

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Daniel Tšarin 164648IABB

**ANALYSIS AND IMPLEMENTATION OF
METHODS FOR FAKE IP ADDRESS
DETECTION**

Bachelor's thesis

Supervisor: Martin Rebane
MSc

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Daniel Tšarin 164648IABB

VÕLTSITUD IP AADRESSI AVASTAMISE MEETODITE ANALÜÜS JA RAKENDUS

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc

Author's declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature, and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Daniel Tšarin

3.01.2021

Abstract

The purpose of this thesis is to analyze modern methods for fake IP detection and briefly describe how does each method work. Also, it is planned to create an application that will show the implementation of each method. Further, the application will be used in experiments with IP masking tools like proxy and VPN.

To achieve the objective, the author will describe the theoretical side of each method in the first half of this thesis. In the second half of this thesis, it is planned to present the application implementation and experiments results, that show the efficiency of selected fake IP detection methods.

During this research, it has been found that methods based on WebRTC, JS Timezone, and Geolocation API can effectively detect the fact of IP substitution if proxy is used for IP masking. However, WebRTC and Geolocation API in most cases cannot detect the fact of IP substitution if VPN is used for IP masking.

This thesis is written in English and is 24 pages long, including 4 chapters, 15 figures, and 5 tables.

Annotation

Käesoleva lõpputöö eesmärk on analüüsida kaasaegseid võltsitud IP aadressi avastamise meetodeid ja lühedalt kirjeldada kuidas iga meetod töötab. Samuti on plaanis luua rakendus mis näitab iga meetodi teostust. Seejärel rakendus kasutatakse eksperimentides koos erinevatega IP aadressi võltsimise tööriistadega nagu proxy ja VPN.

Eesmärgi saavutamiseks, autor kirjeldab iga meetodi teoreetilist osa esimeses töö pooles. Teises töö pooles on plaanis esitada rakenduse teostust ja eksperimentide tulemused, mis näitavad valitud võltsitud IP aadressi avastamise meetodite efektiivsust.

Selle töö käigus oli leitud, et meetodid mis põhinevad WebRTC, JS Timezone ja Geolocation API baasil võivad efektiivselt avastada IP aadressi asenduse fakti juhul kui proxy on kasutatud IP maskeerimiseks. Kuid WebRTC ja Geolocation API enamasti ei saa avastada IP aadressi asenduse fakti juhul kui VPN on kasutatud IP maskeerimiseks.

Lõpputöö on kirjutatud inglise keeles ning sisaldab teksti 24 leheküljel, 4 peatükki, 15 joonist ja 5 tabelleid.

List of abbreviations and terms

Geolocation API	API used by websites to determine user location
Fake IP	IP address substitution with tools like a proxy, VPN, and others
Fingerprinting	Techniques that allow websites or applications to identify user's different device parameters
ICE	Interactive Connectivity Establishment, a technology used for peer-to-peer connection establishment
mDNS	Multicast DNS is a simplified version of traditional DNS which is used by WebRTC
NAT	Network Address Translation provides security for private networks, solves the issue with lack of IPv4 addresses but complicates peer-to-peer connection establishment
Proxy	An intermediary server between a client and target server which can protect the client from attack and provide anonymity but at the same time can be used by criminal for IP substitution
P2P	Peer-to-peer architecture in which hosts transmit data between each other directly, without a centralized server
RFC	Request for Comments documents that define the Internet standards
SDP	Session Description Protocol used by data streaming protocols like WebRTC
STUN	Session Traversal Utilities for NAT can be used to determine the IP address and port of host allocated to it by a NAT
TURN	Traversal Using Relay NAT is a relay that helps two peers to establish a peer-to-peer connection
WebRTC	Web Real-Time Communication technology which allows sending audio, video, or arbitrary data via P2P

Table of contents

1 Introduction	10
1.1 Objective.....	10
2 Theory.....	11
2.1 Overview of fake IP detection methods	11
2.2 HTTP headers	12
2.3 WebRTC.....	13
2.3.1 NAT problem.....	13
2.3.2 ICE and SDP.....	14
2.3.3 Private IP addresses security leak.....	15
2.4 Timezone	16
2.5 Geolocation API	16
2.6 TCP/IP fingerprinting	17
2.6.1 Active and passive OS fingerprinting.....	17
2.6.2 Nmap	18
3 Implementation.....	19
3.1 Application architecture	19
3.1.1 Modules	19
3.1.2 Sequence.....	20
3.1.3 Conceptual diagrams	21
3.2 Fingerprinting methods.....	23
3.2.1 HTTP headers	24
3.2.2 Timezone implementation	24
3.2.3 Geolocation implementation	25
3.2.4 WebRTC implementation.....	28
3.2.5 TCP/IP fingerprinting (Nmap)	28
4 Experiments with fake IP detection tool.....	30
4.1 Test with proxy	30
4.2 Test with VPN	31
4.3 Results clarification and methods comparison	32
Summary.....	33
References	34

Appendix 1 – Public IP from WebRTC.....	35
Appendix 2 – FakeIP application sequence diagram.....	36

List of figures

Figure 1 Fake IP detection methods (author's diagram).....	11
Figure 2 STUN/TURN servers use in WebRTC (author's diagram).....	14
Figure 3 Example of SDP data (author's code snippet).....	15
Figure 4 FakeIP package diagram (author's diagram)	20
Figure 5 “Shared” module component diagram (author's diagram)	22
Figure 6 “Fingerprinting” module component diagram (author's diagram)	23
Figure 7 Request to get IP data from headers (author's code snippet)	24
Figure 8 IPData class (author's code snippet).....	24
Figure 9 User's system Timezone value obtaining (author's code snippet).....	25
Figure 10 Geolocation obtaining (author's code snippet).....	26
Figure 11 Geolocation API response example (author's picture)	26
Figure 12 Geolocation API permissions request (author's picture)	27
Figure 13 Result of Nmap OS scan (author's code snippet).....	29
Figure 14 Public IP obtainment through WebRTC (author's code snippet).....	35
Figure 15 FakeIP sequence diagram (author's diagram)	36

1 Introduction

Nowadays, fraudulent payments cause significant damage to the E-commerce business. Nevertheless, cybercriminals have one thing in common – they use IP address masking solutions such as proxy, VPN, mobile 3G/4G internet, or SSH tunnels. In fact, a regular user has no reason for masking his/her real IP address in most of the cases (exceptions are privacy reasons or desire to get access to restricted resources). So how can we understand if a user uses any IP address masking solution and recognize the fact of IP address substitution to prevent illegal activities? At the same time, are there any mechanisms that may resist fake IP address detection?

An IP address is one of the most important and unique identifiers for worldwide web users. From the transport layer perspective, the IP address solves the simple issue. It helps to address networking packets. At the same time, on the application layer, the IP address helps to uniquely identify users and personalize content, based on their location.

The unique identification part is the most important in the security part of a web application. Developers can implement different fake IP revealing methods on their websites to prevent different harmful and illegal activities.

1.1 Objective

The aim of this work is to analyse methods for fake IP detection, assess reliability of each method and understand whether it might be useful to use these methods separately or together in combination. The thesis itself is logically divided into 3 main sections: the theoretical side of each method, implementation, and the demonstration part with experiments results. The implementation part will have an architecture description and code snippets.

