




Article

Neither Denied nor Exposed: Fixing WebRTC Privacy Leaks

Alexandros Fakis ^{1,†}, Georgios Karopoulos ^{2,*,†} and Georgios Kambourakis ^{2,†}

¹ Department of Information and Communication Systems Engineering, University of the Aegean, Karlovassi, 83200 Samos, Greece; alfa@aegean.gr

² European Commission, Joint Research Centre (JRC), 21027 Ispra, Italy; georgios.kampourakis@ec.europa.eu

* Correspondence: georgios.karopoulos@ec.europa.eu; Tel.: +39-0332-78-6165

† These authors contributed equally to this work.

Received: 2 April 2020; Accepted: 19 May 2020; Published: 22 May 2020



Abstract: To establish peer-to-peer connections and achieve real-time web-based communication, the Web Real-Time Communication (WebRTC) framework requires address information of the communicating peers. This means that users behind, say, Network Address Translation (NAT) or firewalls normally rely on the Interactive Connectivity Establishment (ICE) framework for the sake of negotiating information about the connection and media transferring. This typically involves Session Traversal Utilities for NAT (STUN)/Traversal using Relays around NAT (TURN) servers, which assist the peers with discovering each other's private and public *IP:port*, and relay traffic if direct connection fails. Nevertheless, these *IP:port* pieces of data can be easily captured by anyone who controls the corresponding STUN/TURN server, and even more become readily available to the JavaScript application running on the webpage. While this is acceptable for a user that deliberately initiates a WebRTC connection, it becomes a worrisome privacy issue for those being unaware that such a connection is attempted. Furthermore, the application acquires more information about the local network architecture compared to what is exposed in usual HTTP interactions, where only the public IP is visible. Even though this problem is well-known in the related literature, no practical solution has been proposed so far. To this end, and for the sake of detecting and preventing in real time the execution of STUN/TURN clandestine, privacy-invading requests, we introduce two different kinds of solutions: (a) a browser extension, and (b) an HTTP gateway, implemented in C++ as well as in Golang. Both solutions detect any WebRTC API call before it happens and inform accordingly the end-user about the webpage's intentions. We meticulously evaluate the proposed schemes in terms of performance and demonstrate that, even in the worst case, the latency introduced is tolerable.

Keywords: WebRTC; STUN; TURN; ICE; security; privacy; WWW

1. Introduction

The need for enabling Peer-to-Peer (P2P) communication without the requirement of providing extra extensions or native apps to the peers has been of utmost importance for years. In 2011, WebRTC [1] was developed, offering high-quality real-time communication between web browsers, mobile applications, and IoT devices. WebRTC uses JavaScript Application Programming Interfaces (APIs), defined by the World Wide Web Consortium (W3C) for enabling multi-platform voice, text, and video communications.

Typically, WebRTC is used for realizing real-time audio and video calls, web conferencing between a number of peers, and even direct file transfers, without the need for extra extensions or additional applications. The biggest benefit of this technology is that the whole communication is achieved

via a typical web browser, like Chrome or Mozilla. Although WebRTC is supported by companies such as Google, Microsoft, Mozilla, and Opera, and, despite WebRTC's popularity [2,3], a significant privacy issue remains unresolved: the IP addresses, both private and public, of each client visiting a website which is capable of performing a WebRTC peer connection request, can be potentially revealed to the website host. This enables malicious sites to attack clients which would otherwise be inaccessible, e.g., behind a firewall or NAT. It is worth noting here that such an attack is not possible due to vulnerabilities in the WebRTC protocol per se, but only by abusing its legitimate features.

In more detail, any user browsing the web is able to visit a webpage or use services like sending an email. This is feasible, as web servers have either a public or a port-forwarded IP address, exchanging data with any device. Specifically, in case the client is behind a NAT gateway, NAT maps the device's private IP address to a public IP address and forwards each request to the server and each response back to the client. However, when it comes to P2P services like video telephony where both devices, i.e., the caller and callee, are behind different NAT gateways, an obstacle arises when trying to establish the connection. This is because, when the caller tries to initiate a session, the callee's NAT will not be aware of the callee behind it, and thus will drop the connection [1]. In order to bypass such impediments and explore different connection methods, WebRTC deploys some extra protocols.

WebRTC allows media to get transferred between peers, regardless their network topology, even if the peers are behind NAT. Nevertheless, in order to successfully create a connection, each peer needs to dynamically generate and discover the most effective path for sending media to the rest of the peers. WebRTC achieves that by taking advantage of the ICE [4] framework. ICE uses two different methods, namely Session Traversal Utilities for NAT (STUN) [5] and Traversal using Relays around NAT (TURN) [6], to correspondingly assist with the user's browser identify their public IP address and port, and relay traffic if direct connection fails. A STUN server is used by a peer if the latter needs to identify its IP:port address information as seen from a public perspective, i.e., as it is generated by the NAT box closest to the server, while the TURN server acts as an intermediate when the peer's network forbids P2P connections. The response received by STUN is made available to the JavaScript code that initiated the request; since this code runs locally on the user's system, it has access to the private IP address of the user as well. This procedure is transparent to the end-user, so they are completely unaware of these STUN/TURN requests when they visit a webpage.

An attacker is able to setup their own STUN/TURN server, forcing the user's browser to perform queries to this server, by placing a short JavaScript code to a webpage before the user visits it. As a result, anyone who has access to this rogue or compromised server is able to obtain the private and the public IP of the user, which is a piece of personal information that may lead to the identification, say, at minimum by means of geolocation, of the end-user.

WebRTC technology exists in all modern web browsers, so any security and privacy issues would affect virtually any end-user browsing the web and this is the reason why proposing an effective solution is important. In order to avoid risking their endpoint security, most security-savvy users prefer to disable any WebRTC services. This of course is a temporary and certainly incommodious solution, as real-time communication is an integral part of most messaging platforms, including Messenger and Google Hangouts. Some users even employ a Virtual Private Network (VPN) solution or connect via The Onion Routing (Tor) [7] or the Invisible Internet Project (I2P) [8] anonymity networks, but this may come at the expense of experiencing low quality real-time communication. Furthermore, as detailed in Section 4, the aforementioned solutions are not always effective.

A more reasonable solution would inform the user whether the visited website is trying to use the browser's WebRTC capabilities [9], rather than completely disabling it for all websites. Any website that is not related to web communication between users, for instance a news webpage, should probably not try to make any WebRTC requests and thus the user could safely select to block it.

To solve the aforementioned vulnerability following a user-awareness approach, we implemented and propose two novel solutions: (a) a browser extension and (b) an intermediate trusted gateway, both able to examine a webpage before it is loaded on the browser and inform the user about any WebRTC actions that the webpage is about to execute. We scrupulously evaluate the proposed schemes in terms of performance and show that the additional delay is most of the times negligible, even for websites with an above average size.

The rest of the paper is structured as follows. The next section briefly covers the way STUN and TURN mechanisms work, while Section 3 provides background details on the WebRTC framework. Section 4 provides details on the privacy issue that affects WebRTC by revealing peers' private and public IP addresses. A discussion about how the specific vulnerability can be addressed using two different kinds of solutions is provided in Section 5. Section 6 focuses on implementation issues, while Section 7 presents our results. Section 8 addresses the related work. Finally, the last section concludes and provides some pointers for future work.

2. Media Connections

There are two basic call topology types in WebRTC. In the simplest scenario, the peers are employing the same WebRTC service offered by the same webpage, creating a triangle model as Figure 1 depicts. On the other hand, if the peers' browsers are using services hosted by separate web servers, then the model becomes a trapezoid, inspired by the so-called Session Initiation Protocol (SIP) [10] trapezoid model, as illustrated in Figure 2. In this latter case, the two different web servers need a way to communicate, and thus signaling messages are used to set up, manage, and terminate the communication. The protocol used for this communication is not part of WebRTC; however, a standard protocol, such as SIP, JavaScript Session Establishment Protocol (JSEP) [11], or a proprietary one can be used [12]. In both the triangle and trapezoid models, the caller's and the callee's browser is communicating with the corresponding server, exchanging signaling messages using either HTTP or WebSocket protocol, in a way also external to WebRTC.

