WebRTC Covert Channels by Image Filtering

An Abstract of a

Thesis Presented to the

Department of Computer Science

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]. This type of technology has become a main-stay in all industries, and more specifically WebRTC being open-source has allowed for researchers and developers across the world to discover new advancements and heights of real-time communication without prior installations or requirements. Although WebRTC is increasingly used for real-time video communications, there are still concerns with this being relatively new. This study will look further into security issues regarding these types of applications, along with briefly discussing the issue of solving IP leaks through a distributed hash table. To further develop this topic, a simple WebRTC application has been created with the purpose of discovering if WebRTC is susceptible to covert channel implementation, and if so, how to mitigate this. This is examined through a number of statistics, focusing on the bitrate, framerate, covert channel bandwidth and error rate. Furthermore, this issue will help develop defensive measures to help protect organizations using WebRTC as changes can be done at the client end without needing any additional access by a user.

APPROVAL PAGE

This thesis by **Dennis McMeekan** is accepted in its present form by the Department of Computer Science of Western Illinois University as satisfying the thesis requirements for the degree Master of Computer Science.

Chairperson, Examining Committee

Member, Examining Committee

Member, Examining Committee

Date

WebRTC: Covert Channels by Image Filtering

A Thesis Presented to the

Department of Computer Science

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 ONE

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, “Live chat has become the leading digital contact method for online customers, as a staggering 46% of customers prefer live chat compared to just 29% for email, and 16% for social media” [2]. With the reach of internet to all points of the world, 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. Roughly in the past ten years, there have been a number of open-source examples that have been centered around WebRTC that focused on the altering and transmission of data in real-time. These were vital in providing a sense of direction on how to implement covert channels and initiating/preventing data transformation. 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.

The emphasis for the research and project portion was confidentiality violations related to 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, 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. A prevention method will be implemented to help mitigate covert channel implementation, which involves adding constant or random delay mechanisms to delay data packets in the WebRTC stream. Adding the delays will make it harder to use covert channels. These experiments were done using WebRTC's API and different test elements such as the bitrate, framerate, and covert channel bandwidth in an effort to determine and judge the error rate.

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.

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.

CHAPTER TWO
BACKGROUND AND RELATED WORK

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

2.1 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.

2.2 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.

2.2.1 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. This obviously 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.

2.2.2 Covert Channels via Image Filtering

Expanding upon confidentiality violations, WebRTC real-time image filtering can be implemented to exploit the vulnerability of data being altered from one client to another. 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 [3].

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 without the administrator knowing. Taking this approach of image filtering involves not only one client first implementing a delay to send data, but also the receiving client sensing that delay to receive the sent data. This also to the network/server administrator would only appear to be noise, as the data rates tend to fluctuate very briefly before it returns to normal.

2.2.3 IP Leaks

A large issue with WebRTC applications, is IP leak concerns. This is due to the fact that a Peer-to-Peer connection requires that each client has each other's communication address. A similar issue was discussed and implemented with the creation of the Tor browser [4], but instead of using a simple WebSocket or HTTPS server, a Distributed Hash Table server was implemented that would allow for each client to still communicate directly with each other but remain anonymous. This provides the idea that in a similar fashion, this type of server communication can be implemented with a WebRTC application. Although this ideally would be implemented and have a prototype as is done with covert channel implementation, time did not permit during this thesis. Instead, later explained is a research document further describing this possibility and future courses of action.

CHAPTER THREE

RESEARCH METHODOLOGY & IMPLEMENTATION

This chapter describes the research steps taken to establish an understanding of how to approach and implement the subject of covert channels, as well as describing the steps needed in creating a distributed hash table server with a WebRTC application. With the extensive research and examination of open-source examples, a prototype implementation involving an unsecure and secure version have been created.

3.1 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.

3.1.1 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 [5]. 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 [6]. 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.

3.1.2 Image Filtering

Image filtering is the tool that connects a WebRTC application and 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, and this brought the idea that by transmitting a video element, into a canvas element, that the data would 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 [7]. This provided the basis that instead of implementing a filter, that a delay could be used to alter the data being sent and received. With the examination and analysis of these open-source examples, actual implementation is able to take place through an unsecure prototype, and a secure prototype.

3.2 Unsecure Prototype

The creation of an unsecure prototype is solely emphasized on creating covert channels while establishing a WebRTC connection between one peer to another. To begin, a simple application was created which built the connection using WebRTC API, a great example was found which made this as dead-simple as possible and provided a base to build upon [6]. The application was altered to allow for the connection and remote connection to be simulated and displayed on the same webpage. 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.