As a result, within this project, it is planned to create a web application, which will be available as an open-source fake IP detection tool. The main purpose of the application - is to spot mismatches in different parameters that can point to the fact that the IP address is faked.

2 Theory

This part will describe the theoretical part of the fake IP address detection methods.

2.1 Overview of fake IP detection methods

Over the years, quite a big subset of methods for IP address detection have been invented. Some of the technologies, that allow us to detect the fact of IP address substitution were created for a different purpose at all. For example, WebRTC's main purpose is to allow users to have real-time communications. However, due to technical implementation, it turned out to be a great tool for real public IP address detection. It was possible due to the peer-to-peer approach that is used in this communication technology.

Figure 1 shows the diagram with the methods for fake IP detection, that are going to be considered in this thesis.

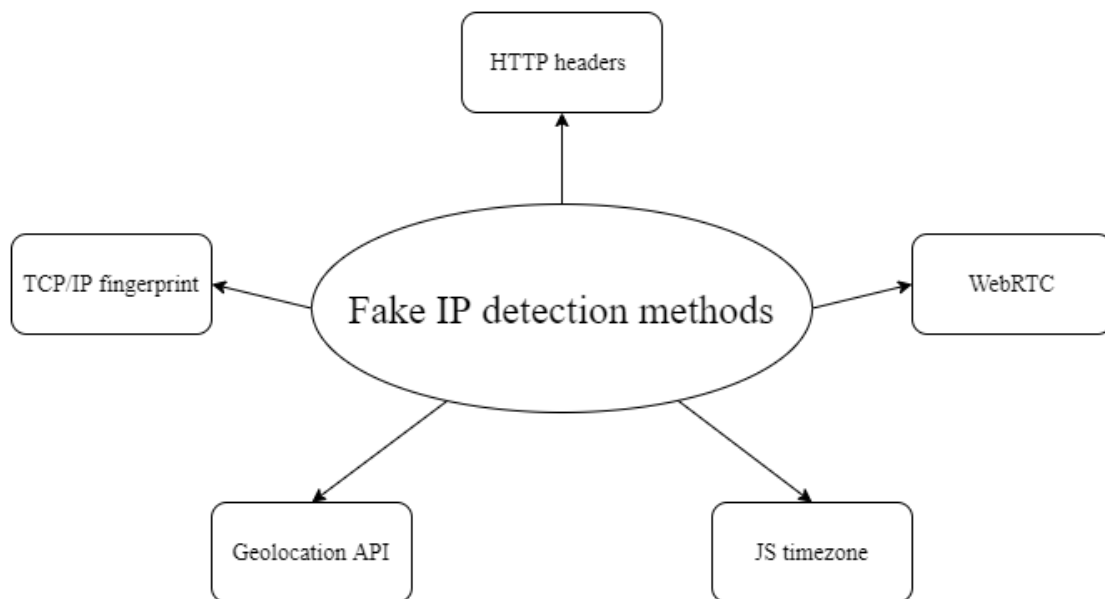


Figure 1 Fake IP detection methods (author's diagram)

It is necessary to understand that by the use of methods, mentioned in Figure 1, the fact of IP address substitution is established indirectly. JS Timezone, Geolocation API, TCP/IP fingerprint can just point out the discrepancy of some parameters that may indicate that the IP address is faked. However, WebRTC is an exception because it directly points to the difference between public IP addresses.

The IP address obtained through HTTP headers and its parameters (location, local Timezone, and its TCP/IP fingerprint) are used for comparison with analogical parameters that were received through corresponding methods. For example, the Timezone of the IP address obtained through HTTP headers, and Timezone, obtained through JS Timezone. Table 1 describes mismatches that are going to be detected by each method.

Table 1 Differences in parameters by method indicating a mismatch

Method	Difference indicating mismatch
WebRTC	Public IP from HTTP headers and public IP from WebRTC
JS Timezone	IP Timezone and System Timezone obtained by JS
Geolocation API	IP location and location obtained by Geolocation API
TCP/IP fingerprint	OS declared in User-agent and OS of public IP host (which might be a proxy server IP)

2.2 HTTP headers

There is a set of headers which provides an opportunity to detect user IP address. IP address detection by HTTP headers is one of the most common methods. At the same time, this is the easiest way to get client IP. Nowadays, there are lots of API services, that provide an opportunity to get a client's IP address from headers. It makes it easier for the website developers to get client IP from headers without the implementation of their HTTP request analysis solution. Examples of such services: www.ipify.org and www.geojs.io.

“The X-Forwarded-For (XFF) header is a de-facto standard header for identifying the originating IP address of a client connecting to a web server through an HTTP proxy or a load balancer” [1]. However, we need to understand that almost all proxies have masked HTTP headers and in most cases, HTTP headers will not reveal the real IP address of the client. IP from headers and its parameters can be used as a starting point in comparison to IP and parameters obtained with other techniques like WebRTC, Geolocation API, and others.

2.3 WebRTC

WebRTC is a technology that allows web applications and websites to capture and optionally stream audio and/or video media, as well as to exchange arbitrary data between browsers without requiring an intermediary. Connections between two peers are represented by the `RTCPeerConnection` interface. Once a connection has been established and opened using `RTCPeerConnection`, `MediaStreams`, or data channels (`RTCDataChannels`) can be added to the connection. [2]

2.3.1 NAT problem

NAT (Network Address Translation) is a mechanism that allows hiding hosts within a local network behind the private IP address. Thereby NAT is an intermediary between private and public networks.

“Primarily NAT was introduced to the world of IT and networking due to the lack of IP addresses, or looking at it from another view, due to the vast amount of growing IT technologies relying on IP addresses. To add to this, NAT adds a layer of security, by hiding computers, servers, and other IT equipment from the outside world.” [3]

However, NAT brings quite a big problem for peer-to-peer connections, since hosts, that would like to communicate with each other may not know the IP address of the opponent, which is hidden behind the NAT. Also, the NAT will act as a firewall, bypassing packets outside, while denying packets from the outside, which makes peer-to-peer connection impossible.

As a solution to the NAT problem, WebRTC comes with a solution of NAT traversal, which uses STUN and TURN servers to detect the real IP of the peer. STUN is a protocol to discover your public address and determine any restrictions in your router that would prevent a direct connection with a peer [4].

The client will send a request to a STUN server on the Internet who will reply with the client’s public address and whether or not the client is accessible behind the router’s NAT. Figure 2 schematically describes the process of WebRTC peer connection establishment with the help of the STUN/TURN server.

