WebSocket Adoption and the Landscape of the Real-Time Web

Paul Murley
University of Illinois at Urbana-Champaign
pmurley2@illinois.edu

Zane Ma
University of Illinois at Urbana-Champaign
zanema2@illinois.edu

Joshua Mason
University of Illinois at Urbana-Champaign
joshm@illinois.edu

Michael Bailey
University of Illinois at Urbana-Champaign
mdbailey@illinois.edu

Amin Kharraz
Florida International University
mkharraz@fiu.edu

ABSTRACT

Developers are increasingly deploying web applications which require real-time bidirectional updates, a use case which does not naturally align with the traditional client-server architecture of the web. Many solutions have arisen to address this need over the preceding decades, including HTTP polling, Server-Sent Events, and WebSockets. This paper investigates this ecosystem and reports on the prevalence, benefits, and drawbacks of these technologies, with a particular focus on the adoption of WebSockets. We crawl the Tranco Top 1 Million websites to build a dataset for studying real-time updates in the wild. We find that HTTP Polling remains significantly more common than WebSockets, and WebSocket adoption appears to have stagnated in the past two to three years. We investigate some of the possible reasons for this decrease in the rate of adoption, and we contrast the adoption process to that of other web technologies. Our findings further suggest that even when WebSockets are employed, the prescribed best practices for securing them are often disregarded. The dataset is made available in the hopes that it may help inform the development of future real-time solutions for the web.

CCS CONCEPTS

• **Information Systems** → *World Wide Web.*

KEYWORDS

WebSocket; Polling; Measurement; Performance; Adoption; Abuse; Security

ACM Reference Format:
Paul Murley, Zane Ma, Joshua Mason, Michael Bailey, and Amin Kharraz. 2021. WebSocket Adoption and the Landscape of the Real-Time Web. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3450063

WWW '21, April 19–23, 2021, Ljubljana, Slovenia

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3450063

1 INTRODUCTION

Websites are increasingly dependent on real-time communication between clients and servers. The modern web has expanded to a broad range of applications that require bidirectional updates between parties—online gaming, advertising, and collaborative document editing are a few examples. For years, the client-server model was stretched to provide these capabilities to developers. Practices such as HTTP polling enabled some real-time applications, but these stopgap solutions present significant inefficiencies in the form of excessive headers and wasted requests. HTTP polling gradually evolved and optimized some of these pain points, but the fundamental architectural challenge remained: receiving real-time updates from a server which cannot initiate connections.

WebSockets were designed to offer developers a built-in, performant solution for real-time applications. The WebSocket API directly addresses these needs, providing low overhead and full duplex communication between web servers and their clients, while avoiding large amounts of unnecessary traffic. Having been implemented in major browsers for almost a decade, WebSockets are now a mature, widespread part of the web landscape. Accordingly, it is important to understand how this web technology is deployed and to what degree WebSockets have been successful in improving the web experience. By identifying WebSocket successes and failures, and comparing WebSockets with their alternatives, we strive to inform future efforts to improve WebSockets and similar web technologies.

This paper is an empirical assessment of the real-time web. We visit the Tranco Top 1M [46] websites, gathering data on 71,637 WebSocket connections across 55,805 websites which use them. We make our full dataset available for download[1]. Using this dataset, we study the types of sites that use real-time technologies, with a particular focus on WebSockets, and we characterize how these sites employ these various technologies. As expected, we find that WebSockets are being used for a diverse set of use cases across popular websites, with some of the most common being chat, analytics, and live updates for sports scores and stock prices. We find that online chat accounts form a majority of WebSocket connections across top sites. Scripts which communicate via the WebSocket API are almost exclusively third-party (94.9%), suggesting that

[1] https://bit.ly/2TeUpYx

most websites do not implement their own WebSocket infrastructure. This is in line with previous work [44], which detailed the increasing complexity of websites due to a growing number of third party inclusions, and particularly third party scripts.

In order to assess how successful WebSockets have been, we quantify the advantages they provide and identify real scenarios where they could be employed to improve current website implementations. We provide calculations on the overhead and wasted request savings of switching from HTTP polling to WebSockets, and we present real-world data to support our calculations. The results suggest that there are at least as many sites still using HTTP polling as there are sites which use WebSockets. As a case study, we provide a concrete example of the reduction of overhead that a website achieves by deploying WebSockets. Our research points to significant room for improvement on the web through the further deployment of WebSockets.

Like many other technologies on the Internet, WebSockets are commonly misconfigured and/or misused. While we did not uncover any new vulnerabilities, our analysis reveals shortcomings in third party libraries and other errors in deployment that degrade the security and performance of many applications which use WebSockets. We discuss several of the best practices laid out in the WebSocket RFC [2] and compare them to the ecosystem we observe. For example, we find that 74.4% of WebSocket servers do not check/verify the HTTP `Origin` header attached to requests to open a connection. 14.1% of the WebSocket servers we observe are accessible over unencrypted (`ws://`) channels, with 0.8% of them using this configuration by default. While these may not always result in blatant vulnerabilities, they are indicators that developers are frequently failing to follow best-practices, leading to a more risky environment overall. Beyond misconfigurations, we uncover evidence that WebSockets are regularly used to facilitate unsavory practices such as user tracking, cryptojacking, and malware delivery. We provide an empirical look at several malicious use cases we observed, along with examples we observed in the wild. Our hope is that this work serves to raise awareness about the importance of real-time technologies such as WebSockets, while also underscoring several problems to be corrected in current, real-world deployments.

2 BACKGROUND

2.1 The Need for Real-Time Communication

The client-server model has long been the architectural backbone of the web: a client requests a resource, and that request is subsequently serviced by a remote server. However, this model has become increasingly inadequate for various applications. As web apps become more dynamic and interactive, servers often require the ability to push messages to the client at will. Examples of these use cases are numerous: browser-based gaming, collaborative document editing such as Google Docs, chat services, continuous updates to sports scores and stock prices, and many more. In these applications, servers may need to update clients multiple times per second, but may also go without sending an update for minutes or hours. Clearly, the web needs mechanisms for bidirectional communication at unpredictable intervals. In this paper, we examine a group of technologies that seek to enable this: HTTP Polling/Streaming, Server-Sent Events (SSE), and WebSockets.

2.2 HTTP Polling

An early solution to server-initiated communication, known as HTTP polling, has been around for more than two decades. In HTTP polling, a client desiring near-real-time updates sends frequent HTTP requests to a server, which usually replies with a empty or baseline response. When an update becomes available, the client receives it from the server with a latency of roughly the time between requests. To further optimize this solution, HTTP offers the `keep-alive` header. This option allows the reuse of a TCP connection for multiple HTTP requests, reducing the overhead of creating a new TCP connection for each HTTP request. The problems with this approach are straightforward and well known [1]. Even with persistent connections, the server cannot push updates to the client directly. While HTTP polling may be sufficient for applications where data arrives at known times, it will lead to many wasted client requests when servers need to update clients at inconsistent intervals. Each of these requests and responses contain HTTP headers, leading to significant amounts of wasted network traffic. In addition, updates from the server can only arrive at the frequency with which it is polled. An optimization on polling, called "long polling", uses the ability of servers to hold client queries open until data becomes available. This reduces the delay in delivering updates to clients, since the server will respond to the request at the moment when the data becomes available. Since requests need to be sent much less frequently (only when the server sends data or when a request times out), this solution offers a significant decrease in bytes on the wire.

