code Stack.

A. *WebRTC Gateways*

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 [8], [9]. Another major problem of WebRTC peer-to-peer calls was IP address leakage [10], [11]. 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 [12].

These are usually called **WebRTC gateways:** media servers that become a central management 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 peer-to-peer connection with the gateway.

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

- *MCU (Multipoint Control Unit)* [13]: 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 [14].
- *SFU (Selective Forwarding Unit)* [15]: the most used approach, based on a technique known as simulcast [16]. 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. SFUs allow different peers to have different speeds/resolutions of video streams according to their network's capabilities.

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.

B. Understanding WebRTC Video Frame Flows

WebRTC is built on top of an open standard. It is 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 [17].

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: **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. T

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, placed in different locations of the WebRTC code stack, and incremented each time that piece of code runs.This variables 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.

C. Instrumenting the WebRTC Receiving and Sending Flows

Taking a deeper dive into the receiving of a frame in the WebRTC Code Stack:

1) The first step is toretrieve 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()*
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.

Now analyzing 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.

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.

IV. TorCloak

In this section, 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.

A. Design Goals and Threat Model

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.

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.

B. Architecture

Figure 2 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.
To illustrate the process, Figure 2 shows Alice, a user located in a censored region controlled by a state-level adversary, using a TorCloak bridge located in a free Internet

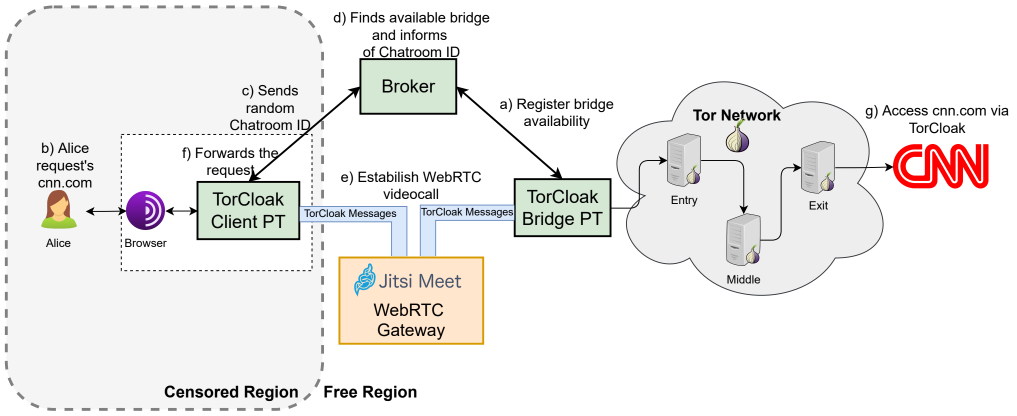

d) Finds available bridge and informs of Chatroom ID
Broker
c) Sends random Chatroom ID
a) Register bridge availability
Tor Network
g) Access cnn.com via TorCloak
b) Alice request's cnn.com
f) Forwards the request
e) Estabilish WebRTC videocall
TorCloak Client PT
TorCloak Bridge PT
TorCloak Messages
TorCloak Messages
Alice
Browser
Entry
Exit
Middle
Jitsi Meet
WebRTC Gateway
Censored Region
Free Region

Figure 2. TorCloak Architecture.

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.

C. 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. 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.

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]. 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[18] proxy dissemination mechanism. Specifically, we deploy

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

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. 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[19].

The rotation of the chatroom RIDs must also be managed and controlled to ensure maximum resistance again censorship. On spin up, the TorCloak Client generates a RID for that session. 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).

D. 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 3 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 [20] 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. 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.
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.
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.