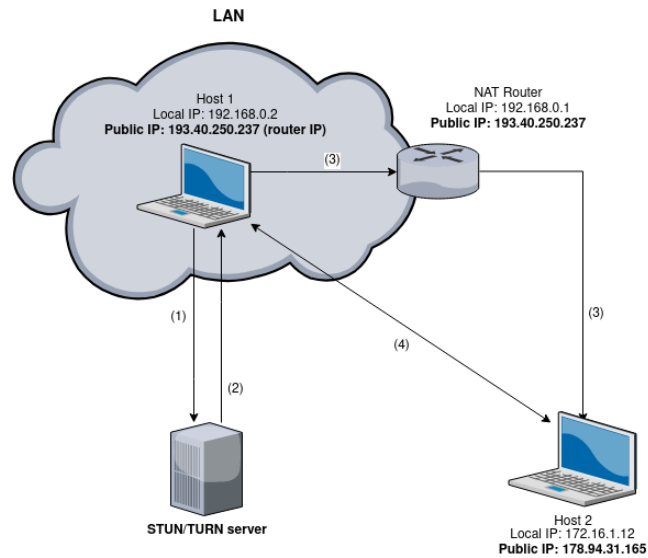


Figure 2 STUN/TURN servers use in WebRTC (author's diagram)

Steps of WebRTC peer connection with STUN/TURN:

1. Host 1 sends a request to the STUN/TURN server
2. STUN/TURN server sends public IP of Host 1, which is the IP address of NAT router
3. Host 1 passes IP address received in step 2 to Host 2
4. Host 1 and Host 2 establish a direct peer-to-peer connection

2.3.2 ICE and SDP

To further move towards WebRTC IP leak, we need to consider two important concepts of WebRTC – SDP, and ICE.

SDP describes a multimedia session, which can be audio, video, whiteboards, fax, modem, and other streams. It provides a general-purpose, standard representation to describe various aspects of the multimedia session, such as media capabilities, transport addresses, and related metadata. [5] In our context, the transport addresses part is the most important since it contains the IP addresses of a peer.

```

v=0
o=mozilla...THIS_IS_SDPARTA-81.0 7322323708244440960 0 IN IP4 0.0.0.0
s=-
t=0 0
a=sendrecv
a=fingerprint:sha-256
77:86:34:AE:EA:79:F6:46:D2:E3:EE:B5:95:DE:74:CF:6A:7D:49:5A:2F:40:1F:24:C6:D8
:5D:E3:85:25:F7:5B
a=group:BUNDLE 0
a=ice-options:trickle
a=msid-semantic:WMS *
m=application 56392 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 193.40.250.235
a=candidate:0 1 UDP 2122252543 2f2d895b-7eff-44e5-b81d-aaf14897cd7b.local
56392 typ host
a=candidate:2 1 TCP 2105524479 2f2d895b-7eff-44e5-b81d-aaf14897cd7b.local 9
typ host tcptype active
a=candidate:1 1 UDP 1686052863 193.40.250.235 56392 typ srflx raddr 0.0.0.0
rport 0
a=sendrecv
a=end-of-candidates
a=ice-pwd:4201e5a5e56e5e455bd6d8dbe9c293af
a=ice-ufrag:3ae78d10a=mid:0
a=setup:actpass
a=sctp-port:5000
a=max-message-size:1073741823

```

Figure 3 Example of SDP data (author's code snippet)

ICE is a framework used by WebRTC for connecting two peers, regardless of network topology. This protocol lets two peers find and establish a connection with one another even though they may both be using NAT. [6] ICE is an extension to the offer/answer model and works by including a multiplicity of IP addresses and ports in SDP offers and answers. The IP addresses and ports included in the SDP and the connectivity checks are performed using the STUN or TURN servers. [7]

In other words, the SDP contains all the peer-related data to perform the further connection.

2.3.3 Private IP addresses security leak

In 2014 there was a bug reported to the chromium project which stated that WebRTC provides quite a severe leak which allows websites to get user private IP address without notifying the user [8].

Therefore, in 2019 WebRTC changed their approach in ICE candidates gathering regarding private IP addresses, since it is quite a serious leak, that might be exposed for fingerprinting or direct attacks:

“As detailed in [IPHandling], exposing client private IP addresses by default to web applications maximizes the probability of successfully creating direct peer-to-peer connections between clients, but creates a significant surface for user fingerprinting.” [9]

As a fix to this security leak, there was implemented a solution with the usage of mDNS and private IP addresses became masked with UUID¹. We can see this from the example of SDP data specified in paragraph 2.3.2, Figure 3.

2.4 Timezone

The Timezone mismatch is quite a trivial solution that allows detecting IP substitution. It is needed to compare Timezone values of public IP, that was received by request headers and get system Timezone via browser by executing simple JavaScript code - the creation of Date instance which will show the real system Timezone.

2.5 Geolocation API

The Geolocation API allows the user to provide their location to web applications if they so desire. The Geolocation API is accessed via a call to navigator.geolocation; this will cause the user's browser to ask them for permission to access their location data. If users accept it, then the browser will use the best available functionality on the device to access this information. [10]

The Geolocation API defines a high-level interface to location information associated only with the device hosting the implementation, such as latitude and longitude. Sources of location information include Global Positioning System (GPS) and location, inferred from network signals: such as IP address, RFID WiFi, and Bluetooth MAC addresses, and GSM/CDMA cell ID's, as well as user input. [11] However, the location obtained with Geolocation API might not be entirely accurate and for this additional parameter –

¹ UUID - Universally Unique Identifier (<https://tools.ietf.org/html/rfc4122>)

“accuracy” is passed together with latitude and longitude values, which shows the position accuracy in meters.

2.6 TCP/IP fingerprinting

Network traffic from a computer can be analyzed to detect what operating system it is running. It is largely due to the differences in how the TCP/IP stack is implemented in various operating systems. A simple but effective passive method is to inspect the initial “Time To Live” (TTL) in the IP header and the TCP window size (the size of the receive window) of the first packet in a TCP session. One reason why the TTL and window size values vary between different OS - because the RFC’s for TCP and IP do not require implementations to use any particular default value for these fields. [12]

Most often, users and their proxy server’s operating systems do not match. It is because - proxy servers are running mostly on Linux, while the highest percentage of users on the web prefer Windows or macOS as main operating systems. Thus, if a user with macOS uses a proxy server, running on Linux, using Nmap we can detect this mismatch. It just needs to perform TCP/IP fingerprint and to compare its result to the operating system value from the User-agent.

2.6.1 Active and passive OS fingerprinting

Tools for OS fingerprinting are separated into 2 types – active and passive. One of the best examples of active fingerprinting is Nmap². The popular port scanner Nmap can identify the operating system of a remote computer, by sending six packets with specially crafted option combinations in the TCP layer (for example window scale, NOP, and EOL options). Nmap then watches how the scanned host responds to these odd packets. [13]

Passive fingerprinting uses most of the same techniques as the active fingerprinting performed by Nmap. The difference is that a passive system simply sniffs the network, opportunistically classifying hosts as it observes their traffic. It is more difficult than active fingerprinting since you have to accept whatever communication happens, rather than designing your own custom probes. [14] An example of tools that can be used for

² <https://nmap.org/>

passive fingerprinting is possible. With passive fingerprinting, we just need a user, somehow, to send a request to our server with configured ports.

However, active fingerprinting can bring negative consequences, because port scanning which is used in active fingerprinting may be illegal in some jurisdictions.