2.3 HTTP Streaming and Server-Sent Events

A further optimization to long polling, known as "HTTP streaming", keeps the underlying TCP connection open even after data is delivered in response to a request, meaning that more data can be delivered to the client if/when it becomes available. This is accomplished using a `Transfer Encoding: chunked` HTTP header, which causes the browser to hold open the TCP connection even as multiple responses are received, eliminating the need for a reconnection whenever the server sends data.

One particular variety of HTTP streaming, called Server-Sent Events (SSE), are offered by the `EventSource` JavaScript browser API. SSE essentially implements HTTP streaming in an easy-to-use interface, with reconnection and event firing built in. It sets the HTTP `mimetype` header to the value `text/eventstream`, indicating to the browser that responses should be delivered as JavaScript events. However, these

solutions still suffer from added overhead due to connection timeouts and HTTP headers attached to each message.

2.4 WebSockets

The WebSocket protocol was standardized in 2011 to address many of the issues described above. WebSockets offer a significant improvement over previous real-time update mechanisms. A single HTTP request/response is required for setup. Subsequently, a full-duplex communications channel is available to both the client and the server. This means a client can receive updates from a server (either text or binary data) in real-time, without polling the server. WebSocket frames sent over an existing connection do contain a header, but this header is much smaller than an HTTP header (usually less than 8 bytes total). Thus, WebSockets are valuable for two main categories of use cases that are not sufficiently handled by existing web technologies: 1) Small, frequent data exchanges between client and server which benefit from smaller per-message overhead, and 2) Server-initiated communications with a client which no longer require the client to poll the server.

A WebSocket connection begins when a client initiates a connection by sending an HTTP `GET` request with the header `Upgrade: websocket`. If the server is capable of serving a WebSocket connection, it responds with HTTP `101: Switching Protocols`, and the connection is established. Both the client and the server may now send data frames at will. According to the WebSocket RFC [2], WebSockets were designed to be "as close to just exposing raw TCP to [a] script as possible". There is a small header (2 to 14 bytes) attached to every WebSocket frame which contains an opcode, the payload size, and a masking bit, but relative to HTTP headers, the overhead of WebSocket frames is quite small. The browser closes WebSocket connections automatically when the client closes (or navigates away from) the page. Major browsers began implementing experimental versions of WebSockets as early as 2010, and since 2013, all major (desktop and mobile) browsers provide full WebSocket support [13].

2.5 Excluded Technologies

There are a few notable techniques and technologies related to real-time communication on the web which we purposefully exclude from our analysis. We do not examine plugins such as Microsoft Silverlight or Adobe Flash. While these protocols do allow for sockets which can be used in websites, we consider them to be end-of-life, as they are either no longer supported by major browsers (Silverlight), or will be phased out within a year (Flash). We also specifically exclude WebRTC from our measurements. WebRTC is an important protocol for real time communication on the web, but it usually satisfies different requirements than the protocols we study here. WebRTC is a peer-to-peer protocol which usually uses UDP. In this paper, we are studying client/server interactions, so we omit it from our analysis. Finally, we do not investigate HTTP/2 Server Push. Server push is a technique by which a server can initiate the sending of resources to the client without a request. However, it is not meant as a real-time communication mechanism, but rather as an optimization technique to decrease the number of `GET` requests the client needs to send on a page load. For this reason, we do not discuss it in the paper.

3 MEASUREMENT

To gather data on the use of real-time technologies in the wild, we crawled the Tranco Top 1M [46][2] using an instrumented version of the Chromium browser. Our browser harness is written in Go. It uses *chromedp*[15], a Go interface for the Chrome DevTools Protocol, to drive the browser and collect data. We use a full (non-headless) version of the browser in an in-memory display, and we use a fresh browser instance with a new user data directory for each website visit.

Our crawl of the Tranco Top 1M was conducted across 10 virtual machines, each running Ubuntu 16.04 and Chromium version 81. Each crawler VM ran 12 browser instances simultaneously, and the crawl took approximately four days (8 thru 11 May 2020) in total. After retrying each failed crawl a second time, we obtained data for a total of 88.1% of websites in the top million. 7.7% of the listed domain names failed to resolve, and the remaining 4.2% had web servers which failed to respond or reset incoming connections. We remained on each web page for 60 seconds, in order to ensure we could observe real-time updates for a reasonable period of time. We gathered metadata on all resources downloaded for each site, including request initiators and timestamps for each request. We gathered data on each WebSocket and EventSource (Server-Side Event) connection down to the data sent in each individual frame. Our full dataset is available for download[3].

3.1 Polling, Long Polling, and HTTP Streaming

Detecting HTTP polling and streaming is tricky because there are many different techniques, intervals, and libraries that are used to accomplish them in the wild. We take a heuristic approach to the measurement of these techniques. To be considered polling, a site must send three or more requests to the same URL, although we allow the query and fragment to differ. The requests must come from the same initiating script. We also require that the time between these requests remain consistent. To enforce this requirement, we calculate the median time between polling requests for a site and compare it to the number of requests and the amount of time spent on the page. If polling is happening at a consistent interval, we expect the product of median polling interval and the number of requests to be near the time spent on the page. We verify this methodology with manual inspection of a sample of the instances identified, finding no false positives.

Our crawler remains on each page for 60 seconds during our main crawl, but this is often is often too brief to detect polling or streaming, given that default browser timeouts for HTTP connections are now as high as five minutes[18]. To address this, we conducted an additional crawl of a subset

[2]Top 1M List: https://tranco-list.eu/list/QJ94
[3]Dataset: https://bit.ly/2TeUpYx

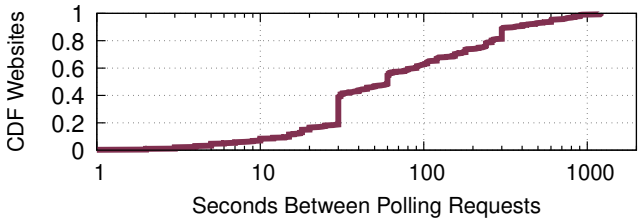

Figure 1: Polling Intervals—A CDF of the median intervals at which sites send new polling requests. No standard interval for long polling exists, and we see wide variation in how developers implement their own polling solutions.

of websites, in which we remained on each page for one hour, rather than one minute. Our subset of sites for this crawl consisted of the top 1000 sites, along with a random sample of 1000 sites from each of the top 10K, top 100K, and top 1M, for a total sample size of 4000 websites. We found that HTTP polling or long polling are in use on 14.8% of these websites. Higher-ranked sites more commonly use polling, with 19.8% of the top thousand sites leveraging the technique compared to 9.2% of sites in the top million. We observed varying polling intervals, shown in Figure 1. The most common choice is an interval of 30 seconds, with the median being 60 seconds. On some sites, JavaScript forces full or near-full page refreshes periodically, which we consider to be a particularly inefficient variety of polling.

