SECURING WEBRTC

An Abstract of a

Thesis Presented to the

School of Computer Sciences

Western Illinois University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

DENNIS MCMEEKAN

May, 2021

ABSTRACT

WebRTC is a breakthrough open-source technology that allows clients and businesses to connect in real-time to communicate through a web application without extra installations [1]. Although WebRTC is increasingly used for real-time video communications, there are still many security concerns with this being relatively new. This study will look further into security issues arising in WebRTC applications, with a focus of exploring confidentiality violations and mitigating IP leaks through a distributed hash table server. A simple WebRTC application has been created with the purpose of discovering if WebRTC is susceptible to covert channels, and if so, how to mitigate this.

APPROVAL PAGE

This thesis by **DENNIS MCMEEKAN** is accepted in its present form by the School of Computer Sciences of Western Illinois University as satisfying the thesis requirements for the degree Master of Computer Science.

Chairperson, Examining Committee: Dr. Binto George

Member, Examining Committee: Dr. Nilanjan Sen

Member, Examining Committee: Dr. Chunying Zhao

Date

SECURING WEBRTC

A Thesis Presented to the

School of Computer Sciences

Western Illinois University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

By

DENNIS MCMEEKAN

May, 2021

ACKNOWLEDGMENTS

Grateful acknowledgment is extended to Dr. Binto George, thesis supervisor, and committee members Dr. Nilanjan Sen and Dr. Chunying Zhao for their valuable suggestions and guidance given in this thesis project.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

In a cultural and technology environment that is continuously evolving, real-time communication is more essential than ever for individuals to complete daily tasks such as school, work, and personal communication. With a profound use of video communication, “78% of corporate companies use video calling software” [2]. The reach of internet and video communication to all points of the world is continuously growing, and it is realistic to see that real-time communication between two or more individuals will be the main product for online communication methods if it is not already there today. In recent years, WebRTC applications have been used to establish real-time communication between two or more peers. This specification provides limitless advantages to web developers across the entire world, but with these advantages, there are also disadvantages. The main focus of this study is to determine the security vulnerabilities that may be exploited with WebRTC and mitigating these issues.

In establishing an understanding of the vulnerabilities, we conducted a study to look further into possible security flaws with WebRTC and real-life examples seen with these applications. Through a series of open-source examples centered around WebRTC applications, an understanding of how data is transferred between two clients in real-time has been established. The most common way to develop a WebRTC application is using the open-source API implemented by the creators Google, focused around the two main languages of JavaScript and HTML5.

We studied security issues of WebRTC. Our focus was mainly on confidentiality violations through covert channels, along with a brief survey on IP leaks. Confidentiality violations are an issue because of WebRTC's open-source model, clients hold the ability to signal information from one peer to another using the API provided. Specifically, for example, we found that the application can be altered to create covert channels via image filtering, a process that takes the video input, effects the transmission of data, and then outputs the video. It was discovered that by implementing a delay in data transfer from one client to another, a bit can be sent and received based on this delay experienced by the client. This would provide a method to transfer data secretly through clients without any control or knowledge by the administrator. We explore various strategies for mitigating covert channels such as introducing constant or random delay mechanisms to delay data packets within the WebRTC stream. We found that mitigations will make it harder to use covert channels. These experiments were done using WebRTC's API and we measured the bitrate, framerate, and covert channel bandwidth in an effort to determine the vulnerability of covert channels and effectiveness of mitigation strategies.

To prove and further research confidentiality violations, two prototypes have been created. An unsecure version, which focuses on creating a covert channel, delaying/sensing data, and then receiving a bit based on the delay. And a secure version which combats this issue, implementing constant and random delays to further increase the error rate at which the bit is being received.

Beyond this, IP leaks is highly discussed as a security flaw with WebRTC applications, as in most cases the user's public IP address is used directly. This is commonly addressed with the use of a VPN (Virtual Private Network), but in almost all instances this can be a costly effort and when free, users are limited by the amount of data that is able to be transmitted privately.

CHAPTER 2
BACKGROUND AND RELATED WORK

This chapter focuses on describing background details and looking into prior research. This sets the groundwork for the thesis and finding next courses of action.

2.1 Background Information

In a constant evolving technology culture, WebRTC has evolved into a powerful real-time communication technology that has exploded in use in recent years due to cheaper high-speed network access and more powerful devices. [3]. This is a specification that is open and royalty-free that provides individuals the ability to use media sources between one or more peers to communicate in real-time. This results in developers and users holding the ability to develop and implement WebRTC applications, but the downfall is not all security exploits have been discovered. Being open-source and royalty-free, Google does not hold accountability for misuse.

2.2 Motivation

The motivation behind examining and exploring security exploits related to WebRTC applications arises from the idea that WebRTC is a new and not fully examined specification. This allowed us to create something that can leave a lasting impact on the field of computer science, and specifically real-time communication. The world-wide pandemic of COVID-19 allows for any advancement in real-time communication to provide immense value considering the drastic increase of virtual environments throughout businesses, schools, and personal lives.

2.3 Related Work

To develop and understanding on how to approach implementation and development of a WebRTC application, it is vital to examine what has already been done and establish a definition of two elements: covert channels and IP leaks.

Before diving into an understanding of a WebRTC application and the surrounding parts for prototype implementation, it is important to see what research has come before in the study of computer science. When doing research and learning from others, it is the role of the student to construct knowledge instead of merely receiving and storing knowledge [4]. This means that while it is important to learn and understand from a teacher or researcher, that knowledge must be used to construct and develop insight not seen from the original sentiment. The second paper that needs to be examined before diving deep into WebRTC, is a paper that confines a program. This sets the basis for our study and establishes that programs want to prevent any series of executions outside of its program [5]. This first touched upon the idea of covert channel implementation and acknowledged that it is possible to unintentionally transfer information.

Specific to a WebRTC application, there have been no studies conducted that research and develop covert channels with this specification, and the wide range of outcomes is frightening to any WebRTC developer.

2.3.1 Covert Channels

Covert channels allow one or more peers to send and receive data in a secretive manner, usually without any notice by an administrator. Although in some cases this may not be harmful, this is becoming more common in use by exploiters for malicious purposes for a wide variety of reasons, such as sending malware [6]. Due to the WebRTC API working in compliment to

JavaScript and HTML5, there are a large number of ways that the application can be altered in an effort to establish covert channels. One that will be focused on is applying filters to a media element, which allows the media source that is being sent between multiple peers to be altered. If it possible to establish covert channels in a WebRTC application, there are countless possibilities on what could be sent and received without proper mitigation.

2.3.2 IP Leaks

A large issue with WebRTC applications, is IP leak concerns. This is due to the fact that "any client using a WebRTC application can retrieve the public IP address of another user or client" [7]. This is a vital concern, because a public IP address can be used to determine sensitive information such as the geographical location of any client using the WebRTC application.

CHAPTER 3

RESEARCH METHODOLOGY & IMPLEMENTATION

This chapter establishes the problem statement and describes the research steps taken to determine if WebRTC is susceptible to covert channels. A prototype has been created in an effort to establish covert channels, and if possible, then to implement mitigations. Beyond this, it is crucial to discuss a prevention method for IP leaks.

3.1 Problem Statement

WebRTC has the foundation to allow for a secure and simple connection to be made by two users without installing native apps or plugins. Being relatively new, not all security vulnerabilities have been examined.

3.1.1 Configuration

All writing, programming, and implementation was done using an Acer Laptop with an Intel® i7-7500 CPU and 64-bit Windows 10 Pro operating system. Testing and programming specifically involved using Visual Studio Code and then served through a command line prompt using a home network Wi-Fi at 5 GHz. Application testing used the Google Chrome browser.

3.2 Security Concerns

We focused on two security issues of WebRTC applications. The first one being confidentiality violations concerns, which involves signaling information through covert channels without any knowledge to the administrator. The second being IP leaks in relation to a connection being established by two or more peers.

3.2.1 Confidentiality Violations

Going beyond server and client authenticity, it is vital to ensure that the *communication stage*, the main segment that is being used throughout this thesis implementation, holds secure. When creating a WebRTC application, two or more peers will create a connection, and send real-time data between each other through the use of a server. The security concern in doing so with a WebRTC application comes mainly as a result of this being open-source project, and the API is available to the public. Open source is a concern because clients hold the ability to manipulate WebRTC API and add a wide variety of outside programming functions and methods, as done in our example with covert channel implementation. This provides great opportunities for developers, and also for hackers and exploits. Specifically, the video can be altered between each client, creating a vital security concern.