2.6.2 Nmap

“Nmap OS fingerprinting works by sending up to 16 TCP, UDP, and ICMP probes to known open and closed ports of the target machine. These probes are specially designed to exploit various ambiguities in the standard protocol RFCs. Then Nmap listens for responses. Dozens of attributes in those responses are analyzed and combined to generate a fingerprint.” [13]

Due to the simplicity of the approach, the Nmap and similar tools are used most frequently for TCP/IP OS fingerprinting.

3 Implementation

This part will describe the implementation part of fake IP address detection methods. The author will use his open-source web application written on Angular, which demonstrates, how each of the methods works. The website used in testing is available at <http://fakeip.xyz/> and the source code can be found at <https://github.com/f1kus97/fake-ip>.

3.1 Application architecture

This section will cover the architectural details of the application.

3.1.1 Modules

The application is separated into two logical modules: “Fingerprinting” and “Shared”. The main purpose of components in the “Fingerprinting” module is to collect IP address related fingerprints. The “Shared” module is responsible for the view part of the application: templates, styles, and responsive grid layout alignment. Furthermore, the “Shared” module unites all templates of components from the “Fingerprinting” module.

The main purpose of services is to provide access to external API and separate all the complicated logic related to fingerprints collection. Services are used by components from the “Fingerprinting” module and return results asynchronously as Observable³ or Promise⁴.

Figure 4 shows a package diagram that describes the main application packages and their components.

³ <https://angular.io/guide/observables>

⁴ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

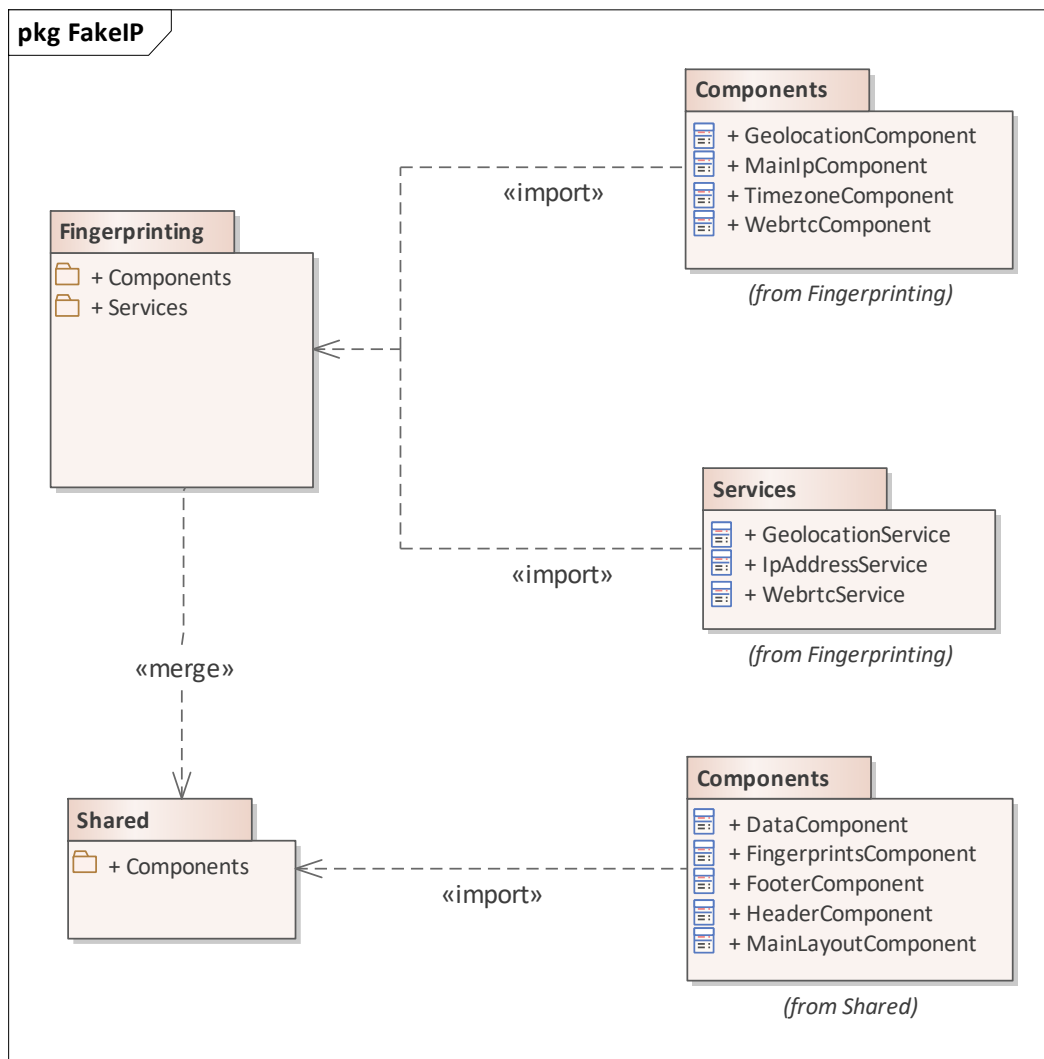


Figure 4 FakeIP package diagram (author's diagram)

3.1.2 Sequence

The sequence diagram (see Figure 15 from Appendix 2) shows the lifecycle of fingerprinting components.

1. `FingerprintsComponent` gathers and initializes all fingerprinting components: `MainIpComponent`, `WebrtcComponent`, `TimezoneComponent`, `GeolocationComponent`. Besides, `FingerprintsComponent` performs mismatch checks for public IP and location.
2. All fingerprinting services perform requests to third party API or perform fingerprinting logic. Services return results asynchronously as `Promise` or `Observable`.

3. Public IP address mismatch check is performed by FingerprintsComponent. As soon as FingerprintsComponent got the IP addresses from MainIpComponent and WebRtcComponent the mismatch check is performed in performIpMismatchCheck().
4. Timezone values mismatch is performed by the TimezoneComponent since local and system Timezone obtaining is performed by this component and there is no need to pass local and system values to FingerprintsComponent component.
5. Geolocation mismatch check is performed by the FingerprintsComponent. As soon as FingerprintsComponent got locations from MainIpComponent and GeolocationComponent the mismatch check is performed in performGeoMismatchCheck().
6. updateIP(), updateIpLocation(), updateWebRTCIP() and updateGeoApiLocation() are event listeners of Fingerprints component that update public IP addresses and location values in FingerprintsComponent.

3.1.3 Conceptual diagrams

Figure 5 shows the conceptual diagram of the “Shared” module. MainLayoutComponent includes HeaderComponent, FooterComponent and DataComponent. All components in this module are responsible for template formation. DataComponent includes FingerprintsComponent, which binds components from the Fingerprinting module.