To determine whether these instances were regular or long polling, we examined the time between request and response. In long polling, we expect the request/response interval to be close to the interval between requests. In regular polling, the server responds immediately, so the interval is much shorter. We consider any group of polling requests in which the median time for a response is greater than half the median time between requests to be indicative of long polling. Applying this methodology to our dataset, we find long polling to be exceedingly rare in practice, occurring on only a single website out of our 4000 site sample. This was a surprise to us, as we found long polling discussed frequently online as a real-time update technique. One explanation for this is that while regular polling sacrifices latency in updates, it also frees servers from the requirement to hold open connections over a long period of time. As discussed further in our limitations section, this finding is likely also influenced by our data collection method, which only visited the front pages of websites. Nonetheless, it was surprising to find a lack of usage of this technique. HTTP streaming, which can be identified by looking for a "`Transfer-Encoding: Chunked`" HTTP header on polling requests, was present on 4.5% of all pages, and roughly a quarter of the sites using some form of HTTP polling/streaming.

3.2 Server-Sent Events

Like long polling, Server-Sent Events, which are implemented using the JavaScript `EventSource` API, were much less prevalent than we expected. We find them in use on only 0.4% of the top thousand and 0.05% of websites in the top million. Of that small percentage in the top million sites, a single advertising/tracking service (*media.net*) accounts for 62.8% of those instances. Clearly, this is not a technology that has found widespread adoption on the web. Usage of SSE and WebSockets offer an intriguing case study into what happens when a technology is not adopted by all major browsers. Although the `EventSource` standard was adopted quickly by Chrome (2010), Safari(2010), and Firefox (2012), it was *never* added to Internet Exporer, and was only added to Microsoft Edge in 2019[5]. Given that Internet Explorer still has a market share of more than 2%, it is understandable that developers have largely avoided this API in favor of more widely supported real-time solutions. By contrast, Internet Explorer (and all other major browsers) had added support for WebSockets by 2012. As we show below, they have become *significantly* more common than SSE. While there are undoubtedly other contributing factors, it seems clear that the decision not to add SSE to Internet Explorer has had a significant adverse impact on SSE adoption.

3.3 WebSockets

We found WebSocket usage on 55,805 websites (6.3%) in total. This is a substantial increase from a study in 2018 [22], which found that only 1.6% to 2.5% of sites used WebSockets. Figure 2 shows the prevalence of WebSocket usage relative to site ranking. Extremely highly ranked sites tend to be slightly more likely to use WebSockets (7.3% of the top 1000 sites), but this difference is marginal. WebSockets are clearly favored by developers relative to SSE and the EventSource API, but they continue to lag well behind polling in terms of adoption.

The expanding usage of WebSockets across the Internet is enabled by third party providers, who account for the vast majority of WebSocket use. Across 71,637 WebSocket connections we observed, 94.9% of connection-initiating scripts and 92.1.% of the WebSocket servers to which they connect are third-party (different second-level domain than the base site). This large number of third party providers means that there is less heterogeneity in deployments than one might expect based on the number of sites using WebSockets. Across all the connections we observed, we find only 7,624 unique WebSocket servers.

3.3.1 Who Uses WebSockets? Next, we investigated the types of websites that use WebSockets and the types of services that are being provided over WebSockets. To perform website categorization, we use the *WebShrinker* API [19], which provides a mapping from domain names to website categories (Sports, Chat, News, etc.). As shown in Table 1, live chat is the most common use case for WebSockets, accounting for thirteen of the top twenty most common third party providers. These products are designed to increase user

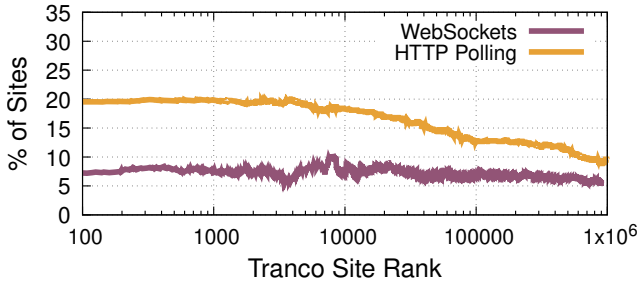

Figure 2: WebSockets and polling by site popularity—We track the cumulative frequency of WebSocket and HTTP Polling use over the Tranco Top 1M. WebSockets are marginally more common in the top 1K sites than the top 1M (7.3% vs. 6.3%), but polling is *much* more common over those same intervals.

WS Service	Count	Percentage	Category
zopim.com	9640	13.5%	Chat
tawk.to	8500	11.9%	Chat
drift.com	5888	8.2%	Tracking
intercom.io	4924	6.9%	Chat
livechatinc.com	4448	6.2%	Chat
visitors.live	3157	4.4%	Tracking
jivosite.com	1847	2.6%	Chat
hotjar.com	1803	2.5%	Tracking
firebaseio.com	1799	2.5%	Analytics
crisp.chat	1715	2.4%	Chat
Others	27916	39.0%	-

Table 1: Most Common WebSocket Services—The most common third party WebSocket service providers. The majority of top services are live chat products, accounting for six of the top ten. Tracking and analytics use cases are also quite common, and are mostly provided as third party services.

Category	# Using WS	% Using WS
Stocks	572	50.8%
Software	1959	29.0%
Gambling	847	25.0%
Shopping	3327	10.0%
Chat	907	10.0%
Real Estate	594	8.7%
Sports	566	6.3%
Adult	593	4.8%
Social	288	3.12%
News/Weather	2196	3.1%

Table 2: WebSocket Usage by Category—We list some interesting website categories, along with the rate at which they use WebSockets. WebSockets are deployed across many different types of websites, led by stock sites, which are often updated in real time.

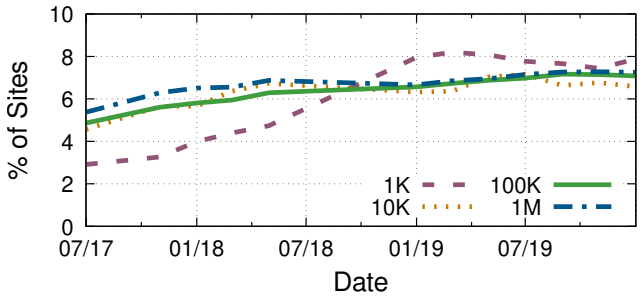

Figure 3: WebSocket Adoption Over Time—The percent of sites using WebSockets over the last three years. Perhaps surprisingly, the top thousand sites were slower to adopt WebSockets than the top million as a whole. As a whole, WebSocket adoption has mostly leveled off over the last two years—an indication that the technology is relatively mature.

engagement across many types of websites, especially for online commerce or company product sites. We find that WebSockets are also used to a lesser extent for ads, tracking, and outright malicious purposes as well. We delve deeper into who is using WebSockets in Section 4, focusing on misconfiguration and malicious use.

3.3.2 How Have WebSockets Been Adopted Over Time? We leveraged data from the HTTP Archive [16] to study WebSocket prevalence over the last three years—the oldest reliable data on WebSocket use we could find. The HTTP Archive publishes data from historical web crawls. While they do not capture fine-grained information about WebSocket connections such as opcodes and payload data, they have data on WebSocket initiation requests for roughly the last three years, which we plot in Figure 3. The data shows that the top one thousand sites were slower, and perhaps more cautious, in adopting the new technology. This could be because the top thousand sites care more about compatibility with all browsers, including very old ones. The data also shows that WebSocket adoption has been stagnant over the last year, indicating that WebSockets are a mature technology and may be reaching their "high-water mark" less than a decade after their standardization.