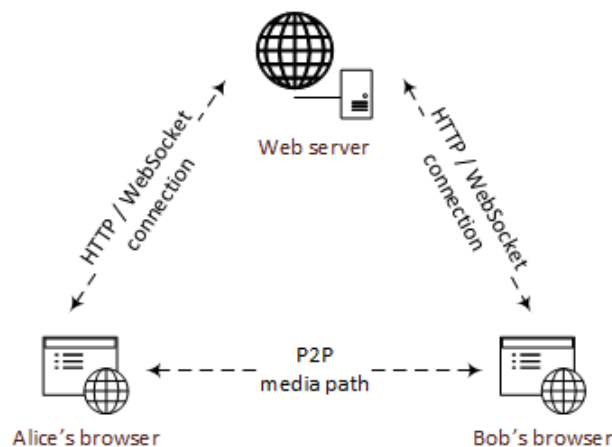


Figure 1. WebRTC triangle network architecture.

Most of the times, P2P communication over the Internet is complicated despite the simplicity of the network architecture. Precisely, the peers might either be behind a firewall restricting network access, sitting behind a symmetric NAT box, or their router may forbid a direct connection to peers, so the use of a relay is necessary. In the context of WebRTC, the ICE framework offers a solution to the aforementioned problems. Specifically, as depicted in Figure 3, ICE enables the caller's and callee's device to be informed about their public IP:port address, by making binding requests to a STUN server. In case of using the same WebRTC service, the peers will most probably use a common STUN server; otherwise, different STUN servers will serve them. After the peers become aware of their IP:port address as seen from the STUN's perspective, they can easily establish a direct communication over

UDP or TCP, if UDP fails. If at least one of them is unable to do so in the presence of one or more of the aforementioned network obstacles, say, a non-STUN compatible NAT, then a TURN server can be used as a fallback mechanism. As illustrated in Figure 4, this server acts as an intermediate node to relay data packets between the peers. Note that, typically, a TURN server offers STUN functionality too. Moreover, due to the provision of real-time voice and video services, WebRTC preferably runs over UDP. Thus, in addition, UDP hole punching may be needed for establishing a direct connection between the peers successfully.

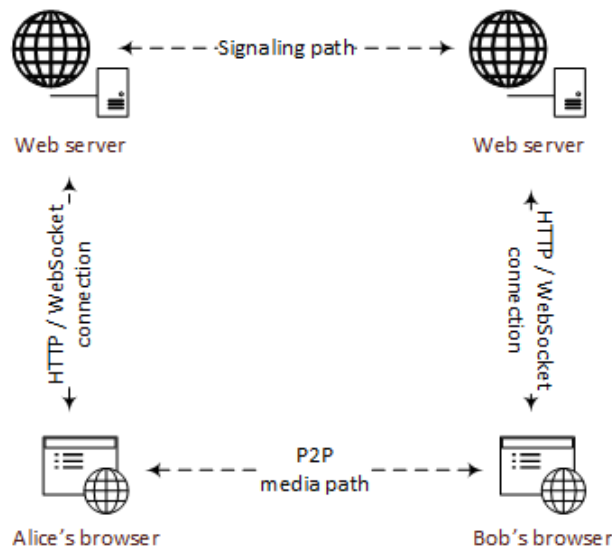


Figure 2. WebRTC trapezoid network architecture.

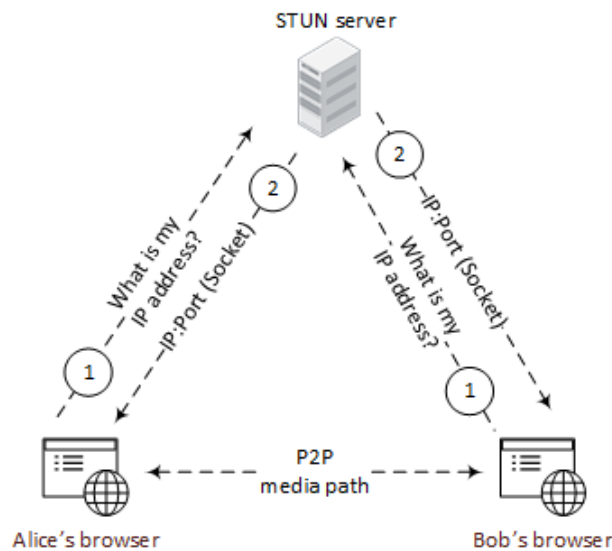


Figure 3. Using a STUN server.

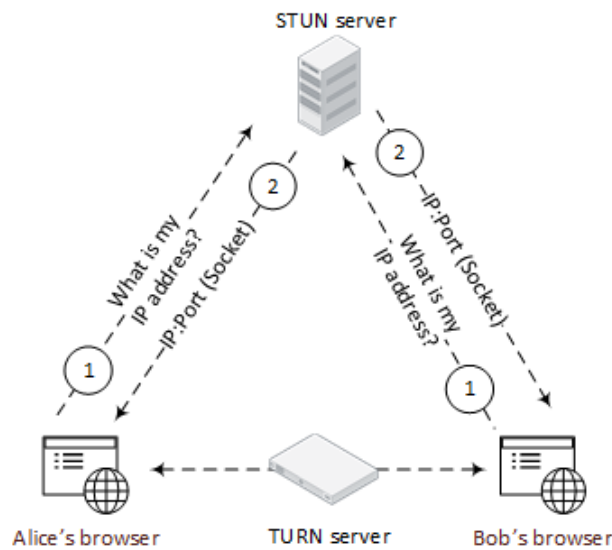


Figure 4. Using a TURN server.

Browsers supporting WebRTC are able to make STUN requests and get informed about the actual private and public IP address of the end-user before passing this information to the JavaScript API. To initialize a connection to a remote peer, the local browser will use the `RTCPeerConnection` interface. Plainly, an `RTCPeerConnection` represents a WebRTC connection between the local machine and a remote peer, and thus provides all the necessary methods to the user's browser to connect to a remote peer and then maintain, manage, and monitor the established connection.

To demonstrate a common scenario, we use an example where a client is behind a symmetric NAT. In such a case, the caller needs to ask the TURN server to allocate some of its resources for them, as the TURN server will be used as a relay to contact the other peer [6]. In the log entry in Listing 1, one can easily observe the information that the TURN server logs after the client initiated an "Allocate transaction", using an Allocate type of request. First, a time handler responsible for the process termination starts. The default lifetime of an allocation is 10 min, but the actual time is defined in the initial Allocate request [6]. Later on, the server logs the public IP and the port of the client, in order to complete the allocation for this particular client, as presented on the "remote" information, while "local" information is the public IP and the port of the TURN server. Following a successful allocation process, the initiator can either keep the connection alive using refresh requests, or terminate it.

Listing 1: TURN server log.

```

1 client_to_be_allocated_timeout_handler:start
2 shutdown_client_connection:start
3 session 00100001: close (2nd stage), user realm <> origin <>,
4 local 198.201.166.150:3478 remote 31.132.100.72:1348, reason: allocation
5 watchdog determined stale session state
6 shutdown_client_connection:end
7 cliend_to_be_allocated_timeout_handler:end

```

3. WebRTC Background

Before explaining how a malicious user can exploit any browser offering WebRTC capabilities, we should describe in detail how WebRTC works. Let's consider an example with two users, Alice and Bob, who both use a WebRTC client, and Alice wishes to call Bob (Figure 3). To create a P2P connection, Alice needs to generate a Session Description Protocol (SDP) [13] offer. Note that SDP is a format

protocol intended to describe media details, transport addresses, and any other session description metadata to the peers. In our example, the SDP message is sent between the peers, using HTTP or WebSockets. That is, Alice and Bob have to exchange SDP data using the existing signaling channel in order to negotiate audio and video media parameters, such as media codecs and video resolution. Additional information regarding setting up, updating, and tearing down the WebRTC session, such as transport addresses and related metadata, will also be exchanged using SDP. This process is initiated along with the creation of an `RTCPeerConnection` object as explained in Section 2.

The SDP offer includes information about Alice's network connection, and as Alice's data may follow different communication paths before they reach the outside, this offer must contain the shortest and most efficient network path. This is where ICE is applied to help gather the different nodes of Alice's network, known as ICE candidates. The reason why the SDP offer is vital in this step is that it includes all the ICE candidates that Bob can use for communicating with Alice, along with others, important for the call information.