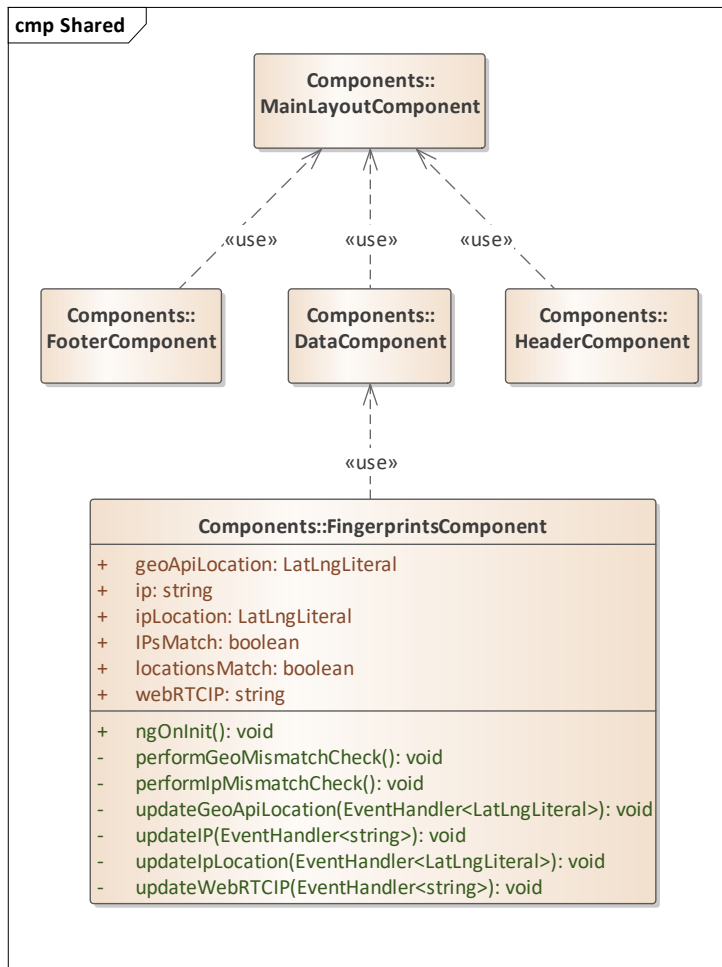


Figure 5 “Shared” module component diagram (author’s diagram)

Figure 6 shows the conceptual diagram of the “Fingerprinting” module. As stated above, the Fingerprinting module’s purpose is to collect IP related fingerprints. These fingerprints are HTTP headers, Geolocation API, Timezone, and WebRTC. All fingerprinting methods are implemented in the corresponding components.

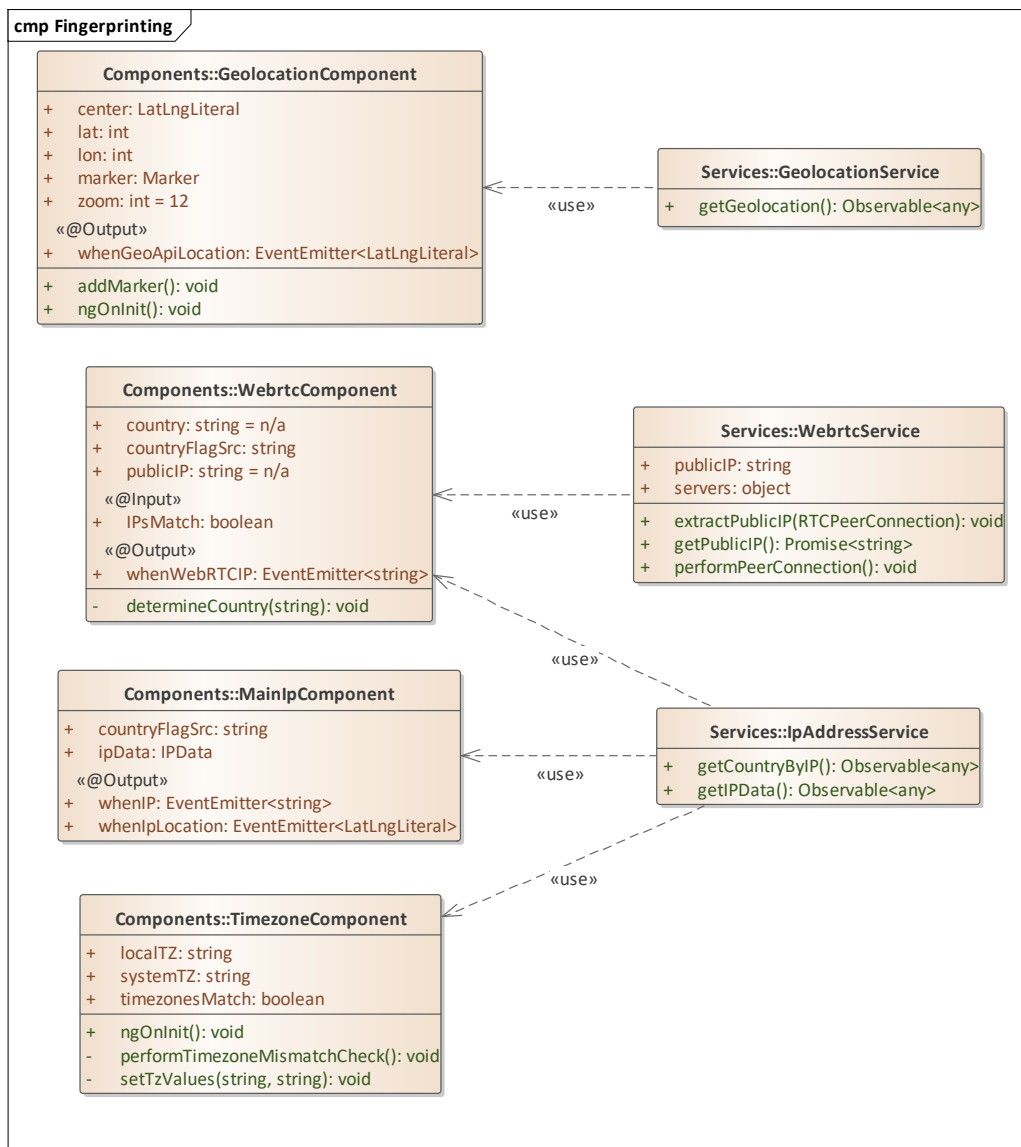


Figure 6 “Fingerprinting” module component diagram (author’s diagram)

3.2 Fingerprinting methods

This part will cover the implementation details of fingerprinting methods that allow detecting the fact of IP address substitution.

3.2.1 HTTP headers

In FakeIP project, IP from HTTP headers is obtained via third-party API provided by GeoJS⁵. IpAddressService has method getIPData which performs JSONP request to GeoJS API. JSONP is used for requests to bypass CORS limitations. Figure 7 shows the code snippet with JSONP request to GeoJS API.

```
getIPData(): Observable<any> {  
    return this.http.jsonp('https://get.geojs.io/v1/ip/geo.js', 'callback');  
}
```

Figure 7 Request to get IP data from headers (author's code snippet)

Response from GeoJS returns an object and it is mapped to IPData class. Figure 8 shows the IPData class structure which represent IP address related data received from GeoJS.

```
class IPData {  
    ip?: string;  
    country?: string;  
    country_code?: string;  
    country_code3?: string;  
    continent_code?: string;  
    city?: string;  
    region?: string;  
    latitude?: string;  
    longitude?: string;  
    accuracy?: number;  
    timezone?: string;  
    organization?: string;  
    asn?: number;  
    organization_name?: string;  
}
```

Figure 8 IPData class (author's code snippet)

3.2.2 Timezone implementation

Timezone mismatch detection is performed in the following steps:

⁵ <https://www.geojs.io/>

1. Obtaining local Timezone value based on the IP address from HTTP headers (in our case GeoJS). GeoJS was selected for IP data collection because it already contains Timezone value in its response.
2. Figure 9 shows the process of the user's system Timezone obtaining through the browser:

```
const systemTZ = Intl.DateTimeFormat().resolvedOptions().timeZone;
```

Figure 9 User's system Timezone value obtaining (author's code snippet)
3. Comparing local and system Timezone values. If values do not match, then this means that a user might have fake IP.