3.2.2 Covert Channels via Image Filtering

Expanding upon confidentiality violations, WebRTC real-time image filtering is meant for implementing cool features but could be misused for signaling through covert channels. This has been seen implemented using two different types of video input, *local* and *remote*. Separating these two types of videos simulates a network connection with WebRTC API in which one peer connects to a server, and then sends data to a receiving client. With image filtering, this instead involves taking the input (local) video and transmitting that through a canvas element. A canvas element is a type of HTML5 element that is intended for the use of drawing graphics or in this case, videos. After the video is transmitted into the canvas element, then that video is sent to the receiving client. There is a very powerful example done previously that does this sort of action, but also applies a filter on top of the incoming video to alter the appearance [8].

Through this examination, it appears that action can be taken similar to having a filter on top of the data, but instead delaying the video being sent from the canvas element to the output (remote) client. This basis of implementing a delay using image filtering, allows the creation of a covert channel. A covert channel allows for individuals to send secret or sensitive information potentially undetected using non-standard methods. In our study, we use the canvas element to approach image filtering by introducing a delay for signaling data, and also to allow the receiving client to sense that delay in an effort to receive the sent data. Since the data rates tend to fluctuate very briefly before it returns to normal, this is hard to detect.

3.3 Open-Source Project Examples

Expanding upon the previous chapter that discussed the two main issues at hand, there are a number of examples that help provide a starting point in establishing covert channels and then preventing these concerns.

3.3.1 WebRTC API

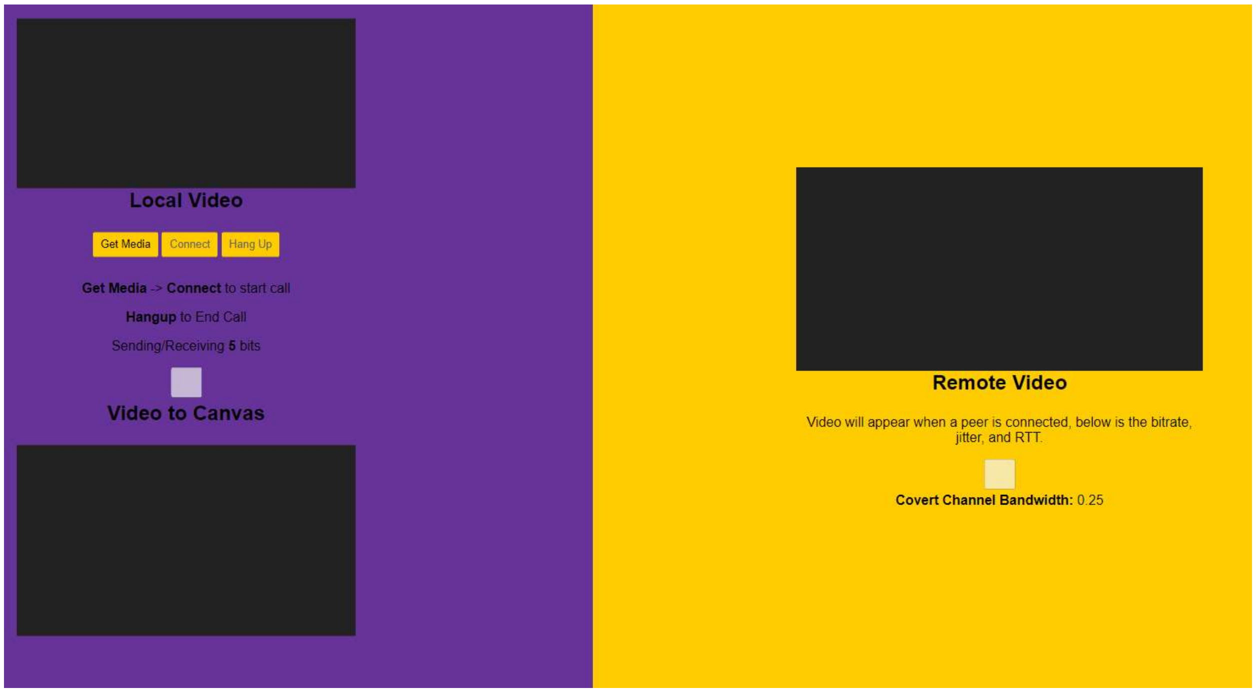
The center of implementation programming revolves around holding a deep understanding of WebRTC API and concepts involved. The beginning of this research begins with a great publicly available API webpage that lays out each function, method, interfaces, and references available in relation to WebRTC [9]. Mainly, the interface 'RTCPeerConnection()' allows a local peer to connect to a remote peer, and that connection being able to be monitored. There also are many methods below this interface that allow for a peer to wait when there is no remote connection or end the call when the connection is dropped. There are also many other properties that the server and each client use to determine if a connection has been made such as creating an offer, 'createOffer()', and 'addStream()' or 'addTrack()', which actually sends the media elements to the other peer. This allowed a base understanding to be established and continued as a reference for attacking specific angles in prototype implementation. Further than API, actual examples also allowed for a great resource for expanding knowledge of a simple, but complex, WebRTC application. Specifically, an “as simple as it gets” application was found that built a client and server connection, and then allowed the media to be transferred through a WebRTC connection using the corresponding protocols [10]. Deeply examining and understanding how the simple connection can be made, while looking through the API page provided allowed for further implementation and transformation of the unsecure prototype.

3.3.2 Image Filtering

Our finding is that the image filtering capability can be misused for covert channel implementation. This first came to discussion by acknowledgment of altering data in a WebRTC application. This process is centered around using a canvas element. A canvas HTML5 element is a common element used throughout web development. The most typical use for a canvas element is to develop graphic images or media on a webpage, but there are extra capabilities. This brought the idea that by transmitting a video element, into a canvas element, that the data could then be altered before being sent. An example that provided a great basis to this implementation took all aspects explained earlier, a local video, canvas element, and an output/remote video but instead of manipulating the pixels, the example allows for filters to be draw on top of the video feed that is being transmitted [11]. This provided the basis that instead of implementing a filter, that a delay could be used to alter the data being sent and received. We implemented an unsecure version of a WebRTC application with a covert signaling mechanism. We also implemented a secure version with covert channel mitigations in place to compare performance.

3.4 Unsecure Prototype

The unsecure prototype implements a signaling mechanism through covert channels during a WebRTC connection between one peer to another. To begin, we based our implementation on a sample WebRTC application. [10]. The application was altered to allow for the connection and remote connection to be simulated. This meant altering the way that the clients interacted with each other, and instead of waiting on the server to send a signal, once a connection is made, a video element is used which conquers a similar action by adding a "Get Media", "Start Call", and "Hangup" button through a series of WebRTC connection functions and methods. This allows the ability to have a local and remote connection appear on the same webpage, instead of having needed two completely separate clients to alleviate testing and trail purposes. This made the timeline much more realistic in terms of proper implementation, due to network, system, and time constraints. Using Figure 3.1 below, we are able to see how exactly this prototype appears.



Get Media -> **Connect** to start call

Hangup to End Call

Sending/Receiving **5** bits

Video will appear when a peer is connected, below is the bitrate, jitter, and RTT.

Covert Channel Bandwidth: 0.25

Figure 3.1 Prototype Implementation

There were two main pieces of code that required being altered, one being HTML5, and one being JavaScript. In terms of HTML5, the focus was mainly formatting and allowing the objects and video elements in relation to the corresponding JavaScript to be acquired and displayed properly. In terms of JavaScript, the focus was to implement the change in functioning of the WebRTC application specifically to implement and mitigate covert channels. Once the connection was altered to simulate local and remote in one webpage, we used the canvas element to implement covert signaling between the local system and the remote system. The canvas element can be used to introduce a delay when sending the bit one, and a small little delay when sending the bit zero. Image filtering utilizes a drawToCanvas function which will relay the data from the local video to the canvas element, which then allows the remote connection to call the data being transmitted from the canvas. Beyond this, using WebRTC API's 'getStats()' it was able for each client to determine certain statistics such as the bitrate. Based on these statistics, the receiving client is able to sense the bit being sent from the fluctuating bitrate. Specifically in our example, we can see when the bitrate goes below 500, we know that the bit of one has been received, while when the bitrate is above 2000, we know that a bit of zero has been received. Jitter rate was useful in measuring video quality but because of the complexity, we did not implement this on our own. The built-in statistics module does not provide jitter calculation when canvas is used. Lastly, it was then vital to delay the data based on user input (one or zero) by adding this into the drawToCanvas function. The delay would then occur if an input bit were one, and a very small delay would occur if the input were zero. A small delay was needed to determine if a zero was sent, because this allows for the bitrate to fluctuate in a manner that is different from a one being sent. Based on these certain inputs, it was seen that the bitrate would either fluctuate to a very high, or very low amount. There was a slight anomaly that propelled this implementation, because when adding in a very small delay, this manipulated the bitrate to increase drastically for a very short period. Data compression can reduce the bit rate and could be mistaken as a one being received. Another vital concern with this method is network traffic, as the bit is being received based specifically on a bitrate, and testing occurs with this prototype at the forefront, where adding or removing network traffic can result in varying bitrates.