There were two main pieces of code that required being altered, one being HTML5, and one being JavaScript. In terms of HTML5 (index.html), 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 (webrtc.js), 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, image filtering needed to occur which would allow for covert channel implementation to be possible. This involved adding an extra step in between the local connection and the remote connection, using a canvas element. This expanded upon the example looked at, and instead of applying a filter to the data being transmitted, a delay occurs. 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. For unsecure and secure prototype purposes, it was determing there were three elements that needed to be examined, the incoming bitrate and framerate. It was hopeful to examine the jitter between the local and remote connection, but this proved to be problematic due to canvas implementation. This is believed to provide an unknown error to a WebRTC application, in terms of masking the network connection data being sent and received. 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. Based on these certain inputs, it was seen that the bitrate would either fluctuate to a very high, or very low amount. This would mean that the remote client would then be able to then sense the bit based on the bitrate because the bitrate would be altered for multiple seconds before stabilizing. There was a slight anomaly propelled this implementation, because when adding in a very small delay, this manipulated the bitrate to increase drastically for a very short period. This is believed to be due to data compression techniques in relation to data transformation with media elements in HTML5 and JavaScript.

3.3 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 two routes in which this approach can be taken. One, with a constant delay, and two, with a random delay. These two methods simulate an implementation by the administrative side through the server. This involves adding in an interval section that will add in a delay, with intentions of increasing the error rate. 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. 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.

3.4 Distributed Hash Table

Outside of prototype implementation, it is vital to address IP leaks and discuss the possibility of implementing a distributed hash table. “Any two devices talking to each other directly via WebRTC, however, need to know each other’s real IP addresses” [8]. 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. Instead, it is possible to shift focus to implementing a similar server side as seen with the Tor browser. 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.

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.

A series of nodes will need to be implemented. An example set of transfer nodes can exist as follows: "connects at random to one of the publicly listed entry nodes, bounces that traffic through a randomly selected middle relay, and finally spits out your traffic through the third and final exit node" [9]

Three levels will need to exist: entry, relay, and exit nodes. Roughly 3 to 5 per level, allowing that when one peer-to-peer connection is established, the nodes do not become overwhelmed by the traffic coming in and out. Through this relay of data transfer, the original address that is being initiated and the receiving end will both remain confidential in relation to each other. With all nodes initiated and being created with the purpose to actively send and receive data, it is vital to create a Distributed Hash Table that creates a series of transfers between each client. In the creation of a Distributed Hash Table, this can be done using C/C++ or Google's programming language Go [10]. 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. This can then be set to alter or change periodically, giving an even greater chance to continuously change course. 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}) mod 2^m) [10]. 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 different levels 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.