However, the accuracy of this method is the weakest since it does not necessarily mean that the user has a fake IP. There might be a case that a user just set the wrong system Timezone.

Especially it applies to the Timezones with the same offset but different names. For example, a user can be physically in Tallinn and he/she should have Europe/Tallinn +02:00 Timezone, however accidentally put Timezone with the same offset but a different name, for example, Europe/Stockholm +02:00.

Therefore, there are two approaches for comparing Timezone values – comparing just offsets and offsets together with Timezone names. The second approach is stricter and may produce false-negative results for normal users without a faked IP address. In the fake-IP project, the comparison is based on both parameters – offset and Timezone name.

At the same time, a user who uses IP substitution can set manually system Timezone to the same value as a proxy server to bypass this detection method. However, he/she will need to perform the system Timezone change each time the country of IP address changes.

3.2.3 Geolocation implementation

Implementation of geolocation is quite trivial task, just need to perform a call to one of the most popular JavaScript interfaces - Navigator.

Figure 10 shows the code snippet from the GeolocationService, which gets the user's location coordinates from the Geolocation API.

```
if (window.navigator && window.navigator.geolocation) {
  window.navigator.geolocation.getCurrentPosition(
    (position) => observer.next(position),
    (err) => /* Error handling */
  );
}
```

Figure 10 Geolocation obtaining (author's code snippet)

First, it verifies that the navigator and geolocation are not null and then calls for the `getCurrentPosition()` which returns a callback with coordinates that we then pass as an observable to the `GeolocationComponent`.

The `getCurrentPosition()` method returns a `GeolocationPosition` object with different parameters like latitude, longitude, accuracy (measured in meters), and others. Figure 11 shows an example of a `GeolocationPosition` object, returned from the Geolocation API.

```
GeolocationPosition
└─ coords: GeolocationCoordinates
   └─ accuracy: 1631
      altitude: null
      altitudeAccuracy: null
      heading: null
      latitude: 59.4509824
      longitude: 24.887296
      speed: null
   └─ <prototype>: GeolocationCoordinatesPrototype { latitude: Getter, longitude: Getter, altitude: Getter, ... }
  timestamp: 1602682281370
  └─ <prototype>: GeolocationPositionPrototype { coords: Getter, timestamp: Getter, ... }
```

Figure 11 Geolocation API response example (author's picture)

As soon as coordinates are received from the Geolocation, API comparison can be performed with the IP address coordinates, obtained previously from request to GeoJS in a form of an `IPData` object. Slight inaccuracy might be because of Geolocation API inaccuracy and because the IP obtained by request headers returns the location of the ISP provider.

After receiving coordinates from both sources – `IPData` and Geolocation API, it is needed to perform a comparison of latitude and longitude values. However, due to the inaccuracy of both methods, the critical distance difference needs to be defined. Each developer can select their distance difference, which will be set as critical because there are no official recommendations for this. The fake-IP project uses 0.5 delta (approx. 55 km) for latitude

and 1.0 delta (approx. 58 km) for longitude. Table 2 shows the ratio between degrees and kilometers with different delta values.

Table 2 Latitude/Longitude and distance (km) ratio

Initial value (lat, lon)	Delta (lat, lon)	Result value (lat, lon)	Approx. difference (km)
58.37, 26.72	0.1, 0	58.47, 26.72	11
58.37, 26.72	0.5, 0	58.87, 26.72	55
58.37, 26.72	1, 0	59.37, 26.72	111
58.37, 26.72	0, 0.1	58.37, 26.82	5.85
58.37, 26.72	0, 0.5	58.37, 27.22	29.4
58.37, 26.72	0, 1	58.37, 27.72	58

Unlike Timezone, the substitution of Geolocation API coordinates is more difficult to perform. It is not possible to easily change coordinates via system or browser settings. However, we need to consider that nowadays, there are applications and browser addons that allow changing Geolocation API coordinates.

Besides, there is an alternative scenario for Geolocation API – a user can just forbid access to the location by setting Geolocation API to “Block”. Usually, blocked Geolocation API should be considered as suspicious from the website perspective. Figure 12 shows an example of the location permission request performed by the Geolocation API.

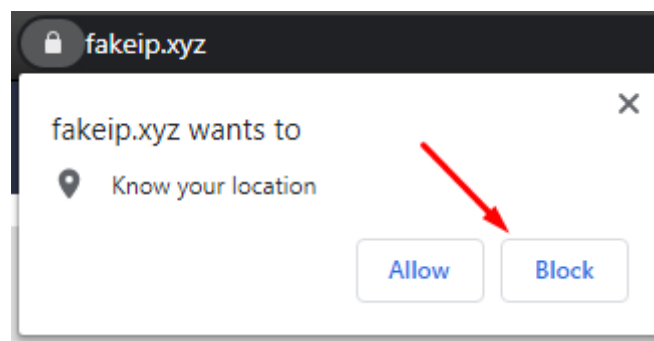


Figure 12 Geolocation API permissions request (author’s picture)

3.2.4 WebRTC implementation

Figure 14 (see Appendix 1) shows the code snippet from the WebRTCService, which performs a WebRTC connection and gets the user's public IP address. To establish a WebRTC connection and get ICE candidates (public IP address, hashed private IP address, etc.) following steps need to be performed:

1. Getting RTCPeerConnection from the global object window.
2. Callback function creation that will pass ICE candidates to our extraction function as soon as we will get any.
3. Empty data channel creation.
4. Creation of offer with SDP data.

After receiving the public IP address from WebRTC, it can be compared to public IP obtained through HTTP headers. If a mismatch in IP addresses occurs, then most probably - this particular user uses IP substitution methods.

The WebRTC method is the most accurate since it shows a mismatch of public IP values.

3.2.5 TCP/IP fingerprinting (Nmap)

The TCP/IP fingerprinting itself is not implemented in the project due to legal and ethical reasons. OS fingerprinting by Nmap implies port scanning of a target machine, which may not be allowed in different countries' jurisdictions.