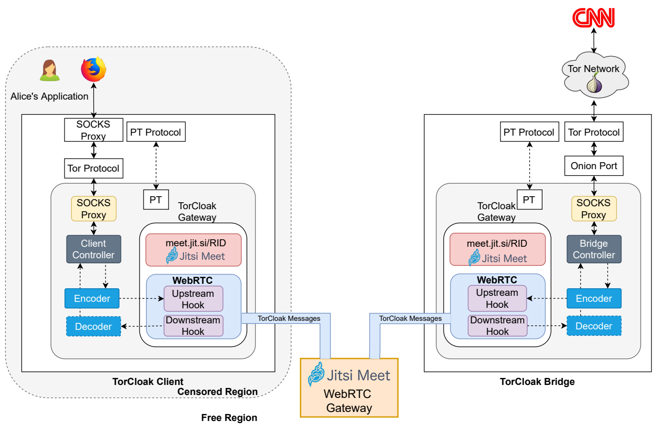


Figure 3. TorCloak's Main Components.

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. 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). 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. 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. This results in severe reduction of the channel bandwidth. 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.

***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. The main difference between the structures of both codecs are: 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 mechanisms, we need to adjust our offsets to account for the new header. Using the VP9 Decoding Specification [21] together with WebRTC's VP9 decoding structure 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. 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.

E. 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 [22].

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[2].

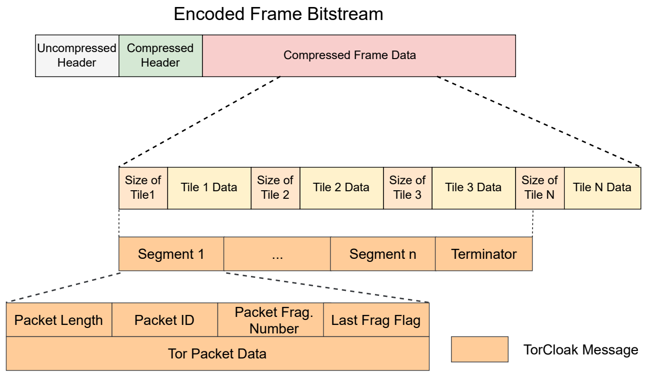


Figure 4. Format of VP9 encoded bitstream with replaced TorCloak data.

V. Implementation

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 [23] and feed recorded video with the help of the ffmpeg video library [24]. We will now describe in more detail about the most critical aspects of the implementation of TorCloak.

TorCloak is structured in five main components: i) Client Pluggable Transport, ii) SOCKS proxy, iii) Bridge/Client Controller, iv) Encoder Hook and v) Decoder Hook. The Pluggable Transport Client implements the Tor PT API instructing Tor how to open and control the TorCloak process. The SOCKS proxy is responsible to receive Tor data which is then routed to the respective encoder/decoder hook, to be encoded/decoded into the WebRTC video stream. The Client and Bridge controllers managed every component of the TorCloak structure.

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

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

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.

VI. Evaluation

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.

A. 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.

o 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* [25] and *HTTPing* [26]. This enables us to stress the covert's channel capacity and latency.

B. Experimental Testbed and Datasets

Our laboratory testbed, illustrated in Figure 5, 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".

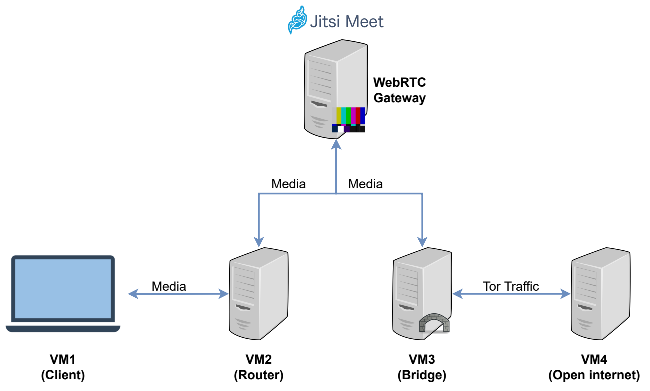


Figure 5. Laboratory setup.

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 [27], 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.

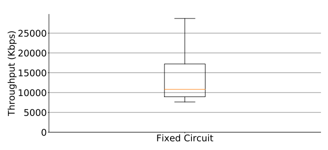
Network configuration : We conducted experiments in a controlled, isolated deployment to benchmark results and avoid outside interference (e.g., network impairments outside of our control). These tests are executed with an artificial network delay of 50 ms to simulate a realistic network delay for connections established within the same continent [28]. We piggybacked our performance measurements over these tests.