3.5 Secure Prototype

After successful implementation of an unsecure prototype, the application needs to be altered to mitigate the low error rates of transmitting of bits through covert channels. There are many routes in which this can be attempted, but instead we tried two. One, with a constant delay, and two, with a random delay. These two methods may be implemented at a proxy between the two ends. This involves adding in an interval section that will add in a delay, with intentions of increasing the error rate of bits sent through the covert channel. With a constant delay, this may appear to work somewhat, but it would be very common that the sender and receiver could see the constant delay, and then alter the sending and receive methods accordingly. We still wanted to test the results of a constant delay, even though this could be overridden, in an effort to further compare performance. With a random delay, this is the best approach to take when mitigating covert channels via image filtering, due to the inability of the exploiter to sense the random delay each time.

3.6 Distributed Hash Table

The second issue that we addressed is IP leaks. For devices to communicate with each other using WebRTC, each needs to know each other's real IP addresses [12]. This will allow for the other client, and more frightening, a 3rd party application the ability to detect and misuse the original client's real IP address. The current approach of the server-side implementation of a WebRTC application involves sending data through a series of *STUN/TURN* protocols using web sockets. Instead, it is possible to shift focus to implementing a similar server side as seen with the Tor browser.

A new approach would be taking similar action to the Tor browser's implementation which has been made famous due to its ability for users to remain completely anonymous. Tor involves using a series of nodes, and a similar action can be taken using WebRTC. An example set of transfer nodes can exist as follows: connecting to a random publicly listed node, relaying that traffic through another random middle node, and finally transmitting that information through a final exit node [13].

Three levels of nodes will need to exist when prior to implementing a distributed hash table with a WebRTC application: entry, relay, and exit. To allow for randomness each time, it is ideal to have a minimum of three to five nodes per set (entry, relay, and exit). Through this relay of data transfer, the original address that is being initiated and the receiving end will both remain private in relation to each other. With all nodes initiated and being created with the purpose to actively send and receive data, next is to implement a distributed hash table. This can be done using C/C++ or Google's programming language Go [14]. This process involves creating a lookup function, which will randomly allow for any of the 9-15 data transfer nodes being online, with at least more than 60% being required to be online to ensure that the route is different each time. It is vital to have a different route each time, to have anonymity between the sender and receiver. Once the Distributed Hash Table has the correct nodes initiated, a lookup function needs to be created. This lookup function will have to choose, at random, an entry or exit level node dependent upon who is receiving data, and then create a route based on a DHT algorithm $((n+2^{i-1}) \bmod 2^m)$ [14]. Extending upon this, which is something that exactly may not have any prior work in relation to using a DHT, a send and receive function need to be created. These both will have to send and receive the data that is being transmitted by each client, properly eliminating data after being sent. In doing so, once the data is transferred throughout the nodes, using WebRTC API a peer connection can be built and the audio and video tracks can be sent, with complete anonymity in relation to the client's IP address.

To sum up the structure of how this process would work:

1. Nodes are initiated at three levels (entry, relay, and exit) and made online/offline.
2. A peer connection is initiated.
3. Lookup function determines a functional route with at least one entry node, one relay node, and one exit node.
4. Send function takes place, which will relay the IP address from each node, tossing away the data at each point until each entry and exit node are given to the respective client.
5. Using the IP address of the entry and exit node, a peer connection is built.
6. Data transfer will occur using the WebRTC API, sending tracks of video and audio to the nodes.

7. Receive Function will take place which will act the same as the send function but relay it back to the respective client.

Once each and every step is completed, this will correlate into eliminating IP leaks with a WebRTC application. A distributed hash table will prevent the public IP address of the sending and receiving clients to be known. Although this will ultimately eliminate this concern, through proper implementation and examination it may be found that this is not ideal when looking at performance.

CHAPTER 4

FINDINGS & ANALYSIS OF DATA

This chapter discusses the findings that were discovered through the analysis of the unsecure prototype that establishes covert channels, and the secure prototype which mitigates covert channels with a series of delays.

4.1 Covert Channels

Expanding upon an unsecure prototype implementation, proper testing needs to occur in which the effectiveness can be determined of this method. There are two main issues which can occur from implementing delays in a real-time media application, video quality and the error rate at which the bit is being sensed. To test this, baseline statistics need to be measured in which no delay or prevention method have occurred. For baseline measurements, ten different tests were taken for each chart, five using a *black screen* as the media source, and five using a *sample presentation*. One main difference that was noticed between these two methods was that when using the sample presentation method, the bitrate remained much more consistent which led to a small increase in error rate. We believe this is different than black screen testing because the bitrate is easier to manipulate with black screen testing, due to data compression techniques.

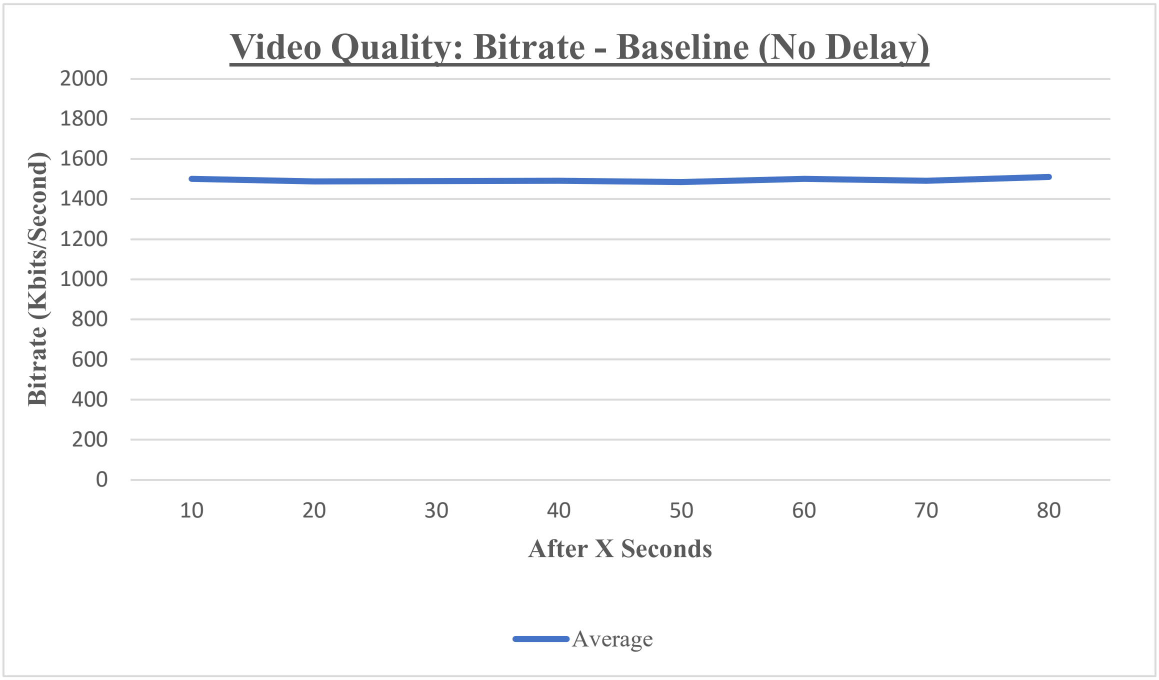


Figure 4.1 Video Quality: Bitrate – Baseline (No Delay)