Figure 13 shows the result of Nmap execution on Linux Ubuntu 18 server, which is owned by the author. Please notice that the author warned system administrators of this machine before performing the Nmap scan. Parameter -O stands for enabling OS detection.

```

$ sudo nmap -O 45.9.190.101
Starting Nmap 7.80 ( https://nmap.org ) at 2020-10-15 16:08 EEST
Nmap scan report for 45.9.190.101
Host is up (0.075s latency).
Not shown: 981 closed ports
PORT      STATE SERVICE
1/tcp    open  tcpmux
22/tcp   open  ssh
25/tcp   filtered smtp
79/tcp   open  finger
80/tcp   open  http
...
Device type: general purpose|specialized
Running (JUST GUESSING): Linux 3.X|4.X (86%), Oracle VM Server 3.X (85%)
OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
cpe:/o:oracle:vm_server:3.4.2 cpe:/o:linux:linux_kernel:4.1
Aggressive OS guesses: Linux 3.10 - 4.11 (86%), Oracle VM Server 3.4.2 (Linux
4.1) (85%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 22 hops

OS detection performed. Please report any incorrect results at
https://nmap.org/submit/.
Nmap done: 1 IP address (1 host up) scanned in 11.13 seconds

```

Figure 13 Result of Nmap OS scan (author's code snippet)



As we can see, Nmap identified the OS as Linux with a 86% probability. In addition to OS detection, which actually can be done by sending random packets to just one TCP port, Nmap performed a scan on all ports of the target machine. However, only the first 5 ports are displayed in this code snippet.

4 Experiments with fake IP detection tool

This part will show the results of proxy and VPN experiments performed on the <http://fakeip.xyz> website. All the pictures in the tables are screenshots from the author's website and show the results of the corresponding methods. All the results were made from the same IP address in Estonia. Proxy server from Belgium and Japanese VPN server were used for IP substitution in experiments.

4.1 Test with proxy



Table 3 IP mismatch results with proxy

Method	Result	Mismatch?
HTTP headers	<p>IP address</p> <p>Public IP: 45.148.102.16</p> <p>Country: Belgium </p>	-
WebRTC	<p>WebRTC</p> <p>Public IP: 88.196.24.172</p> <p>Country: Estonia </p>	Present, WebRTC public IP and public IP from HTTP headers do not match.
TimeZone	<p>Local timezone</p> <p>Europe/Brussels UTC+1</p> <p>System timezone</p> <p>Europe/Tallinn UTC+2</p>	Present, Timezone names, and offsets are different.
Geolocation	<p>Location by IP address</p> <p>Lat: 50.85</p> <p>Lng: 4.34</p> <p>Location by Geolocation API</p> <p>Latitude: 59.45</p> <p>Longitude: 24.88</p>	Present, the difference of coordinates is about 9 degrees for latitude and 20 for longitude.

According to the test results, all three detection methods pointed to mismatches in parameters. As a result, we can infer that a proxy used for IP substitution by a user - can be easily detected by the website, using the methods considered in this thesis.

4.2 Test with VPN

Table 4 IP mismatch results with VPN

Method	Result	Mismatch?
HTTP headers	IP address Public IP: 5.181.235.188 Country: Japan 	-
WebRTC	WebRTC Public IP: 5.181.235.188 Country: Japan 	Absent, WebRTC public IP and public IP from HTTP headers match.
TimeZone	Local timezone Asia/Tokyo UTC+9 System timezone Europe/Tallinn UTC+2	Present, Timezone names, and offsets are different.
Geolocation	Location by IP address Lat: 35.62 Lng: 139.75 Location by Geolocation API Latitude: 35.61 Longitude: 139.73	Absent, the difference of coordinates is within tolerable limits (geolocation inaccuracy applies).

In contrast to a proxy, IP substitution with VPN showed different results. Firstly, WebRTC public IP is equal to the public IP from HTTP headers. Also, coordinates mismatch is not detected since the difference between latitude and longitude is not significant, and this slight difference might be because of Geolocation API inaccuracy. However, a Timezone mismatch was detected.

4.3 Results clarification and methods comparison

The experiment results with VPN showed that WebRTC and Geolocation mismatches might not be detected. Most probably mismatches are not detected because the VPN server used in the experiment has corresponding leak protection⁶.

Difference and ambiguity of results with proxy and VPN lead to the point that none of the methods can provide 100% accuracy of mismatch determination separately. This leads to the idea that all three methods need to be used and analyzed together.

Table 5 shows a comparison of methods according to results from the experiment.

Table 5 Methods results comparison

Method	Proxy detection	VPN detection	Complexity of counterfeiting
WebRTC	Yes	No, if VPN server has corresponding protection	Moderate, by the aid of third party browser extensions
TimeZone	Yes	Yes	Easy, changeable through OS settings
Geolocation	Yes	No, if VPN server has corresponding protection	Moderate, by the aid of third party browser extensions

⁶ Namely leak of real public IP from WebRTC or leak of real coordinates from Geolocation API.

Summary

In this thesis, the author introduced and described methods that can be used for fake IP detection. These methods are WebRTC leak, JS Timezone, Geolocation API, and TCP/IP fingerprint. Each method, considered in this thesis has its pros and cons and the author tried to bring it on.

One of the main purposes of this work was to create an open-source web application, which demonstrates fake IP detection methods. Application development started from the design part (UI and architecture design), then the main challenge was to investigate and implement each method. As a result, the application was deployed to the production environment and is available by address <http://fakeip.xyz/>. Also, the revision control was done with the Github VSC system, so the source code and the history are publicly available at <https://github.com/f1kus97/fake-ip/>.

During this research, it was found that some methods are more difficult to bypass. Also, some of the methods indicate mismatch more explicitly than other methods. Both metrics show whether these methods can be considered as more trustworthy. The author considers WebRTC as the most reliable method for fake IP detection since it explicitly indicates that public IP addresses do not match. The second method by the level of reliability is Geolocation API because we can detect IP mismatch by the difference of IP location (obtained through HTTP headers) and location from Geolocation API. The last method is Timezone since it can be easily changed in the OS settings.

Notwithstanding the different level of trust in each method, the result of the test with VPN has shown us, that Timezone was the only method which detected the mismatch. Therefore, it should be concluded, that all methods and their results need to be used and analyzed together.

Moreover, we need to understand, that today it is possible to bypass each method in some way and it is just a matter of effort and patience. For example, there are browser addons, which allow to disable or substitute WebRTC and change coordinates of Geolocation API. This fact brings us back to the idea that fake IP detection methods need to be used together in a combination.

