MASTER'S THESIS COMPUTING SCIENCE

Misuse of WebRTC on the Web: A Measurement Study

RICO TE WECHEL
S4773039

June 10, 2026

First supervisor/assessor:
dr. Gunes Acar

Second assessor:
dr. Erik Poll

Radboud University

Abstract

Web Real-Time Communication (WebRTC) requires browsers to explicitly bypass Network Address Translation (NAT) and firewalls to directly establish peer-to-peer connections, creating privacy risks that allow tracking and fingerprinting actors to bypass safeguards. Motivated by a recent disclosure of WebRTC being abused to leak first-party cookies, this research investigates the underlying mechanisms and scale of these vulnerabilities and exposes novel tracking methods through data exfiltration.

We deployed an automated web crawler across 7,884 popular websites confirmed to utilize WebRTC and performed live network traffic analysis to chart the behavior and identities of third-party script initiators on those websites. We analyzed the WebRTC API calls invoked during our crawls to search for tracking-related activity or potential exfiltration of data. In addition, we identified the most prevalent WebRTC script initiators within our crawl data and manually browsed websites containing those scripts for network traffic analysis. We then analyzed network packets associated with WebRTC traffic for any privacy-sensitive identifiers or signs of data exfiltration. Our findings show that nine of the top ten WebRTC script initiators are focused entirely on user tracking rather than providing a means of online communication. In particular, we found evidence that WebRTC is systematically being exploited for consistent and structural leakage and exfiltration of privacy-sensitive data on a large scale.

Contents

Chapter 1

Introduction

The modern web relies on various measures to protect user privacy, but advanced frameworks like Web Real-Time Communication (WebRTC) challenge these measures. WebRTC is an open-source framework that allows users to initiate direct peer-to-peer connections for audio, video, and other data communication. To successfully accomplish this, it has to bypass obstacles like Network Address Translation (NAT) and firewalls. This fundamental principle of WebRTC creates a significant privacy risk, as the protocols that are used to establish a peer-to-peer connection bypass regular privacy measures by design. A good example is the STUN protocol, which WebRTC utilizes to discover a user's public IP address [20]. STUN operates completely outside of standard HTTP traffic, which allows it to bypass browser-level protections like HTTP proxies or even VPNs [7]. While protocols like STUN are designed with good intentions, they can also be abused by third parties. This threat has been emphasized by various researchers and even regular users for years [32, 1, 12, 24].

A recent disclosure shows how malicious actors can abuse WebRTC to covertly exfiltrate[1] first-party web cookies to native background applications on Android devices, circumventing traditional HTTP safeguards [9]. This issue, strengthened by the caveats of WebRTC described above, warrants the desire for more research on potential abuse. While there is quite some research available on WebRTC's flaws that allow for IP leakage and user tracking and fingerprinting [32, 1, 12], no literature exists that conducts a large-scale observational study explicitly investigating the active deployment of WebRTC for data exfiltration and tracking on the web.

In this research we crawl the most popular websites containing WebRTC traffic and try to investigate what the traffic is intended for while identifying the third-party scripts initiating this traffic. In doing so, we make the following key contributions:

[1] `https://www.ibm.com/think/topics/data-exfiltration`

- We measure the current WebRTC use and misuse on the web.
- We expose a novel data exfiltration method involving WebRTC, which we show is actively abused on a large scale by the most prevalent WebRTC traffic initiators on the web.
- In particular, we show that both actors and data involved in exfiltration hint toward tracking purposes.

We start by giving a background (chapter 2), in which we cover the WebRTC basics, including its underlying mechanisms and protocols, and zoom in on the earlier mentioned disclosure [9] as the main motivation for our research. In chapter 3 we address research papers that have been published regarding the privacy concerns of WebRTC and use them to define our knowledge gap. Following up in chapter 4 we formalize our methodology and scope our research. Chapter 5 presents the first findings of the proposed methodology, introduces a follow-up data collection and analysis method based on the outcomes, and presents the findings of the follow-up research as well. Chapters 6 and 7 provide a summary of our work and findings, the most essential conclusions and implications of our research, and several limitations of our research, raising motivations for future work.

Chapter 2

Background

We begin by explaining the purpose and fundamentals of WebRTC, including essential frameworks, standards, protocols, and API calls that are used by WebRTC.

2.1 WebRTC

Web Real-Time Communication (WebRTC) is a set of protocols [15] that allows users to set up a direct peer-to-peer connection with another user to exchange audio, video, and other multimedia or generic data [15]. Traditionally, communications like video calls relied on a client-server connection where all data is first sent to an intermediate host server, which forwards all data to the other user. WebRTC introduces a direct peer-to-peer (P2P) connection to improve stability and speed and to minimize bandwidth costs. However, establishing a direct connection between two peers on the public internet comes with challenges and complications. For instance, firewalls and NAT suddenly become obstacles to work around. To overcome these, WebRTC follows a standardized negotiation process involving the Session Description Protocol (SDP), Interactive Connectivity Establishment (ICE), Session Traversal Utilities for NAT (STUN), and Traversal Using Relays around NAT (TURN). This negotiation results in a so-called candidate pair, which resembles a pair of endpoints that forms the best P2P connection.

2.1.1 Session Description Protocol (SDP)

SDP is not really a protocol but a data format in which two peers describe the metadata of what data will be exchanged in the P2P connection. SDP messages take care of exchanging between the peers what type of data will be exchanged, in what format that data is structured, and what the endpoints of the two peers are that the data should be sent to in the P2P connection [16]. These SDP messages are usually exchanged using an intermediate signaling server outside the scope of what WebRTC specifies and are exchanged before the P2P connection is set up. The developer supplies the signaling server and is free to use whatever technology

to set up this signaling server. The WebSockets technology [13] is often used for this [30].

2.1.2 Interactive Connectivity Establishment (ICE)

ICE is designed to set up and manage the P2P connection between the two peers [14]. To establish a P2P connection, both peers must find their own reachable IP address and a network path that can reliably, and preferably quickly, guide messages to that IP address. For both peers, ICE collects all possible addresses at which a user can be reached. These are called ICE candidates, which are exchanged with the other peer through the same signaling server described above. In fact, ICE candidates are even part of SDP messages themselves in a later stage, when ICE candidates have been found.

When all candidates from both sides are exchanged between the two peers, ICE tries to make a candidate pair to form the best connection. It tests which candidate pairs can communicate with each other and picks the pair with the most stable and fast connection to use as the address pair for the P2P connection. ICE candidates come in three flavors: host candidates, server or peer reflexive candidates, and relay candidates. Host candidates are candidates resembling a local IP, server or peer reflexive candidates resemble public IP addresses, and relay candidates resemble public IP addresses of a public TURN server, in case no direct path between the two peers could be found (see TURN below). Ideally, ICE tries to form pairs of symmetric types (for instance, two host candidates if both peers are on the same LAN), but asymmetric pairs are also possible.

2.1.3 Session Traversal Utilities for NAT (STUN)

The STUN protocol is used by the ICE framework to perform network address discovery and connectivity checks. It allows users behind NAT to discover their own public IP address and port as seen by the public internet through a STUN server and enables peers to make connectivity checks and pinpoint any router settings or NAT table that restrict a P2P connection [20, 18]. This allows ICE to select the most efficient candidate pair. STUN servers are relatively simple mirroring servers. The client sends a “Binding Request”, and the STUN server replies with a “Binding Success Response”, carrying the client’s publicly visible IP and port number. These binding requests can also be sent from one peer to another for the connectivity checks. Depending on various replies from the STUN servers and remote peers, ICE can determine if NAT and router settings are illegible for a P2P connection.

2.1.4 Traversal Using Relays around NAT (TURN)

It is possible that either or both of the peers trying to set up a P2P connection are behind NAT or firewall rules that prevent direct communication with the peer. In other words, the connectivity check described in the STUN protocol has failed. In

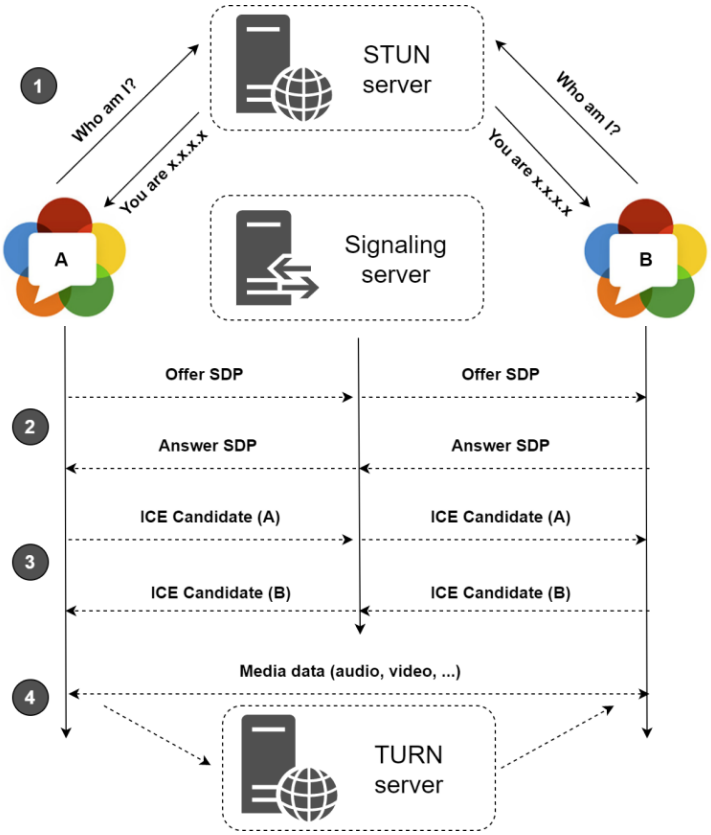


Figure 2.1: A simplified WebRTC connection lifecycle. Peers first discover their public IP addresses via STUN (1) and exchange SDP media capabilities through a signaling server (2), enabling the ICE framework to test and establish a direct connection path (3) or ultimately fall back to a TURN relay server for media routing if strict NAT or firewalls block direct communication (4).
Source: `https://docs.mirotalk.com/coturn/stun-turn/`

this case WebRTC uses the TURN protocol as a fallback mechanism [21]. TURN servers are essentially authenticated relay servers that forward any traffic coming through them to the peer one is trying to connect to. Before a peer can use a TURN server as a relay, it must send an "Allocate Request". The server will reply with an "Allocate Success Response", together with the relay IP that is allocated to the user on the TURN server. This is, of course, not a real P2P connection anymore and introduces overhead; hence, it is used as a last fallback option.

2.1.5 Setting up a peer-to-peer WebRTC connection

Establishing the WebRTC P2P connection comes down to combining all previously described protocols and standards in one offer & answer model [22]. Figure 2.1 shows a visualization of this model. This is precisely what WebRTC provides

using dedicated API calls. During establishment, a chosen signaling server assists in connecting the two peers. SDP messages are continuously being sent back and forth between the two peers through the signaling server until all agreements about data transfer and connectivity are made. When a WebRTC connection is initiated, the following steps are followed:

1. Peer A initializes an `RTCPeerConnection` object and calls `createDataChannel`. This initializes ICE, but no further network traffic is generated yet.

2. Peer A decides what kind of connection (audio, video, or data) they want to set up and configures this in the created data channel. They then call `createOffer`, which places these media capabilities in an SDP message called the initial SDP offer.

3. By calling `setLocalDescription`, peer A then sets the media capabilities for its own browser and triggers the first network activity: ICE starts gathering host, server/peer reflexive, and possibly relay candidates (in that priority) using binding requests to STUN and possibly allocate requests to TURN servers. This is an ongoing parallel process.

4. While the ICE gathering stage has started, peer A continues by sending the initial SDP offer, which still contains only the media information, to the signaling server, which forwards it to peer B. As ICE candidates are being gathered, they are trickled one by one in subsequent SDP messages to peer B via the signaling server.

5. Peer B receives the initial SDP offer and calls `setRemoteDescription`. This informs peer B's browser of peer A's media capabilities.

6. Peer B calls `createAnswer`, which creates a response to peer A's offer. The SDP block generated should be a set of agreements that is compatible with both peer A's offer and peer B's capabilities. Usually, peer A's offer contains multiple options that can be chosen from to create the definitive answer.

7. Peer B then calls `setLocalDescription` to set the chosen media capabilities to the browser, send back the created SDP answer, and start their own ICE gathering phase.

8. Peer A receives the answer and calls `setRemoteDescription` to also update their browser with peer B's preferences.

9. Meanwhile, both peers should possess a list of each other's ICE candidates. This starts the connectivity check phase, in which both peers send STUN binding requests directly to each other's ICE candidate addresses (or indirectly via the TURN server) to find the best networking path. The result is

one candidate pair (containing one candidate from A and one from B) that forms the best P2P connection, or a connection via a TURN server.

10. Once this is achieved, the signaling server and STUN servers are not needed anymore, and the peers can start communication data towards each other directly.

2.2 Motivation

Technologies like WebRTC are revolutionary to developers regarding connectivity on the web because of their P2P architecture, low latency, compatibility, and easy integration. However, they require deep access to underlying networking systems to function. Through protocols like ICE and STUN, browsers are allowed to directly communicate with local network interfaces, bypassing NAT and working around firewalls. This creates a risk: The same protocols that allow WebRTC to function properly might be powerful tools for malicious actors to invade privacy or security.