C. Performance Evaluation

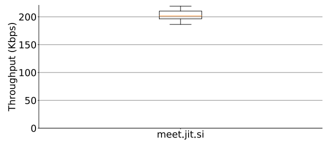
This section describes the methodology used to perform our performance evaluation. We will describe the setup used to measure TorCloak's throughput, latency and resource utilization. Next, we present and analyze our obtained results obtained.

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 VI-B). Even though relays advertise bandwidth



(a) Regular Tor



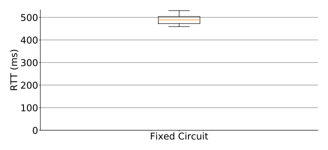
(b) TorCloak

Figure 6. Regular Tor and TorCloak throughput comparison.

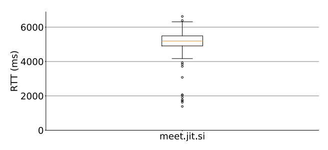
as part of the Tor consensus, relays may also impose bandwidth restrictions per user/relay connection using the `TORRC` options: `PERCONNBWRATE`, `PERCONNBWBURST`, `RELAYBANDWIDTHRATE` and `RELAYBANDWIDTHBURST` which use token buckets to rate the bandwidth of client or relay data evenly to every connected 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 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.

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 [26] 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 7 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 7. Regular Tor and TorCloak latency comparison.

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: i) **Service Provider:** Whereby and Jitsi may allow their users to use different video bandwidths, which limits the amount of covert data we can transfer; ii) **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. iii) **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. This can seriously harm our throughput as well since they is not so much space as in the VP8 frames.

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

Table I
PERFORMANCE COMPARISON WITH RELATED WORK.

VII. 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 has been proposed, leveraging the video streams of WebRTC for tunnelling covert traffic. However, Protozoa on its own, cannot perform the discovery of trusted proxies.

We present 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 and can be fully integrated into the Tor browser. It offers users an automated and safe way to discover bridges (proxies) to receive and forward their covert data.

References

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

[2] T. 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)*, Baltimore, MD, Aug. 2018, pp. 169–185.

[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*, 2020, pp. 35–48.

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

[8] 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.

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

[10] A. Fakis, G. Karopoulos, and G. Kambourakis, "Neither denied nor exposed: Fixing webrtc privacy leaks," *Future Internet*, vol. 12, no. 5, p. 92, 2020.

[11] C. Castro, "Webrtc leaks: What they are and how to prevent them," Mar 2022, accessed: 2022-10-31. [Online]. Available: https://www.techradar.com/vpn/webrtc-leaks

[12] T. Levent-Levi, "Seven Reasons for WebRTC Server-Side Media Processing," Tech. Rep., 2015.

[13] M. Westerlund and S. Wenger, "RTP Topologies," Internet Requests for Comments, RFC 7667, 2015. [Online]. Available: https://tools.ietf.org/html/rfc7667

[14] 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.

[15] 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.

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

[17] M. Foundation, "Rtcstatsreport - web apis: Mdn," accessed: 2022-10-31. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/RTCStatsReport

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

[19] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson, "Blocking-resistant communication through domain fronting." *Proc. Priv. Enhancing Technol.*, vol. 2015, no. 2, pp. 46–64, 2015.

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

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

[22] "Documentation — android developers," https://developer.android.com/docs, accessed: 2022-10-31.

[23] "V4L2LOOPBACK," https://github.com/umlaeute/v4l2loopback, 2005, accessed: 2022-10-31.

[24] "FFMPEG," https://ffmpeg.org, 2000, accessed: 2022-10-31.

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

[26] Pjperez, "Pjperez/httping: Httping - a tool to measure rtt on http/s requests." [Online]. Available: https://github.com/pjperez/httping

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

[28] "Monthly ip latency data," 2022, accessed: 2022-10-31. [Online]. Available: https://www.verizon.com/business/terms/latency/