In Figure 4, we compare the adoption rate of WebSockets over the last three years to that of SSE and HTTP Strict Transport Security (HSTS), a HTTP header which causes browsers to enforce using HTTPS on a particular page. While HSTS is not a real-time technology, we thought it worthwhile to provide an adoption comparison since there are both optional web improvements released as RFCs at approximately the same time. All three of these technologies were introduced in a three year period from 2010-2012, but they have seen very different rates of adoption over the last decade. For reasons explained above, SSE has remained almost nonexistent in the wild. HSTS, on the other hand, has seen usage grow significantly, even relative to WebSockets. We hypothesize that this is attributable to the simplicity and ease-of-implementation

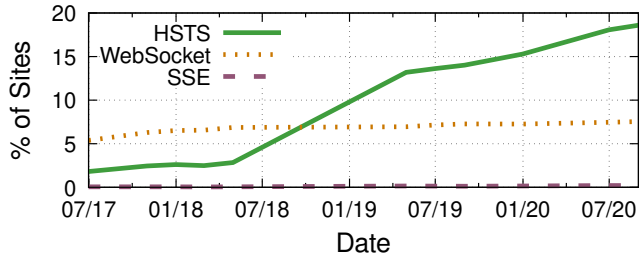

Figure 4: Web Tech Adoption Over Time—The rates at which WebSockets have been adopted over the last three years relative to Server-Side Events (SSE) and HTTP Strict Transport Security (HSTS). SSE is barely visible on the x-axis here, due to extremely low adoption. HSTS provides a contrast to WebSockets in adoption rate, likely because of the ease and simplicity of adoption relative to WebSockets and SSE.

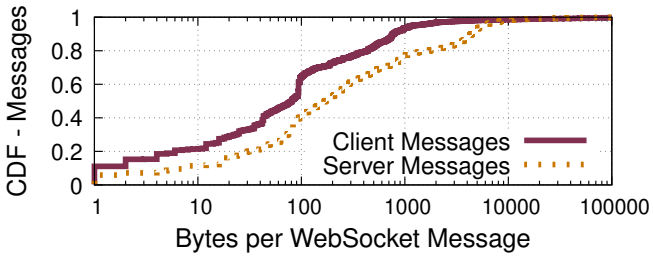

Figure 5: CDF: Bytes Per Message—We show the distribution of the number of bytes in each WebSocket message payload. The overall median frame size is 85 bytes, while the largest single message we captured was over 5 megabytes. The small size of most messages underscores the value of the reduced per-message overhead offered by WebSockets, since WebSocket headers are generally an order of magnitude smaller than HTTP headers.

of HSTS for web developers, but we leave investigation of this for future work.

3.3.3 How Are WebSockets Used? To characterize how developers are using WebSockets today, we present some statistics on the traffic we observed. Figure 6 shows the distribution of the number of WebSocket messages sent over each connection by clients and servers. This scatter plot shows that while there is diversity in the way messages are sent, it is clear that it is more common for servers to send multiple messages for each client message than the reverse. In other words, the directionality of WebSocket traffic leans towards servers sending more messages to clients. This aligns with our expectations, given that one of the primary motivations for WebSockets was the ability for servers to push messages to clients without a corresponding request. 3.1% of connections sent zero messages, and the median connection saw between 7 and 8 messages exchanged over the 60 seconds we remained on the page. Interestingly, there were multiple connections that exchanged thousands of messages during a single site visit. In the most extreme case, *popsplit.us*, a web-based game, sent 31,324 messages (from server to client) over the course of our visit, averaging an update every 1.4 milliseconds. While this particular case is extreme and is likely the result of poor development or misconfiguration, the fact that the site still functions highlights the ability of WebSockets to facilitate high frequency communication with websites. Despite this capability, only 2,512 connections (2.9%) averaged more than a frame per second, suggesting that cases requiring high frequency updates may actually be relatively rare.

On a frame-by-frame level, we observed that most messages are relatively small, with the median message size being 85 bytes. However, more than 5% of the messages were larger than a kilobyte, and there were rare instances of multi-megabyte WebSocket frames. Manual inspection revealed that some of these were the result of tracking/analytics scripts sending the content of the DOM back to a server. Putting aside the privacy concerns, which are discussed in more detail

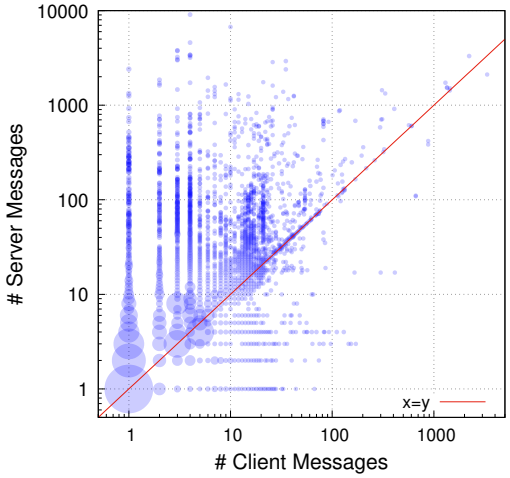

Figure 6: Client and Server Messages per Connection—The number of messages sent relative to the number of messages received for each WebSocket connection. The size of each mark scales with the number of connections with those particular x and y values. Although connections are quite diverse in their patterns, it is clearly more common for server messages to outnumber client messages.

in Section 4, this seems to be a highly inefficient method for accomplishing the goals of these companies. Other sites use large payloads for a variety of reasons. We observe instances where sites use WebSockets to load resources which would traditionally be loaded via normal HTTP `GET` requests. One site, *moviemovie.com.hk*, achieved an ever-changing display of movies by sending megabytes worth of movie poster images over WebSockets. Sites which load many of their resources over WebSockets may find an increase in efficiency, but there is a danger of having to re-implement many features that HTTP provides within the JavaScript handling these WebSocket connections.

The structure of data flowing over WebSockets on the modern web is relatively homogeneous. WebSockets can transmit data in text or binary. Clearly, it is more bandwidth-efficient to transmit data in a binary format. However, we find that 88.4% of messages, and 96.4% of total bytes, are made up of text data. Further, the majority of this text data (69%) is JSON, indicating that developers using WebSockets value ease of development over optimization. We also observed that some of this JSON contained re-implementation of features already provided by the WebSocket protocol. We identify 433 sites which are sending `PING/PONG` messages enclosed in JSON, even though the WebSocket protocol itself provides dedicated opcodes specifically for this purpose. All of this together serves to confirm the anecdote that web developers often introduce inefficiencies in their code to ease development, or because they simply lack a firm understanding of the features and tools they use.

3.3.4 How Much Could WebSockets Improve Sites Currently Using Polling? Previous work has studied the performance benefits of WebSockets under controlled laboratory settings [52, 53]. Here, we focus on applying real-world data to understand how changes could affect websites as they are currently deployed. In schemes such as HTTP polling and long polling, where a unique HTTP request is required for each message, HTTP headers become a significant source of overhead. Across our data, we find that the mean size of HTTP request headers is 184.9 bytes. Response headers are more than twice as large on average, at 403.1 bytes. Given that our crawls are unauthenticated and thus our requests contain significantly fewer cookies/tokens, these are definitely underestimates of average header size. As a concrete example to demonstrate the differences in traffic requirements between polling and WebSockets, consider *tradingview.com*, a currency exchange website. This website uses a single WebSocket connection to push updated currency exchange rates to clients roughly once per second. As WebSocket use cases go, this is not a particularly high update frequency. We found that some WebSockets deliver tens or hundreds of updates per second. However, the bandwidth savings achieved through WebSocket usage here are still significant. With WebSockets, updates require an average of 82 bytes sent across the network (an 8 byte WebSocket header plus a variable sized binary payload averaging 74 bytes). No request is required—the update is delivered at the moment it becomes available on the server. Using our data on HTTP header sizes, we estimate that implementing the same functionality with long polling would require an average of 499 bytes on the network for each update, given that both request and response headers would be required and binary data would need to be base64 encoded at a 4:3 ratio. Therefore, we conservatively estimate that WebSockets offer a decrease in network overhead of at least 417 bytes per client per second. Given that *tradingview.com* is estimated to have approximately 4,835 visitors on their page at a time in September 2019 [10], WebSockets represent a decrease of 16Mbps in required server bandwidth for this relatively benign traffic load.