The direct motivation for our research comes from a recent disclosure by researchers from the Radboud University, KU Leuven, and IMDEA Networks Institute (Girish et al., 2025), which demonstrated an up-until-then never-before-seen tracking method using WebRTC [9]. Tech giant Meta abused the STUN protocol by sending binding requests to the localhost address on mobile Android devices. In the requests, first-party cookies were placed that were collected from browser sessions on the mobile device. If a user visited a website with the Meta Pixel[1] script active, the script would take the `_fbp` cookie (which is supposedly used for "providing advertising and site analytics services") from the website and put it in a STUN binding request using a technique called SDP munging [10], which comes down to illegally modifying SDP messages. The binding request would be sent to the device's localhost address while the Facebook app, if installed on the device, was listening to localhost in the background and thus exposing the cookie to Meta. The entire process circumvents HTTP safeguards that would normally prevent leakage of first-party cookies. The disclosure has, at the time of writing this research, been formalized in a research paper that is yet to be presented [33].

The method described above has remained covert for roughly six months, and with the `_fbp` being active on about a fourth of the top one million websites, it formed a user data exfiltration operation on a massive scale. In figure 2.2 we present a graph showing the percentage of page loads in Chrome browsers that used the `setLocalDescription` method at least once between 2024 and 2026. The spike visible corresponds with the time this tracking method was active and visualizes the scale of the exploit.

The exploit in the Local Mess disclosure exposes a blind spot in current-day web security and privacy standards: malicious actors are no longer relying solely

[1] `https://www.facebook.com/business/tools/meta-pixel`

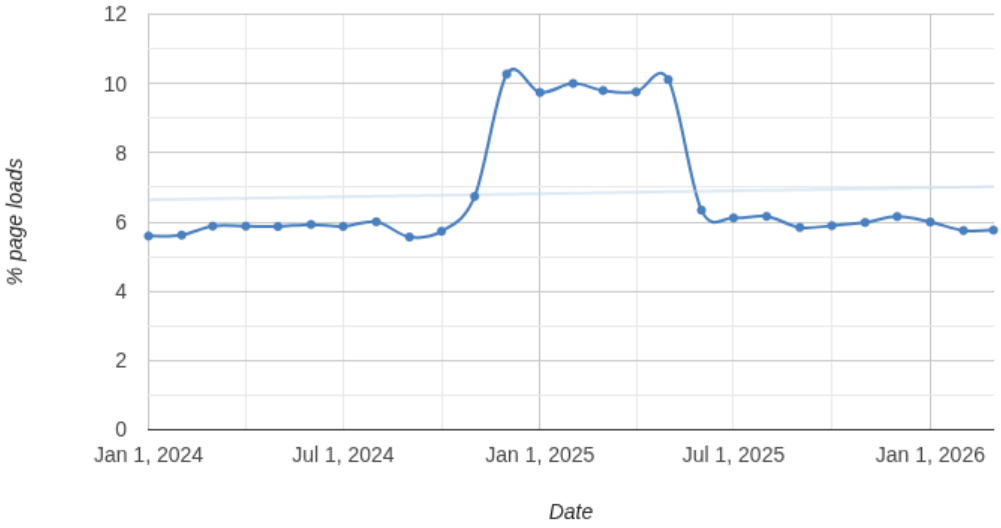


Figure 2.2: A graph showing the use of `setLocalDescription` on Chrome browsers between 2024 and 2026.
Source: `https://chromestatus.com/metrics/feature/timeline/popularity/3452`

on traditional cross-site tracking methods but are instead hiding within legitimate and standardized web APIs. Worries and signs about WebRTC posing a privacy risk by circumventing standard safeguards have existed for a long time [24][32]. The exfiltration method and the activity spike of the `setLocalDescription` method lead us to question what other WebRTC exfiltration methods and leaks are currently active and undetected on the web. This thesis therefore investigates the prevalence and methods of WebRTC-related privacy risks on the web.

Chapter 3

Related work

In this chapter we address related academic literature in the field of web tracking, fingerprinting, data exfiltration, and WebRTC in general. Our goal is to give a clear picture of the current research landscape regarding these fields and use it to define a research gap. This research gap helps us scope our research.

3.1 Current state of web fingerprinting

To get an insight into the current universe of tracking and fingerprinting, research by Iqbal et al. provides context on the evolution and current-day state of tracking and fingerprinting activities on the web [23]. By deploying a tool called FP-INSPECTOR, a fingerprinting detector based on machine learning, across the Alexa top 100K websites[1], the researchers found that browser fingerprinting has grown in popularity and is very prevalent across the web. Fingerprinting scripts are currently active on over a quarter of the Alexa top 10K sites and more than 10% of the top 100K sites. The study also shows how tracking methods are constantly evolving to evade detection. The tool used in the study detected several previously unknown fingerprinting methods that abuse relatively new APIs. This demonstrates that actors are continuously finding new ways to exploit web standards for privacy-invading aims. We see this research as an additional motivation to investigate tracking and fingerprinting activities on the web via WebRTC. WebRTC is briefly mentioned in the study as one of the APIs scanned to identify scripts as "fingerprinting scripts", but no in-depth research was done on how WebRTC specifically plays a role.

[1] `https://www.expireddomains.net/alexa-top-websites/`

3.2 Existing research on WebRTC and its privacy concerns

Work published by Reiter & Marsalek of the Graz University of Technology [32] provides a good overview of multiple fundamental privacy risks of WebRTC. It describes the core of most of the already performed research papers on WebRTC worries. The research explores how WebRTC's underlying peer-to-peer technology inherently exposes new elements to the set of assets that are explicitly protected by a browser. These assets mostly include public and private IP addresses, which WebRTC can share with any remote client contacted by a JavaScript script. Because the protocol reveals local IP addresses in the signaling phase of ICE, the study indicates that attackers can abuse WebRTC to reliably scan someone's local network, reveal IP addresses of users behind a VPN, and even fingerprint cross-browser by combining different exposed private and public IP addresses. The study raises concerns and worries about the underlying architecture of WebRTC and suggests more control for the user on what information is shared by WebRTC.

There is various in-depth research to be found building upon WebRTC's fundamental need to reveal public and private IP addresses to establish peer-to-peer connections, which bypasses regular privacy mechanisms. Al-Fannah of Royal Holloway, University of London [1] explicitly researches the privacy implications of WebRTC allowing visited websites to leak private IP addresses via the SDP protocol and public IP addresses via the STUN protocol, of which the latter can completely negate anonymity provided by VPNs, which enables better user tracking. The research compares different browsers and VPN configurations. Research by Hazhirpasand & Ghafari, University of Bern [12], further extends the implications of local IP leakage by introducing a browser- and network-independent JavaScript scanner that can scan private networks and discover internal network ports, identify active nodes in the network, and map the internal infrastructure of the network, even if behind a firewall. This can all be done by having the victim visit a webpage, implying this can also be used for device fingerprinting very well. These papers together show that WebRTC's IP leakage forms a privacy issue that can compromise the identity of a victim and expose their entire local network to automated scanners.

Most of the current research structurally shows that WebRTC can be used on a large scale to discover private or hidden IP addresses, identify users, break through anonymity, reveal information about local networks, and, in turn, track users across the web using this information, allowed by the design choices made to enable peer-to-peer connection establishment.

3.3 Large-scale research on data exfiltration

This thesis does not only focus on tracking and fingerprinting using WebRTC but primarily on an explicit strategy to achieve those goals: data exfiltration. Bashir et

al. [3] studied similar privacy risks in WebSocket, a stateful, continuous communication protocol allowing low-latency communication between client and server [13]. In their study, a large-scale automated crawl was performed on 100,000 of the most popular websites, intercepting any WebSocket traffic and inspecting what kind of data is shared with what (third-party) actors. The study shows how WebSocket is abused on a large scale to bypass ad-blockers and explicitly exfiltrate data. The research mentions that 65-72% of all WebSocket connections are related to tracking or advertising. The researchers identified many advertising and analytics companies behind scripts that initiated WebSocket connections and were able to show that among the data exfiltrated are cookies, IP addresses, unique identifiers, and even entire Document Object Models, all used for user tracking and browser fingerprinting. Similar to the Local Mess disclosure that motivates our research [9], WebSocket was found to be able to circumvent traditional HTTP safeguards, allowing data exfiltration. This study demonstrates a methodology that could be very effective in finding similar results in the context of WebRTC.

3.4 Research gap

Although the focus of the Local Mess disclosure [9] was the abuse of the loopback address on mobile devices rather than data exfiltration via WebRTC, the disclosure shows that WebRTC is in fact used for it systematically. The disclosure proves active data exfiltration via WebRTC, in this case allowing tracking and possibly fingerprinting by first-party cookies and other identifiers. This kind of development is precisely what the research paper described in 3.1 warns us about [23].

While existing research extensively describes privacy risks of WebRTC relating to IP leakage, the ability to bypass VPNs, its use in network scanning, and, in turn, its role in browser fingerprinting and user tracking [32][1][12], we were unable to find any research papers that explicitly investigate the deployment of WebRTC for active exfiltration of data, nor research that identifies actors behind any WebRTC abuse for tracking- or fingerprinting-related reasons. As far as we know, numerous research papers have been published on *how* WebRTC can be abused for tracking or fingerprinting purposes, but there is currently no study that actually *measures* the current state and scale of WebRTC abuse on the web. Although this exact type of large-scale research has been conducted for the abuse of WebSockets by advertising and analytics companies [3], the same large-scale crawl and analysis has not been conducted yet within the scope of WebRTC. This defines the research gap that our research strives to bridge.

Chapter 4

Methodology

Our goal is to characterize WebRTC usage on the web. We are mostly interested in popular websites in terms of visit counts that are confirmed to use WebRTC. This can be either WebRTC traffic originating from the website itself or from a third-party script loaded on the website. In this chapter we explain how we decide on the relevant pool of websites, how we perform our data collection on them, and what kind of data we aim to collect.

4.1 Source of research data

We start with selecting our data source. To maintain significance, we need often-visited websites. A dataset often used by researchers when it comes to web traffic analysis is the HTTP Archive [2]. It contains millions of structured, JSON-formatted logs with network requests, responses, and browser data crawled monthly.

A useful tool that we can use to access this dataset while sorting on popularity and filtering on WebRTC usage is Google BigQuery [25]. This is a platform on which we can perform queries across a broad scale of large, publicly available datasets. We follow a popular guide on how to set up BigQuery for access to the HTTP archive [11].

For a measure of website popularity, we utilize the Chrome UX Report (CrUX), which ranks the most visited websites using the Chrome browser [6]. The websites are ranked on popularity in buckets, a bucket with rank 10,000 being a set containing all top 10,000 most-visited websites, ordered randomly. This data is all available on BigQuery as well [26].

In listing 4.1 we placed the SQL query that we ran in BigQuery to obtain a list of popular websites that are confirmed to use WebRTC. This query filters for three essential WebRTC calls: `SetLocalDescription`, `SetRemoteDescription`, and `CreateOffer`. These calls are needed to set up a connection, so if a website contains them, we are confident a connection is possible. We ensure we include `is_root_page` in the query to filter out homepages only. This way we scope our research to WebRTC connections being made directly on the user's arrival. By

```
SELECT
  rank,
  page,
  client,
  features.feature,
  features.type,
  features.id
FROM
  `httparchive.crawl.pages`,
  UNNEST (features) AS features
WHERE
  date = '2025-06-01' AND
  rank < 1000000 AND
  is_root_page AND
  features.feature IN (
    "RTCPeerConnectionSetLocalDescription",
    "RTCPeerConnectionSetRemoteDescription",
    "RTCPeerConnectionCreateOffer"
  )
ORDER BY
  rank asc;
```

Listing 4.1: The SQL query we used to obtain a list of websites that use common WebRTC features/JavaScript API methods.

filtering for rank $<$1,000,000, the results are scoped to websites that are in popular buckets, effectively buckets with rank $\leq$500,000, since there's no other rank bin in between.

The result is a JSON file containing 29,785 entries. Listing 4.2 contains an example of one of the entries in the JSON file. These entries consist of many duplicates, as our query lists websites that are popular on a mobile client as well as on a desktop client twice and lists the website once for every WebRTC call present on the website. We thus want to filter out these duplicates so we are left with a list of distinct eTLD+1[1] domain names. After doing this, we obtained a list of 7,884 distinct domain names. The most popular domain names are in bucket 1,000, and the least popular domain names are in bucket 500,000. This essentially means that in the top 500K most popular websites, about 1.58% of them are confirmed to use WebRTC.

4.2 Data collection

To collect network traffic data from our curated list of domain names, we use an automated web crawler. We are especially interested in intercepting WebRTC network requests and responses and WebRTC API function calls, arguments, return

[1] https://developer.mozilla.org/en-US/docs/Glossary/Registrable_domain

```
{
  "rank": "1000",
  "page": "https://www.cnn.com/",
  "client": "mobile",
  "feature": "RTCPeerConnectionCreateOffer",
  "type": "default",
  "id": "3453"
}
```

Listing 4.2: An example JSON entry of the results of SQL query.

values, and payloads in JavaScript. This way, we can investigate which scripts trigger WebRTC traffic and what kind of data is sent to what party, if any. DuckDuckGo's open-source Tracker Radar Collector, based on the Puppeteer library [4], is a suitable candidate for this, as it allows the capturing of many different types of data during web crawls. It contains so-called 'collectors' for different data types that can be turned on or off. Examples are API calls, HTTP traffic, cookies, and cookie walls (CMPs). New collectors can easily be defined and added to the crawler.