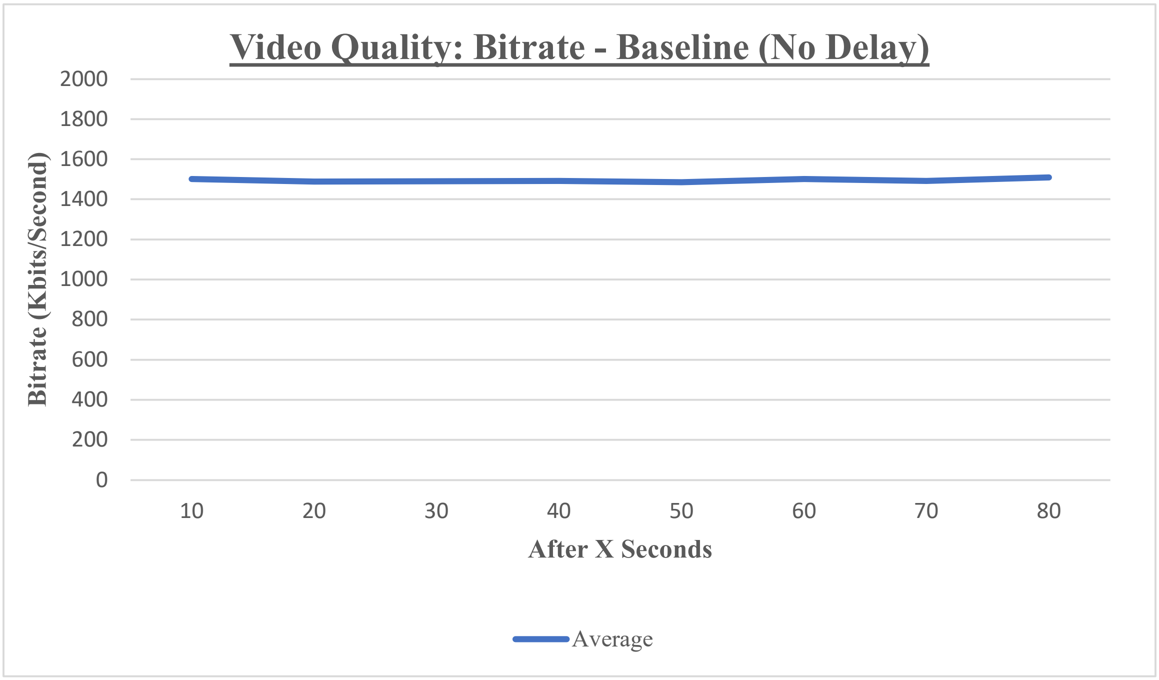
CHAPTER FOUR

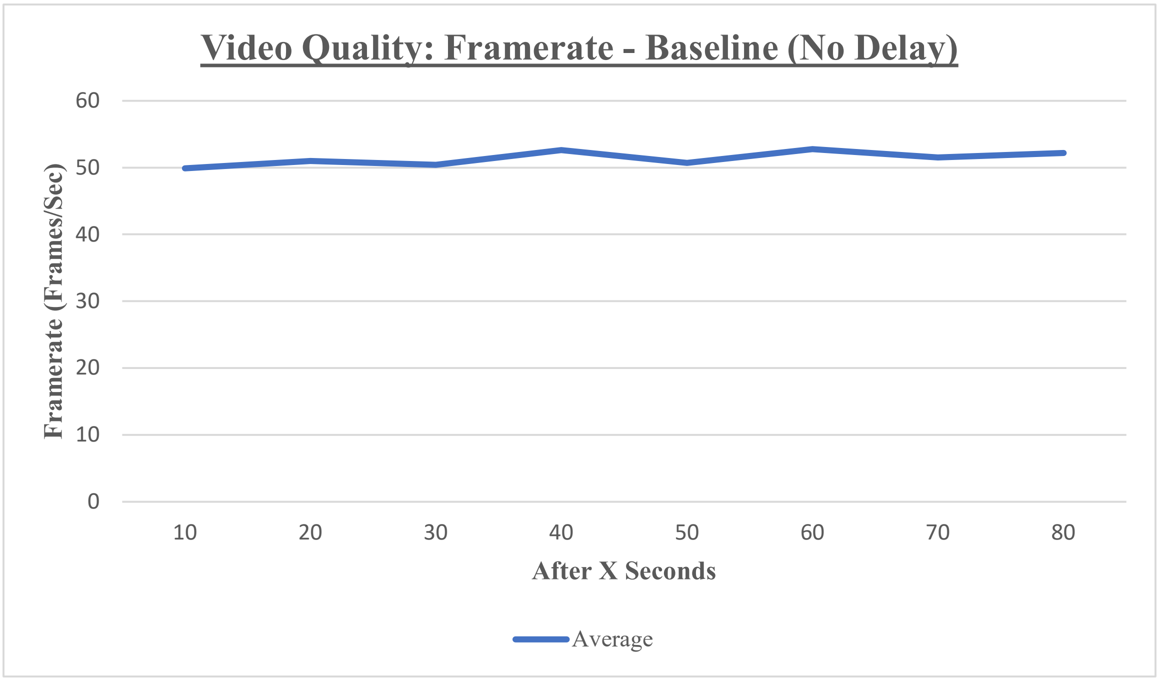
FINDINGS & ANALYSIS OF DATA

This chapter discusses the findings that were discovered in the implementation of an unsecure prototype, establishing covert channels, and a secure prototype which mitigates covert channels with a series of delays.

4.1 Covert Channels

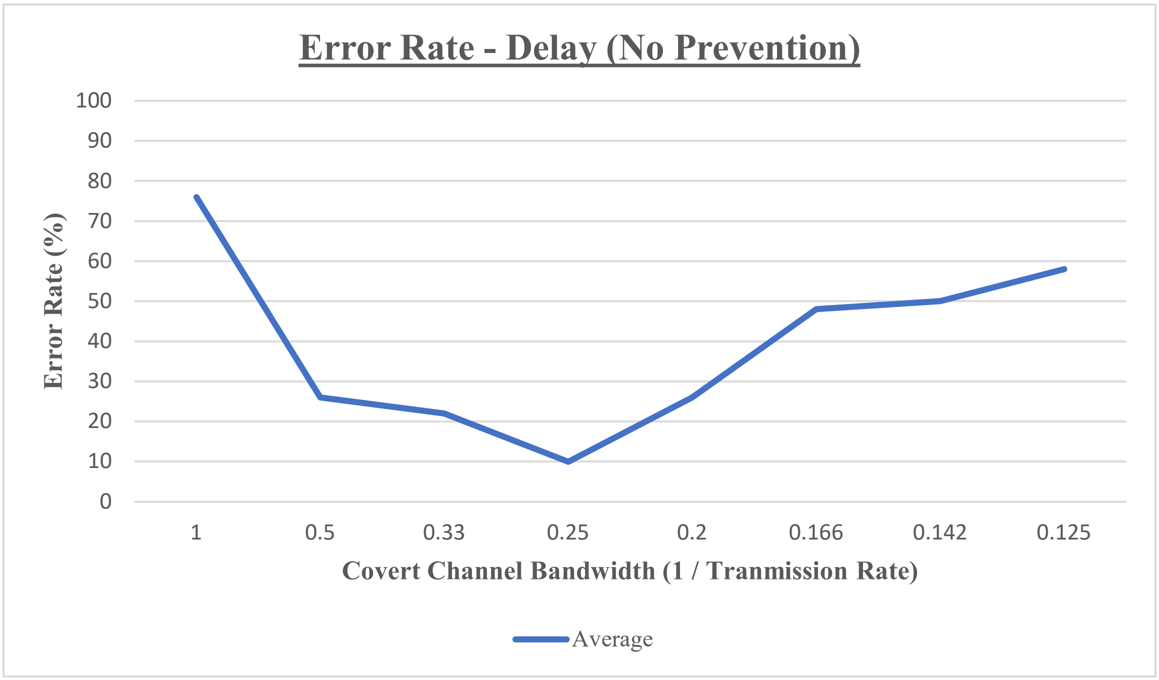
Expanding upon an unsecure prototype implementation, proper testing needs to occur in which the effectiveness can be determined of this method. There are two main issues which can occur from implementing delays in a real-time media application, video quality and the error rate at which the bit is being 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.*