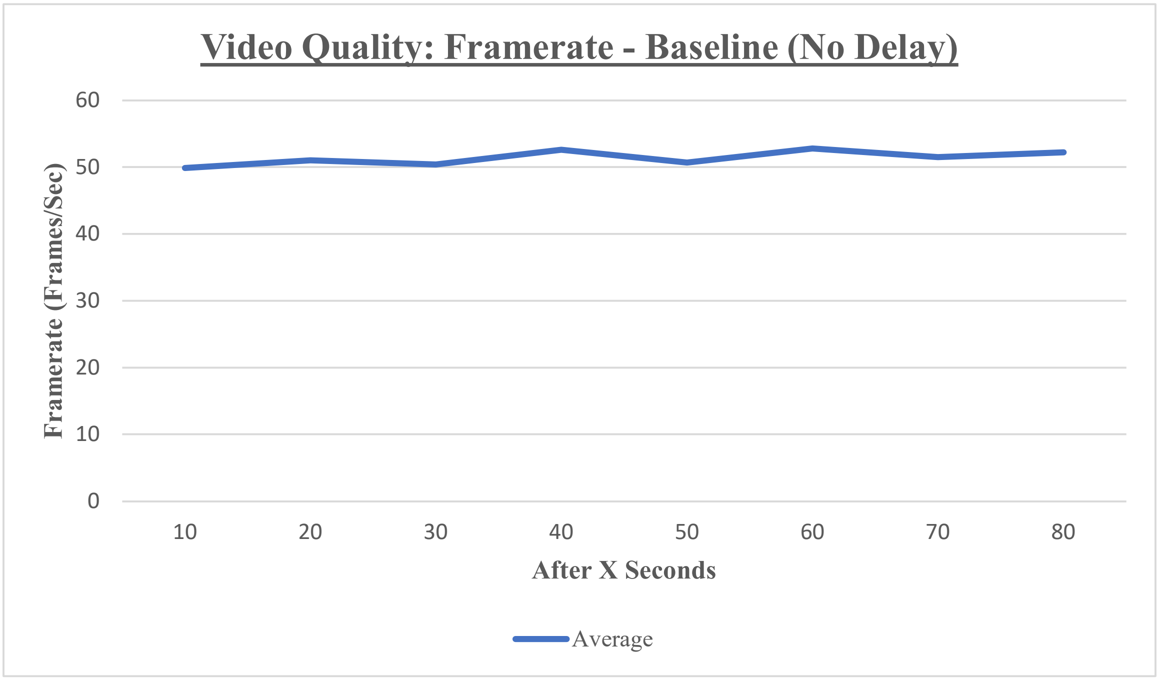


Figure 4.2 Video Quality: Framerate – Baseline (No Delay)

Based on the testing results above, the baseline measurements for this WebRTC application without any delay or mitigation methods involved is the bitrate sitting around 1500 kbits/second, and the framerate sitting around 50 frames/second. This provides a baseline measurement to judge performance statistics in comparison to the delay method implemented along with the mitigation/prevention method. For all testing purposes, these tests and later tests were completed using the same network, equipment, and a series of ten different results. The network still played a role even though both clients were done on the same computer, because the WebRTC API called still requires a network connection be used in sending and receiving data.

Next, testing needs to occur involving using the delay method which allows covert channels to occur and a bit to be sent and received from one client to another. Each test involved is measuring two separate elements, the bitrate and framerate based on the covert channel bandwidth. Covert channel bandwidth is a statistic that takes the number of bits being sent divided by the time at which the input is being calculated (1 / inputRate or T). The sampling at the receiving end is scheduled at the regular interval of T (inputRate), which means that one bit can be received per amount of T leading to the equation.

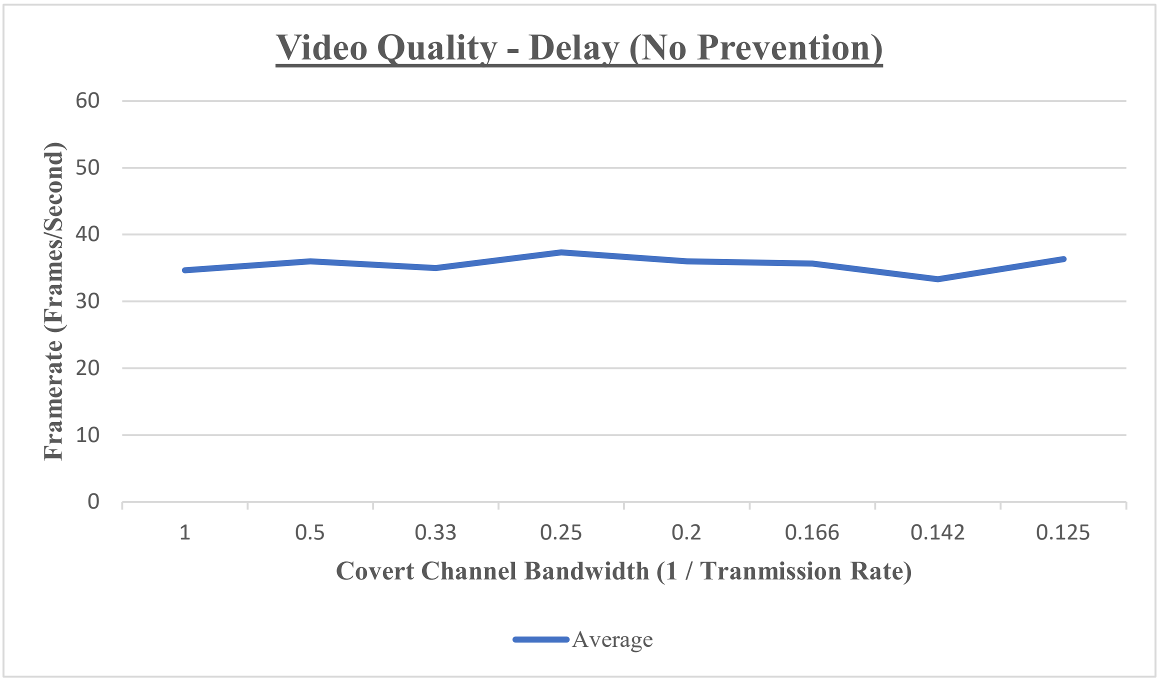


Figure 4.3 Video Quality – Delay (No Prevention)

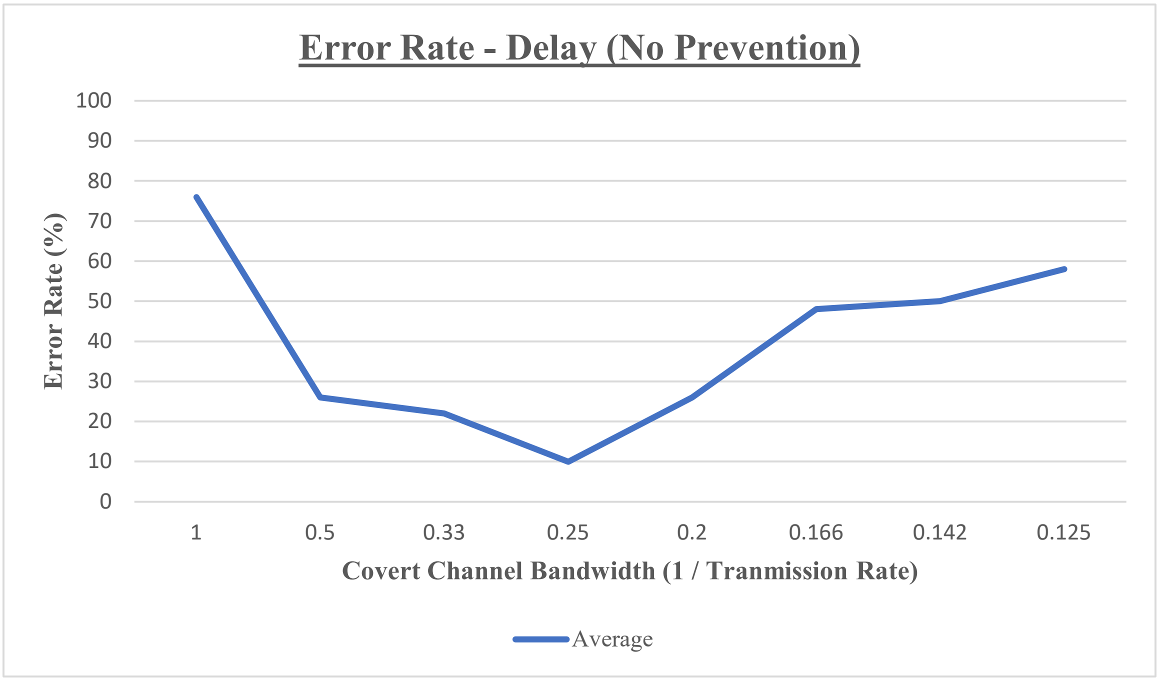


Figure 4.4 Error Rate – Delay (No Prevention)

When looking at Figure 4.4, the error rate appears to flucuate greatly depending upon the amount of time between each bit being sent (covert channel bandwidth). The error rate is high for low value of T (inputRate) because it takes time for the bitrate to stabilize. A similar action occurs when the bitrate is able to stablize for an extended period, as the error rate also increases when too much time is given between a bit being sent and received. It appears that roughly two to four seconds is the ideal time in being able to send, and then receive the bit on the remote connection. When looking at the video quality, it seems that the average is roughly 15 frames less per second in comparison to the baseline. Although this is drastic in comparison, this is a small price to pay when implementing covert channels with a WebRTC Application. Many of these statistics and variables used throughout implementation can vary greatly depending on the network and equipment being used.