For building the list of ICE candidates, Alice makes a STUN request to a STUN server which is expected to respond with her public IP address and port(s). At this point, the STUN server allows Alice's client to discover its public IP address and, say, the type of NAT it is behind from, by sending back to her a success response which contains all the appropriate information.

All in all, every node that forwards Alice's traffic from the local network to the Internet is considered an ICE candidate, and the whole process of IP and port identification is called ICE candidate gathering. There are three types of candidates:

1. **Host Candidate:** Contains the private IP address and local UDP and TCP ports which are associated with the user's local interface. They are generated by the client itself.
2. **Server Reflexive Candidate:** Contains the public IP address and UDP and TCP port of the user that is returned by the STUN server. In contrast to Host Candidate, the client sends query messages to the STUN server, which will pass through the NAT, creating a NAT binding that is a public-private IP address mapping. The response contains the public IP and port (IP:port) generated for the binding.
3. **Relayed Candidate:** Similar to Server Reflexive Candidate, this type of candidate contains the translated public address of the user; however, the NAT binding is obtained by a TURN server, instead of a STUN.

An example of an SDP message after an "`RTCPeerConnection`" initiation procedure is presented in Listing 2. As observed from the listing, three different candidates were gathered. The first one consists of the user's interface (private) IP address, along with the port the browser is listening to. In this case, if the remote peer, i.e., the callee, is part of the same intranet this candidate will be selected, and the Real-time Transport Protocol (RTP), which is the protocol used for delivering audio and video, will run over UDP. The second candidate is identical to the first; however, TCP will be used instead of UDP. The last candidate is a reflexive one, as a STUN server was used to return the user's public IP:port. Therefore, it is a lower priority candidate, but it will be nonetheless probed if the peers reside in different networks. It is to be noted that the returned number of candidates could be augmented depending on whether the peer's machine is multi-homed or is connected via a VPN or an anonymity network.

Listing 2: List of ICE Candidates (available to JavaScript).

```

1 a=candidate:0 1 UDP 2122252543 192.168.1.229 59914 typ host
2 a=candidate:2 1 TCP 2105524479 192.168.1.229 9 typ host tcptype active
3 a=candidate:1 1 UDP 1686052863 31.132.100.72 1348 typ srflx raddr 192.168.1.229 rport 59914

```

After receiving Alice's SDP message, Bob needs to follow the same procedure to produce an SDP message as a response and send it back to Alice. After the two users obtain the appropriate information via the exchange of SDP messages, they both perform a number of connectivity checks. These checks

comprise a series of STUN requests to each IP and port pair of the ICE candidates. If the other party's browser responds, the originated request is considered successful and the checked pair is marked as valid. After the checks are completed, the ICE algorithm will decide which of the valid candidate pairs are the most efficient, based on a list of rules, including the IP address family, the utilization of media intermediary, say, a TURN server, and peer's connection security, i.e., the use of VPN or not. If no valid pair is found, then the peers will make TURN requests to the TURN server in order to use it as an intermediate. After the connection has been established, the TURN server will forward any packets transferred between the two parties, as depicted in Figure 4. More specifically, in case of using a TURN server, TURN will remain in the media path, even after the connection has been established and will act as a relay between the two ends.

4. IP Disclosure

4.1. Adversary Model

The present study mainly targets the privacy threats while users are operating a typical web browser. Consequently, adversaries are individuals or organizations which attempt to compromise the privacy of any Internet user. We consider an adversary with the following capabilities: (1) they can intercept, modify, or inject any message in the public communication channels; (2) they adhere to all cryptographic assumptions, e.g., an adversary is not able to decrypt an encrypted message without knowledge of the decryption key; (3) they are able to setup and operate their own STUN/TURN server; (4) they are able to inject JavaScript code to any webpage; (5) they can lure individuals into visiting certain webpages by, say, exercising social engineering techniques. An extensive threat model for WebRTC can also be found in [14].

4.2. Problem Statement

Bear in mind that there are two possible ways for the browser to execute a piece of JavaScript code on a webpage: (a) either immediately in the order it appears or (b) wait for a triggered event to be executed. In any case, injected JavaScript code will always be executed transparently in the background without the user's permission. Taking this into account, a webpage that supports WebRTC will execute some WebRTC API calls in the background and will try to initiate a call to the specified remote party. A code snippet responsible for such an action is given in Listing 3.

This small piece of JavaScript code actually configures the STUN server that will be used for the initiation of the call and then constructs an *RTCPeerConnection* object, which executes the STUN or TURN requests. Specifically, the code first initializes a JavaScript object with a list of possible ICE servers that could be used during the procedure, adding for each candidate the IP/domain of the server and the user's password, if needed (lines 2 to 4). As observed, in line 3, the URI scheme "stun" is used, followed by the server domain, while the credential field in line 4 carries the authentication password. In most cases, the selected STUN server will be one of the available servers that are offered by organizations like Google, e.g., `stun.l.google.com:19302`. In lines 7 to 8, the *RTCPeerConnection* object gets instantiated and saved in the "servers" variable. The "onicecandidate" in line 11 is an *EventHandler*, where a function is specified and it is called when an "icecandidate" event occurs, namely, when an ICE candidate is discovered. This is the function via which the caller/callee will retrieve any information about themselves, namely any local or external IPs they are using. Finally, the "createDataChannel" function in line 20 creates a new communication channel with the callee, over which any data can be transmitted. On the other hand, the "createOffer" function in line 22 initializes an SDP offer, with the purpose of creating a WebRTC connection with the callee. After the JavaScript code is executed, the peer will communicate with the declared STUN server and exchange data with it.

Listing 3: Initializing a WebRTC call.

```
1 //initialize list of ICE servers
2 var servers = {iceServers:
3   [{url:"stun:stunserver.org",
4     "credential":"my_password"}]};
5
6 //construct a new RTCPeerConnection
7 var rtc = new RTCPeerConnection(
8   servers);
9
10 //Event Handler for new ICE candidate
11 rtc.onicecandidate = function (ice) {
12   if (ice.candidate) {
13     //Returns a DOMString describing
14     //the candidate in detail
15     console.log(ice.candidate.candidate);
16   }
17 };
18
19 //create a bogus data channel
20 rtc.createDataChannel("");
21 //create an offer sdp
22 rtc.createOffer(function (result) {
23   //trigger the stun server request
24   rtc.setLocalDescription(result,
25     function () { }, function () { });
26 }, function () { });
```

However, setting up and running a STUN server is a task that even a script-kiddie can easily perform aiming to expose the IP addresses of any user who executes the JavaScript code in Listing 3. Thus, if one installs a STUN server on the “malicious_stun_server.org” domain, they could easily replace lines 2 to 4 of Listing 3 with lines in Listing 4.

Listing 4: Choosing the preferred STUN or TURN server.

```
1 var servers = {iceServers: [{url:"stun:eve@malicious_stun_server.org",
2   "credential":"my_password"}]};
```

By doing so, the attacker is able to force the caller’s browser to use a STUN or TURN server controlled by them. Now, the aggressor could inject the script in the source code of a webpage that they own or have compromised, and as a result any browser visiting this page will find executing a STUN request to the STUN server controlled by the attacker. For instance, think of a watering hole variant attack scenario or the use of specially selected advertisements displayed to the victim. Accordingly, any webpage containing the JavaScript code of Listing 3 and the corresponding STUN server can instantly acquire the public IP:port of the peer. Bear in mind that the JavaScript code becomes knowledgeable of all ICE candidates of the peer, and as each one of them is associated with a peer’s private interface, the mapping of the user’s connection topology is feasible. The script injection mentioned above can be performed using a Cross-Site Scripting (XSS) attack. This is feasible on a web application where user input is directly included in the webpage without previous validation or encoding. If the XSS attack is successful and the script executes any STUN requests, then those requests will not be blocked by extensions such as Adblock [15], given that STUN requests are made outside of the normal XMLHttpRequest procedure.