Moti et al. extended this crawler with a fingerprint collector, a link collector, a video collector, and an ad collector for their research on tracking and malicious advertising on children's websites [27]. Especially the fingerprint collector is a very useful addition for our research to the Tracker Radar Collector, as it allows interception of various Web API function calls and property accesses that are often used in fingerprinting and tracking methods. The researchers from the Radboud University focused on various APIs, WebRTC included. This modified crawler will thus form a solid base for our research. We took the Targeted and Troublesome crawler and made our own modifications to it to fit our needs:

- Removed all non-WebRTC intercepts, like Canvas and AudioContext fingerprinting intercepts, to focus solely on WebRTC traffic
- Added some WebRTC function calls to the intercept list to broaden our search within the WebRTC scope
- Modified the `postLoad()` function of the CMP collector so that the crawler waits 2,5 seconds after page load, giving cookie walls time to load properly
- Modified the user agent string to show Chrome version 132.0.0.0 on both mobile and desktop crawls and show Android version 14 on mobile crawls
- Applied bug-fixes to prevent crashes that we encountered while crawling

Table 4.1 shows the WebRTC-related API methods and properties instrumented in our modified crawler. In the second column, the reason for our interest in the intercept is noted. Most often, the reason for including the intercept is that the

method can be used to send arbitrary data to a STUN or TURN server or to a remote peer.

Table 4.1: The WebRTC function calls and property accesses that we added to the crawler

Function call or property access	**Reason for interest**
`RTCPeerConnection.createDataChannel`	Parameters potentially allow crafted strings
`RTCPeerConnection.onicecandidate`	Indicates a new `candidate` object is found, which might contain crafted strings
`RTCPeerConnection.addIceCandidate`	Argument is a `candidate` object that might contain crafted strings
`RTCPeerConnection.setLocalDescription`	Argument is a `sessionDescription` object that might contain crafted strings
`RTCPeerConnection.createOffer`	Return value is supposed to be passed to `setLocalDescription`, could give more insight into data flow
`RTCPeerConnection.setRemoteDescription`	Similar to `setLocalDescription`
`RTCPeerConnection.createAnswer`	Similar to `setLocalDescription`
`RTCPeerConnection.setConfiguration`	Argument contains a configuration object in which server endpoints can be included
`RTCDataChannel.send`	Sends actual data over the peer-to-peer connection. Could be anything
`RTCRtpSender.setParameters`	Sets parameters that could contain crafted strings

In our crawl, we enable the fingerprint collector, the CMP (cookie wall) collector, and the screenshot collector. The fingerprint collector tries to intercept the calls in table 4.1, the CMP collector will try to detect and handle any cookie walls that appear, and the screenshot collector takes a screenshot of the page when the page is fully loaded. For debugging purposes, the CMP collector takes a screenshot moments before handling the cookie wall and moments after. The screenshot collector does the same as a very last thing before the crawled page is closed.

We run the command with the following flag: `--autoconsent-action "optIn"`. The crawler's CMPCollector class utilizes DuckDuckGo's Autoconsent library [5]. This library defines rules for detecting and handling cookie walls and contains succession self-tests for cookie walls of known Consent Management Platforms. The CMP collector also contains regular expression patterns for detection of cookie walls. By including this flag, the CMP collector will be tasked to try and recognize cookie walls and implement the policy of always accepting all cook-

ies. We do this because WebRTC traffic might be triggered by third-party scripts that will not load if the cookie wall is not accepted.

Furthermore, we add two lines to a configuration file that will be considered while the crawler runs: `"emulateMobile": true` and `"extraExecution TimeMs": 10000`. The first line ensures that the crawler will try to emulate a mobile device while visiting the website. Important to note is that for this research, we only stuck to mobile website visits. We chose to start with this parameter (motivated by the Local Mess disclosure) but never got to extend the research to also crawl and analyze desktop website visits. More on this in chapter 7. The second adds ten seconds of waiting time on the page after the page is loaded. We do this to ensure that potential WebRTC traffic that still has to be initiated by a script will have time to start. Running the crawler in a folder containing a text file with all our 7,884 domain names will essentially trigger the following steps:

1. The crawler reads the next domain name to open from the text file
2. The crawler loads and start running the collectors passed to it
3. The crawler starts a headless Chrome browser and tries to load the website
4. After the page is loaded, the crawler will wait 2.5 seconds for any cookie walls to appear
5. The CMP collector takes a screenshot, tries to detect and handle the cookie wall by accepting the cookies, and takes another screenshot
6. When the cookie wall is handled, the crawler will idle ten seconds on the website with the fingerprint collector listening
7. After ten seconds, a final screenshot is taken, and the headless Chrome browser is closed

The collectors dump the information they harvested in the folder. We executed the crawl in three batches. This happened on the 2nd of July ’25, the 25th of July ’25, and the 7th of August ’25, all from the same IP address (the Netherlands, EU) on a Windows 11 machine, emulating mobile website visits. In the following chapter, we go over all results. In section 5.1 we present the results of the crawls and analyze the information in the dumped data. After analyzing the crawl results, we decided on a follow-up data collection and analysis round on the same websites, but with a different strategy. This will be explained in section 5.2 with more context.

Chapter 5

Results

In this chapter we will go over all results obtained by the automated crawls and perform manual network traffic analysis. We employ these two complementary methods to better characterize WebRTC usage on the web.

5.1 Crawl analysis

As explained in the previous chapter, we ran a query on Google BigQuery to extract all websites in buckets ranked $<$ 1,000,000 that are confirmed to use either the `setLocalDescription()`, `setRemoteDescription()`, or `createOffer()` WebRTC methods. The result is a list of 7,884 domain names. We crawled all of those websites to save intercepted WebRTC calls, including their arguments and return values, in structured JSON files.

5.1.1 Crawler statistics

Earlier we introduced our modified version of the Targeted and Troublesome's fingerprint collector [27] and the WebRTC function calls and property accesses it tries to intercept (table 4.1). For each website that was crawled, if WebRTC traffic was intercepted, a JSON entry called `"fingerprints"` is placed in the JSON file. The entry contains metadata on what calls were intercepted how many times, by what source they were initiated (`"callStats"`), and more insightful data on those calls, including the arguments and return values of the calls (`"savedCalls"`). In listing 5.1 an example is given of a WebRTC fingerprint entry belonging to a website visit to `https://sunfere.com/`.

In total, 7,829 out of 7,884 websites were crawled without run-time error and resulted in a JSON file. Analyzing the data in callStats, we found that WebRTC traffic was intercepted on 4,419 different websites.

Given that we filtered our list of websites on WebRTC usage, it is unexpected that the crawler was only able to intercept WebRTC traffic from 4,419 of the 7,829. For about twenty websites that contained empty `"fingerprints"` entries, we

```
"fingerprints": {
  "callStats": {
    "https://r2cdn.myshopline.com/static/rs/adff/prod/latest/
        bundle_dee.iife.js?dt=20250702": {
      "RTCPeerConnection.createDataChannel": 1,
      "RTCPeerConnection.createOffer": 1,
      "RTCPeerConnection.setLocalDescription": 1
    }
  },
  "savedCalls": [
    {
      "source": "https://r2cdn.myshopline.com/static/rs/adff/prod/
          latest/bundle_dee.iife.js?dt=20250702",
      "description": "RTCPeerConnection.createDataChannel",
      "arguments": {
        "0": ""
      },
      "returnValue": {},
      "accessType": "call"
    },
    {
      "source": "https://r2cdn.myshopline.com/static/rs/adff/prod/
          latest/bundle_dee.iife.js?dt=20250702",
      "description": "RTCPeerConnection.createOffer",
      "arguments": {},
      "returnValue": {},
      "accessType": "call"
    },
    {
      "source": "https://r2cdn.myshopline.com/static/rs/adff/prod/
          latest/bundle_dee.iife.js?dt=20250702",
      "description": "RTCPeerConnection.setLocalDescription",
      "arguments": {
        "0": {
          "type": "offer",
          "sdp": "v=0\r\no=- 6579818913818385299 2 IN IP4
              127.0.0.1\r\ns=-\r\nt=0 0\r\na=group:BUNDLE 0\r\na=
              extmap-allow-mixed\r\na=msid-semantic: WMS\r\nm=
              application 9 UDP/DTLS/SCTP webrtc-datachannel\r\nc=
              IN IP4 0.0.0.0\r\na=ice-ufrag:B5Vj\r\na=ice-pwd:
              wjyYCStn7nyFcaTMrOgF/eIG\r\na=ice-options:trickle\r\
              na=fingerprint:sha-256 EF:7C:B2:E5:70:00:63:B6:FF:3F
              :43:89:67:F4:38:B6:DB:38:C8:9A:E8:CD:92:8A:00:EC:D6:
              EF:3D:66:51:75\r\na=setup:actpass\r\na=mid:0\r\na=
              sctp-port:5000\r\na=max-message-size:262144\r\n"
        }
      },
      "returnValue": {},
      "accessType": "call"
    }
  ]
}
```


Listing 5.1: An example of a fingerprints JSON entry

manually visited the homepage and waited for 30 seconds to see if any WebRTC traffic would be visible in Wireshark or in Chrome's WebRTC diagnostic tool (`chrome://webrtc-internals/`). We did not observe any WebRTC traffic on these sites. This could be for a number of reasons:

- More time on or interaction with the website is needed before the WebRTC traffic actually initializes.
- We used our crawler with the `emulateMobile` flag. It could be that mobile versions of the website do not trigger the WebRTC traffic
- In contrast, it could be that websites are configured to only run on mobile websites, but the website detected that our headless browser is not running on an actual mobile device.
- The Targeted and Troublesome Crawler is currently unmaintained and outdated compared to DuckDuckGo's Tracker Radar Collector, which could cause unwanted behavior. Similarly, there could also be a bug in the automated headless Chrome browser.
- Finally, since the crawls were done in three separate batches in July 2025, and the WebRTC activity data request from Google BigQuery happened on June 1st, some websites might have changed or removed WebRTC-initiating code meanwhile.

In table 5.1 the intercepted traffic is categorized per WebRTC call. The "# Sites" column contains the number of websites that show at least one entry of a specific call, and the "# Calls" column shows the total number of calls over all websites. This provides an insight into how popular each method is across websites and also how often it is used. We can see that it is probably most relevant to focus on the top four methods, as they are significantly more present across websites.

5.1.2 API call content analysis

To investigate what data is shared over WebRTC, we inspected the `"arguments"` and `"returnValue"` fields of the entries in `"savedCalls"`. Below are our findings for the kind of intercepted call. We omitted `RTCDataChannel.send`, `RTCPeerConnection.setConfiguration`, and `RTCRtpSender.setParameters` as the number of found calls is negligible. For the allowed arguments and possible return values, we refer to the documentation page [28].

`createOffer` Almost no arguments were recorded, aside from some legitimate optional arguments. No single captured return value is shown in the JSON files, probably because this call returns an object [28], which is not directly printable.

`createDataChannel` All calls recorded an argument, of which most are empty strings or decimal numbers between 0 and 1, such as "0.44767611973532984".

Table 5.1: Counts on intercepted WebRTC function calls and property accesses

Function call or property	# Sites	# Calls
`RTCPeerConnection.createOffer`	4,413	11,508
`RTCPeerConnection.createDataChannel`	4,130	7,014
`RTCPeerConnection.onicecandidate`	3,783	8,708
`RTCPeerConnection.setLocalDescription`	3,691	7,988
`RTCPeerConnection.setRemoteDescription`	505	1,895
`RTCPeerConnection.createAnswer`	280	1,556
`RTCPeerConnection.addIceCandidate`	259	1,454
`RTCDataChannel.send`	2	109
`RTCPeerConnection.setConfiguration`	1	1
`RTCRtpSender.setParameters`	0	0

This looks like it is coming from JavaScript's `Math.random` method[1]. Other labels we see are "dummyChannel", "fake_data_channel", "test" or other short, arbitrary strings. Return values were not recorded, again possibly because they are not directly printable.

`onicecandidate` Either the argument was missing from the call, or it was an empty string. No return values were recorded, similar to the cases above.

`setLocalDescription` We often see that SDP messages are recorded as arguments. As explained in chapter 2, SDP messages contain a lot of information and are prone to SDP munging [10]. It is difficult to automate analysis on if these messages contain privacy-sensitive information, as their structures are very dynamic and the identifiers found in them are from browser sessions we cannot access anymore. We address this issue and how we tackle it in section 5.2. From what we can see by manual inspection, though, nothing looks out of the ordinary compared to the description and example given in RFC 8839 [22]. Most SDP messages found are similar to the one found in Listing 5.1. Furthermore, we did extract hostnames and IP addresses from the SDP messages for further inspection, which we describe in section 5.1.3. No return values were recorded, similar to the cases above.

`setRemoteDescription` Same as for `setLocalDescription`.

`createAnswer` No arguments were captured, which is logical according to the documentation. No return values were recorded, similar to the cases above.

`addIceCandidate` Except for four calls in total, function calls contained no arguments, in which case an end-of-candidates signal is sent [28]. The ICE candidates found in the four other calls all look in accordance with RFC 8839 [22]. We were unable to tie the two IP addresses found in the ICE candidates to any party,

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random`

but no records were found reporting the IPs as malicious or privacy-invading.

To summarize, the arguments and return values of the WebRTC calls captured in the JSON files either contain no information at all or contain information that is in accordance with the SDP's and ICE's RFC pages and with WebRTC's documentation page. However, there are multiple reasons for more research. The SDP messages contained in the `setLocalDescription` and `setRemoteDescription` calls require investigation from a different approach to conclude that nothing suspicious is happening, as the crawler provides data from expired browser sessions and the data is not structured enough to analyze easily. Besides that, simply looking at these individual call intercepts does not give a clear view on why websites are invoking these WebRTC calls and what exactly is happening. Most of these websites do not seem to give the impression they provide real-time communication functionalities, yet so much WebRTC traffic is captured. It could also be that our crawler is missing information or intercepts calls improperly. Just as mentioned in the paragraph on `setLocalDescription`, a live browser session and interaction with a website while monitoring network traffic are required to draw conclusions (section 5.2).