4.2 Prevention Method

Expanding upon a secure prototype implementation, proper testing needs to occur. This will be compared to the previous results seen with baseline measurements as well as the delay method. The same number and variables are going to be used in extension of secure testing, and the results are as followed.

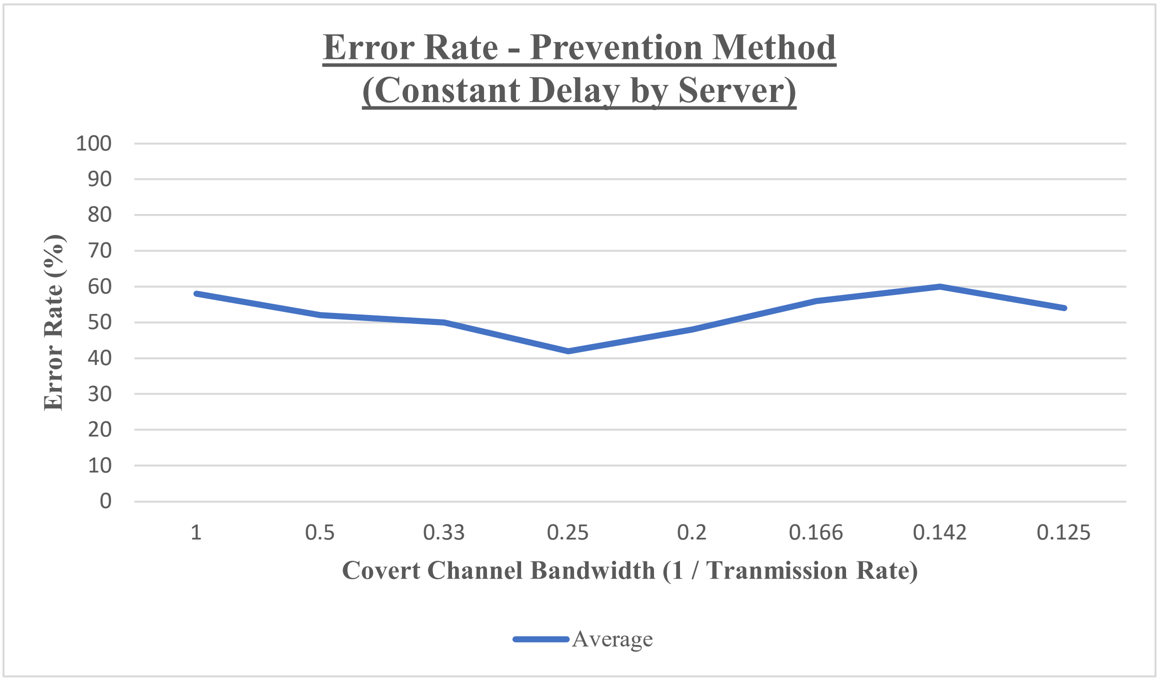


Figure 4.5 Error Rate – Prevention Method (Constant Delay by Server)

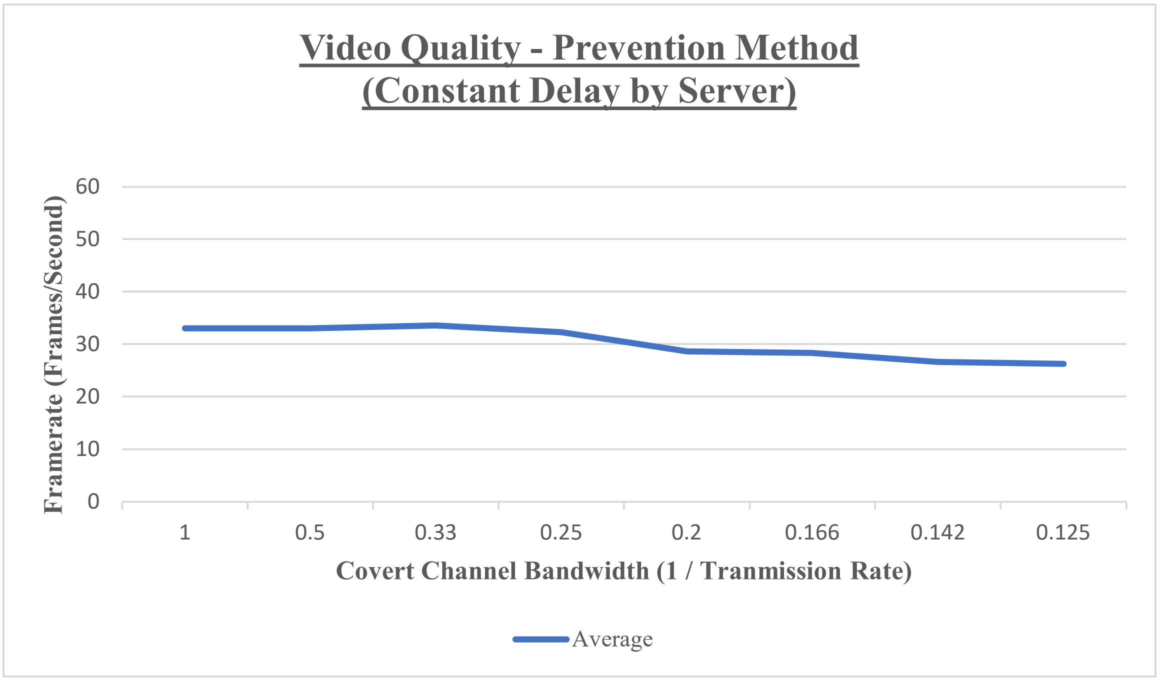


Figure 4.6 Video Quality – Prevention Method (Constant Delay by Server)

First, the error rate and video quality will be examined using a constant delay. It is seen when looking at Figure 4.5, the error rate actually decreases when using a constant delay. This is believed to be as a result of the bitrate stabilizing quicker than expected due to the constant noise. Although the error rate is not 100% for each covert channel bandwidth, and in some instances the error rate actually decreased, it is generally still much better in comparison to the baseline. The error rate seems to have stabilized when implementing a delay on the server side. This would appear to be a result of the bitrate not being able to fluctuate as frequently due to the noise. Although this is does not conquer at achieving a 100% error rate, it appears to be somewhat stable throughout each test group. When looking at Figure 4.6, it is seen that video quality is a little bit worse in comparison to the delay method, but not enough to allow for added confidentiality violations.

Next, the same tests have been conducted using the same variables and measures but instead with a random delay.

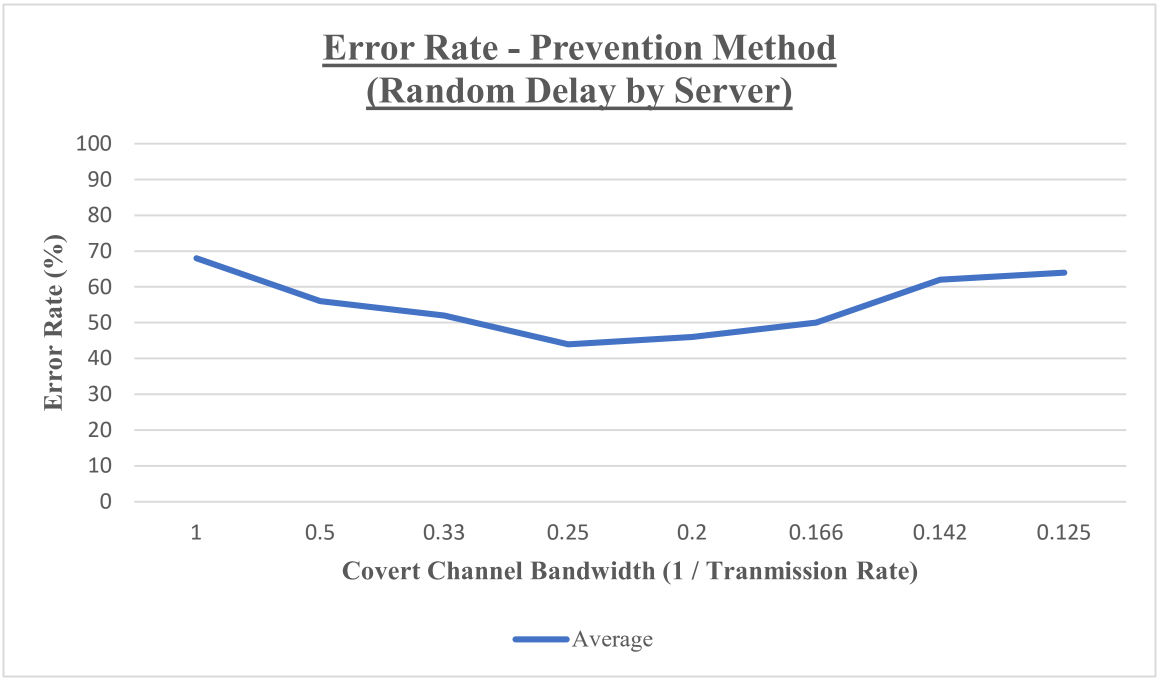


Figure 4.7 Error Rate – Prevention Method (Random Delay by Server)