While the main role of a STUN server is informing the peer of their public IP address and port as seen from its viewpoint, there are no completely safe ways for a user to hide their public or private IP address, even if a VPN or an anonymity network like Tor is employed. Bear in mind that, when a user connects to a VPN, a virtual network interface is created to handle all the network traffic, redirecting any data inside the VPN tunnel. Linux systems use the network tunnel (TUN) or network tap (TAP) [16] to provide packet reception and transmission when VPN is enabled. When a new virtual network interface is created, two or more adapters will be present and the operating system has to decide which one to use for sending traffic. The selection is based on the properties of each adapter, which, for example, in the MS Windows operating system are determined by a feature called *Automatic Metric*. After this process, the VPN adapter will be selected as the default in order to achieve secure and confidential communication. However, the previous main network interface that is the Ethernet or WiFi adapter is still active. In every operating system, the software is able to choose any active network interface for communication, but usually the default one selected by the operating system is used. Therefore, in the context of this adapter selection process, a STUN request may also reveal the end-user's public IP as allocated by the ISP to the JavaScript running on the webpage. Current trials over well-known VPN services, including *TunnelBear* [17] and *Hotspot Shield* [18], verify this situation. Although the MS Windows OS is the main victim of such a flaw, Mac and Linux systems may be vulnerable as well, depending on the user's VPN client and how it is configured [19]. In a similar research [20], the authors claimed that both Firefox and Chrome were vulnerable regardless of the underlying operating system, with iOS being the only exception. They also mentioned that the only browsers that support WebRTC, but were immune to this vulnerability, were Safari on Mac and Microsoft Edge on Windows.

A straightforward solution for protecting the users' IP addresses would be the use of a VPN firewall, but still without ensuring full anonymity. Generally, if a VPN connection fails, any data will be sent unsecured. A VPN firewall will ensure that any outgoing traffic will be sent through the established VPN tunnel, forbidding any connection in case the tunnel breaks down for any reason. In addition, a VPN firewall offers more benefits, including tight firewall rules and defeating the shared VPN/Tor server leak bug [21]. However, even this approach does not ensure full anonymity. This is because some browsers can store data from past user activities, like previously opened tabs. In that case, if a tab is opened before the user connects to the VPN, the public IP of the user will be cached in memory. In addition, even if the public IP address is not leaked, the end-user's private IP is still left unprotected, since JavaScript is able to identify it. As a result, the server will retrieve and log all the available information that could be used to identify the user.

Based on the WebRTC API [22], apart from public and private IP addresses, media information can also be retrieved by the "MediaDeviceInfo" interface. Precisely, WebRTC can leak the existence of any microphone and camera that the user may utilize. After obtaining an array of the available "MediaDeviceInfo" objects, one per media device, the browser can collect information for input and output devices through the object's properties. For instance, the "kind" property will return an enumerated value that is either "videoinput", "audioinput" or "audiooutput", while the "label" property will return a DOMString describing the device. By default, the "label" is an empty string; however, if one or more media streams are active, or if the user granted persistent permissions to the browser, then the DOMString will contain information about any employed peripheral device, including its name and type, e.g., "External USB Webcam". For example, the enumeration of "MediaDeviceInfo" array may produce an output similar to that in Listing 5. Thus, if two audio and one video input were attached to the client's machine, the kind of each input along with its ID would be outputted.

Listing 5: Example of MediaDeviceInfo output.

```

1 videoinput: id = 56430b9613fcb1ac822fd53a6c25
2 audioinput: id = 321668ae7aebb94d0d2b90bee995
3 audioinput: id = 0fd5b84ae87e3420486e2e2c4d9f

```

On the other hand, each item of Listing 5 would contain all the available information in case one or more media streams were active or permissions were granted by the user, as presented in Listing 6. That is, for each media input, apart from device's ID and its type, the "label" describing the device and a "groupID" would also be returned. The "groupID" represents the group of the device and two devices have the same group identifier if they are part of the same physical device, for example, if both a camera and a microphone belong to the same monitor.

Listing 6: MediaDeviceInfo output with granted permissions.

```

1 kind: videoinput
2 label: Integrated Webcam (1bcf:28b0)
3 deviceId: 56430b9613fcb1ac822fd53a6c25
4 groupId: 85632d755cfb1dfb30228124124ec
5
6 kind: audioinput
7 label: Microphone (Realtek Audio)
8 deviceId: 0fd5b84ae87e3420486e2e2c4d9f
9 groupId: 40756e2116ee3d4d75183136bd03e
10
11 kind: audioinput
12 label: Headset (HD 4.40BT Hands-Free AG Audio)
13 deviceId: 321668ae7aebb94d0d2b90bee995
14 groupId: b0d25385669f2a63bfe9d556fee46

```

5. Dealing with IP Leaks

As discussed in Section 2, a browser informs the actual WebRTC application about the location of STUN and TURN servers using the `RTCPeerConnection` object. The peer can later use this object to connect to the STUN server, get informed about its own IP addresses through this connection, and create SDP offers for initializing a call. It is therefore straightforward that a webpage that is not destined to execute WebRTC requests should not use by any means the `RTCPeerConnection` object. In that case, if a binding request to a STUN server takes place, then the user should be informed that the visited website needs to access private information (the public IP address) in order to use WebRTC.

Taking the latest Android update as an example, a "runtime" permission system would be the ideal approach against such kind of actions. More precisely, a user should be informed that the webpage is trying to access their IP address information to initiate a WebRTC call and the exact time the user is placing the call. A similar type of permission request is used in some browsers, like Chrome, to inform the user that the current webpage is trying to access their location or wishes to send notifications. In such circumstances, if the user approves the request, then the browser will be able to get the appropriate user's information. Contrariwise, if the user rejects the request, the action should get blocked, also meaning they will not be able to make a call, if that is the case.

To address the above issue and detect whether a webpage attempts to execute a STUN or TURN request, we propose two diverse kinds of solutions, which are implemented differently, but, in essence, use the same detection technique. The first approach is a browser extension, which basically uses a preload mechanism to prevent such JavaScript calls before the actual HTML Document Object Model (DOM) loading starts. The second one is using a gateway, either local or third-party, to inspect the JavaScript code the user is about to download.

5.1. Browser Extension

The implemented browser extension is able to prevent any WebRTC requests before they are executed by a webpage. This is achieved by dereferencing any WebRTC objects that could be used by a webpage for executing STUN or TURN requests. Initially, when a user visits a webpage, the WebRTC requests are blocked, while the rest of the page loads normally without any user interruptions. After the page loading is completed, the extension inspects for any suspicious WebRTC requests, and if any is detected, the browser asks for WebRTC permissions from the user. If the user wishes to use WebRTC capabilities, the extension will restore any WebRTC objects that were previously dereferenced, allowing the user to initiate a WebRTC call. As browser extensions are implemented differently on each browser, the drawback of this approach is that a different extension per browser is needed. As detailed in Section 7, our extension has been implemented for use with the popular Chrome browser.

5.2. Gateway

A centralized and more generic solution is to implement a gateway, which examines the webpage source code before it is delivered to the user's browser. The user has the ability to choose the gateway that will be used, and thus the gateway is considered trusted. This way, a user would again be informed by the trusted gateway if the website that they would like to visit is trying to make a WebRTC request. If the user is unaware about that website actions, they may block the relative JavaScript or blacklist the website by storing the domain on the gateway's list. Then, this list will be checked every time a user visits a website, so the latter will be prevented from loading if its domain already exists in the list. In contrast to the browser extension, any check will be conducted by the gateway before the webpage loads to the user, and thus an extra delay will be added to the load time of the webpage. Those extra delays are presented later in Section 7.

6. Implementation

The source code of both the implemented solutions is correspondingly given under the European Union Public Licence (EURL) [23] at: <https://github.com/IncredibleMe/WebRTC-IP-Leak-Chrome-Extension> and <https://github.com/IncredibleMe/WebRTC-IP-Leak-Gateways>.