4 MISCONFIGURATION AND MISUSE

The security community has long known that misconfiguration and unintended use of technologies can lead directly to vulnerabilities [32, 48]. Given the diversity of applications on the web, and the importance they hold for users, it is valuable to periodically measure and understand misconfiguration and misuse of web technologies in the wild. This section looks back at the WebSocket RFC and reflects on how well real-world deployments have complied with some best-practices and recommendations laid out in the original standard.

4.1 Misconfiguration

4.1.1 Checking Origin Headers. Since WebSockets are not restricted by the Same Origin Policy (SOP), a malicious script can use existing cookies for a host to authenticate in a manner similar to cross-site request forgery attacks [27]. Consequently, a WebSocket server should, for many typical use cases, validate the HTTP `Origin` header, which is set by the web browser based on the origin of the script opening the connection. To facilitate this process, major WebSocket libraries such as *socket.io* and *sockjs* provide a specific function for setting the allowed origins [11, 12]. However, the default behavior of these libraries is to allow any origin to access the server.

We measured whether WebSocket servers in the wild are checking `Origin` headers by attempting to connect to each of the WebSocket servers we observed using an arbitrary HTTP `Origin`. To do this, we used a dedicated script as a WebSocket client and used the applicable domain name as the host, and specified arbitrary (incorrect, unrelated) domain as the HTTP `Origin`. We observed that of successfully connect to (and receive data frames from) 74.4% of the 7,624 distinct WebSocket servers we encountered. While this widespread lack of `Origin` header validation in initial WebSocket requests might sound shocking at first. There is no doubt that an overlooked `Origin` header check could jeopardize user security and privacy. However, our analysis showed that the scripts accepting these connections were mostly trackers, web analytics, and advertising networks that were loaded during a website visit. These entities operate more effectively when they provide universal access to their servers and allow all incoming connections in order to attain more visibility over the behavior of users across different websites. This behavior is in line with well-known tracking methodologies such as pixel tracking [30] where fetching a cross-origin request is necessary to load a remote image. Servers are often configured to allow cross origin requests using Cross-Origin Resource Sharing (CORS) to allow the image fetching and complete the tracking process successfully. Trackers and analytics servers often omit a check on the `Origin` header, but applications where security is important to the provider, such as cryptomining and gaming, tend to reject connections without the proper header.

4.1.2 Unencrypted Connections. The large number of tracking and analytics scripts we found emphasizes the fact that

WebSocket connections often transmit sensitive data to remote servers. In this section, we seek to understand whether the adoption of WebSockets aligns with the push in the web community to move the web traffic to TLS-protected channels. WebSockets can use unencrypted (`ws://`) or TLS-protected (`wss://`) connections [2]. However, in all modern browsers, if a website is served over an HTTPS connection, an attempt to open a non-TLS-protected WebSocket connection will fail [8], so unencrypted WebSocket connections can only be initiated by websites who are served over HTTP. Of the 55,805 websites we observed using WebSockets, 438 of them (0.8%) used unencrypted WebSockets by default. We also attempted to create unencrypted connections to each of the WebSocket servers using HTTP WebSockets solely with the goal of locating servers that do not enforce a secure WebSocket communication on the incoming requests. This analysis showed that 14.1% of the servers allowed WebSockets upgrade request over clear text communication channel. In the best case, these are unnecessary services exposed on the Internet, exposing additional attack surfaces without conferring any apparent benefits. In the worst case, they represent a vulnerability in that sensitive user data may be sent across the web unencrypted without users ever knowing.

4.2 Malicious Use

In the course of our study, we found that a significant percentage of WebSocket use in the wild was related to tracking, analytics, and even worse, scams and malware delivery. To be clear, we do not assert that these practices would not be possible without WebSockets. However, WebSockets clearly make these harmful practices easier and more discreet. This section outlines our findings on the darker side of WebSocket use.

4.2.1 Data Leaks. The topic of web tracking has been well studied, and the pervasiveness of trackers and their privacy implications have been extensively documented [26, 31, 34, 37, 42, 43]. Trackers are known to use intrusive techniques to gather information about online users and their behavior patterns. Examples include the use of HTML5 APIs such as Canvas, Battery Status, and Audio context [31] for device fingerprinting [35, 45, 51], cross-device tracking [25], and even exfiltrating user data from unsubmitted forms [55].

We extended our experiments in Section 3 to investigate how WebSockets, as an efficient data transmission mechanism, are used by trackers. Personally Identifiable Information (PII) is a blanket term describing information about users that could be used to trace an individual's identity. PII includes first and last names, email address(es), phone number(s), and IP addresses. We define a PII leak as any instance in which this information is transmitted to a third-party without user consent. Beyond PII, there is a wide variety of information that scripts can collect which might be considered objectionable to users on privacy-related grounds. Session recording, for example, is the process of deep copying the DOM objects and serializing those objects to a specific format (e.g., JSON) for transmission to a remote server. In addition to sending a full snapshot of the DOM tree at the start of a site visit, the JavaScript code periodically records the interaction of the user as a set of snapshots that contain mouse coordinates and keyboard strokes. Session recording has been studied by other researchers [29] and is considered by many to be a serious privacy violation. Session recorders, PII leakers, and other unsavory (but increasingly common) scripts use WebSockets as an efficient way to extract the data they collect.

We find that leaked data in the frames' payloads is usually sent in a structured format (i.e., key/value pairs) such as `password=mypass` or `email=reg@example.com`. Accordingly, we performed simple string matching to identify PII being sent inside of WebSocket payloads. Although values such as username, password, and email address were often not present because this was an automated crawl, we were able to measure PII leakage based on clearly-named keys. Table 3 illustrates the type of information sent over WebSocket channels. While we observed the use of unique identifiers across third-party code, the data frames also included key/-value pairs such as locations and email address. As an example, *https://vapesociety.com*, an online shopping website, loads a script that establishes a WebSocket connection to *zopim.com*—a marketing automation entity. The code sent 21 frames (approximately one frame every two seconds) to the server during the site visit. The exchanged frames contained information about website visitors (e.g., mouse movements, client IP, and geolocation data). Note that PII leakage was not seen in every instance of the third-parties listed in Table 5. For example, *hotjar.com*, one of the most frequently contacted domains, leaked user data in only 79 out of 2,337 cases. It was also common that these third parties received more than one tracking parameter (e.g., visitor ID, phone number). The analysis on the type of PII data sent over WebSockets shows that visitor ID, email address, and IP address were the most common PII data extracted among the five libraries listed in Table 3. We identified 211 unique incidents where PII data was extracted as key/value pairs, and 160 (76%) of these cases contained information about the visitor ID, email address, or IP address of the visiting user.