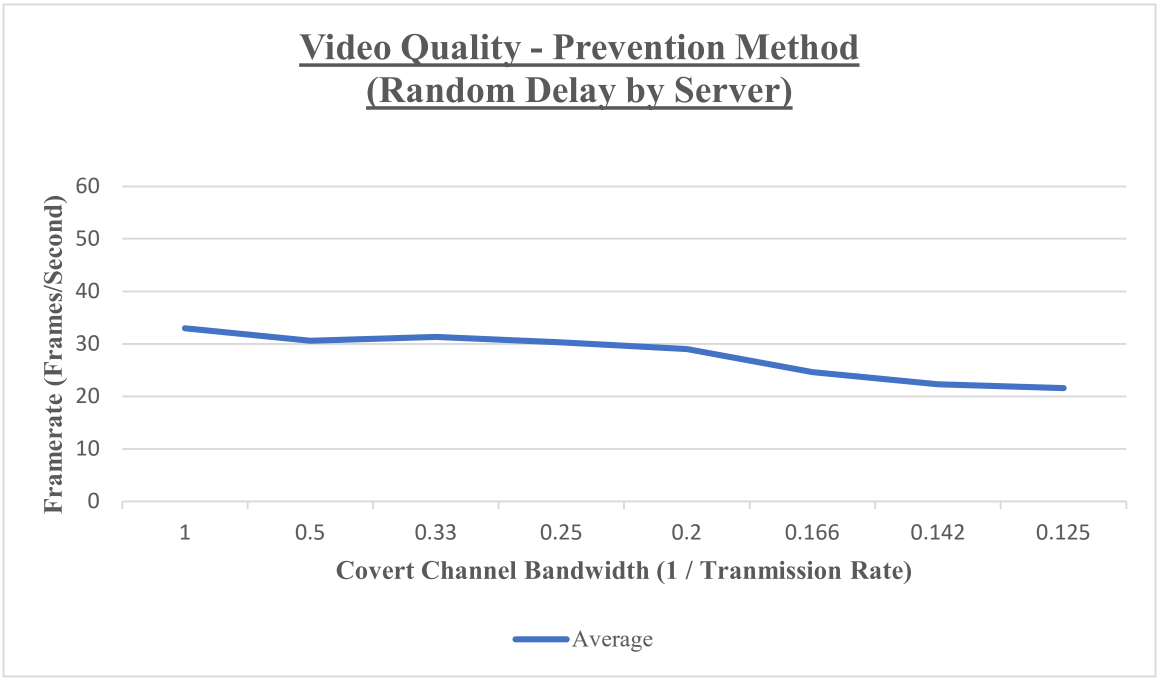


Figure 4.8 Video Quality – Prevention Method (Random Delay by Server)

These results are quite similar to the delay implementation as a constant, but in terms of error rate this seems to fluctuate more than as a constant. I believe that this is due to the random element that is applied to the noise, where it cannot always be assumed this is better than a constant rate. Although this may appear to be very similar in performance in comparison to a constant delay, this would definitely be the ideal choice due to the fact that when using a covert channel, the client would not be able to determine the delay and alter the code specifically to counter the delay.

CHAPTER 5

SUMMARY AND CONCLUSIONS

Through the development of this thesis, we studied two large security concerns surrounding a WebRTC application: confidentiality violations and IP leaks. Our study found that WebRTC is susceptible to covert channel attacks. This seemed to provide some decrease in performance when exfiltrating data through covert channels, but overall, nothing too over the top. This was then mitigated through the result of constant and random delays mechanisms. Mitigations further resulted in performance concerns, but greatly increased the error rate at which a bit can be sent and received through convert channel implementation. It is important to notice that results may vary greatly considering each testing procedure was done with a series of random bits being sent each time. This is because for testing purposes only two bits were tested, one and zero. Since there were only two bits being used, it is definitely possible that the receiving end could almost guess the bit being received, instead of being precise each time. It may be better to implement characters or such to get more of a variance in results. Also, with data transfer, the latency between two network connections can vary greatly. It also is vital to acknowledge that since we are only sensing two different data rates, it would be good to reset the bit when no bit is received. This would possibly increase the error rate for the secure prototype because the bit being sensed would be able to be differentiated at a much higher rate. It is also noticed that using the sample presentation testing method, which is a realistic scenario, the effect of data compression in error rate is minimal.

5.1 Future Work

In terms of future work, there are quite a bit of possibilities on the direction of which this project could continue. First, it is vital to address the network concern that this was not done throughout two separate network connections. This is the thought that instead of implementing a WebRTC application with covert channel implementation through one webpage, this be done using separate networks and with completely separate clients. Although this is a concern, and should be tested appropriately, our results still are valid to due to the fact that with WebRTC API, this still involves sending data through a network connection. When establishing covert channels in a WebRTC application using two completely separate clients, it is believed that be simply adjusting variables such as the minimum and maximum bitrate in which the bit is being sensed could solve any concerns. Next, ideally, an audio delay would need to match up with the video delay. Currently, only the image and video being displayed is altered, while matching up the audio would mask the data transformation even more. At first glance, this may involve dropping audio packets to match up with the drop in video packets. Finally, it would be incredible to see an actual prototype implementation of the distributed hash table server implementation with a WebRTC application that was examined previously. This could provide a drastic change in the privacy of real-time communication applications if IP leaks are mitigated and may be groundbreaking to the technology industry.

References

[1] B. Jansen, T. Goodwin, V. Gupta, F Kuipers, G. Zussman, “Performance Evaluation of WebRTC-based Video Conferencing,” in *Performance Evaluation Review*, *ACM*, vol. 45, no. 3, pp. 56-68, 2018.

[2] K. Stone, (2020, July 7). *The State of Video Conferencing in 2020* [Online]. Available: https://getvoip.com/blog/2020/07/07/video-conferencing-stats/

[3] N. Blum, S. Lachapelle, H. Alvestrand, “WebRTC – Realtime Communication for the Open Web Platform,” in *Queue*, *ACM*, vol. 19, pp. 1-30, 2021.

[4] B. Mordechai, “Constructivism in Computer Science Education,” in *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science*, *SIGCSE ’98*, pp. 257-261, 1998.

[5] B. Lampson, “A Note on the Confinement Problem,” in *Comunications of the ACM*, 1973.

[6] C. Heinz, W. Mazurczyk, L. Caviglione, “Covert Channels in Transport Layer Security,” in *Proceedings of the European Interdisciplinary Cyber Security Conference*, *EICC ’20*, pp. 1-6, 2020.

[7] A. Reiter, A. Marsalek, “WebRTC: Your Privacy is at Risk,” in *Proceedings of the Symposium on Applied Computing*, SAC ‘17, pp. 664-669, 2017.

[8] Donaldson, Scott (2018, April 10). *Using WebRTC for Real-Time Image Filtering.* [Online]. Available: https://sudo.isl.co/webrtc-real-time-image-filtering/

[9] Mozilla (2021, Apr. 10). *WebRTC API.* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API

[10] Tully, Shane (2018, Aug. 24). *GitHub-Shanet-WebRTC-Example.* [Online]. Available: https://github.com/shanet/WebRTC-Example

[11] Mozilla (2021, Feb. 21). *Pixel Manipulation with Canvas* [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas

[12] N. Yoshiuara, K. Lida, "Identification of Writing on Bulletin Board via Tor," in *Proceedings of the 3rd International Conference on Software Engineering and Information Management*, *ICSIM '20*, pp. 135-139, 2020

[13] Porup, J.M. (2019) *What is the Tor Browser? And how can it help protect your identity.* Retrieved: Dec. 10, 2020 https://www.csoonline.com/article/3287653/what-is-the-tor-browser-how-it-works-and-how-it-can-help-you-protect-your-identity-online.html

[14] Khan, Farhan Ali (2018) *Chord: Building a DHT in Golang* Retrieved December 10, 2020 https://medium.com/techlog/chord-building-a-dht-distributed-hash-table-in-golang-67c3ce17417b

Appendix

Communication Stage

There are two separate stages involved with WebRTC applications, a communication stage, and a signaling stage. The communication strictly focuses on how the two clients interact with each other, while the signaling stage focuses on how the server side is implemented. This project's almost entire focus is on the communication stage.