6.1. Browser Extension Implementation

As already pointed out in Section 4, STUN requests are made outside of the normal XMLHttpRequest procedure, so a common extension responsible of ad blocking, for instance, is not able to detect such requests. The implemented extension should prevent the webpage for such possible STUN requests by inspecting any JavaScript code. The main hurdle is that any JavaScript residing on the source code will be immediately executed after the webpage has been downloaded, so blocking is not feasible, and, as a result, a different approach should be followed. While implementing a browser extension, lots of information about it should be declared on the "manifest.json" file, such as the name of the extension, its version, and its permissions. In the manifest file, one can also force the extension to be executed as fast as possible, using the "run_at : document_start" option. We also deployed the same "run_at" option, since we need to inject a JavaScript piece of code preventing WebRTC requests in the webpage's source code before any other DOM is constructed or any other script is executed.

This option gives us the ability to inject JavaScript code as a script at the top of the website's HTML document and execute it immediately after the DOMContentLoaded event. Bear in mind that in an HTML document all scripts are treated the same and thus they are executed in the order they appear in the document. Having our defensive script placed on top of everything else, every malicious action that may appear later in the document can be prevented.

The next step is to create a JavaScript element and backup the "RTCPeerConnection" object value, along with the compatible objects for Firefox and Chrome, "mozRTCPeerConnection" and "webkitRTCPeerConnection" respectively, to some temporary values. After that, one can set those

object values to null so any later scripts will be unable to use them. By using this technique, we are able to break any malicious code during execution. In addition, if any use of those objects is detected in the source code, the user will be informed, and, if they wish to proceed, the script can revert on-the-fly the initial value to the RTCPeerConnection object.

6.2. Gateway Implementation

The gateway can either be a SOCKS [24] or a HTTP proxy, and while a plethora of programming languages can be used to develop such a software, there is no actual limitation about the implementation details. However, the main question that needs to be answered is what the delay is that this process adds to the webpage loading process. With that in mind, a high-performance programming language should be used, making the code-examining process a really quick task. For the sake of this work, we developed two different versions of an HTTP proxy: one written in C++ and one in Golang, two of the most high-performance server programming languages. Using the aforementioned implementations, we measured the time penalty produced by different webpages of various sizes (data volume) and characteristics. To dereference the induced time penalty from any network latency, the gateway runs in the same machine where the user is located.

No less important, the gateway is able to examine the content of any webpage using either HTTP or HTTPS. Precisely, for HTTPS traffic, the gateway works as a transparent proxy, which acts as a man-in-the-middle (MITM) at the TLS tunnel, and thus is able to read the traffic even if the content is encrypted with TLS. Typically, a client would connect to a proxy by executing a CONNECT request, for instance “CONNECT example.com:443 HTTP/1.1”, and open a P2P connection between the client and the destination web server.

Using a modern programming language, the parsing of an average webpage’s source code will be completed in a few milliseconds. Table 1 presents a comparison of Golang and C++ when it comes to processing text files of diverse sizes and characteristics. As observed, the results pertaining to both languages are almost identical, and even on much larger file sizes than the average webpage size which is ≈ 3 MB [25]. In addition, searching of certain keywords can be achieved in less than a second even in files of 100 MB that is ≈ 534 ms and ≈ 513 ms on average for C++ and Golang correspondingly. We should also mention that on each search the whole text file was examined, to reach the conclusion that the document is safe. However, if a malicious JavaScript was included in a document, the result would be obtained before reaching the end of the file, and thus the delay is expected to be even less. It is to be noted that the results reported in Table 1 have been acquired after conducting 500 searches per different text file size and computing the mean value. As observed in Table 1, the average value is almost half a second for files even thirty times as big as the size of the average webpage. This leads us to conclude that most of the delay of this gateway-based approach is expected to be mainly due to the bandwidth of the gateway and the network speed rather than the actual examination of the webpage’s source code.

Table 1. Performance comparison of C++ and Golang in file processing.

Filesize (MB)	Delay (ms)	
	Golang	C++
0.5	3	3
1	6	6
5	28	28
10	56	58
50	283	269
100	513	534

7. Evaluation

7.1. Browser Extension Evaluation

To test the performance of the proposed browser extension, we created a proof-of-concept implementation for the Chrome browser running on Ubuntu Linux 18.10 on a laptop machine equipped with an Intel Core i7-6700HQ 3.4 GHz CPU and 16 GB of RAM. Chrome extensions tend to be a heavy burden to the overall performance of the browser, as they currently work for each active tab. For instance, the popular Adblock extension consumes ≈ 100 MB of memory [26]. We used Chrome’s task manager to measure the CPU and memory usage of our extension when visiting a webpage. Regarding CPU usage while the extension loads, it only happens once, and according to our measurements was negligible, i.e., less than 0.1%. Upon the examination of the webpage, the extension consumed $\approx 0.15\%$ of the available CPU, while the memory footprint reached ≈ 3 and 5 MB for websites having correspondingly a size of ≈ 3 and 10 MB. This is due to the fact that the extension initially blocks any STUN or TURN requests before they execute, rather than checking the whole webpage’s source code for such requests.

7.2. Gateway Evaluation

To measure the time penalty induced by the examination of the webpage source code, we used the Resource Timing API that is part of the Chrome Developers Tools and made measurements during webpage loading both with and without the use of the gateway. To approximate the actual time it takes for each HTTP/HTTPS GET request to be examined before it is sent back to the user, we need to identify the different components that contribute to the final webpage load time. To this end, we employed the Resource Timing API, which provides a way to get detailed network timing data about the resource loading. The interface splits the webpage’s loading time in different parts, each one representing a network event, such as DNS lookup delay, response start and end times, as well as the actual content downloading. Modern browsers use Resource Timing API [27], which is usually embedded by default, to offer developers the ability to calculate a web request load and serving time. In Chrome, a user can observe detailed information about the time each request needs to be fulfilled, using the Network Analysis Tool that is placed in the Network panel. An example of a request’s timing breakdown is presented in Figure 5.

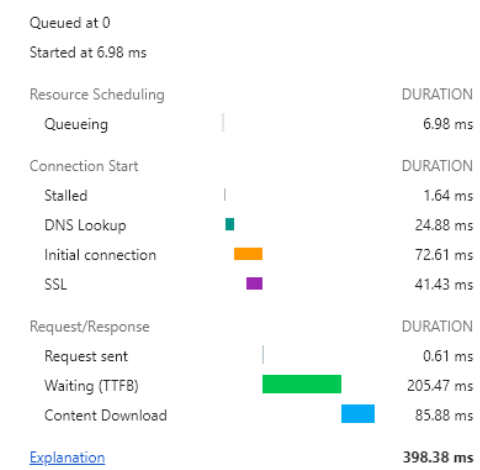


Figure 5. Preview of an indicative timing breakdown in Chrome.

In the following, we succinctly describe each of the webpage phases for the sake of understanding the whole lifecycle of a web request. Before each request is sent to its final destination it may need to be delayed for several reasons, like other higher priority requests waiting to be served, the allowed number of TCP connections has been exceeded, or the disk cache is full. For one of the aforementioned

reasons, a request may be queued or even stalled. The “DNS Lookup” delay comprises the time spent for a request to perform a DNS lookup for a certain domain in a webpage. However, if the same webpage along with the carrying domains has been visited in the past, most or even all of its domains already exist in the DNS cache of the browser, and thus the DNS Lookup delay will be close to zero. The connection process with the remote webserver is measured by the “Initial connection” metric, which represents the time the connection needs to be initialized, including the TCP handshake and the needed retries, and, if applicable, the negotiation of TLS, that is, the TLS handshake between the client and the server. Finally, the “SSL” is the time spent until the TLS handshake protocol completes, that is, the change cipher spec message, which is used to switch to symmetric key protection. The “Request Sent” is the actual time spent for the request to be sent, overcoming any network issues, which usually is negligible. The subsequent two phases are probably the most significant ones when it comes to the delay of the request. The “Waiting” represents the time spent waiting for the initial response to be processed, also known as Time to First Byte (TTFB). This pertains to the number of milliseconds it takes for a browser to accept the first byte of the response after a request has been sent to a web server, that is, the latency of a round trip to the server. Last but not least, the “Content Download” is the actual time spent before the client receives the complete web content from the server, a phase which is obviously the most critical, as most of the web request’s time is spent here.