5.1.3 Identifying actors

Another important element to investigate is what actors are involved in all the network traffic. We look at two kinds of possible actors: the endpoints that are contained in the SDP messages and the third-party scripts that initiate the WebRTC traffic. Endpoints in SDP messages are found under the `setLocalDescription` and `setRemoteDescription` intercepts under the `savedCalls` section of the generated JSON files. Script initiators are visible in the structured `callStats` section of the JSON files.

5.1.3.1 Endpoints

Using a script, we searched through all JSON files, scanned for either `setLocal Description` or `setRemoteDescription` intercepts, opened its argument `sdp` for the SDP message, and scanned for IP addresses and hostnames in places where we would expect them. According to the RFC [16], we can expect endpoints of four different kinds: endpoints in the originator info field (`o=`), endpoints in a connection info field (`c=`), ICE candidates in an attributes field (`a=candidate:`), and endpoints in an attributes field for RTCP routing (`a=rtcp:`). Note that in one SDP message, only one originator info field can be found, but all other fields can occur multiple times, as SDP supports multiple media description sections in one message. Each media description can contain one connection field and multiple attribute fields as well. We do not go into technical detail about what all these fields mean, but they are all fields that contain IP addresses used for routing, so all are relevant. In tables 5.2 and 5.3 we list the counts for every found end-

point, categorized per kind. Duplicate SDP messages in one JSON file have been skipped. Note that we make a distinction between `setLocalDescription` and `setRemoteDescription` because the former represents outgoing SDP messages, thus containing only our own endpoints, and the latter represents incoming messages, thus containing only endpoints from the remote peer. Hence, the endpoints found in `setRemoteDescription` might be more interesting through the perspective of finding endpoints that we might have sent data to while crawling.

Table 5.2: Endpoints and counts found in `setLocalDescription`

Endpoint	Source field	Count
`127.0.0.1`	`o=`	3,684
`0.0.0.0`	`c=`	3,683
`0.0.0.0`	`a=rtcp:`	170
`*.local`	`a=candidate:`	4

Table 5.3: Endpoints and counts found in `setRemoteDescription`

Endpoint	Source field	Count
`127.0.0.1`	`o=`	248
`0.0.0.0`	`o=`	2
`*.local`	`a=candidate:`	223
`127.0.0.1`	`a=candidate:`	27
`200.160.6.45`	`a=candidate:`	1
`2001:12ff:0:6000::45`	`a=candidate:`	1
`130.61.210.204`	`a=candidate:`	1
`130.61.33.174`	`a=candidate:`	1
`2603:c020:8012:dd13:213e:cfde:808c:1d76`	`a=candidate:`	1
`0.0.0.0`	`c=`	150
`172.16.23.4`	`c=`	1
`0.0.0.0`	`a=rtcp:`	2

The results from tables 5.2 and 5.3 do not directly reveal many relevant third-party actors.

Addresses `127.0.0.1` and `0.0.0.0` come forward most often in both tables. Address `127.0.0.1` refers to localhost and is the loopback address; anything sent to it is sent back. The presence of the loopback address in the table could be an indication that actors apply the same kind of method that Meta used to leak the `_fbp` cookie on Android devices, described in the Local Mess disclosure [9].

Even though Meta stopped doing this, it could be that other parties apply or have applied the same method. In our crawler results, we do not find any proof for this, though. It might also just be a placeholder. The other address, `0.0.0.0`, is the initially set transport address for the media section in SDP messages [22], implying a suitable ICE candidate has not been found yet, according to the documentation.

Then there are the addresses ending with the `.local` suffix. The asterisks in both tables indicate that all found addresses are unique. An example of such a unique address is `1f3b467a-f539-4c4f-9ea9-86f33b6db4bc.local`. These are masked private IP addresses, either belonging to the host or to the peer, depending on the type of call. It has been acknowledged that exposure of private IP addresses from a network allows for fingerprinting and thus poses a privacy risk. Since private IP addresses are used in WebRTC to set up a peer-to-peer connection as efficiently as possible, modern web browsers mask these IP addresses using Multicast DNS UUIDs to enhance privacy while maintaining WebRTC functionality [19][17][34]. Since they are private IP addresses, they do not provide us any information on what parties are behind these endpoints. Note that since UUIDs are used for masking, multiple of these addresses could originate from the same local IP.

Finally, we find six other IP addresses in the `setRemoteDescription` calls. Address `172.16.23.4` is again in a range of private addresses, so that does not provide us any information. The other three addresses are public, but we were unable to identify them other than their WHOIS information, which leads to service providers in Germany and Brazil. No records were found reporting the IPs as malicious or privacy-invading.

5.1.3.2 Script initiators

Each JSON file keeps track of which script source the intercepted WebRTC calls originate from. For example, in listing 5.2 we see the script sources that were found in the JSON file belonging to a website visit to `https://iam8bit.com/`. It is notable that we can see many arguments and identifiers in the URLs of the script sources. We can clearly see that some of the arguments are tied to parties associated with user tracking, such as Shopify, and those same arguments contain a string that matches our public IP. One of the identifiers, together with the detected calls, is truncated for demonstration purposes.

To get a clear view of what script initiators we are dealing with, we wrote a script that goes over all JSON files and keeps track of how many distinct websites are running the same script. Besides investigating if a script is active on a website, we also kept track of the WebRTC calls that are invoked by those scripts. In total, we found 1,102 third-party actors running scripts containing WebRTC on our crawled websites. In table 5.4 we recorded the top ten most active script initiators in terms of how many websites the script was found running on, sorted by activity. We omitted `RTCDataChannel.send`, `RTCPeerConnection.setConfiguration`, and `RTCRtpSender.setParameters` as these are not used by the script ini-

```
"callStats": {
  "https://static.kyc.red/lite.js?version=lite&sessionId
      =145116133229iam8bitmyshopifycom&ping=false&profile=true&
      pageURL=https%253A%252F%252Fwww.iam8bit.com%252Fweb-pixels
      %254073b305c4w82c1918fpb7086179m603a4010%252Fcustom%252Fweb-
      pixel-19660874%25402%252Fsandbox%252Fmodern%252F": {
    "RTCPeerConn...
  },
  "https://static.kyc.red/lite.js?version=lite&sessionId
      =145116133229iam8bitmyshopifycom&ping=false&profile=true&
      pageURL=https%253A%252F%252Fwww.iam8bit.com%252F": {
    "RTCPeerConn...
  },
  "https://imgs.signifyd.com/fp/check.js;CIS3SID=6395
      CF454BD5CD477B72C65B5E20EB65?org_id=w2txo5aa&session_id
      =145116133229iam8bitmyshopifycom&nonce=42c2782b49535259&jb
      =363726266a716f773f4e6b667d782e6271673d4966667a65696c273a3
      ...": {
    "RTCPeerConn...
  }
}
```

Listing 5.2: An example of a JSON entry displaying multiple script sources

tiators in the table.

Table 5.4: Script sources ranked on popularity together with call counts

Script source	# Sites	% Sites	create DataC.	onice cand.	create Offer	setLocal Descr.	setRem. Descr.	create Answer	addIce Cand.
adsco.re	829	10.59	829	829	828	813			
signifyd.com	434	5.54	434	433	434	434			
kyc.red	256	3.27	253	253	256	256	256	256	256
bounceexchange.com	253	3.23	252	251	253	252			
strpst.com	204	2.61			204				
richpanel.com	157	2.01	157	157	157	156			
ttwstatic.com	144	1.84	144	144	144	144			
myshopline.com	119	1.52	119		119	119			
licdn.com	98	1.25	98	98	98	98			
kaptcha.com	79	1.01	79	79	79	79			

At the top of table 5.4 is `adsco.re`, by far the most active WebRTC utilizer in the list, accounting for more than 10% of all crawled websites. This number, just like the others, is probably underestimated since our crawler was not always able

to intercept WebRTC traffic, even though websites are confirmed to use WebRTC, as mentioned before.

As shown in the table, the four calls we marked 'most significant' before are, for a big part, the only calls used by the script initiators in the table. This is an interesting find, since to set up a functional real-time media communication channel, `setRemoteDescription` is essential to use [29]. Without it, such a channel cannot exist. The `setLocalDescription` and `setRemoteDescription` call counts not being on par was also visible in table 5.1, but the implication here is stronger: The most active WebRTC actors on the web show signs that they are not using WebRTC in the intended way. Below is a brief description of all actors found in table 5.4.

adsco.re Adscore is a very well-known service for advertising companies. Its services include detecting click fraud and distinguishing between human traffic, proxies, and bots[2]. Often, user data is collected for these purposes.

signifyd.com Signifyd is a business that offers fraud protection services for e-commerce. It focuses on digital payment fraud risk decisioning. On their website they state that "Consumer behavior is monitored over time to build a constantly evolving profile of each shopper within our network. This allows Signifyd to accurately detect anomalies in shopper behavior"[3]. When searching through our JSON files, it appears that this script is often triggered on web stores, which aligns with their mission.

kyc.red Very little is known about this domain, aside from that the domain has not been active for very long[4]. Going through our own JSON files, it appears that this script almost only triggers together with Signifyd scripts on web stores. It could be that this domain is associated with them, especially given that 'KYC' is an abbreviation for 'Know Your Customer', a technology often used as a digital payment fraud prevention system.

bounceexchange.com This domain redirects to `https://wunderkind.co/`, which claims it can boost digital marketers' revenue by 25% by analyzing anonymous network traffic, combining behavioral signals, and utilizing their identity network. Their product is a personalized experience in messages and advertisements[5]. Another party that could benefit from user data.

strpst.com This domain appears to have a relation to pornographic websites. According to contributors on a popular DNS blocklist on GitHub, the domain is used

[2] `https://adscore.com/`
[3] `https://signifyd.com/`
[4] `https://directory.cside.com/nl/kyc.red`
[5] `https://wunderkind.co/`

as a pornographic webcam platform and content delivery network for adult websites[6]. WebRTC could rightfully be utilized for its real-time media communication capabilities, given that webcamming websites access this domain. No records were found reporting the domain name as malicious or privacy-invading.

richpanel.com This seems to be an AI-driven customer experience platform[7]. While they do not mention anything about analyzing digital behavior or network traffic, it is not implausible that this company is still interested in user data.

ttwstatic.com Searching for this domain in a search engine results in many hits for cookie statement pages stating that "TikTok uses this domain for tracking the use of embedded services"[8][9]. Other network analysis websites also report that the domain has connections to TikTok[10][11]. TikTok offers third-party tracking scripts (e.g., the TikTok pixel) for website owners services, which may explain the behavior.

myshopline.com This appears to be an all-in-one platform called Shopline to set up a web store[12]. Similar to earlier examples of web stores and customer management platforms, collection of user data would not be unexpected.

licdn.com Browsing to `https://licdn.com/` directly redirects us to the LinkedIn homepage. Being a social media site, although employment- and profession-oriented, it realizes marketing purposes using code snippets, as mentioned on their advertising page [13].

kaptcha.com A quick search for this domain leads to a support page belonging to a company called Kount, explaining how this domain "should be added to a list of directives to implement fingerprinting correctly, as it is the recommended domain for their data collector"[14]. Kount is a company owned by Equifax, a major multinational data, analytics, and technology company[15].

In conclusion, given the descriptions of all these actors and the deviant statistics of the WebRTC calls they invoke, it is likely that WebRTC functionalities are not used by these companies for peer-to-peer communications, except for `strpst.com`.

[6] `https://github.com/hagezi/dns-blocklists/issues/2721`
[7] `https://richpanel.com/`
[8] `https://ice.org.uk/cookie-statement`
[9] `https://matchis.nl/en/cookies`
[10] `https://netify.ai/resources/domains/ttwstatic.com`
[11] `https://semrush.com/website/ttwstatic.com/overview/`
[12] `https://shopline.com/`
[13] `https://business.linkedin.com/advertise/ads/insight-tag`
[14] `https://developer.kount.com/hc/en-us/articles/14564335823764-Content-Security-Policies-CSP-and-the-Device-Data-Collector`
[15] `https://equifax.com/about-equifax/who-we-are/`

Data analyzed in this section does not reveal what endpoints are contacted by the mentioned scripts, nor does it reveal what data they send. To achieve this, we visit and interact with websites that have scripts running from the above parties in a live browser setting while monitoring network traffic in section 5.2.

5.2 Live network analysis

In the previous section we analyzed our crawler's results. We saw that it was unable to intercept calls on every website, while all were confirmed to use WebRTC in some way. We also saw that handling and accepting cookie walls was not always successful, which might have influenced the intercepted calls. Of the websites the crawler was able to intercept calls from, the arguments and return values did not reveal exfiltration of information initially. The `setLocalDescription` and `setRemoteDescription` calls contain many endpoints in their SDP arguments, but none hint towards suspicious activities or actors. Other identifiers found in the SDP messages are difficult to distinguish from illegitimate traffic, as they come from a browser session we cannot access anymore. Finally, nine out of the top ten script initiators are confirmed to be domains belonging to companies that do not have a relation with real-time media communication but instead with advertising, fraud or bot detection, e-commerce, customer platform management, and social media services. The statistics on the used calls by these actors also indicate unintended use of WebRTC.

These are all solid reasons to perform more in-depth research on how and why websites use WebRTC. In this section we perform live network analysis using Wireshark. We not only try to get an insight into what traffic our crawler has potentially missed but also aim to investigate the underlying intents of the third-party entities initiating these WebRTC connections.

5.2.1 Traffic data analysis

For each script initiator found in table 5.4, we chose three websites on which the script is active and visited and interacted with the homepage. The websites are chosen on two criteria: they should have a descriptive hostname (many of them look suspicious or have meaningless names, such as `https://32d.cc/`) and the page should look like it has a valid purpose (such as a web store or a news page). This is to ensure a degree of relevance for our research.