Local

Local means that when looking at a connection being made between two clients, the local connection is the initial client to be sending a connection request, and in doing so, is the client who will send a bit.

Remote

Remote means that when looking at a connection being made between two clients, the remote connection is the former client in which establishes a connection based on a previous request. In this implementation, this is simulated through WebRTC API and this client will be receiving the bit based on the local connection.

STUN/TURN

STUN/TURN are protocols that are extremely common with WebRTC applications. These are simply called using WebRTC API and most famously created by Google.

Black Screen

This is a type of testing that occurs by using a webcam and covering the camera, to allow for a media element to still be sent but to find hold similar results throughout various tests.

Sample Presentation

This is a type of testing that involves real-life examples of connection testing, and involved a webcam being on and an individual (myself) to be "giving a presentation" to simulate a real-time example of a WebRTC application.

APPENDIX A

SOURCE CODE

Webpage Implementation

Index.html

```
// Developed Using the Simple Example [8] and WebRTC API [7]
<div class="split left">
<div class="centered local">
        <video id="localVideo" playsinline autoplay muted width="640" height="360"></video>
        <h2>Local Video</h2>
        <div>
                <button id="getMedia">Get Media</button>
                <button id="connect" disabled>Connect</button>
                <button id="hangup" disabled>Hang Up</button>
        </div>
        <P><Strong>Get Media</Strong> -> <Strong>Connect</Strong> to start call</P>
        <P><Strong>Hangup</Strong> to End Call</P>
        <div id="inputBuffer"></div>
        <P></P><div>
        <textarea id = "dataChannelSend" maxlength="1" rows="2" cols="2" disabled></textarea
        >
        </div>
        <h2>Video to Canvas </h2>
                <canvas id="inputCanvas" width="640" height="360"></canvas>
        </div></div>

<div class="split right">
<div class="centered remote">
        <video id="remoteVideo" playsinline autoplay muted width="640" height="360""></vid
        eo>
        <h2>Remote Video</h2>
        <p>Video will appear when a peer is connected, below is the bitrate, jitter, and RTT.</p>
        <div id="bitrate"></div>
        <div id="jitter"></div>
        <div id="RTT"></div>
        <div id="frames"></div>
        <textarea id = "dataChannelReceive" rows="2" cols="2" disabled></textarea>
        <div id="ccBandwidth"></div>
        <div id="errorRate"></div>
</div>
</div>
```

Server-Side Implementation

Server.js – As Simple as It Gets Example [8]

```
const HTTPS_PORT = 8443;

const fs = require('fs');
const https = require('https');
const WebSocket = require('ws');
const WebSocketServer = WebSocket.Server;

const serverConfig = {
	key: fs.readFileSync('key.pem'),
	cert: fs.readFileSync('cert.pem'),
};

const handleRequest = function(request, response) {
console.log('request received: ' + request.url);

if(request.url === '/') {
	response.writeHead(200, {'Content-Type': 'text/html'});
	response.end(fs.readFileSync('client/index.html'));
} else if(request.url === '/webrtc.js') {
	response.writeHead(200, {'Content-Type': 'application/javascript'});
	response.end(fs.readFileSync('client/webrtc.js'));
}};

const httpsServer = https.createServer(serverConfig, handleRequest);
httpsServer.listen(HTTPS_PORT, '0.0.0.0');

const wss = new WebSocketServer({server: httpsServer});

wss.on('connection', function(ws) {
ws.on('message', function(message) {
	console.log('received: %s', message);
	wss.broadcast(message);
	});
});
```

```
wss.broadcast = function(data) {
	this.clients.forEach(function(client) {
	if(client.readyState === WebSocket.OPEN) {
		client.send(data);
	}
	});
};

console.log('Server running. Visit https://localhost:' + HTTPS_PORT + ' in Firefox/Chrome.\n\n\
');
```