It is expected that, as the whole webpage source is downloaded and parsed by our proxy, the size of the source code will affect the final delay. However, the browser can also employ several techniques to speed up the content downloading process, including caching. Namely, the browser cache is being used to store website documents like HTML files, Cascading Style Sheets (CSS), and JavaScript code in order to avoid downloading it again in future site visits. This means that, if a user revisits a webpage, the browser will try to download only new content, loading the rest from its cache. The previous action will naturally lead to negligible delays when it comes to security checks on a webpage, since the content has already been checked in the past.

As already mentioned, the average webpage size is ≈ 3 MB, which makes source code parsing a really quick task. It is also worth mentioning that the heavier part of the webpage is being bound by images or other multimedia type objects, while our interest lies solely on the JavaScript source code possibly existing in the webpage. With that in mind, the proxy needs to only deal with the parsing of the source code, whose actual size will be significantly reduced, hence introducing less delay.

The gateways were implemented in C++ and Golang and installed on Ubuntu Linux 18.10 on a laptop machine equipped with an Intel Core i7-6700HQ 3.4 GHz CPU and 16 GB of RAM. The Internet connection used provides a bandwidth of 34 Mbps downstream and 28 Mbps upstream. The laptop machine was connected to the Internet via an IEEE 802.11n access point. We also configured a Coturn server [28], which is an open-source TURN/STUN server for Linux. Additionally, we created a webpage, which executes requests to our STUN server, and thus we were able to examine the exact log entry at the server side.

We compared the two gateway implementations and logged key metrics that characterize a website visit. The acquired results are summarized in Table 2. We tested our gateway implementations against some of the most popular websites which mandate HTTPS, as well as websites that are plain HTTP.

The time each solution needs to download all of the (possibly compressed) contents of the webpage is of course dependent upon its size. The “content size” column represents the actual size of the resource, while “response size” represents the number of MB transferred on the wire or the wireless medium. As long as most of the modern web servers use compression to the response body before they are sent over the network, the size of the transferred data are significantly reduced. The “load time” column is the total time delay that the webpage spent in order to load all of its contents, including JavaScript and CSS code, images, as well as the main HTML document. In detail, the “load time” is divided into the “DOM content load time” and the webpage rendering. The first represents the time that the webpage needs to build the DOM. That is, when the browser receives the HTML document from the server, it stores it into the local memory and automatically parses it for creating the DOM tree.

During this time, any synchronous scripts will be executed and static assets, such as images saved on the server side, will be downloaded as well. The webpage rendering phase is the amount of time that the browser needs to show the content of the webpage to the user. However, in that stage, no extra delay will be added since the initial code will be sent directly to the client's browser.

Table 2. Performance comparison of C++ and Golang proxy implementations vis-à-vis to the typical proxyless setting. Content and response current approximate sizes are in MB, while load times are in seconds and represented as a 95% confidence interval over 50 measurements done in different days and times.

URL	Content Size	Response Size	Type of Proxy	Load Time	DOM Content Load Time
https://wikipedia.org	0.22	0.09	C++	3.91 ± 1.36	5.18 ± 1.27
			Golang	4.05 ± 1.37	4.89 ± 1.33
			Proxyless	2.41 ± 1.26	2.54 ± 1.29
https://office.com	1.46	0.77	C++	1.69 ± 0.28	2.43 ± 0.31
			Golang	1.65 ± 0.25	2.31 ± 0.38
			Proxyless	1.13 ± 0.16	1.65 ± 0.25
https://ebay.com	2.66	1.25	C++	2.51 ± 0.26	3.21 ± 0.41
			Golang	2.15 ± 0.22	2.57 ± 0.27
			Proxyless	1.78 ± 0.14	2.41 ± 0.34
https://vk.com	3.27	1.16	C++	4.48 ± 0.80	6.16 ± 0.77
			Golang	4.09 ± 0.94	5.70 ± 0.85
			Proxyless	3.09 ± 0.64	4.32 ± 0.85
https://instagram.com	3.44	1.11	C++	2.39 ± 0.19	3.93 ± 0.34
			Golang	2.14 ± 0.15	3.66 ± 0.30
			Proxyless	1.11 ± 0.16	2.51 ± 0.31
https://aliexpress.com	5.40	3.12	C++ proxy	4.49 ± 0.24	7.45 ± 0.52
			Golang	2.78 ± 0.21	7.40 ± 0.96
			Proxyless	1.92 ± 0.13	6.17 ± 0.75
https://yahoo.com	0.31	0.14	C++	1.57 ± 0.18	2.00 ± 0.23
			Golang	1.78 ± 0.16	2.62 ± 0.22
			Proxyless	0.66 ± 0.14	0.87 ± 0.29
https://netflix.com	5.37	2.24	C++	3.97 ± 0.27	4.74 ± 0.40
			Golang	3.38 ± 0.25	4.26 ± 0.34
			Proxyless	2.47 ± 0.32	2.86 ± 0.52
https://amazon.com	9.84	2.93	C++	2.80 ± 0.24	3.02 ± 0.35
			Golang	3.57 ± 0.22	4.85 ± 0.38
			Proxyless	1.41 ± 0.19	1.93 ± 0.37
https://twitch.tv	22.73	17.46	C++	5.60 ± 0.56	13.09 ± 2.08
			Golang proxy	5.1 ± 0.56	12.45 ± 1.99
			Proxyless	3.76 ± 0.58	10.74 ± 2.14
http://wired.com	20.20	11.61	C++	6.09 ± 1.53	21.18 ± 5.54
			Golang	5.76 ± 1.54	20.77 ± 5.52
			Proxyless	4.83 ± 1.49	19.57 ± 5.44
http://nba.com	23.60	14.50	C++	6.82 ± 1.06	19.10 ± 3.25
			Golang	7.52 ± 0.97	20.59 ± 3.09
			Proxyless	5.85 ± 0.95	17.29 ± 3.22
http://espn.com	1.56	0.55	C++	4.15 ± 0.74	8.64 ± 2.06
			Golang	3.77 ± 0.78	8.28 ± 2.13
			Proxyless	2.98 ± 0.79	7.31 ± 1.89

As observed from Table 2, checking a website's intentions, in terms of JavaScript code, comes with a cost, although most of the times it will not be noticeable. Both C++ and Golang have proven good choices for a proxy implementation, producing negligible delay vis-à-vis to a non-proxy scenario. Precisely, neglecting the web page size factor, in the case of the C++ proxy, the average time across all the websites we tested was 6.25 s with a standard deviation of 3.41 s, while for the Golang one the corresponding times were 5.83 and 2.69 s. These times compared to those of the proxyless configuration

(4.18 s and 1.65 s) are rather insignificant. However, the delay becomes noticeable on sites almost 300% bigger than the average one, which in our table exceed the size of 8 MB. In addition, no major difference appears between HTTP and HTTPS websites, which means that the client's gateway acting as a "TLS Termination Proxy" adds a negligible delay. All in all, the most critical parameter is the "content size" of each website. The most significant time delay was observed on amazon.com, nba.com, and espn.com whose size is among the biggest in Table 2. As expected, the larger the size of the source code to be examined, the greater the webpage's loading time.

8. Related Work

Thus far, the WebRTC API privacy issue has been explored by a handful of works in the literature. The work in [19] provides details on the privacy threat occurring when WebRTC is exploited by browsers. The authors conduct different experiments in order to examine which types of IP addresses can get compromised. To this end, they tested a variety of widely used VPN services on different browsers and presented the results of their experiments, concluding that none of the tested VPN services has proven to be a perfect privacy solution. From the results offered by the authors, one can realize that each VPN service exposes a different set of user's IPs, with TorGuard being the preferred privacy-preserving VPN service, exposing just the private IPv4 of the user assigned by the VPN server. However, based on their results, a VPN solution is neither a silver bullet to the IP leaking issue, nor efficient, as VPN networks are highly dependent on the bandwidth of their VPN server, which are often geographically dispersed.

The issue of using WebRTC in order to map the intranet topology from an external attacker is discussed in [9]. The authors present a JavaScript code, namely *MaliceScript*, which collects intranet information abusing the features of WebRTC, and then infiltrates a targeted website from its own intranet. This work focuses on the attack implementation and only provides recommendations for end-user protection. One of the defensive measures is to notify the user of the browser and ask for explicit authorization before calling the WebRTC API, which is actually what our implementations provide.