The main goal is to look for identifiers or other data shared in the network packets that could possibly pose a privacy risk. We want to do this in a structured way using Wireshark. All website visits are done from the same network for consistency.

1. Start a fresh browser session using a dedicated Chrome profile.

2. Start capturing packets in Wireshark on any interface.

3. Browse to the website and accept all cookies, if applicable.
4. Scroll all the way down to the bottom of the page and ensure the entire website is loaded.
5. Verify that the relevant script(s) is/are loaded using the browser's network tab.
6. Filter on STUN traffic in Wireshark to see if WebRTC traffic appears. This should reveal any connections made to STUN or TURN servers.
7. If it does, inspect payloads of STUN packets and look for suspicious traffic. Use the browser's network, storage, and cookies tabs to scan for any matching identifiers.
8. If it does not, try interacting with a website accordingly. For instance, visiting a product page on a web store. Repeat from step 6.
9. Optional is to inspect the browser's source tab to try and interpret what the script is trying to do.
10. Clear cache, cookies, and other browser data; close browser session.

In Appendix A.1 we document our findings. For all the websites analyzed in Wireshark, we noted the endpoints to which STUN connections are being made, the types of messages being sent, and the attributes in the messages, including their values. A description of all types of messages found is also included in the appendix. Below, a summary per script initiator is given that explains our findings in the Wireshark captures.

5.2.1.1 Findings per script initiator

adsco.re In the Wireshark captures (table A.1) we can see that on all three websites visited, 'Binding Request' messages are being sent to either `stun.cloudflare.com` or `stun.l.google.com` over UDP. These are general STUN servers available for public use. In most cases, multiple binding requests are sent, which are all answered by a 'Binding Success Response' message. The response messages always contain the XOR-MAPPED-ADDRESS attribute, which shows our public IP. We did not find any identifiers in these messages. A potential explanation for these connections is to bypass proxies and VPNs, as we discuss in section 5.2.2.

signifyd.com In all websites that we visited that contain a third-party script from Signifyd (most happen to be web stores), we found multiple STUN and TURN connections being made to `eu-aa.online-metrix.net`. We also found that the domain is included in the Signifyd script source code as a literal TURN server. Searching for this domain results in several forum posts that discuss the presence

of this domain on blocklists because of tracking reasons. A network intelligence website[16] indicates that the domain is associated with LexisNexis Risk Solutions, a company specialized in detecting digital fraud and identity theft, financial crime compliance, and digital risk assessment[17]. These goals relate very much to the services that Signifyd offers to their clients. Inspecting the network tab in the browser indeed confirms that the Signifyd scripts are the initiators of the traffic to the `online-metrix.net` domain. LexisNexis and their domains are not publicly known to provide regular WebRTC services, let alone host STUN or TURN servers. Other connections visible in the capture are again to Google's public STUN servers, containing messages containing our public IP.

In the traffic to `eu-aa.online-metrix.net` (table A.2) we can see 'Binding Request', 'Allocate Request' and 'Refresh Request' messages, all with responses from the server. The binding requests and allocate requests often do not contain any interesting attributes, and these seem to fail because of a '401 (Unauthenticated)' error response. The allocate requests are resent after, this time containing a 'Username' attribute. These messages authenticate correctly and are answered by 'Allocate Success Response' messages, again containing our public IP. Suspicious here is that the 'XOR-RELAYED-ADDRESS' attribute in which it is found should not contain our own IP, but rather one of the TURN servers, as described in section 2.1.

Notably, the username attribute's string value always follows the structure of a concatenation of identifiers, separated by colons and semicolons. This structure looks non-default, and the content varies across messages, indicating the username strings are likely crafted using SDP munging [10], similar to what happens in the Local Mess disclosure [9]. In table 5.5, we summarize all identifiers found in table A.2. The string values contain (encoded[18]) store IDs, timestamps[19] and UUIDs[20]. The '`w2txo5aa`' identifier appears in all traffic that involves Signifyd's scripts. When searching for the string, we find many automated malware network scans that tie Signifyd and the `online-metrix.net` domain together exclusively[21]. We suspect it to be a hard-coded string to identify Signifyd to LexisNexis.

Finally, all traffic with `eu-aa.online-metrix.net` is taking place twice: once over UDP and once over TCP, concurrently. This is interesting, as STUN and TURN both primarily run over UDP. We suspect that the use of STUN and TURN over TCP is used as a fallback should UDP messages fail to be delivered.

kyc.red The traffic we found on the websites that run scripts from `kyc.red` (table A.3) is very similar to traffic on websites that run scripts from Signifyd. STUN and TURN connections are made to `eu-aa.online-metrix.net` and public

[16] `https://www.netify.ai/resources/domains/online-metrix.net`
[17] `https://risk.lexisnexis.com/global/en`
[18] `https://en.wikipedia.org/wiki/Base64`
[19] `https://www.unixtimestamp.com/`
[20] `https://en.wikipedia.org/wiki/UUID`
[21] `https://www.joesandbox.com/analysis/1685714/0/executive`

Table 5.5: Summary of identifiers found in the 'Username' attribute for websites with Signifyd script

Identifier	Category	Possible purpose
	watchwarehouse.nl	
w2txo5aa	32-bit string	As this string turns up in every web store, probably an ID specific to/for Signifyd
stores/28d61-rwi8rkh3aezjgtxa	Arbitrary string	Based on its value, probably a store ID to identify this specific store
dc43423ff7a204c0	64-bit hex string	Seems arbitrary. Might be a hash output, checksum, or signature
	metallica.com	
w2txo5aa	32-bit string	As this string turns up in every web store, probably an ID specific to/for Signifyd
TWV0YWxsaWNhMWMwYzZkZm JkZDFhZWRjNzQzYzYwZDc1ZGQ	Base64 UTF-8 encoding	Probably a store ID. Decodes to Metallica1c0c6dfbdd1aedc743c60d75dd
1770223149808:1770223149808	Integer	Timestamps. These cohere to the time of capturing the traffic
6832cba63087549b1ae9e22a30be92d5	128-bit hex string	Same as 64-bit hex string above
4ce10c43e959c64d	64-bit hex string	Idem
	philips.nl	
w2txo5aa	32-bit string	As this string turns up in every web store, probably an ID specific to/for Signifyd
e7fabe03-2a05-4f5b-974b-dd4f89c5a25d	UUID	Probably this store's ID, given UUID's purposes and presence of other store IDs
1770223582802:1770223582802	Integer	Timestamps. These cohere to the time of capturing the traffic
7972e758f2a55dc85c047e9a65d8a220	128-bit hex string	Same as 64-bit hex string above
96ac0ae9cabcfd96	64-bit hex string	Idem
d799ad2fcd0b4148a7951c1c71adc26c	128-bit hex string	Idem

Google servers, the 'Username' attribute in allocate requests and refresh requests is again abused to share information, UDP and TCP traffic take place simultaneously, and the '`w2txo5aa`' identifier always appears. Everything is almost identical, aside from other string values contained in the username attribute. Opening the network tab in our browser reveals that Signifyd scripts are the initiators of the `kyc.red` scripts, confirming our suspicion in section 5.1.3.

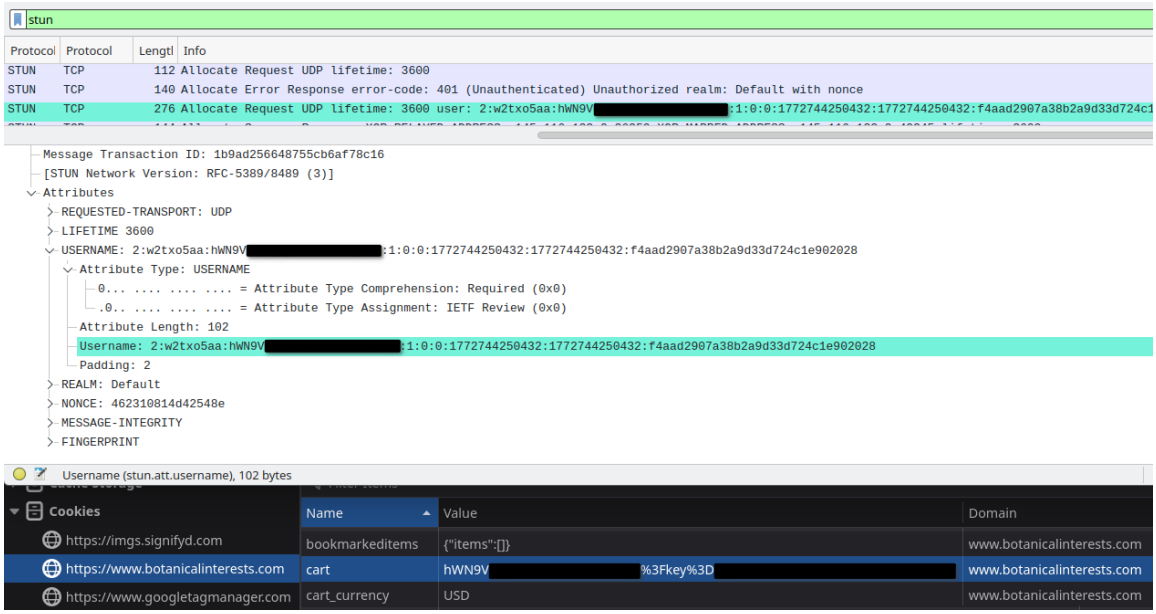


Figure 5.1: A screenshot of Wireshark on top of a Firefox browser session, displaying exfiltration of the cart cookie. Relevant lines are highlighted. Note that this screenshot is from a different browser session than those displayed in table A.3. The cookie IDs are hidden for privacy reasons.

In table 5.6 we again summarize all identifiers found in table A.3. This time, we find something worrying: a first-party cookie is being exfiltrated from the website via the username attributes of TURN allocate requests and refresh requests. In figure 5.1 we can see that part of the crafted string contained in the 'Username' field in Wireshark matches the first-party cookie called 'cart', visible in Firefox's cookies tab. Sometimes we need to add an item to our cart before we see the cart cookie being exfiltrated, while other times it is shared instantly. In case it is not shared instantly, we see that our public IP concatenated with a store identifier is shared instead. Since first-party cookies are set to the website's own domain and are mainly used to allow basic website functionality, they should not leave the context of the website. However, in this case we see that they are exfiltrated from multiple websites using WebRTC. This allows the parties the cookies are exfiltrated to (in this case the `online-metrix.net` domain belonging to LexisNexis Risk Solutions) to link them together and create a tracking profile of a user. This exfiltration is very similar to what is described in the Local Mess disclosure [9].

We wanted to do some research into what the relation is between Signifyd and the `kyc.red` domain. It is not easy to see what traffic originates from the Signifyd script and what originates from the `kyc.red` script, but we can say with high certainty that the two parties work together: We see that the cart cookie is only exfiltrated on websites that load both the Signifyd script and the `kyc.red` script, implying that the presence of both scripts creates a higher privacy risk. We also did

an analysis on our crawl data to see what websites load either or both of the scripts: In our dataset there are 434 websites that load Signifyd scripts. Of those websites, 250 websites load Signifyd scripts as well as `kyc.red` scripts, and 184 of them load Signifyd only. That is 57.6% with both scripts and 42.4% with only Signifyd. In our crawl results, we see that four JSON files indicate only `kyc.red` scripts were loaded. This is likely an error, as when manually browsing those websites, we saw both loaded.

Table 5.6: Summary of identifiers found in the 'Username' attribute for websites with kyc.red script

Identifier	**Category**	**Possible purpose**
	botanicalinterests.com	
w2txo5aa	32-bit string	Same identifier as in table 5.5
192021epicgardeningukmyshopifycom	Arbitrary string	String containing our public IP concatenated with a store identifier
492311d315ee425a	64-bit hex string	Seems arbitrary. Might be a hash output, checksum, or signature
hwn62[redacted]	**Cart cookie**	First-party cookie used for handling the website's cart functionality
3576fa1acf2a2daf	64-bit hex string	Same as 64-bit hex string above
	iam8bit.com	
w2txo5aa	32-bit string	Same identifier as in table 5.5
hwn5u[redacted]	**Cart cookie**	First-party cookie used for handling the website's cart functionality
fa5c9661c3c78fb6	64-bit hex string	Same as 64-bit hex string above
	lyleandscott.com	
w2txo5aa	32-bit string	Same identifier as in table 5.5
192021lyleandscottdevelopmentmyshopifycom	Arbitrary string	String containing our public IP concatenated with a store identifier
4d98bc56fa4cffad	64-bit hex string	Same as 64-bit hex string above
hwn6c[redacted]	**Cart cookie**	First-party cookie used for handling the website's cart functionality
c8f3eef7b6bef556	64-bit hex string	Same as 64-bit hex string above

bounceexchange.com While our crawler results contain JSON files displaying WebRTC activity, we were unable to find any WebRTC traffic caused by this script by visiting affected websites. As such, we did not include any tables in Appendix

A.1. Our suspicion is that this script requires certain triggers to initiate the WebRTC traffic, which we do not meet in the current setup. To name two examples, this script could have detected our crawler as a bot before, triggering a bot detection script utilizing WebRTC, or the traffic only triggers from mobile devices. Inspecting the script's source code reveals WebRTC code containing a regular expression that scans for local IP addresses, but it seems not to trigger either.

strpst.com For `strpst.com`, we were unable to reproduce any WebRTC traffic on the websites in question, even when opening a video stream on any of the websites. As such, we did not include any tables in Appendix A.1. We are unsure why we find no traffic, but there are no hints of any data exfiltration. We remain confident that the script does not intend to abuse WebRTC for this cause, as the websites affected support live video streams.

richpanel.com For the Richpanel script, we found the same results as for `bounce exchange.com`, and we again suspect that scripts might trigger in different conditions. As such, we did not include any tables in Appendix A.1.

ttwstatic.com Websites affected by this script show the same behavior as we see in the traffic to and from websites affected by the Adscore script, as can be seen in table A.4. Only our public IP is explicitly revealed in the XOR-MAPPED-ADDRESS attribute of binding success responses from public STUN servers. More on this in section 5.2.2.

myshopline.com Websites affected by Shopline show similar traffic to the Adscore websites, but with a wider variety of STUN servers and attributes in the STUN messages. The source code of the script initiating the WebRTC traffic reveals that the STUN servers listed in table A.5 are hard-coded to be made connections to. Some of these are verified, publicly available STUN servers (Amazon, Google, Cloudflare, Nextcloud, Xiaomi (on `miwifi.com`)), but others are publicly unknown and could hide suspicious activity (`biliapi.net`, `smartgslb.com`). All IP addresses that are found in the table other than our own public IP address belong to the STUN servers, and their presence is considered normal STUN behavior.