4.2.2 Web Tracking. Web tracking has become a well-known practice in recent years. The canonical web tracking technique assigns an identifier to the user's browser for a third-party domain, say `tracker.com`, and generates a request to `tracker.com` using that identifier when the user visits a webpage that contains a resource from `tracker.com`. Figure 7 illustrates a real-world example of cross-site web tracking based on WebSockets. The third-party code belonging to *truconversion.com*, a web tracking entity, collects information about device properties, the IP address, and the geographical location of the user. It sends this information (via a WebSocket) to a remote server. The server sends a periodic heartbeat which contains a unique 16 digit user ID to check whether the browser tab is still open on user's machine. We identified 54 websites which were using the same script from *truconversion.com*. We tested all 54 websites using the same

Library	PII	Location	Fingerprint	Session Recording
cux.io	•	•	•	•
beusable.com		•	•	•
hotjar.com	•	•	•	•
inspectlet.com	•	•	•	•
webspectator		•	•	•
colpirio.com	•	•	•	
firecrux.com	•	•	•	•
zopim.com		•	•	

Table 3: Prevalent third-party trackers—Some of the most common third party trackers, along with the types of data they export. We observe that many scripts export data at regular intervals. WebSockets allow them to accomplish this less conspicuously, since they avoid sending frequent HTTP requests, especially in cases like session recording.

browser profile and confirmed that the assigned user ID exchanged over WebSockets was identical in all the websites. If the remote server receives heartbeat responses from different websites for the same user ID at the same time, it allows the tracker to create a list of websites that a user has visited in a specific time period. We extend this experiment by deleting all cookies of the browser profile used in the previous experiment, and run the experiment again while monitoring the assigned user ID of the device. Our analysis showed that clearing cookies would not prevent trackers from identifying the same device because the same user ID was assigned to the visiting device across each of the 54 websites.

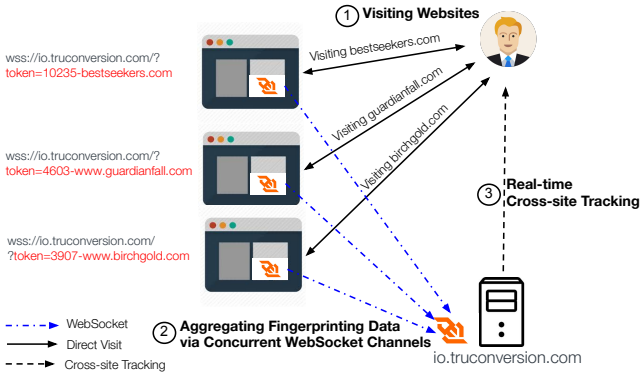

Figure 7: Example: Cross-Site Tracking—An example where WebSockets are used to track users across multiple sites. By correlating user IDs sent at aligning times, a third party script provider can build lists of websites which users have visited.

In this experiment, we observed that *truconversion.com* uses persistent browser-based fingerprinting to make defending against online tracking more difficult. WebSockets make this possible by allowing remote servers to push low-cost heartbeat requests across different websites. Real-time cross-site tracking can have significant monetary value [23] due to the rise of Real-Time Bidding (RTB), in which advertising and tracking companies are incentivized to collaborate in order to exchange real-time data about users and facilitate bidding on impressions [23, 33]. *Bashir et al.* [24] studied this behavior by implementing a simulation of an online ad ecosystem, demonstrating how online tracking and analytics incorporate various techniques to collect users' browsing patterns and use them that in the RTB market. We found five other tracking companies that were using similar techniques for real-time cross-site web tracking.

4.2.3 Exposing Users to Malicious Pages. Prior work [49, 50, 57, 59] has discussed various techniques that adversaries use to distribute malicious links across websites, luring users to visit their scam pages. However, this approach lends itself to rapid identification and removal of these malicious pages, as discussed in prior work on emerging online scams [39, 49]. An alternative for adversaries is to expose users to malicious content via real-time notifications – a form of server to client messaging often built on WebSockets. Open source PushWoosh [9], Google Firebase notification [6], and Amazon Simple Notification Service (SNS) [3] are just examples of implementations which are widely deployed. Using such a service, web applications can send data to thousands of remote user devices with relatively small server-side overhead. Real-time notifications are deployed on top of this service to provide publishers with a flexible notification platform that is supported across a variety of end-user devices. Since WebSockets work on all major browsers [14], they allow for broad code reuse across many browser versions and device types.

The ability to send real-time notifications to remote targets and avoid leaving evidence of malicious activity in website source code allows adversaries to hide from search engines and active web scans by security researchers. We identified 1,466 websites that were using third-party libraries to deliver WebSocket-based notifications. Visitors to the corresponding websites are encouraged to register for real-time notifications to receive special discounts on products or software.

We performed an experiment on 400 websites which used these third-party libraries. Inside of a dedicated virtual machine, we registered for notifications and monitored traffic from these sites for 30 days. Of these 400 websites, we identified 32 cases where push notifications were used to deliver Potentially Unwanted Programs (PUPs), scam pages, adult, or affiliated websites. Across these 32 websites, we logged 123 individual payloads.

Table 4 shows the types of malicious payloads distributed by WebSocket push notifications. While we observed multiple types of malicious practices, a large number of collected samples (52.7%) were distributing PUPs such as Amonetize, Mac Keeper, and TotalAV. While the number of identified cases is not huge, the finding is in line with prior work [50] where the authors analyzed a large number of social engineering and web-based attacks and found that PUPs and extensions

Categories	Dist.
Adult pages	12 (10%)
Affiliate Programs	33 (27%)
Malware	3 (2.5%)
PUPs	42 (34%)
Malicious Extensions	23 (18.7%)
Technical Support Scams	10 (8%)
Total	123 (100%)

Table 4: Malicious payloads delivered by push notifications—The distribution of malicious payloads we observed being delivered during our notification experiment. Across 32 out of 400 websites which delivered unwanted content, we observed 123 total payloads. Among these payloads were scams and malware which could have serious negative impacts on individual users.

Binary Type	Dist.	Type
Installcore	3	PUP
Speed Dial	12	Extension
Mac Keeper	20	PUP
Easy Convert	11	Extension
Search Defender	10	PUP
TotalAV	9	PUP
Flash update	3	Malware
Total	69 (100%)	-

Table 5: List of unique downloaded binaries distributed via WebSocket-based notifications—We observed these downloads as part of our 30-day experiment on WebSocket-based push notifications. While most of these downloads simply cause annoyance for users, there are a handful of examples of actual malware being distributed through these channels.

are the most common forms of malicious payloads. We also found 10 cases where pop-up widgets claimed that the visitor's computer was infected with malware. These websites are entry points to technical support scams, an ongoing problem recently explored by other researchers [49].

5 DISCUSSION AND LIMITATIONS

Many of the canonical difficulties in measuring website functionality apply to this study. In particular, WebSockets are difficult to measure in full for two main reasons. First, initiating WebSocket connections may require specific interactions with a website, such as pressing a "connect to updates" button. Predicting these interactions heuristically, either beforehand or during site visits, is quite difficult, and we do not attempt it in this work. Second, many of the more interesting WebSocket applications sit behind some form of authentication, or simply deeper than the front pages of websites. Measurement of authenticated services is a long standing problem in the community, and finding ways to effectively measure authenticated WebSockets at scale remains an interesting topic for future research. In particular, we believe that incorporating a server-side vantage point into measurements could shed additional light on the value of various real-time technologies on the modern web.