The authors in [20], among other security and privacy risks, concentrate on that of WebRTC. Specifically, the authors were able to perform a network scan on victim networks and collect sensitive network information about other internal nodes, such as open ports based on heuristics like round-trip delays. They also claim that they developed a browser extension that is able to warn users about possible malicious activities, when a certain number of connections made to the private IP address of a user is surpassed. However, the countermeasure they propose for the specific privacy issue is just disabling WebRTC or using other extensions that can expose only the public IP of the user.

In addition, the work in [29] capitalizes on WebRTC in an attempt to disclose the user's private IP and execute network scans using an open source JavaScript scanner, called *jslanscanner* [30]. The network scan is supposed to detect not only possible online local network nodes, but routers as well. The authors extend the way a malicious user can use WebRTC to expose user's IP; nonetheless, they omit proposing a solution to safeguard against these kinds of attacks.

A Browser Scanner is proposed in [31] where an online service uses an XMLHttpRequest and WebRTC for the purpose of collecting information about the network a user resides in. The authors present the results they were able to gather from the scanned networks; however, they suggest that JavaScript and WebRTC should be disabled for avoiding such scans by malicious websites.

In [32], the use of mDNS [33] is proposed for protecting private IP addresses. In the proposed mechanism, the web browser registers an ephemeral mDNS name for the private IP address of the host and then provides this name in its ICE candidates instead of the IP address. The ICE candidates negotiations can proceed as usual and the mDNS names can be resolved to IP addresses only when necessary and without revealing the actual address to the other endpoint. The main issue with this approach is that it can break WebRTC applications. There are various reasons for that: (a) mDNS is built on the assumption that the names never leave the local domain, (b) the SDP protocol expects IP

addresses instead of DNS names, and (c) if a DNS server tries to resolve an mDNS name it will fail as no relevant resource record exists.

WebRTC's IP leak issue is also addressed by some browser add-ons for content-filtering, including ad-blocking. Two of them, namely Privacy Badger [34] and uBlock Origin [35], both multi-platform and open-source, seem to stand out. However, while these extensions prevent WebRTC from leaking the peer machine's private IP address by blocking it, they do not solve the problem of public IP leakage happening through STUN requests.

There is also a volume of works which utilize the WebRTC leak in order to fulfil other purposes. In [36–38], WebRTC is used to collect private IP addresses for user tracking and device fingerprinting. The authors in [39] utilize WebRTC for device fingerprinting in order to trace Web attackers even when they are hidden behind VPN or anonymous networks. WebRTC is also used in [40] for device fingerprinting in order to provide augmented user authentication. In all the previous cases, no countermeasures against the WebRTC leak have been implemented.

It is worth noting here that, as explained in Section 2, the signaling protocol between web servers is not part of WebRTC. This means that, even if a competent solution that protects users' privacy is employed on WebRTC, personal user information can leak from the signaling protocol [41]. On the application level, this can be prevented with solutions like [42,43] when SIP is utilized as the signaling protocol. On the network level, Tor-based solutions ([44,45]) can prevent leaking IP addresses to unauthorized entities.

Overall, while the given vulnerability of WebRTC is addressed by a number of works in the literature, most of them either use this vulnerability to fingerprint clients inside a network or propose countermeasures that partially protect user privacy, such as the use of a VPN or a combination of browser extensions. In addition, some works propose the complete deactivation of WebRTC on the browser, which would obligate the user to re-enable it every time they need it. To the best of our knowledge, the work at hand is the first to propose two different practical and effective schemes which can detect the use of WebRTC by a webpage in time, i.e., before any WebRTC requests are made. The proposed solutions can inform the user on-the-fly about whether their privacy is at risk by WebRTC when they try to visit a webpage. Thus, if the user is really in need of WebRTC capabilities, then the webpage is allowed to load; otherwise, the user can choose to continue by disabling the corresponding JavaScript code.

9. Conclusions

As web real-time communications gain increasingly more attention because of their simplicity of use via a typical browser, more websites turn their interest to WebRTC. However, although WebRTC is standardized through the W3C and the Internet Engineering Task Force (IETF) [46] and is being used by the most popular browsers, such as Chrome and Firefox, major privacy issues remain unresolved. Namely, in its attempt to bypass network obstacles, such as NAT and firewalls, WebRTC will silently leak peers' IP addresses without asking for permission. This is because WebRTC employs the ICE protocol, and therefore STUN or TURN mechanisms, in order to determine the most efficient connection path between peers. Furthermore, any website is able to force the user's browser to execute such requests to a STUN or TURN server, without the user being notified. Consequently, the owner of the server will be able to log each user's private and public IPs, even if the user is using VPN or an anonymity network.

While the existing solutions propose the complete deactivation of WebRTC along with its full capabilities, this work proposes and evaluates two diverse schemes to identify WebRTC communication intentions of a website. The first scheme is a browser's extension which by default disables any STUN or TURN request that a visited webpage may attempt to execute. The second proposes the use of a gateway, which we implemented in both C++ and Golang. The gateway acts as a MITM, inspecting the contents of the webpage before they are sent to the client. If the webpage the user is about to visit is trying to perform any STUN or TURN requests, the user will be informed and asked if they wish

to proceed. The first scheme is lightweight with no extra delay as it disables WebRTC by default and re-enables it on-the-fly if the user requires it. However, a different custom-tailored implementation is necessary per browser. The second constitutes a more generic solution as it is browser-independent; however, it adds a more significant delay depending on the size of the webpage, since the source code of the webpage needs to be examined before served. In absence of a built-in browser solution, both the proposed schemes could be proven an acceptable solution as long as WebRTC protocol continues to use untamed STUN or TURN servers for building ICE candidates.

Both of the proposed solutions protect the privacy of end-users who utilize WebRTC-capable browsers by completely disabling WebRTC. Obviously, this protection is not provided to users who choose to authorize the use of WebRTC after they have been informed that a webpage is about to use this protocol. In addition to privacy, our proposals safeguard communicating peers from other security-related attacks as well. An example is a script that forces the browser to make a call to a recording device without the user being aware, transforming their computer into a surveillance device. Since the port used by the web browser is open, another attack is to use this port to send arbitrary data bypassing firewalls, thus leaking information outside the intranet. A further example is to abuse the aforementioned open firewall port and turn the browser into a botnet client.

From the discussion given in the IP Disclosure subsection, one can mark down some directions for future work. Another line of research could investigate potential security issues that can arise when a user is tricked into using a rogue STUN or TURN server: potential attacks include session hijacking, MITM, and DoS. Future work can also include other solutions to be tested as browsers' extensions, such as white-listing STUN and TURN servers. This way, by defining trusted servers, IP logging from malicious users could be prevented.

Author Contributions: All the authors have contributed equally to the various stages of this work, including conceptualization, methodology, information gathering, implementation, discussion of the results, and writing of the original version of the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. WebRTC 1.0: Real-Time Communication between Browsers. 2019. Available online: <https://www.w3.org/TR/webrtc/> (accessed on 26 April 2020).
2. WebRTC Market. Available online: <https://www.acumenresearchandconsulting.com/webrtc-market> (accessed on 26 April 2020).
3. Global WebRTC Market Will Reach USD 21,023 Million By 2025: Zion Market Research. Available online: <https://www.globenewswire.com/news-release/2019/02/15/1725959/0/en/Global-WebRTC-Market-Will-Reach-USD-21-023-Million-By-2025-Zion-Market-Research.html> (accessed on 26 April 2020).
4. Keränen, A.; Holmberg, C.; Rosenberg, J. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*; RFC 8445; IETF: Wilmington, DE, USA, 2018. [CrossRef]
5. Petit-Huguenin, M.; Salgueiro, G.; Rosenberg, J.; Wing, D.; Mahy, R.; Matthews, P. *Session Traversal Utilities for NAT (STUN)*; RFC 8489; IETF: Wilmington, DE, USA, 2020. [CrossRef]
6. Reddy, K. T.; Johnston, A.; Matthews, P.; Rosenberg, J. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*; RFC 8656; IETF: Wilmington, DE, USA, 2020. [CrossRef]
7. Syverson, P.; Dingleline, R.; Mathewson, N. *Tor: The Second Generation Onion Router*; Usenix Security: San Diego, CA, USA, 2004; pp. 303–320.
8. Zantout, B.; Haraty, R. I2P data communication system. In Proceedings of the ICN, St. Maarten, the Netherlands, 23–28 January 2011; pp. 401–409.