Table A.5 also shows a connection being made to `eu-aa.online-metrix.net` again. This connection was, however, made while visiting the checkout page of this web store. It is therefore likely that this connection originates from a different script, as it does not appear in other website visits with the Shopline script. The identifiers contained in the regarding request do not seem to exfiltrate data and instead look like shop or organization identifiers.

licdn.com Similar to Adscore, the script triggers a connection to STUN servers and doesn't transmit any identifiers. See table A.6 and section 5.2.2.

kaptcha.com Again, similar behavior to the Adscore script. See table A.7 and section 5.2.2.

5.2.1.2 Categorization of WebRTC traffic

To summarize our findings in Wireshark, we can categorize the scripts that we analyzed the network traffic of in three different categories:

Scripts that actively exfiltrate data using WebRTC In this category are scripts from `signifyd.com` and `kyc.red`, which both make connections to `online-metrix.net`. This is the most interesting category, given that we have shown these scripts are responsible for active exfiltration of data that is not intended for tracking purposes. The fact that WebRTC is used to covertly share tracking IDs is worrying enough, but sharing actual first-party cookies using a technology intended for voice and video calls is highly subversive. We do not know what is done with the data shared with `online-metrix.net`, nor do we know if LexisNexis is the final recipient of this data, but the fact that all parties involved are related to consumer behavior is debatable. It should be noted, though, that Signifyd, being the party that initiates both scripts, is the only party we were able to identify exfiltrating cookies using WebRTC.

Scripts that initiate WebRTC traffic without exfiltrating data This is the largest category, containing the scripts from `adsco.re`, `ttwstatic.com`, `myshopline.com`, `licdn.com`, and `kaptcha.com`. In our research data, we could only identify simple STUN traffic originating from these scripts in the form of binding requests. These binding requests are always answered by a binding success response that includes the user's public IP address as an attribute value, which is default STUN behavior. No other identifiers or data are exfiltrated. In the next section, we try to determine why this traffic takes place at all, given that the websites involved do not support audio or video communication.

Scripts that show no traffic in Wireshark The scripts from `bounceexchange.com`, `strpst.com`, and `richpanel.com` are in this category. We were unable to reproduce WebRTC traffic originating from these scripts in our experiment setup, though they are confirmed to contain WebRTC code. We suspect that the WebRTC code in these scripts triggers under different conditions, such as website visits from mobile devices.

5.2.2 The 'innocent' STUN traffic

In the previous section we saw that many websites run third-party scripts that make connections to publicly available STUN servers without exfiltrating data or any other privacy-invading action at first sight. We want to research why third-party scripts consistently contact STUN servers without exfiltrating data or setting up a

WebRTC connection. The Wireshark traffic in question can be found in tables A.1, A.4, A.5, A.6 and A.7. Note that the traffic to `eu-aa.online-metrix.net` in table A.5, which contains tracking IDs, is, as mentioned before, caused by a different script and should not be considered in this section.

5.2.2.1 STUN as a proxy server bypass

Since any script running on a website already has access to a user's public IP (the script has to be downloaded over an HTTP connection, revealing the public IP address), and no audio or video communication is supported on any of the relevant websites, we suspect that the traffic we found might resemble attempts to circumvent IP hiding technologies, such as the use of VPNs and proxies. It has long been known that WebRTC can leak a hidden public IP, even from behind a VPN server [7][1], but so far this is not known to be actively abused on a large scale. We also want to test for IP leakage from behind proxy servers, as a paper exploring device fingerprinting, published by researchers from KU Leuven and the University of California, describes how Flash applications on websites were able to circumvent browser-set HTTP proxies and contact a remote host directly, revealing the user's real public IP address to the remote host [31].

We want to research if the STUN traffic we found reveals our hidden public IP address if we visit the pages either from behind a proxy server or behind a VPN server. We used VPN software and freely available online proxy servers to which we connected using Firefox' built-in proxy settings. The latter supports three modes: HTTP, SOCKS4, and SOCKS5. HTTP proxies are specifically designed to handle and interpret HTTP web traffic and only support TCP traffic. SOCKS4 proxies operate on a lower layer and can handle any TCP connection, supporting a broader range of protocols. SOCKS5 is an upgraded version of SOCKS4 that, in addition to TCP connections, can also handle UDP connections and IPv6 addresses.

We suspect different outcomes per VPN and proxy server, as the STUN traffic we found in this research presents itself as both UDP and TCP traffic. As such, we tested the VPN and proxy connections on two different websites: one that loads scripts from Signifyd and `kyc.red`, and one that loads Shopline scripts. Signifyd and `kyc.red` scripts load STUN over UDP as well as TCP, while Shopline scripts only load STUN over UDP. We connected to a VPN server and different kinds of proxy servers, started Wireshark captures, visited the websites, and saved the captures.

5.2.2.2 Comparing the results

We compared the STUN attributes in the captures containing the public IP as seen by the STUN server to the IP address returned by `https://www.whatsmyip.org/`, which is the IP visible to the website we visited. We recorded our findings in Appendix A.2. Comparing the screenshots of the Wireshark results to our public IP address gives a clear insight in wether the VPN or proxy server was bypassed

by the STUN server. In table 5.7 we give a summary of the outcomes.

Table 5.7: Summary of our findings in testing proxy and VPN bypasses in STUN

Connection type	**Scripts**	**STUN over UDP**	**STUN over TCP**
VPN server	Signifyd & kyc.red	Routed through VPN server	Routed through VPN server
	Shopline	Routed through VPN server	N/A
HTTP proxy	Signifyd & kyc.red	Bypasses proxy server	Routed through proxy server
	Shopline	Bypasses proxy server	N/A
SOCKS4 proxy	Signifyd & kyc.red	Bypasses proxy server	Dropped / Unknown
	Shopline	Bypasses proxy server	N/A
SOCKS5 proxy	Signifyd & kyc.red	Dropped / Unknown	Routed through proxy server
	Shopline	Dropped / Unknown	N/A

We see that in our setting, the VPN server we used is immune to leakage of true public IP. In figure A.1 we can only detect the VPN server's IP address. This is not a surprising finding, given that Al-Fannah also found combinations of VPN servers and browsers that were not vulnerable to this leak [1].

For HTTP proxy servers, we see that scripts that initiate STUN traffic over UDP can reveal our true IP address and compromise anonymity in our case. HTTP proxies operate on the application layer and are designed to only handle connections over TCP. Therefore, the configured proxy server simply cannot handle the STUN traffic. The browser knows this and just routes the UDP traffic as it normally would, which completely bypasses the STUN server. Following the same logic, STUN traffic over TCP is successfully routed through the configured proxy server, and anonymity would be maintained. However, the scripts in this context initiate UDP traffic as well, still leaking our IP address. Both the leakage via UDP and anonymity preservation over TCP can be seen in figure A.2.

SOCKS4 proxy servers show similar behavior to the HTTP proxy servers. SOCKS4 is designed to operate strictly over TCP on the session layer. The browser again recognizes that the proxy server cannot handle UDP traffic and thus defaults to simply bypassing the configured proxy server, leaking our hidden IP. Interesting in this case, though, is that we would expect the STUN over TCP traffic to show up in Wireshark just like it did with HTTP proxy servers, but it does not. In figure A.3a we can see that the website containing Signifyd and `kyc.red` scripts does not initiate any TCP traffic while connected to a SOCKS4 proxy, while we saw in all earlier cases that both UDP and TCP traffic were captured. We were unable to identify why this happens. We suspect it relates to how the `online-metrixx.net` STUN server or its script is configured.

With SOCKS5 proxy servers, we see an improved factor of anonymity in our setup. SOCKS5 proxies also operate on the session layer like SOCKS4, but they are inherently designed so that they can also handle UDP traffic, solving a large

part of the problem. However, in figure A.4 we do not see any UDP traffic at all. A likely explanation for this is that while SOCKS5 is designed to support UDP traffic, modern-day browsers like Firefox have not implemented this feature[22]. Consequently, rather than allowing the UDP traffic to bypass the proxy and leak the true IP address, our theory is that the browser drops all UDP traffic as a security feature. With the TCP traffic being the only traffic left originating from these scripts, and that traffic being routed through the proxy server, anonymity is preserved in this case.

Of the researched IP hiding mechanisms, VPN servers and HTTP proxy servers are probably the most prevalent ones, as they are easy to set up and provide the most use cases for the wide public. We therefore think it is safe to conclude that WebRTC is, among others, used on a large scale to at least try to circumvent IP hiding via both proxy and VPN. Out of ten of the largest WebRTC script initiators in our crawled dataset, seven are confirmed to leak a public IP hidden behind an HTTP or SOCKS4 proxy through STUN in our research setup. SOCKS5 does provide some resistance to it, but as UDP support is not always implemented by modern-day browsers, we suspect the practical effect is small. In our experiments the usage of a VPN prevented public IP leakage, but this does not exclude its possibility, as has been shown in the past [1].

[22] `https://bugzilla.mozilla.org/show_bug.cgi?id=1882018`

Chapter 6

Conclusion

In this research, we aimed to bridge a gap in the existing research field by conducting a large-scale crawl on websites that are confirmed to initiate WebRTC traffic, mostly by third-party scripts, to investigate to what extent WebRTC is used for data exfiltration and how this could aid tracker and fingerprinting actors.

From the 500,000 most popular websites following the CrUX ranking [6], we identified 7,884 websites that show activity of WebRTC traffic. We modified the Targeted and Troublesome crawler [27] to our needs, such that it specifically intercepts a range of WebRTC API calls, including their arguments and return values. The crawler was deployed on all 7,884 websites on an emulated mobile Chrome browser while accepting all cookies.

At first sight, the API calls that were intercepted did not reveal any data being exfiltrated, nor were there any hints for tracking or fingerprinting in the arguments or return values of those calls. Through the crawl data on third-party script initiators, we were able to identify the top ten most prevalent script initiators responsible for WebRTC traffic, and we found that nine out of these ten script initiators are parties related to user profiling, tracking, advertising, fingerprinting, and marketing, with no relation to real-time media communication. Our analysis indicated that the scripts initiated by these parties did not invoke the API calls needed to start a functional, valid peer-to-peer connection, which leads us to the conclusion that it is evident that the most active WebRTC actors on the web do not use WebRTC for what it is designed for and instead hint towards using WebRTC for tracking and possibly fingerprinting purposes.

To investigate what exactly WebRTC is instead used for by these top ten parties, we performed manual network traffic analysis on websites running the involved scripts with the aim to gain more insight into possible data exfiltration not shown by our crawler. By visiting the homepages of several websites per script initiator and filtering on STUN and TURN traffic, we defined three categories of WebRTC abuses among the top ten script initiators. Most alarmingly, we found proof of active data exfiltration by scripts from `signifyd.com` and `kyc.red`, both making connections to `online-metrix.net`, all being domains associ-

ated with digital data analysis companies. These scripts make use of a technique known as SDP munging [10] to exfiltrate identifiers, timestamps, unique UUIDs, and even first-party web cookies (particularly a shopping cart cookie) via TURN allocate requests. This outcome justifies our motivation for this research following the Local Mess disclosure [9] and the growing rate of advanced tracking and fingerprinting methods [23].

Furthermore, we identified a large category of scripts that initiate STUN binding requests to public servers (like Google or Cloudflare) without explicitly exfiltrating data but still show suspicious traffic. Our experiments indicated that the purpose of this traffic is likely to circumvent IP-hiding technologies like proxies and VPNs. In this research we were unable to reproduce WebRTC traffic that breaks anonymization by VPN as shown in earlier research [1], but we were able to show that WebRTC is actively being abused to bypass certain proxy configurations.

A part of the WebRTC traffic in our data set remains unexplained, which we provide further thoughts on in chapter 7. It is clear, though, that the vast majority of actors initiating WebRTC have no interest in setting communication channels at all but are instead interested in tracking and possibly fingerprinting goal ends on a large scale.

We want to close off by addressing preventative measures against the issues highlighted in this research paper. Currently, WebRTC API calls, ICE, STUN, and TURN requests are generated in the background, completely transparent to the user. Since we showed that the largest part of online WebRTC traffic is not used for regular communication, warning users by asking permission when browsers are trying to initiate WebRTC calls would greatly improve their privacy. Most popular browsers do not currently support an easy option to warn about or completely block WebRTC traffic. Researchers from the University of the Aegean (Fakis et al., 2020) already propose two methods to supply such options [8]. Furthermore, we could argue that the WebRTC specification itself should contain more privacy- and security-enhancing measures, such as sanitation of API call methods and SDP messages.

Chapter 7

Limitations & Future Work

In this chapter we go over some of the limitations of our research, possibly having an effect on the quantitative and qualitative results and conclusions that we described in this research. Whilst doing so, we make recommendations for further research in this field.