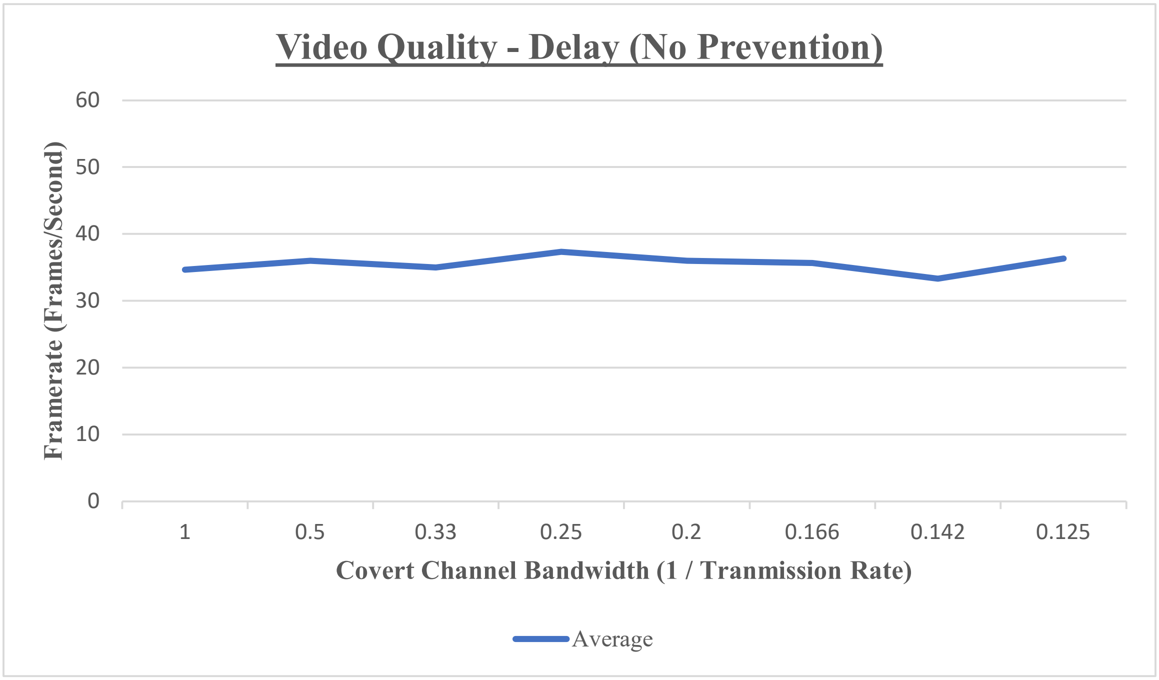




Based on the testing results above, the baseline measurements for this WebRTC application without any delay or mitigation methods involved is the bitrate sitting around 1500 kbits/second, and the framerate sitting around 50 frames/second. This provides a baseline measurement to judge performance statistics 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.

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).