9. Liu, C.; Cui, X.; Wang, Z.; Wang, X.; Feng, Y.; Li, X. MaliceScript: A Novel Browser-Based Intranet Threat. In Proceedings of the 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC), Guangzhou, China, 18–21 June 2018; pp. 219–226.
10. Rosenberg, J.; Schulzrinne, H.; Camarillo, G.; Johnston, A.; Peterson, J.; Sparks, R.; Handley, M.; Schooler, E. *SIP: Session Initiation Protocol*; RFC 3261 (Proposed Standard); IETF: Wilmington, DE, USA, 2002; Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665, 6878.
11. Uberti, J.; Jennings, C.; Rescorla, E. *JavaScript Session Establishment Protocol*; Internet-Draft draft-ietf-rtcweb-jsep-26; IETF, Network Working Group: Wilmington, DE, USA, 2019; Work in Progress.
12. Johnston, A.B.; Burnett, D.C. *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*, 3rd ed.; Digital Codex LLC: St. Louis, MO, USA, 2012; Volume 4, pp. 11–12.
13. Begen, A.C.; Kyzivat, P.; Perkins, C.; Handley, M.J. *SDP: Session Description Protocol*; Internet-Draft draft-ietf-mmusic-rfc4566bis-37; IETF: Wilmington, DE, USA, 2019; Work in Progress.
14. Rescorla, E. *Security Considerations for WebRTC*; Internet-Draft draft-ietf-rtcweb-security-12; IETF, RTC-Web: Wilmington, DE, USA, 2019; Work in Progress.
15. AdBlock Plus. Available online: <https://github.com/adsblock/adsblockplus> (accessed on 26 April 2020).
16. Krasnyansky, M.; Yevmenkin, M. Virtual Point-to-Point (TUN) and Ethernet (TAP) Devices. Available online: <http://vtun.sourceforge.net/tun/index.html> (accessed on 26 April 2020).
17. TunnelBear LLC. Tunnelbear. Available online: <https://www.tunnelbear.com/> (accessed on 28 April 2020).
18. AnchorFree, Hotspot Shield. Available online: <https://www.hotspotshield.com/> (accessed on 28 April 2020).
19. Al-Fannah, N.M. One leak will sink a ship: WebRTC IP address leaks. In Proceedings of the 2017 International Carnahan Conference on Security Technology (ICCST), Madrid, Spain, 23–26 October 2017; pp. 1–5.
20. Mohammadreza, H.; Mohammad, G. One leak is enough to expose them all. In Proceedings of the Engineering Secure Software and Systems: 10th International Symposium, Campus Paris-Saclay, France, 26–27 June 2018; pp. 664–669.
21. Fix Shared VPN/Tor Server Leak Bug. Available online: <https://github.com/adrelanos/vpn-firewall/issues/12> (accessed on 26 April 2020).
22. Media Capture and Streams. Available online: <https://www.w3.org/TR/mediacapture-streams/> (accessed on 26 April 2020).
23. European Union Public Licence. Available online: https://ec.europa.eu/info/european-union-public-licence_en (accessed on 26 April 2020).
24. Leech, M.D. *SOCKS Protocol Version 5*; RFC 1928; IETF: Wilmington, DE, USA, 1996. [CrossRef]
25. The Average Web Page Is 3MB. How Much Should We Care? Available online: <https://speedcurve.com/blog/web-performance-page-bloat> (accessed on 26 April 2020).
26. 10 Ad Blocking Extensions Tested for Best Performance. Available online: <https://www.raymond.cc/blog/10-ad-blocking-extensions-tested-for-best-performance/3/> (accessed on 26 April 2020).
27. A Primer for Web Performance Timing APIs. 2019. Available online: <https://w3c.github.io/perf-timing-primer/> (accessed on 26 April 2020).
28. Coturn TURN Server Project. Available online: <https://github.com/coturn/coturn> (accessed on 26 April 2020).
29. Reiter, A.; Marsalek, A. WebRTC: Your privacy is at risk. In Proceedings of the Symposium on Applied Computing, Marrakech, Morocco, 4–6 April 2017; pp. 664–669.
30. JSLanScanner. Available online: <https://code.google.com/archive/p/jslanscanner/> (accessed on 26 April 2020).
31. Hosoi, R.; Saito, T.; Ishikawa, T.; Miyata, D.; Chen, Y. A browser scanner: Collecting intranet information. In Proceedings of the 2016 19th International Conference on Network-Based Information Systems (NBIS), Ostrava, Czech Republic, 7–9 September 2016; pp. 140–145.
32. Fablet, Y.; Borst, J.D.; Uberti, J.; Wang, Q. *Using Multicast DNS to Protect Privacy When Exposing ICE Candidates*; Internet-Draft draft-ietf-rtcweb-mdns-ice-candidates-04; IETF, RTCWEB: Wilmington, DE, USA, 2019; Work in Progress.
33. Cheshire, S.; Krochmal, M. *Multicast DNS*; RFC 6762; IETF: Wilmington, DE, USA, 2013. [CrossRef]
34. EFForg/Privacy Badger. Available online: <https://github.com/EFForg/privacybadger> (accessed on 26 April 2020).
35. Hill, R. uBlock Origin. Available online: <https://github.com/gorhill/uBlock> (accessed on 26 April 2020).

36. Klein, A.; Pinkas, B. From IP ID to Device ID and KASLR Bypass. In Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19), Santa Clara, CA, USA, 14–16 August 2019; USENIX Association: Santa Clara, CA, USA, 2019; pp. 1063–1080.
37. Al-Fannah, N.M.; Li, W. Not All Browsers are Created Equal: Comparing Web Browser Fingerprintability. In *Advances in Information and Computer Security*; Obana, S., Chida, K., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 105–120.
38. Englehardt, S.; Narayanan, A. Online Tracking: A 1-Million-Site Measurement and Analysis. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16), Vienna, Austria, 24–28 October 2016; Association for Computing Machinery: New York, NY, USA, 2016; pp. 1388–1401. [[CrossRef](#)]
39. Liu, X.; Liu, Q.; Wang, X.; Jia, Z. Fingerprinting Web Browser for Tracing Anonymous Web Attackers. In Proceedings of the 2016 IEEE First, International Conference on Data Science in Cyberspace (DSC), Changsha, China, 13–16 June 2016; pp. 222–229.
40. Alaca, F.; van Oorschot, P.C. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods. In Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16), Los Angeles, CA, USA, 5–9 December 2016; Association for Computing Machinery: New York, NY, USA, 2016. [[CrossRef](#)]
41. Kambourakis, G. Anonymity and closely related terms in the cyberspace: An analysis by example. *J. Inf. Secur. Appl.* **2014**. [[CrossRef](#)]
42. Karopoulos, G.; Kambourakis, G.; Gritzalis, S.; Konstantinou, E. A framework for identity privacy in SIP. *J. Netw. Comput. Appl.* **2010**, *33*, 16–28. [[CrossRef](#)]
43. Karopoulos, G.; Kambourakis, G.; Gritzalis, S. PrivaSIP: Ad-hoc identity privacy in SIP. *Comput. Standards Interfaces* **2011**, *33*, 301–314. [[CrossRef](#)]
44. Karopoulos, G.; Fakis, A.; Kambourakis, G. Complete SIP Message Obfuscation: PrivaSIP over Tor. In Proceedings of the 2014 Ninth International Conference on Availability, Reliability and Security (ARES), Fribourg, Switzerland, 8–12 September 2014; pp. 217–226. [[CrossRef](#)]
45. Fakis, A.; Karopoulos, G.; Kambourakis, G. OnionSIP: Preserving Privacy in SIP with Onion Routing. *J. Univ. Comput. Sci.* **2017**, *23*, 969–991.
46. Rodriguez, P.; Cerviño, J.; Trajkovska, I.; Salvachúa, J. Advanced videoconferencing services based on webRTC. In Proceedings of the IADIS International Conferences Web Based Communities and Social Media, Lisbon, Portugal, 19–21 July 2012; pp. 180–184.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).