We notice that the results we got from our crawler did not provide much insight into what data is actually shared with what actors, while this insight was largely available in Wireshark. The crawls mostly guided us towards the most prevalent script initiators of WebRTC traffic, which we could use for our live network analysis in Wireshark. However, using a more effective crawler with more variation in setup parameters would possibly result in intercepting more of the actual data being sent to specific endpoints. This could allow for better automated research and would potentially dismiss the need to perform a live network analysis, which right now is only tailored to a few websites per script initiator.

The `CMPCollector` was tasked with detecting and handling the cookie walls. A small note must be made that it turns out the collector was not always successful in recognizing cookie walls, and even if it was, it was not always successful in handling them. We found that in some cases, the collector did not detect anything at all, resulting in an empty CMP entry, even though a cookie wall was present. This could be due to a bug in the browser or in our crawler. For this reason, it is possible that some scripts are not executed because no consent for third-party cookies was given, as seen, for example, with the Facebook Pixel code described in the Local Mess disclosure [9]. We suspect that these failures have minimal impact on our results, since the crawler was still able to identify a significant amount of WebRTC traffic.

In section 5.1.1 we mentioned that only 4,419 of the 7,829 crawls intercepted WebRTC traffic, while all websites crawled were confirmed to contain WebRTC calls. This makes a relevant quantitative analysis on the prevalence of tracking and fingerprinting via WebRTC on the web infeasible. Other research found that general fingerprinting prevalence was present on 10.18% of the Alexa top 100K websites in 2021 [23]. We mentioned in section 4 that about 1.58% of the top

500K websites are confirmed to use WebRTC. Extrapolating these numbers would crudely indicate that 1/10th of fingerprinting is done using (among others) WebRTC. Verifying this claim could be interesting further research.

We already listed a few possible explanations in section 5.1.1 as to why some crawls fail to result in useful data. We would like to emphasize some of those explanations, as we think they are more substantial than others:

First, the crawler that we used from the Targeted and Troublesome paper [27] was behind on the Tracker Radar Collector of DuckDuckGo [4] by 3.5 years by the time of crawling. As browsers evolve, many bugs could have been introduced and patched by now. We recommend further researchers interested to use the latest version of the Tracker Radar Collector as a starting point.

Secondly, there are multiple parameters that could have been tweaked in this research, which we did not implement. In further research, we recommend emulating both mobile and desktop devices and actually doing research from mobile devices as well as desktop devices if possible, as scripts might be able to recognize that we are using emulations. Other factors that might make a difference are the use of different browsers, making a distinction between accepting all cookies and denying them, and browsing from multiple vantage points. In this research we only crawled from an EU vantage point.

As for the live network analysis in Wireshark, the last mentioned factors might also apply. They could explain why some traffic did not show up in Wireshark even though script sources confirmed the presence of code. In further research, it might be worth tweaking these parameters to see if it makes any difference.

Moreover, we did not address an issue pointed out by Reiter & Marsalek [32]: the lack of specification on the use of the signaling channel in the WebRTC documentation. WebRTC relies on the trust of the signaling server used during the signaling phase of ICE, the choice of which is left to the programmer. Our research did not zoom in on what channels were used during that phase. In further research, it makes sense to do more live network analysis on what signaling servers are involved under what protocols to analyze the security and privacy of used signaling channels.

To summarize, we think that our live network analysis gave most of the insights in this research when it comes to what data is actually shared and what the intentions of script initiators are. Despite the limitations, our research proved some significant findings. However, spending time on a more effective crawler and doing further research on used signaling channels might give a better and broader insight into the implications and scale on which data exfiltration takes place.

Bibliography

[1] Nasser Mohammed Al-Fannah. One leak will sink a ship: Webrtc ip address leaks. In *2017 International Carnahan Conference on Security Technology (ICCST)*, pages 1–5, 2017.

[2] The Internet Archive. The http archive. `https://httparchive.org/`. Accessed: 2026-03-10.

[3] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. How tracking companies circumvented ad blockers using websockets. In *Proceedings of the Internet Measurement Conference 2018*, pages 471–477, 2018.

[4] DuckDuckGo. Duckduckgo tracker radar collector. `https://github.com/duckduckgo/tracker-radar-collector/`. Accessed: 2026-01-21.

[5] DuckDuckGo. Duckduckgo's autoconsent library. `https://github.com/duckduckgo/autoconsent/`. Accessed: 2026-01-21.

[6] Zakir Durumeric and David Adrian. Chrome (crux) top million websites. `https://github.com/zakird/crux-top-lists`. Accessed: 2026-03-10.

[7] ExpressVPN. Webrtc leak test. `https://www.expressvpn.com/webrtc-leak-test`. Accessed: 2026-04-05.

[8] Alexandros Fakis, Georgios Karopoulos, and Georgios Kambourakis. Neither denied nor exposed: Fixing webrtc privacy leaks. *Future Internet*, 12(5), 2020.

[9] Aniketh Girish, Gunes Acar, Narseo Vallina-Rodriguez, Nipuna Weerasekara, and Tim Vlummens. Disclosure: Covert web-to-app tracking via localhost on android. `https://localmess.github.io/`. Accessed: 2026-01-26.

[10] Philipp Hancke. Not a guide to sdp munging. `https://webrtchacks.com/not-a-guide-to-sdp-munging/`. Accessed: 2026-01-31.

[11] har.fyi. Getting started. https://har.fyi/guides/getting-started/. Accessed: 2026-03-10.

[12] Mohammadreza Hazhirpasand and Mohammad Ghafari. One leak is enough to expose them all: From a webrtc ip leak to web-based network scanning. In *International Symposium on Engineering Secure Software and Systems*, pages 61–76. Springer, 2018.

[13] Internet Engineering Task Force (IETF). *RFC 6455: The WebSocket Protocol*. Accessed: 2026-03-12.

[14] Internet Engineering Task Force (IETF). *RFC 8445: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. Accessed: 2026-03-11.

[15] Internet Engineering Task Force (IETF). *RFC 8835: Transports for WebRTC*. Accessed: 2026-03-11.

[16] Internet Engineering Task Force (IETF). *RFC 8866: SDP: Session Description Protocol*. Accessed: 2026-03-11.

[17] Internet Engineering Task Force (IETF). Using multicast dns to protect privacy when exposing ice candidates. https://www.ietf.org/archive/id/draft-ietf-mmusic-mdns-ice-candidates-02.html. Accessed: 2026-02-01.

[18] Internet Engineering Task Force (IETF). *RFC 5780: NAT Behavior Discovery Using Session Traversal Utilities for NAT (STUN)*. Accessed: 2026-02-22.

[19] Internet Engineering Task Force (IETF). *RFC 6762: Multicast DNS*. Accessed: 2026-02-01.

[20] Internet Engineering Task Force (IETF). *RFC 8489: Session Traversal Utilities for NAT (STUN)*. Accessed: 2026-02-22.

[21] Internet Engineering Task Force (IETF). *RFC 8656: Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. Accessed: 2026-02-22.

[22] Internet Engineering Task Force (IETF). *RFC 8839: Session Description Protocol (SDP) Offer/Answer Procedures for Interactive Connectivity Establishment (ICE)*. Accessed: 2026-01-31.

[23] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1143–1161. IEEE, 2021.

[24] Anne van Kesteren. W3c standard for web security github issue. https://github.com/w3c/webappsec-csp/issues/92. Accessed: 2026-04-05.

[25] Google LLC. Bigquery: From data warehouse to autonomous data and ai platform. https://cloud.google.com/bigquery. Accessed: 2026-03-10.

[26] Google LLC. Crux on bigquery. https://developer.chrome.com/docs/crux/bigquery. Accessed: 2026-03-10.

[27] Zahra Moti, Asuman Senol, Gunes Acar, Hamid Bostani, Frederik Zuiderveen Borgesius, Veelasha Moonsamy, and Arunesh Mathur. Targeted and troublesome crawler. https://github.com/targeted-and-troublesome/targeted-and-troublesome-crawler/. Accessed: 2026-01-21.

[28] Mozilla Developer Network. Rtcpeerconnection documentation page. https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection. Accessed: 2026-01-29.

[29] Mozilla Developer Network. Webrtc api documentation page. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API. Accessed: 2026-02-01.

[30] Mozilla Developer Network. Webrtc api signaling and video calling documentation page. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Signaling_and_video_calling. Accessed: 2026-04-05.

[31] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting . In *2012 IEEE Symposium on Security and Privacy*, pages 541–555, Los Alamitos, CA, USA, May 2013. IEEE Computer Society.

[32] Andreas Reiter and Alexander Marsalek. Webrtc: your privacy is at risk. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 664–669, New York, NY, USA, 2017. Association for Computing Machinery.

[33] Tim Vlummens, Aniketh Girish, Nipuna Weerasekara, Frederik Zuiderveen Borgesius, Gunes Acar, and Narseo Vallina-Rodriguez. Bridges to self: Silent web-to-app tracking on mobile via localhost. In *35th USENIX Security Symposium*, 2026.

[34] Qingsi Wang. Chromium issue: mdns service for ip handling in webrtc. https://issues.chromium.org/issues/40591226. Accessed: 2026-04-05.

Appendix A

Wireshark captures

A.1 STUN traffic during manual page visits

In this appendix all STUN traffic intercepted with Wireshark is displayed in tables. We include short descriptions of all the different types of STUN message attributes found in the Wireshark captures.

Username Is used in STUN together with an ICE password as a shared secret to identify a single ICE interaction, allowing an integrity check [20].

MAPPED-ADDRESS Is included in the response to a STUN binding request, containing the IP and port the server saw in the binding request of the client [20].

XOR-MAPPED-ADDRESS Identical to MAPPED-ADDRESS, but obfuscated by the XOR function [20].

XOR-RELAYED-ADDRESS Essentially the same as XOR-MAPPED-ADDRESS, but in the context of the response to a TURN allocate request [21].

OTHER-ADDRESS A secondary address the STUN server can place in a binding response that would be used if the client requests a change of IP or port. [18].

RESPONSE-ORIGIN Placed in the response to a STUN binding request, containing the IP and port of the STUN server. [18].

Finally, some notes to keep in mind while interpreting the tables:

Note 1: To hide our own IP for privacy reasons, we censored the last three blocks of our IP address as such: 86.[redacted]. The ports shown are unchanged. We applied the same censoring to the cookie IDs, which are always in the format of 'hwnXX[redacted]'.

Note 2: Asterisks (*) are used to indicate that a particular message was sent repeatedly with the same IP but with different port numbers.

Note 3: In all tables, control attributes essential to the STUN traffic but irrelevant to data flow analysis were excluded for the sake of readability. These attributes include error code, lifetime, message integrity hash, realm, and nonce attributes.

A.1.1 adsco.re

Table A.1: Manual STUN traffic network analysis for websites with adsco.re script

Endpoint	**Message type**	**Attributes**	**Value**
		thepiratebay0.org	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:42966*
		nerdtorrenthd.net	
stun.cloudflare.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:58098*
		popads.net	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:33763*

A.1.2 signifyd.com

Endpoint	Message type	Attributes	Value
		watchwarehouse.com	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa:stores/28d61 -rwi8rkh3aezjgtxa; dc43423ff7a204c0
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:8298*
		XOR-MAPPED -ADDRESS	86.[redacted]:34651*
	Refresh Request	Username	1:w2txo5aa:stores/28d61 -rwi8rkh3aezjgtxa; dc43423ff7a204c0
	Refresh Success Response	N/A	N/A
		metallica.com	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A

Endpoint	Message type	Attributes	Value
	Allocate Request	Username	2:w2txo5aa:TWV0YWxsaWNhMWMwYzZkZmJkZDFhZWRjNzQzYzYwZDc1ZGQ=:1:0:0:1770223149808:1770223149808:6832cba63087549b1ae9e22a30be92d5
	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa:twv0ywxsawnhmwmwyzzkzmjkzdfhzwrjnzqzyzywzdc1zgq;4ce10c43e959c64d
	Allocate Success Response	XOR-RELAYED-ADDRESS	86.[redacted]:40832*
		XOR-MAPPED-ADDRESS	86.[redacted]:48293*
	Refresh Request	Username	1:w2txo5aa:twv0ywxsawnhmwmwyzzkzmjkzdfhzwrjnzqzyzywzdc1zgq;4ce10c43e959c64d
	Refresh Success Response	N/A	N/A
stun1.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:39920*
		philips.nl	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A

Endpoint	Message type	Attributes	Value
	Allocate Request	Username	2:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d:1:0:0:1770223582802:1770223582802:7972e758f2a55dc85c047e9a65d8a220
	Allocate Success Response	XOR-RELAYED-ADDRESS	86.[redacted]:56755*
		XOR-MAPPED-ADDRESS	86.[redacted]:52446*
	Refresh Request	Username	2:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d:1:0:0:1770223582802:1770223582802:7972e758f2a55dc85c047e9a65d8a220
	Refresh Success Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d;96ac0ae9cabcfd96
	Allocate Success Response	XOR-RELAYED-ADDRESS	86.[redacted]:9632*
		XOR-MAPPED-ADDRESS	86.[redacted]:34443*
	Refresh Request	Username	1:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d;96ac0ae9cabcfd96
	Refresh Success Response	N/A	N/A

Endpoint	Message type	Attributes	Value
	Allocate Request	Username	2:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d:0:d799ad2fcd0b4148a7951c1c71adc26c
	Allocate Success Response	XOR-RELAYED-ADDRESS	86.[redacted]:65246*
		XOR-MAPPED-ADDRESS	86.[redacted]:50151*
	Refresh Request	Username	2:w2txo5aa:e7fabe03-2a05-4f5b-974b-dd4f89c5a25d:0:d799ad2fcd0b4148a7951c1c71adc26c
	Refresh Success Response	N/A	N/A
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:55811*