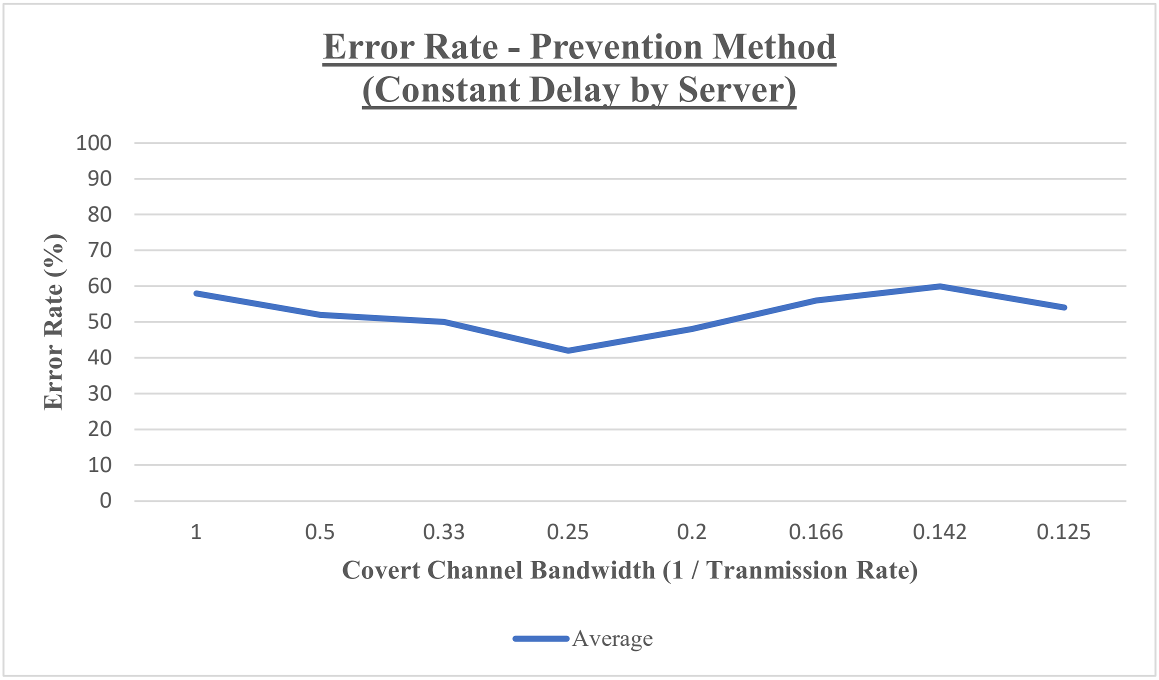


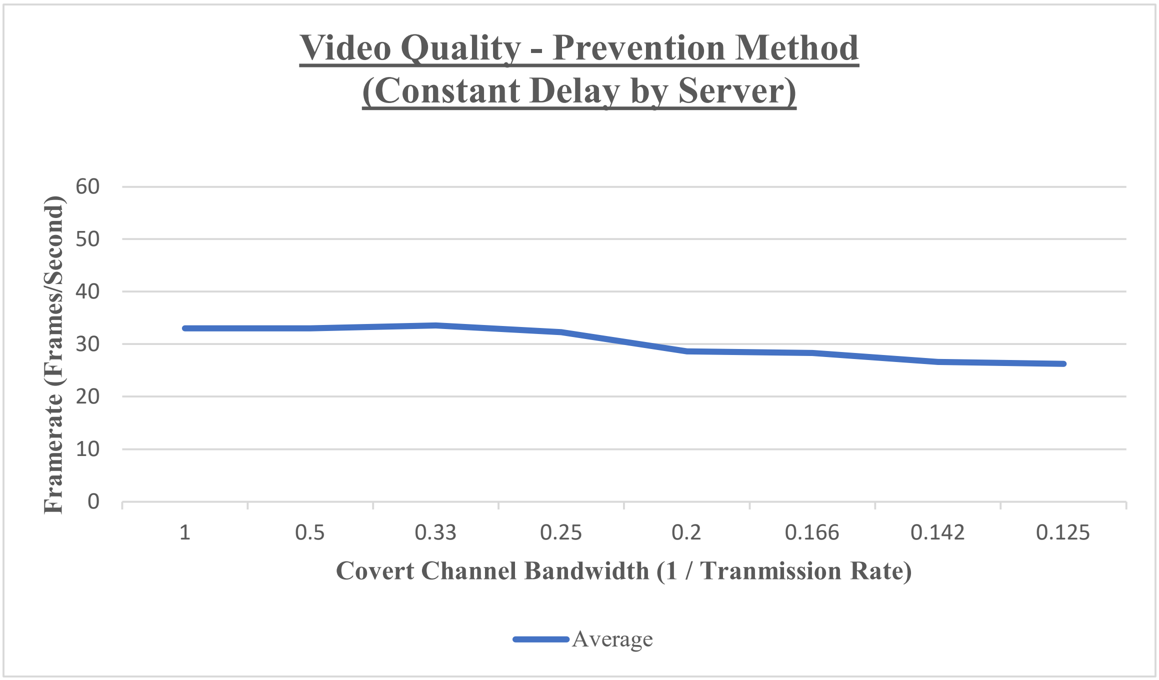


The error rate appears to flucuate greatly depending upon the amount of time between each bit being sent (covert channel bandwidth). This is because after a small amount of time, the bitrate stabilizes more to the averages we see around 1500, which means the error rate would greatly increase if the bitrate is not able 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, which requires a small effort in altering data to more of the norm of testing and prototype equipment and statistics.