References

- [1] Mozilla Developer Network, "X-Forwarded-For," 23 March 2019. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Forwarded-For>. [Accessed November 2020].
- [2] Mozilla Developer Network, "WebRTC API," Mozilla, 1 September 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API. [Accessed October 2020].
- [3] Internet-Computer-Security.com, "What is NAT and how does it work tutorial," 2008. [Online]. Available: <http://www.internet-computer-security.com/Firewall/NAT.html>. [Accessed October 2020].
- [4] Mozilla Developer Network, "Introduction to WebRTC protocols," 1 July 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols.
- [5] S. Jennings and C. Nandakumar, "SDP for the WebRTC," Cisco, 23 February 2013. [Online]. Available: <https://tools.ietf.org/id/draft-nandakumar-rtcweb-sdp-01.html#rfc.section.3>. [Accessed October 2020].
- [6] Mozilla Developer Network, "ICE," Mozilla, 16 June 2020. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Glossary/ICE>. [Accessed October 2020].
- [7] J. Rosenberg, "Interactive Connectivity Establishment (ICE)," April 2010. [Online]. Available: <https://tools.ietf.org/html/rfc5245#page-7>. [Accessed October 2020].
- [8] J. K. Singh and C. Wong, "333752 - Google Chrome WebRTC IP Address Leakage," 13 January 2014. [Online]. Available: <https://bugs.chromium.org/p/chromium/issues/detail?id=333752>. [Accessed October 2020].
- [9] Y. Fablet, J. de Borst, J. Uberti and Q. Wang, "Using Multicast DNS to protect privacy when exposing ICE candidates," 16 October 2019. [Online]. Available: <https://tools.ietf.org/html/draft-ietf-rtcweb-mdns-ice-candidates-04>. [Accessed October 2020].
- [10] Mozilla Developer Network, "Geolocation API," 24 July 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API. [Accessed October 2020].
- [11] A. Popescu, "Geolocation API Specification 2nd Edition," 8 November 2016. [Online]. Available: <https://www.w3.org/TR/geolocation-API/>. [Accessed October 2020].
- [12] E. Hjelmvik, "Passive OS Fingerprinting," Netresec, 5 November 2011. [Online]. Available: <https://www.netresec.com/index.ashx?page=Blog&month=2011-11&post=Passive-OS-Fingerprinting>. [Accessed October 2020].
- [13] G. Lyon, "TCP/IP Fingerprinting Methods Supported by Nmap," 2017. [Online]. Available: <https://nmap.org/book/osdetect-methods.html>. [Accessed October 2020].
- [14] G. Lyon, "Fingerprinting Methods Avoided by Nmap," 2017. [Online]. Available: <https://nmap.org/book/osdetect-other-methods.html>. [Accessed October 2020].

Appendix 1 – Public IP from WebRTC

```
private performPeerConnection() {
  let RTCPeerConnection;

  if (window.document.body) {
    RTCPeerConnection = (window as unknown as Window).RTCPeerConnection ||
      (window as unknown as Window).mozRTCPeerConnection ||
      (window as unknown as Window).webkitRTCPeerConnection;
  }

  if (!RTCPeerConnection) {
    const iframe = document.createElement('iframe');
    iframe.setAttribute('id', 'iframe');
    iframe.sandbox.value = 'allow-same-origin';
    iframe.style.display = 'none';

    const win = iframe.contentWindow as Window;
    RTCPeerConnection = win.RTCPeerConnection || win.mozRTCPeerConnection
    || win.webkitRTCPeerConnection;
  }

  const peerConnection = new RTCPeerConnection(this.servers);
  peerConnection.onicecandidate = (ice: RTCPeerConnectionIceEvent) => {
    if (ice.candidate) {
      this.extractPublicIP(peerConnection);
    }
  };

  peerConnection.createDataChannel('');
  peerConnection.createOffer().then(offer => {
    peerConnection.setLocalDescription(new RTCSessionDescription(offer));
  });
}

/* Function to extract exactly the IP address from incoming ICE candidates */
private extractPublicIP(peerConnection: RTCPeerConnection) {
  console.log(peerConnection.localDescription.sdp);
  const lines = peerConnection.localDescription.sdp.split('\n');
  lines.forEach(line => {
    const exp = new RegExp('(\\d{1,3}\\.){3}(\\d{1,3})');
    if (line.startsWith('a=candidate:') && line.match(exp)) {
      this.publicIP = line.match(exp)[0];
    }
  });
}
```

Figure 14 Public IP obtainment through WebRTC (author's code snippet)

Appendix 2 – FakeIP application sequence diagram

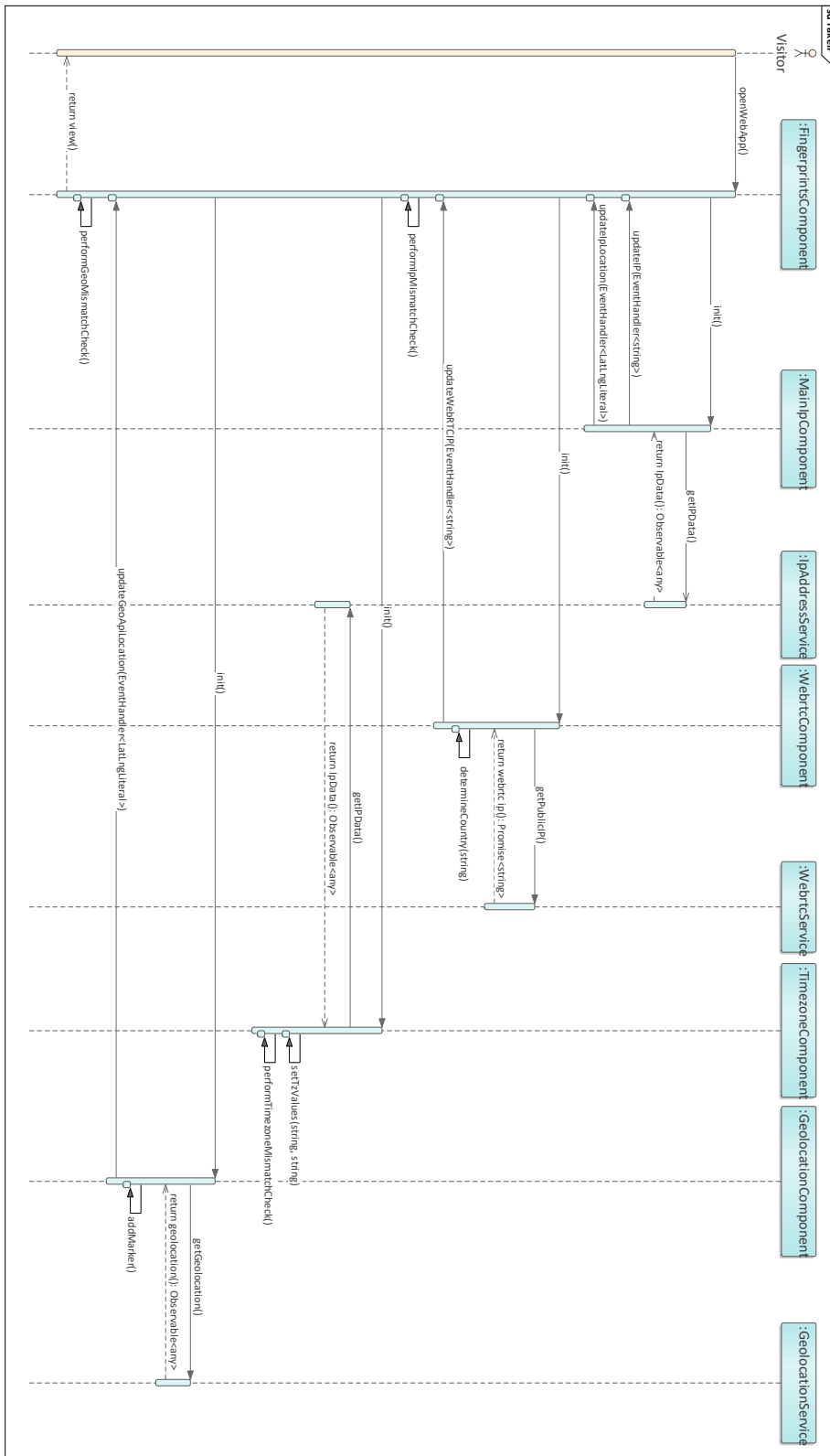


Figure 15 FakeIP sequence diagram (author's diagram)