Adoption of HTTP/2.0 is currently at about 45% among the top million websites [7] and rising. It is interesting to consider how this will impact WebSocket usage, given that HTTP/2.0 includes server push, a feature whereby servers can asynchronously send data to clients without an explicit request. WebSocket usage is still much more common than server push. This is not surprising, given that server push is not supported on older desktop browsers such as Internet Explorer, or on common mobile browsers such as iOS Safari (both of which support WebSockets) [4]. WebSockets and server push fill slightly different roles as well. Server push does not offer a truly bidirectional channel, and the data it sends is not always directly accessible to scripts. In fact, in late 2020, Google announced the removal of HTTP/2.0 and gQUIC server push due to high maintenance costs and low usage [17]. Considering all of this, our assessment is that while the proliferation of HTTP/2.0 may replace WebSockets for some specific use cases, it is unlikely to significantly impact the amount of WebSocket deployments in the foreseeable future.

There is clearly room for improvement in terms of best practices in the WebSocket ecosystem, but we do not believe this to be principally (or even primarily) the responsibility of first-party developers. We fear that web developers are often forced to learn how to implement security in WebSockets via online forum posts and other non-official sources. Current documentation for popular third party libraries such as `socket.io` and `sockjs` lacks clear explanations on how to implement security features such as rate limiting, logging, and authentication. Consequently, developers are often left with powerful functionality features but a sparse knowledge base on creating hardened applications. This seems to be a recipe for dangerous deployments, and we encourage code providers to improve documentation related to securing their WebSocket implementations.

This study follows in the footsteps of numerous efforts to understand how new standards and technologies are changing the web. Given the complexity of modern browsers and websites, it is essential that additions to the ecosystem are both motivated by and evaluated with empirical, evidence-based investigations to understand the need for new features and the impact of those features once they have been deployed. Based on our findings, it is clear that the WebSocket API is providing substantial benefits to developers and users across a broad range of applications. However, it is important to constantly weigh benefits of a technology with the negative effects it generates. In Section 4, we presented evidence of less-than-desirable uses of WebSockets, which should be monitored by the security and privacy community going forward. In particular, WebSockets significantly increase the potency

of tracking and analytics scripts, and we highlight them in particular as a subject for continued research.

6 RELATED WORK

Although the WebSocket protocol was added to major browsers almost a decade ago, empirical analyses focusing specifically on the WebSocket ecosystem in the wild remain relatively sparse. *Snyder et al.* [54] measured WebSocket usage over the Alexa Top 10K websites, finding that 5.4% of them used WebSockets, with 64.6% of those usages being blocked by anti-tracking software. *Bashir et al.* [22] showed how trackers and advertisers could use WebSockets to elude ad blockers. In the process, they also measured WebSocket prevalence and found less WebSocket usage, stating that only about 2% of sites used WebSockets. This disagreement on the commonality of WebSockets is notable. Our data shows a larger WebSocket ecosystem, with 6.3% of WebSockets in the Tranco Top 1M using WebSockets. This makes the misuse and vulnerabilities in the WebSocket ecosystem all the more concerning.

In recent years, browser API usage has become a prevalent way to study various phenomena on the web. Researchers have used traces of these browser API calls to forensically reconstruct web-based attacks [47, 57], analyze malicious extensions [21], better understand anti-ad and anti-tracking extensions [54], and more. We see this measurement technique as extremely potent for understanding dynamic phenomena on the web, and we expect researchers to continue to expand the applications of these traces. We utilize the powerful Chrome DevTools protocol interface to drive the browser and gather fine-grained data.

Our work exists in a larger context of research which tracks abuse of emerging web technologies. Recent work has studied online scams [49], scareware [40, 60], PUPs [41, 50, 56], and the identification of risky websites [36, 58]. Online tracking and privacy violations have been studied extensively in recent years [20, 28, 30, 38]. In particular, *Bashir et al.* explored how WebSockets can assist trackers in bypassing ad blockers. We expand on this work by studying a broader set of troubling use cases for WebSockets and offering a methodology to automatically identify abuse of the API. We believe that our work informs web developers and other researchers of the problems with WebSockets, and equips them with techniques to counter malicious usage.

7 CONCLUSION

This study examined the modern real-time web ecosystem, offering an up-to-date picture of how WebSockets and other real-time protocols are currently being used in the wild. Reflecting on the goals of the WebSocket protocol designers a decade ago, we provided an assessment of the successes and failures of these technologies from an empirical perspective. We compared WebSocket use with the use of other real-time solutions, showing tangible benefits of switching to WebSockets and highlighting some remaining room for improvement. When websites do adopt WebSockets, they should be mindful of best practices which will create safer, more stable applications. We show that web developers are often failing to follow these practices and, in some cases, using WebSockets in questionable or malicious ways. We believe WebSockets provide significant overall benefit to users on the web, and we advocate for expanded adoption, along with improved documentation.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their thoughtful feedback.

REFERENCES

[1] 2011. rfc6202: Known Issues and Best Practices for the use of Long Polling and Streaming in Bidirectional HTTP. https://tools.ietf.org/html/rfc6202. (April 2011).
[2] 2011. rfc6455: The WebSocket Protocol. https://tools.ietf.org/html/rfc6455. (December 2011).
[3] 2019. Amazon Simple Notification Service. https://aws.amazon.com/sns/. (September 2019).
[4] 2019. Can I Use: Push API. https://caniuse.com/#feat=push-api. (October 2019).
[5] 2019. Can I Use: Server-Sent Events. https://caniuse.com/#feat=server-sent-events. (October 2019).
[6] 2019. Firebase Cloud Messaging. https://firebase.google.com/docs/cloud-messaging/. (September 2019).
[7] 2019. HTTP/2 + Push Adoption Measurements. https://http2.netray.io/stats.html. (September 2019).
[8] 2019. Mixed-content WebSockets. https://bugs.chromium.org/p/chromium/issues/detail?id=85271. (September 2019).
[9] 2019. PushWoosh: An Open Source Push Notification. https://pushwoosh.com/. (September 2019).
[10] 2019. SimilarWeb: Traffic Analysis for tradingview.com. https://www.similarweb.com/website/tradingview.com. (September 2019).
[11] 2019. Sockjs-node. https://github.com/sockjs/sockjs-node. (September 2019).
[12] 2019. The Socket.io: real-time, bidirectional and event-based communication. https://socket.io/docs/. (September 2019).
[13] 2019. Web Sockets. https://caniuse.com/#feat=websockets. (October 2019).
[14] 2019. The WebSocket API (WebSockets). https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. (September 2019).
[15] 2020. chromedp. https://github.com/chromedp/chromedp. (May 2020).
[16] 2020. The HTTP Archive. https://httparchive.org. (May 2020).
[17] 2020. Intent to Remove: HTTP/2 and gQUIC server push. https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/vOWBKZGoAQAJ. (November 2020).
[18] 2020. Stop Connection Timeouts from Happening. https://support.mozilla.org/en-US/questions/998088. (May 2020).
[19] 2020. WebShrinker Website Categorization API. https://webshrinker.com. (May 2020).
[20] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. [n. d.]. FPDetective: dusting the web for fingerprinters. In *20th ACM Conference on Computer and Communications Security (CCS)*.
[21] Sajjad Arshad, Amin Kharraz, and William Robertson. 2016. Identifying Extension-based Ad Injection via Fine-grained Web Content Provenance. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.
[22] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. 2018. How Tracking Companies Circumvented Ad Blockers Using WebSockets. In *ACM Internet Measurement Conference (IMC)*.
[23] Muhammad Ahmad Bashir, Sajjad Arshad, William Robertson, and Christo Wilson. 2016. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium (USENIX Security)*.
[24] Muhammad Ahmad Bashir and Christo Wilson. 2018. Diffusion of user tracking data in the online advertising ecosystem. *Proceedings on Privacy Enhancing Technologies (PETS)* (2018).