Client-Side Implementation

Webrtc.js

```
// Basis Provided Using the Simple Example [8] and WebRTC API [7]
var localVideo;
var remoteVideo;
var uuid;
var serverConnection;

let sendChannel;
let receiveChannel;

var delayTime;
var delay;
var data;
var received = false;
const dataChannelSend = document.querySelector('textarea#dataChannelSend');
const dataChannelReceive = document.querySelector('textarea#dataChannelReceive');

var randomDelay;                                        //Prevention Method
var start;
var end;

let bitrate;
let jitter;
let RTT;
let framesPer;

let inputBuffer = new Array(5);                         // Size of Input Buffer
let outputBuffer = new Array(inputBuffer.length);       // Output Match Input

var x = 0;                                              // inputBuffer
var y = 0;                                              // outputBuffer
var firstSet = 0;                                       // firstSet of Input & Output Rates
var e = 0;                                              // error Count
var c = 0;                                              // Count when Calculating Error
var inputRate = 4000;                                   // Beginning Input Rate
var outputRate = inputRate + (inputRate / 2);           // Beginning Output Rate
var t = inputRate;                                      // Input Rate Variable
var statRate = 850;                                     // statRate
```

```
let multiplyRate = 1;                              // Increment Rate
var prevIn;                                        // Previous Input Rate
var currIn;                                        // Current Input Rate
var current = 0;                                   // Determing Current Input Rate
var online;                                        // Connection is Online/Offline
var inputOccured;                                  // Input Has Begun
var ccBandwidth = (1/t) * 1000;                    // Covert Channel Bandwidth

var i;
for (i = 0; i < inputBuffer.length; i++){
        var random = Math.random();
        if(random > 0.5){
                inputBuffer[i] = 1;
        }
        else if(random < 0.5){
                inputBuffer[i] = 0;
        }
}

var inputCanvas = document.getElementById('inputCanvas').getContext( '2d' );
const outputStream = document.getElementById('inputCanvas').captureStream();
var width = 640;
var height = 360;

let localPeerConnection;
let remotePeerConnection;
let localStream;
let bytesPrev;
let jitterPrev;
let timestampPrev;

const getUserMediaConstraintsDiv = document.querySelector('div#getUserMediaConstraints');
const bitrateDiv = document.querySelector('div#bitrate');
const jitterDiv = document.querySelector('div#jitter');
const rttDiv = document.querySelector('div#RTT');
const frameDiv = document.querySelector('div#frames');
const errorRateDiv = document.querySelector('div#errorRate');
const inputBufferDiv = document.querySelector('div#inputBuffer');
const ccBandwidthDiv = document.querySelector('div#ccBandwidth');
ccBandwidthDiv.innerHTML = `<strong>Covert Channel Bandwidth: </strong>${ccBandwidth
}`;
inputBufferDiv.innerHTML = `Sending/Receiving <strong>${inputBuffer.length}</strong> bits`
;
```

```
const peerDiv = document.querySelector('div#peer');
const senderStatsDiv = document.querySelector('div#senderStats');
const receiverStatsDiv = document.querySelector('div#receiverStats');

const getMediaButton = document.querySelector('button#getMedia');
const connectButton = document.querySelector('button#connect');
const hangupButton = document.querySelector('button#hangup');

getMediaButton.onclick = getMedia;
connectButton.onclick = createPeerConnection;
hangupButton.onclick = hangup;

// Establish Peer Connection [8]
var peerConnectionConfig = {
	'iceServers': [
		{'urls': 'stun:stun.stunprotocol.org:3478'},
		{'urls': 'stun:stun.l.google.com:19302'},
	]
};

// Stop Peer Connection (Hangup)
function hangup() {
	console.log('Ending call');
	localPeerConnection.close();
	remotePeerConnection.close();

	// Query stats One Last Time.
	Promise
		.all([
			remotePeerConnection
			.getStats(null)
			.then(calcStats, err => console.log(err))
		])
		.then(() => {
			localPeerConnection = null;
			remotePeerConnection = null;
		});

	// Receive all Tracks
	localStream.getTracks().forEach(track => track.stop());
	localStream = null;
```

```
        // Disable Buttons
        hangupButton.disabled = true;
        getMediaButton.disabled = false;
}

// On Media Success, Create Stream
function getUserMediaSuccess(stream) {
        connectButton.disabled = false;
        console.log('GetUserMedia succeeded');

        localStream = stream;
        localVideo.srcObject = stream;

        //preventionMethod();
        drawToCanvas();
}

// Acquire Media from Client
function getMedia() {
        getMediaButton.disabled = true;
        localVideo = document.getElementById('localVideo');
        remoteVideo = document.getElementById('remoteVideo');

        if (localStream) {
                localStream.getTracks().forEach(track => track.stop());
                const videoTracks = localStream.getVideoTracks();
                for (let i = 0; i !== videoTracks.length; ++i) {
                        videoTracks[i].stop();
                }
        }

        var constraints = {
                video: true,
                audio: true   }
        navigator.mediaDevices.getUserMedia(constraints)
                .then(getUserMediaSuccess)
                .catch(e => {

        const message = `getUserMedia error: ${e.name}\nPermissionDeniedError may mean inv
        alid constraints.`;
                alert(message);
                console.log(message);
                getMediaButton.disabled = false;
    });}
```

```
// Create Peer Connection
function createPeerConnection() {
	connectButton.disabled = true;
	hangupButton.disabled = false;

	bytesPrev = 0;
	timestampPrev = 0;
	localPeerConnection = new RTCPeerConnection(null);
	remotePeerConnection = new RTCPeerConnection(null);

	// Acquire the outputCanvas's Video Stream, which will then be brought into the Remote
	Stream
	localStream = outputStream;
	localStream.getTracks().forEach(track => localPeerConnection.addTrack(track, localStre
	am));

	console.log('localPeerConnection creating offer');
	localPeerConnection.onnegotiationneeded = () => console.log('Negotiation needed – local
	PeerConnection');
	remotePeerConnection.onnegotiationneeded = () => console.log('Negotiation needed – re
	motePeerConnection');
	localPeerConnection.onicecandidate = e => {
		console.log('Candidate localPeerConnection');
		remotePeerConnection
			.addIceCandidate(e.candidate)
			.then(onAddIceCandidateSuccess, onAddIceCandidateError);
	};

	remotePeerConnection.onicecandidate = e => {
		console.log('Candidate remotePeerConnection');
	localPeerConnection
		.addIceCandidate(e.candidate)
		.then(onAddIceCandidateSuccess, onAddIceCandidateError);
	};
```

```
                remotePeerConnection.ontrack = e => {
                if (remoteVideo.srcObject !== e.streams[0]) {
                        console.log('remotePeerConnection got stream');
                        remoteVideo.srcObject = e.streams[0];
                }
        };

        localPeerConnection.createOffer().then(
        desc => {
                console.log('localPeerConnection offering');
                localPeerConnection.setLocalDescription(desc);
                remotePeerConnection.setRemoteDescription(desc);
                remotePeerConnection.createAnswer().then(
        desc2 => {
                console.log('remotePeerConnection answering');
                remotePeerConnection.setLocalDescription(desc2);
                localPeerConnection.setRemoteDescription(desc2);
        },
                err => console.log(err)
        );
        },
        err => console.log(err)
        );
}

function preventionMethod(){
  randomDelay = Math.floor(Math.random() * 100000000) + 10000000;
}

// Draw to Canvas
function drawToCanvas() {
        // Draw Video from Input Canvas
        inputCanvas.drawImage( localVideo, 0, 0, width, height );

        data = dataChannelSend.value;

        if (data == 1){
                delayTime = 100;
                setTimeout(pause, 3000);
        }
        else if (data == 0){
                delayTime = 5;
                setTimeout(pause, 3000);
		}
		else{
			clearTimeout(fx);
		}

		delay = setTimeout(drawDelay, delayTime);
		}

	function drawDelay(){
		requestAnimationFrame( drawToCanvas );
	}

	function pause(){
		//console.log("Paused");
	}

	function input(){
		if (x == inputBuffer.length){
			return;
		}
		dataChannelSend.value = inputBuffer[x];
		console.log("inputBuffer: " + inputBuffer[x]);
		x++;
	}

	function output(){
		outputBuffer[y] = dataChannelReceive.value;
		if (y == inputBuffer.length){
			return;
		}
		console.log("Delay: " + randomDelay);
		console.log("outputBuffer: " + outputBuffer[y]);
		y++;
	}

	// Calc Error when Output Buffer is Full
	function calcError(){
		let errorRate;
		for (c; c < inputBuffer.length; c++){
			console.log("in loop input: " + inputBuffer[c]);
			console.log("in loop output: " + outputBuffer[c]);
		if (inputBuffer[c] != outputBuffer[c]){
			e++;
			console.log("Error Count: " + e);}
		else if (outputBuffer[c] == null || outputBuffer[c] == ""){
			e++;
			console.log("Error Count: " + e);
		}
		}
		errorRate = (e / inputBuffer.length) * 100;
		errorRateDiv.innerHTML = `<strong>Error Rate: </strong>${errorRate}%`;
}

// Mozilla API Calls
function getRandomInt(max) {
		return Math.floor(Math.random() * Math.floor(max));
}

// Ice Canidate Success
function onAddIceCandidateSuccess() {
		console.log('AddIceCandidate success.');
}

// Ice Canidate Failure
function onAddIceCandidateError(error) {
		console.log(`Failed to add Ice Candidate: ${error.toString()}`);
}

// Calculate Video Bitrate, Jitter, and Network Latency (RTT)
function calcStats(results){
		results.forEach(report => {
			const now = report.timestamp;

		// Bitrate caluclated by using Incoming RTP Connection
		if (report.type === 'inbound-rtp' && report.mediaType === 'video') {
			const bytes = report.bytesReceived;

		if (timestampPrev) {
			bitrate = 8 * (bytes - bytesPrev) / (now - timestampPrev);
			bitrate = Math.floor(bitrate);
		}
		if (bitrate < 500){
			if (received == false){
				received = true;
			return;
		}
dataChannelReceive.value = 1;
//console.log('Received Bit: 1');
}
else if (bitrate > 1600){
        if (received == false){
        received = true;
        return;
}
dataChannelReceive.value = 0;
//console.log('Received Bit: 0');
}
bytesPrev = bytes;
timestampPrev = now;
}
// Jitter calculated by using Incoming RTP Connection
// Delay in Data Transfer
if (report.type === 'inbound-rtp') {
        jitter = report.jitter;
}
// Frames per Second
if (report.type === 'inbound-rtp') {
        framesPer = report.framesPerSecond;
}

// Total Round Trip Time by using Candidate Pair Connection
// Amount of time it takes a packet to get from client to server and back
if (report.type === 'candidate-pair') {
        RTT = report.totalRoundTripTime;
}

if (bitrate) {
        //bitrate += ' kbits/sec';
        bitrateDiv.innerHTML = `<strong>Bitrate: </strong>${bitrate} kbits/sec`;
}
if (framesPer) {
        //framesPer += ' frames/sec';
        frameDiv.innerHTML = `<strong>Framerate: </strong>
        ${framesPer} frames/sec`;
}
if (jitter) {
        //jitter += ' milliseconds';
        jitterDiv.innerHTML = `<strong>Jitter: </strong>${jitter} milliseconds`;)}
        if (RTT) {
                //RTT += ' milliseconds';
                rttDiv.innerHTML = `<strong>Total Round Trip Time: </strong>
                ${RTT} milliseconds`;
        }
        });
}

// Display statistics
setInterval(() => {
        if (localPeerConnection && remotePeerConnection) {
        remotePeerConnection
                .getStats(null)
                .then(calcStats, err => console.log(err));
                online = true;
        } else {
                console.log('Not connected yet');
        }
        }, statRate);

setInterval(() => {
        if (online == true && inputOccured == true){
        //online and input has occured
        }
        else{
                return;
        }
        if (outputBuffer[inputBuffer.length] != undefined){
                return;
        }
        if (localPeerConnection && remotePeerConnection){
                output();
        }
        if (firstSet == 0){
                setTimeout(pause, 0);
                multiplyRate++;
                prevIn = inputRate;
                inputRate = multiplyRate * t;
                currIn = inputRate;
                firstSet++;
        }
        else {
                multiplyRate++;
                outputRate += t;)}
        clearInterval();
        }, outputRate);

setInterval(() => {
        if (online == true){
                inputOccured = true;
        }
        else{
                return;
        }
        prevIn = inputRate;
        inputRate = multiplyRate * t;
        currIn = inputRate;
        if (current == 0){
                current++;
        }
        else if (currIn == prevIn){
                if (current == 1){
                        current++;
                }
                else{
                        return;
                }
        }
        if (outputBuffer[inputBuffer.length] != undefined){
                calcError();
                return;
        }
        if (localPeerConnection && remotePeerConnection){
                input();
        }
        clearInterval();
}, inputRate);
```