4.2 Prevention Method

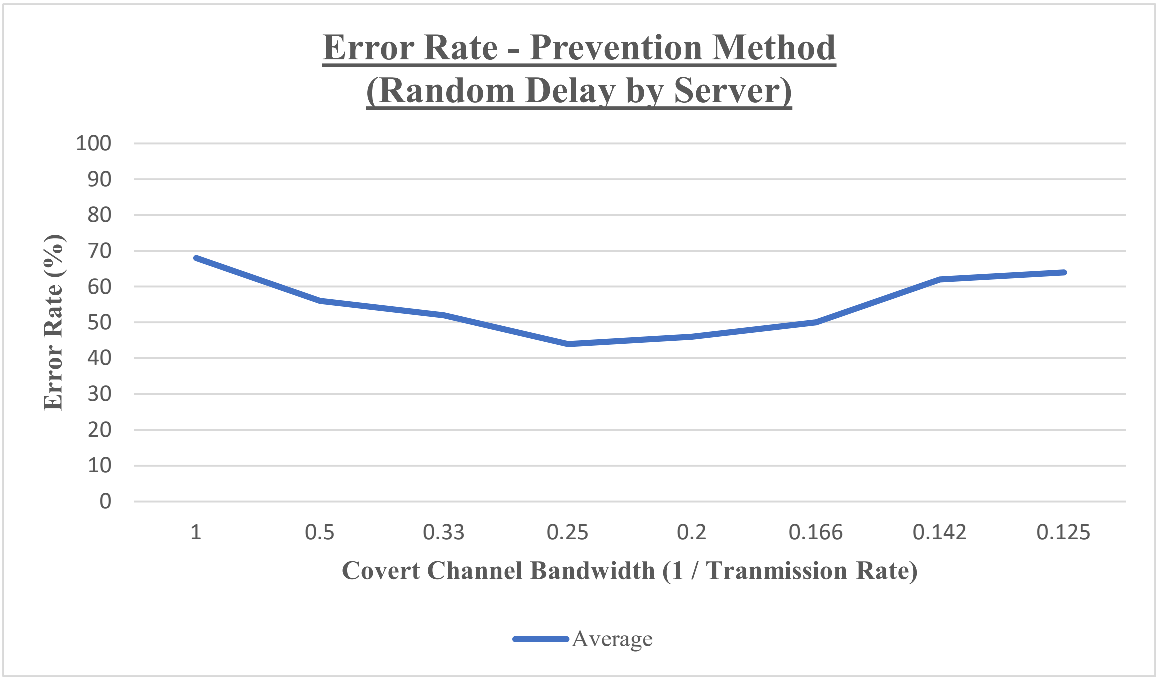
Expanding upon a secure prototype implementation, proper testing needs to occur. This will be compared to the previous results seen with baseline measurements as well as the delay method. The same number and variables are going to be used in extension of secure testing, and the results are as followed.

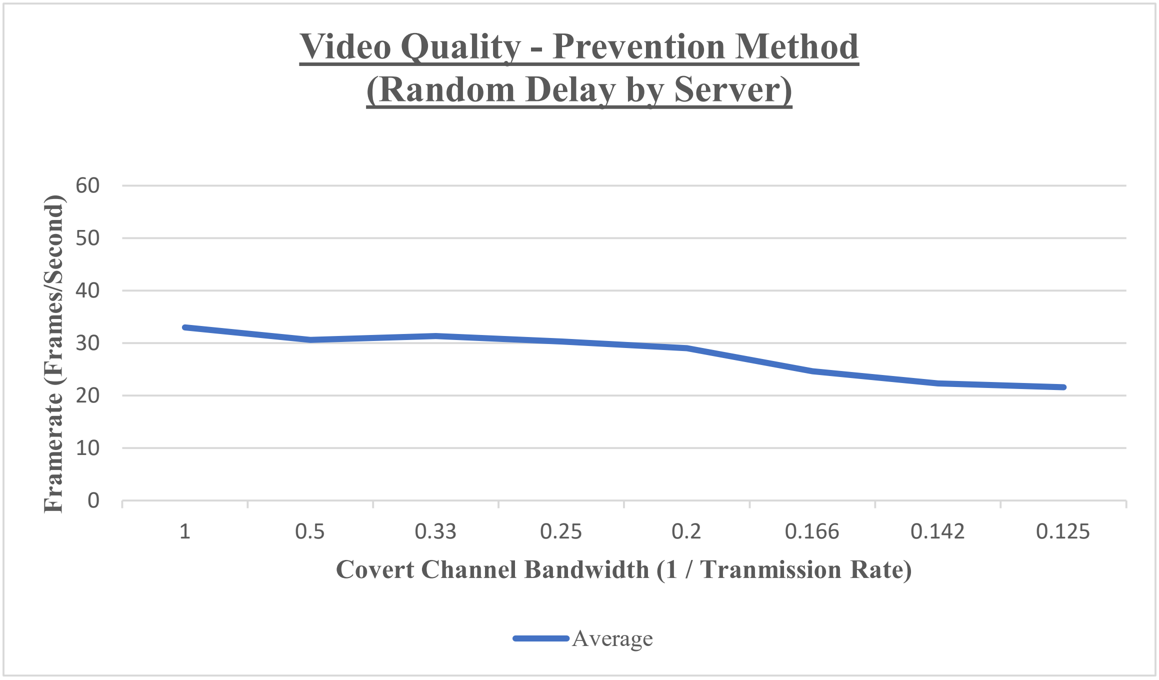




First, the error rate and video quality will be examined using a constant delay. Although the error rate is not 100%, it is still an increase in comparison to the error rate from the previous results of having no prevention method. The error rate seems to have stabilized due to the fact that when implementing a delay on the server side, in this case a constant one, it is believed that this greatly increases the rate at which a bit can be sensed. But although this is not exactly a low number, and isn't a very high number, it appears to be somewhat stable throughout each test group. In terms of framerate, 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