[25] Justin Brookman, Phoebe Rouge, Aaron Alva, and Christina Yeung. 2017. Cross-Device Tracking: Measurement and Disclosures. *Proceedings on Privacy Enhancing Technologies* 2017, 2 (2017), 133–148.

[26] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. 2016. An Empirical Study of Web Cookies. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*. 891–901.

[27] Jianjun Chen, Jian Jiang, Haixin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. 2018. We Still Don't Have Secure Cross-Domain Requests: an Empirical Study of CORS. In *27th USENIX Security Symposium (USENIX Security)*.

[28] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. [n. d.]. The Web's Sixth Sense: A Study of Scripts Accessing Smartphone Sensors. In *25th ACM Conference on Computer and Communications Security (CCS)*.

[29] Steven Englehardt. [n. d.]. No boundaries: Exfiltration of personal data by session-replay scripts. https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/. ([n. d.]).

[30] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *23rd ACM Conference on Computer and Communications Security (CCS)*.

[31] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1388–1401.

[32] Sascha Fahl, Yasemin Acar, Henning Perl, and Matthew Smith. 2014. Why eve and mallory (also) love webmasters: a study on the root causes of SSL misconfigurations. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 507–512.

[33] Arpita Ghosh and Aaron Roth. 2011. Selling Privacy at Auction. In *12th ACM Conference on Electronic Commerce (EC)*.

[34] Phillipa Gill, Vijay Erramilli, Augustin Chaintreau, Balachander Krishnamurthy, Konstantina Papagiannaki, and Pablo Rodriguez. 2013. Best Paper – Follow the Money: Understanding Economics of Online Aggregation and Advertising. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC '13)*. ACM, New York, NY, USA, 141–148. https://doi.org/10.1145/2504730.2504768

[35] Alejandro Gómez-Boix, Pierre Laperdrix, and Benoit Baudry. 2018. Hiding in the Crowd: An Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *Proceedings of the 2018 World Wide Web Conference (WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 309–318.

[36] Luca Invernizzi and Paolo Milani Comparetti. 2012. Evilseed: A guided approach to finding malicious web pages. In *33rd IEEE Symposium on Security and Privacy (IEEE S&P)*.

[37] Sakshi Jain, Mobin Javed, and Vern Paxson. 2015. Towards Mining Latent Client Identifiers from Network Traffic. *PoPETs* 2016 (2015), 100–114.

[38] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *17th ACM Conference on Computer and Communications Security (CCS)*.

[39] Amin Kharraz, , William Robertson, and Engin Kirda. 2018. Surveylance: Automatically Detecting Online Survey Scams. In *2018 IEEE Symposium on Security and Privacy (SP)*.

[40] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (RAID)*.

[41] Platon Kotzias, Leyla Bilge, and Juan Caballero. 2016. Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services. In *25th USENIX Security Symposium (USENIX Security)*.

[42] Balachander Krishnamurthy, Delfina Malandrino, and Craig E. Wills. 2007. Measuring Privacy Loss and the Impact of Privacy Protection in Web Browsing. In *Proceedings of the 3rd Symposium on Usable Privacy and Security (SOUPS '07)*. ACM, New York, NY, USA, 52–63.

[43] Balachander Krishnamurthy, Konstantin Naryshkin, and Craig Wills. 2012. Privacy leakage vs. Protection measures: the growing disconnect. (05 2012), 123–144.

[44] Deepak Kumar, Zane Ma, Zakir Durumeric, Ariana Mirian, Joshua Mason, Michael Bailey, and J. Alex Halderman. 2017. Security Challenges in an Increasingly Tangled Web. In *26th International World Wide Web Conference (WWW)*.

[45] P. Laperdrix, W. Rudametkin, and B. Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *2016 IEEE Symposium on Security and Privacy (SP)*. 878–894.

[46] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23386

[47] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2018. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *25th Network and Distributed System Security Symposium (NDSS)*.

[48] Yang Liu, Armin Sarabi, Jing Zhang, Parinaz Naghizadeh, Manish Karir, Michael Bailey, and Mingyan Liu. 2015. Cloudy with a chance of breach: Forecasting cyber security incidents. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 1009–1024.

[49] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. 2017. Dial One for Scam: A Large-Scale Analysis of Technical Support Scams. In *24th Network and Distributed System Security Symposium (NDSS)*.

[50] Terry Nelms, Roberto Perdisci, Manos Antonakakis, and Mustaque Ahamad. 2016. Towards Measuring and Mitigating Social Engineering Software Download Attacks. In *25th USENIX Security Symposium (USENIX Security)*.

[51] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 541–555. https://doi.org/10.1109/SP.2013.43

[52] V. Pimentel and B. G. Nickerson. 2012. Communicating and Displaying Real-Time Data with WebSocket. *IEEE Internet Computing* 16, 4 (2012), 45–53.

[53] D. Skvorc, M. Horvat, and S. Srbljic. 2014. Performance evaluation of Websocket protocol for implementation of full-duplex web streams. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1003–1008.

[54] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. 2016. Browser feature usage on the modern web. In *16th ACM Internet Measurement Conference (IMC)*.

[55] Oleksii Starov, Phillipa Gill, and Nick Nikiforakis. 2016. Are you sure you want to contact us? quantifying the leakage of pii via website contact forms. *Proceedings on Privacy Enhancing Technologies* 2016, 1 (2016), 20–33.

[56] Kurt Thomas, Juan A. Elices Crespo, Ryan Rasti, Jean-Michel Picod, Cait Phillips, Marc-André Decoste, Chris Sharp, Fabio Tirelo, Ali Tofigh, Marc-Antoine Courteau, Lucas Ballard, Robert Shield, Nav Jagpal, Moheeb Abu Rajab, Panayiotis Mavrommatis, Niels Provos, Elie Bursztein, and Damon McCoy. 2016. Investigating Commercial Pay-Per-Install and the Distribution of Unwanted Software. In *25th USENIX Security Symposium (USENIX Security)*.

[57] Phani Vadrevu, Jienan Liu, Bo Li, Babak Rahbarinia, Kyu Hyung Lee, and Roberto Perdisci. 2017. Enabling Reconstruction of Attacks on Users via Efficient Browsing Snapshots. In *24th Network and Distributed System Security Symposium (NDSS)*.

[58] Thomas Vissers, Wouter Joosen, and Nick Nikiforakis. 2015. Parking Sensors: Analyzing and Detecting Parked Domains. In *21st Network and Distributed System Security Symposium (NDSS)*.

[59] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *24th International Conference on the World Wide Web (WWW)*.

[60] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. 2014. The dark alleys of madison avenue: Understanding malicious advertisements. In *14th ACM Internet Measurement Conference (IMC)*.