Table A.2: Manual STUN traffic network analysis for websites with signifyd.com script

A.1.3 kyc.red

Endpoint	Message type	Attributes	Value
		botanicalinterests.com	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa:86[redacted] epicgardeninguk myshopifycom; 492311d315ee425a
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:5642*
		XOR-MAPPED -ADDRESS	86.[redacted]:44804*
	Refresh Request	Username	1:w2txo5aa:86[redacted] epicgardeninguk myshopifycom; 492311d315ee425a
	Refresh Success Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa: hwn62[redacted]; 3576fa1acf2a2daf
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:34353*
		XOR-MAPPED -ADDRESS	86.[redacted]:43296*
	Refresh Request	Username	1:w2txo5aa: hwn62[redacted]; 3576fa1acf2a2daf

	Refresh Success Response	N/A	N/A
stun2.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:59387*
stun1.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49249*
		iam8bit.com	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa: hwn5u[redacted]; fa5c9661c3c78fb6
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:46108*
		XOR-MAPPED -ADDRESS	86.[redacted]:51041*
	Refresh Request	Username	1:w2txo5aa: hwn5u[redacted]; fa5c9661c3c78fb6
	Refresh Success Response	N/A	N/A
stun2.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:40836*
		lyleandscott.com	
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A

	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa:86[redacted] lyleandscottdevelopment myshopifycom; 4d98bc56fa4cffad
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:32944*
		XOR-MAPPED -ADDRESS	86.[redacted]:59876*
	Refresh Request	Username	1:w2txo5aa:86[redacted] lyleandscottdevelopment myshopifycom; 4d98bc56fa4cffad
	Refresh Success Response	N/A	N/A
	Allocate Request	Username	1:w2txo5aa: hwn6c[redacted]; c8f3eef7b6bef556
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:33219*
		XOR-MAPPED -ADDRESS	86.[redacted]:59653*
	Refresh Request	Username	1:w2txo5aa: hwn6c[redacted]; c8f3eef7b6bef556
	Refresh Success Response	N/A	N/A
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:56981*

Table A.3: Manual STUN traffic network analysis for websites with kyc.red script

A.1.4 ttwstatic.com

Table A.4: Manual STUN traffic network analysis for websites with ttwstatic.com script

Endpoint	**Message type**	**Attributes**	**Value**
		college.harvard.edu	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:47806*
		dailyexpress.com.my	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:53058*
		tiktok.com	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:48590*

A.1.5 myshopline.com

Endpoint	Message type	Attributes	Value
		sunfere.com	
stun.cloudflare.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
hw-v2-web-player-tricker.biliapi.net	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
stun2.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
ec2-13-115-244-22.ap-northeast-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
		MAPPED -ADDRESS	86.[redacted]:34401
ec2-52-215-127-253.eu-west-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
		MAPPED -ADDRESS	86.[redacted]:34401
stun.smartgslb.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
		MAPPED -ADDRESS	86.[redacted]:34401
		RESPONSE -ORIGIN	0.0.0.0:19302

stun.miwifi.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
		MAPPED -ADDRESS	86.[redacted]:34401
		RESPONSE -ORIGIN	111.206.174.3:3478
		OTHER -ADDRESS	111.206.174.2:3479
stun.nextcloud.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:34401
		MAPPED -ADDRESS	86.[redacted]:34401
		RESPONSE -ORIGIN	159.69.191.124:443
		OTHER -ADDRESS	159.69.191.124:3478
eu-aa.online-metrix.net	Binding Request	N/A	N/A
	Binding Error Response	N/A	N/A
	Allocate Request	N/A	N/A
	Allocate Error Response	N/A	N/A
	Allocate Request	Username	1:k8vif92e:yyriskgo317f09bcd67511f0b17ada1ec9bf4387;7b018e7139aede73
	Allocate Success Response	XOR-RELAYED -ADDRESS	86.[redacted]:31702*
		XOR-MAPPED -ADDRESS	86.[redacted]:54812*
	Refresh Request	Username	1:k8vif92e:yyriskgo317f09bcd67511f0b17ada1ec9bf4387;7b018e7139aede73

	Refresh Success Response	N/A	N/A
honeylovedoll.com			
stun.cloudflare.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
hw-v2-web-player-tricker.biliapi.net	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
stun2.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
ec2-13-115-244-26.ap-northeast-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
		MAPPED -ADDRESS	86.[redacted]:49896
ec2-52-215-127-253.eu-west-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
		MAPPED -ADDRESS	86.[redacted]:49896
stun.smartgslb.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
		MAPPED -ADDRESS	86.[redacted]:49896
		RESPONSE -ORIGIN	0.0.0.0:19302
stun.miwifi.com	Binding Request	N/A	N/A

	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
		MAPPED -ADDRESS	86.[redacted]:49896
		RESPONSE -ORIGIN	111.206.174.2:3478
		OTHER -ADDRESS	111.206.174.3:3479
stun.nextcloud.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:49896
		MAPPED -ADDRESS	86.[redacted]:49896
		RESPONSE -ORIGIN	159.69.191.124:443
		OTHER -ADDRESS	159.69.191.124:3478
		nonbinarystyle.com	
stun.cloudflare.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:53068
hw-v2-web-player-tricker.biliapi.net	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:53068
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:53068
ec2-54-65-63-219.ap-northeast-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:53068
		MAPPED -ADDRESS	86.[redacted]:53068

ec2-18-156-18-172.eu-central-1.compute.amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:53068
		MAPPED-ADDRESS	86.[redacted]:53068
stun.smartgslb.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:53068
		MAPPED-ADDRESS	86.[redacted]:53068
		RESPONSE-ORIGIN	0.0.0.0:19302
stun.miwifi.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:53068
		MAPPED-ADDRESS	86.[redacted]:53068
		RESPONSE-ORIGIN	111.206.174.2:3478
		OTHER-ADDRESS	111.206.174.3:3479
stun.nextcloud.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED-ADDRESS	86.[redacted]:53068
		MAPPED-ADDRESS	86.[redacted]:53068
		RESPONSE-ORIGIN	159.69.191.124:443
		OTHER-ADDRESS	159.69.191.124:3478

Table A.5: Manual STUN traffic network analysis for websites with myshopline.com script

A.1.6 licdn.com

Table A.6: Manual STUN traffic network analysis for websites with licdn.com script

Endpoint	Message type	Attributes	Value
		linkedin.com	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:57277*
		resumeworded.com	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:52948*
		mygov.in	
stun.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:42539*

A.1.7 kaptcha.com

Table A.7: Manual STUN traffic network analysis for websites with kaptcha.com script

Endpoint	Message type	Attributes	Value
ca.ecco.com			
stun1.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:39492*
my.equifax.co.uk			
stun1.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:51599*
nativeshoes.com			
stun1.l.google.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:57830*
ec2-52-23-111-175. compute-1. amazonaws.com	Binding Request	N/A	N/A
	Binding Success Response	XOR-MAPPED -ADDRESS	86.[redacted]:58530*

A.2 Proxy and VPN bypassing

This section contains screenshots that prove that STUN servers are used to bypass established connections to proxy servers.

The public IP addresses of the VPN and proxy servers we used in our experiments are included. On the next pages, we included those addresses together with screenshots of Wireshark captures, in which the IP addresses are visible as STUN attributes.

By comparing the IP addresses to the Wireshark traffic, we can easily see in what contexts the proxy or VPN servers are bypassed by STUN (thus leaking our real public IP address) and in what contexts not. Exfiltrated first party cookies are also visible in the traffic, partially hidden for privacy reasons.

A.2.1 VPN server

Protocol	Protocol (IP)	Info
STUN	UDP	Binding Request
STUN	UDP	Binding Error Response error-code: 401 (Unauthenticated) Unauthorized
STUN	UDP	Allocate Request UDP
STUN	UDP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	UDP	Allocate Request UDP user: 2:w2txo5aa:hWN9c[redacted]:1:0:0:
STUN	UDP	Allocate Success Response XOR-RELAYED-ADDRESS: 193.142.201.182:32841
STUN	TCP	Allocate Request UDP
STUN	TCP	Allocate Request UDP
STUN	UDP	Binding Request
STUN	TCP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	TCP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 193.142.201.182:43926
STUN	TCP	Allocate Request UDP user: 2:w2txo5aa:hWN9c[redacted]:1:0:0:
STUN	TCP	Allocate Request UDP user: 2:w2txo5aa:hWN9c[redacted]:1:0:0:
STUN	TCP	Allocate Success Response XOR-RELAYED-ADDRESS: 193.142.201.182:63057
STUN	TCP	Allocate Success Response XOR-RELAYED-ADDRESS: 193.142.201.182:23010

(a) Visit of website with Signifyd and kyc.red scripts while connected to a VPN server

Protocol	Protocol (IP)	Info
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 193.142.201.182:27888
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 193.142.201.182:9832

(b) Visit of website with Shopline scripts while connected to a VPN server

(c) The IP of the VPN server we used while browsing, as shown by `https://www.whatsmyip.org/`

(d) Our real public IP as shown by `https://www.whatsmyip.org/` before hiding our IP, partially hidden for privacy reasons

Figure A.1: Screenshots of Wireshark captures with STUN traffic from two different websites, together with the VPN's IP address and our own IP address.

A.2.2 HTTP proxy server

Protocol	Protocol (IP)	Info
STUN	UDP	Allocate Request UDP lifetime: 3600
STUN	UDP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	UDP	Allocate Request UDP lifetime: 3600 user: 1:w2txo5aa:hwn77[redacted]
STUN	UDP	Allocate Success Response XOR-RELAYED-ADDRESS: 86.[redacted] XOR-
STUN	TCP	Allocate Request UDP lifetime: 3600
STUN	TCP	Allocate Request UDP lifetime: 3600
STUN	TCP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	TCP	Allocate Request UDP lifetime: 3600 user: 1:w2txo5aa:hwn77[redacted]
STUN	TCP	Allocate Success Response XOR-RELAYED-ADDRESS: 46.161.6.165:1210 XOR-

(a) Visit of website with Signifyd and kyc.red scripts while connected to an HTTP proxy

Protocol	Protocol (IP)	Info
STUN	UDP	Binding Request
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 86.[redacted]
STUN	UDP	Binding Request
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 86.[redacted]

(b) Visit of website with Shopline scripts while connected to an HTTP proxy

(c) The IP of the HTTP proxy server we used while browsing, as shown by `https://www.whatsmyip.org/`

(d) Our real public IP as shown by `https://www.whatsmyip.org/` before hiding our IP, partially hidden for privacy reasons

Figure A.2: Screenshots of Wireshark captures with STUN traffic from two different websites, together with the HTTP proxy IP address and our own IP address.

A.2.3 SOCKS4 proxy server

Protocol	Protocol (IP)	Info
STUN	UDP	Allocate Request UDP lifetime: 3600
STUN	UDP	Allocate Error Response error-code: 401 (Unauthenticated) Unauthorize
STUN	UDP	Allocate Request UDP lifetime: 3600 user: 1:w2txo5aa:hwn77[redacted]
STUN	UDP	Allocate Success Response XOR-RELAYED-ADDRESS: 86.[redacted] XOR-
STUN	UDP	Binding Request
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 86.[redacted]

(a) Visit of website with Signifyd and kyc.red scripts while connected to an SOCKS4 proxy

Protocol	Protocol (IP)	Info
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Request
STUN	UDP	Binding Success Response XOR-MAPPED-ADDRESS: 86.[redacted]

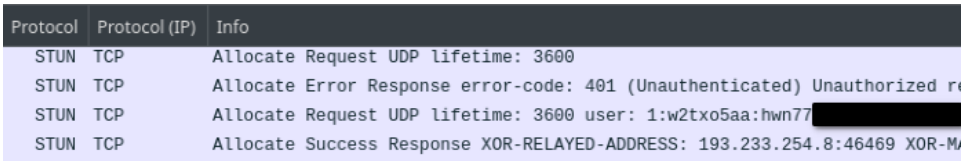
(b) Visit of website with Shopline scripts while connected to an SOCKS4 proxy

(c) The IP of the SOCKS4 proxy server we used while browsing, as shown by `https://www.whatsmyip.org/`

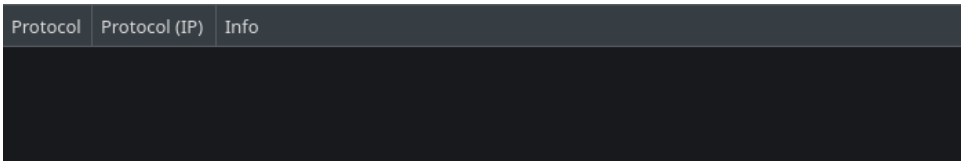
(d) Our real public IP as shown by `https://www.whatsmyip.org/` before hiding our IP, partially hidden for privacy reasons

Figure A.3: Screenshots of Wireshark captures with STUN traffic from two different websites, together with the SOCKS4 proxy IP address and our own IP address.

A.2.4 SOCKS5 proxy server



(a) Visit of website with Signifyd and kyc.red scripts while connected to an SOCKS5 proxy



(b) Visit of website with Shopline scripts while connected to an SOCKS5 proxy

(c) The IP of the SOCKS5 proxy server we used while browsing, as shown by `https://www.whatsmyip.org/`

(d) Our real public IP as shown by `https://www.whatsmyip.org/` before hiding our IP, partially hidden for privacy reasons

Figure A.4: Screenshots of Wireshark captures with STUN traffic from two different websites, together with the SOCKS5 proxy IP address and our own IP address.