These results are quite similar to the delay implementation as a constant, but in terms of error rate this seems to fluctuate a tad bit more than as a constant. 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 FIVE

SUMMARY AND CONCLUSIONS

Through the development of this thesis, security protocols surrounding a WebRTC application have been examined in terms of confidentiality violations and IP leaks. Covert channel implementation allowed for a bit to be sent and received by a peer-to-peer connection without any knowledge to the administrative side. This seemed to provide some decrease in performance, but overall, nothing too over the top. This was then mitigated through the result of constant and random delays simulating a server-side implementation. This 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. Also, with data transfer, it is very common to fluctuate the ability at which a network is able to transfer the media being sent and received. It also is vital to acknowledge that since we are only sensing two different data rates, that ideally using a middle ground to reset the bit received would greatly decrease the error rate for the delay method, while increasing the error rate involved with the mitigation method. This is due to the fact that the value being sensed would be "reset" at a much higher rate, which would prove to be an advantage for both sides of the prototype implementation. It is also noticed that using the sample presentation testing method allowed for the data to remain more consistent, due to not using such a high consistent level of data compression.

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 network concerns. 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. This may raise quite a few concerns and issues, but simply tweaking variables and measures would be a possible outlook. 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. 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 how real-time communication applications are implemented and may be groundbreaking to the technology industry.

References

[1] NTT Communications (2015). *A Study of WebRTC Security*. Retrieved August 20, 2020 (https://webrtc-security.github.io)

[2] MacDonald, Steven (2021). *25 Reasons Live Chat Can Help You Grow Your Business in 2021*. Retrieved February 3, 2021 (https://www.superoffice.com/blog/live-chat-statistics)

[3] Mozilla (2021). *Pixel Manipulation with Canvas.* Retrieved February 12, 2021 (https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/Pixel_manipulation_with_canvas)

[4] Anon, Dennis (2018). *Everything you wanted to know about Tor but were afraid to ask.* Retrieved October 10, 2020 (https://privacy.net/what-is-tor/)

[5] Mozilla (2021). *WebRTC API.* Retrieved November 1, 2020 (https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API)

[6] Tully, Shane (2018). *GitHub-Shanet-WebRTC-Example.* Retrieved October 1, 2020 (https://github.com/shanet/WebRTC-Example)

[7] Donaldson, Scott (2018). *Using WebRTC for Real-Time Image Filtering.* Retrieved November 10, 2020 (https://sudo.isl.co/webrtc-real-time-image-filtering/)

[8] ExpressVPN (2021). *How to use the WebRTC leak checker.* Retrieved December 4, 2020. (https://www.expressvpn.com/webrtc-leak-test)

[9] 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)

[10] 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)

Appendices:

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 which 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

```
<!DOCTYPE html>
<html>
  <head>
    <title>WebRTC Prototype</title>
    <style>
      body {
        font-family: Arial;
        color: black;
      }

      .split {
        height: 100%;
        width: 50%;
        position: fixed;
        z-index: 1;
        top: 0;
        overflow-x: hidden;
        padding-top: 20px;
      }

      .left {
        left: 0;
        background-color: rgb(102, 51, 153);
      }

      .right {
        right: 0;
        background-color: rgb(255, 204, 0);
      }

      .centered {
        position: absolute;
        top: 50%;
        left: 50%;
        transform: translate(-50%, -50%);
        text-align: center;
      }
```

```
.centered video {
  display: block;
  margin: 0 auto;
}
button {
  background-color:  rgb(255, 204, 0);
  border: none;
  border-radius: 2px;
  color: black;
  font-family: 'Roboto', sans-serif;
  font-size: 0.8em;
  margin: 0 0 1em 0;
  padding: 0.5em 0.7em 0.6em 0.7em;
}

button:active {
  background-color:  rgb(204, 168, 21);
}

button:hover {
  background-color: rgb(204, 168, 21);
}

button[disabled] {
  color: rgb(103, 101, 101);
}

button[disabled]:hover {
  background-color: rgb(255, 204, 0);
}
video {
  background: #222;
  margin: 0 0 20px 0;
  --width: 100%;
  width: var(--width);
  height: calc(var(--width) * 0.75);
}
canvas {
  background: #222;
  margin: 0 0 20px 0;
  --width: 100%;
  width: var(--width);
  height: calc(var(--width) * 0.75);
}
```

```
  div#getUserMedia {
    padding: 0 0 8px 0;
  }

  div.output {
    background-color: #eee;
    display: inline-block;
    font-family: 'Inconsolata', 'Courier New', monospace;
    font-size: 0.9em;
    padding: 10px 10px 10px 25px;
    position: relative;
    top: 10px;
    white-space: pre;
    width: 270px;
  }
  section p:last-of-type {
    margin: 0;
  }

  section {
    border-bottom: 1px solid #eee;
    margin: 0 0 30px 0;
    padding: 0 0 20px 0;
  }

  section:last-of-type {
    border-bottom: none;
    padding: 0 0 1em 0;
  }
  section#statistics div {
  display: inline-block;
  font-family: 'Inconsolata', 'Courier New', monospace;
  vertical-align: top;
  width: 308px;
}

section#statistics div#senderStats {
  margin: 0 20px 0 0;
}

section#constraints > div {
  margin: 0 0 20px 0;
}
```

```
section#video > div {
  display: inline-block;
  margin: 0 20px 0 0;
  vertical-align: top;
  width: calc(50% - 22px);
}

section#video > div div {
  font-size: 0.9em;
  margin: 0 0 0.5em 0;
  width: 320px;
}

h2 {
  margin: 0 0 1em 0;
}

section#constraints label {
  display: inline-block;
  width: 156px;
}

section {
  margin: 0 0 20px 0;
  padding: 0 0 15px 0;
}

section#video {
  width: calc(100% + 20px);
}
@media screen and (max-width: 720px) {
  button {
    font-weight: 500;
    height: 56px;
    line-height: 1.3em;
    width: 90px;
  }

  div#getUserMedia {
    padding: 0 0 40px 0;
  }
```

```
      section#statistics div {
        width: calc(50% - 14px);
      }

      video {
        height: 96px;
      }
    }
      div#container {
        margin: 0 auto 0 auto;
        max-width: 60em;
        padding: 1em 1.5em 1.3em 1.5em;
      }

      canvas {
        height: auto;
        max-width: 400px;
      }
      textarea {
        resize: none;
      }
    </style>

  </head>
    <script src="https://code.jquery.com/jquery-
3.1.1.min.js"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.0/jq
uery.min.js"></script>
  <body>
    <!-- Left side Video (Local) -->
    <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>

    <!-- Right side Video (Remote) -->
    <div class="split right">
      <div class="centered remote">

        <video id="remoteVideo" playsinline autoplay muted width="640"
 height="360""></video>
        <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>
    <script src="https://webrtc.github.io/adapter/adapter-
latest.js"></script>
    <script type ="text/javascript" src="webrtc.js"></script>
  </body>
</html>
```

Server-Side Implementation

Server.js

```
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\
Some important notes:\n\
  * Note the HTTPS; there is no HTTP -> HTTPS redirect.\n\
  * You\'ll also need to accept the invalid TLS certificate.\n\
  * Some browsers or OSs may not allow the webcam to be used by multip
le pages at once. You may need to use two different browsers or machin
es.\n'
);
```

Client-Side Implementation

Webrtc.js

```
'use strict';
// Global/Function Scoped
var localVideo;
var remoteVideo;
var uuid;
var serverConnection;

let sendChannel;
let receiveChannel;

// Implementing Delay Variables
var delayTime;
var delay;
var data;
var received = false;
const dataChannelSend = document.querySelector('textarea#dataChannelSe
nd');
const dataChannelReceive = document.querySelector('textarea#dataChanne
lReceive');

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
// The number of bits the covert channel can send per second

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;
   }
}

/****************************************************************/

// Image Filtering
var inputCanvas = document.getElementById('inputCanvas').getContext( '
2d' );
const outputStream = document.getElementById('inputCanvas').captureStr
eam();
var width = 640;
var height = 360;

// Block Scoped
let localPeerConnection;
let remotePeerConnection;
let localStream;
let bytesPrev;
let jitterPrev;
let timestampPrev;

// Format Get User Media
const getUserMediaConstraintsDiv = document.querySelector('div#getUser
MediaConstraints');

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.le
ngth}</strong> bits`;
                                              // Sending X Number of Bits

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

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

// Buttons to Get Media, Connect, and Hangup, and Send Data
getMediaButton.onclick = getMedia;
connectButton.onclick = createPeerConnection;
hangupButton.onclick = hangup;

// Establish Peer Connection
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}\nPermissionDen
iedError may mean invalid 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 brou
ght into the Remote Stream
   localStream = outputStream;
   localStream.getTracks().forEach(track => localPeerConnection.addTra
ck(track, localStream));

   console.log('localPeerConnection creating offer');
   localPeerConnection.onnegotiationeeded = () => console.log('Negotia
tion needed - localPeerConnection');
   remotePeerConnection.onnegotiationeeded = () => console.log('Negoti
ation needed - remotePeerConnection');
   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 (Possible Pixel Manipulation)
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} mil
liseconds`;
```

```
      }
      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)
```