12-2022

# Creating a Better Browser Fingerprint

Scott Reiling
*University of Nebraska at Omaha*, sreiling.sr@gmail.com

# Creating a Better Browser Fingerprint

A Thesis

Presented to the

School of Interdisciplinary Informatics

and the

Faculty of the College of Interdisciplinary Science & Technology

University of Nebraska

In Partial Fulfillment of the Requirements for the Degree

M.S. Cybersecurity

University of Nebraska at Omaha

By

Scott Reiling

December 2022

# Creating a Better Browser Fingerprint

Scott Reiling, M.S. Cybersecurity

University of Nebraska at Omaha, 2022

Web browser fingerprinting is used to analyze client behavior through retrieval of browser attributes unique to the user's browser, network and hardware profile. Third-party trackers are prevalent on the top Alexa sites and use JavaScript to retrieve and store user machine information in a stateless fashion. Stateless fingerprinting is performed through acquisition of client machine specifiers through an embedded JavaScript, which then forwards the information to a server. The client information is purportedly used to provide tailored advertising and enhance the browsing experience. However, the depth of captured client information often extends into the realm of personally identifiable information. The user is often unaware of privacy issues and how their information is disseminated for profit, or the risk of such data being used by hackers to exploit divulged vulnerabilities.

We review fingerprinting techniques from previous works that delineate seminal methods and countermeasures, and present a novel fingerprinting JavaScript that measure over 200 Windows and Navigator object properties. The results reveal new parameters that can be used to generate unique user identifiers, and accurately track individual browsing behavior. These findings may be used by developers of anti-tracking software to improve efficacy and preserve individual privacy.

## Table of contents

# 1. Introduction

Web browser fingerprinting is a technique used to analyze client behavior through retrieval of characteristics concerning the user’s software and hardware profile through web browser Application Programming Interfaces (APIs). Third-party tracking scripts are the primary means of fingerprinting, which use algorithms to retrieve and store such parameters in a stateless or stateful fashion.

Currently, there are two general methods used to do fingerprinting; these use cookies, which are stored locally on a client machine, and browser fingerprinting, which uses hyper-text transfer protocol (HTTP) requests to measure browser and machine properties. Both are typically used simultaneously to obtain an accurate fingerprint, even in presence of user-installed obfuscation add-ons meant to defeat such techniques.

Cookies are text files websites store locally on a user’s computer and are thus considered a “stateful” tracking tool; they are written with permission granted by the browser settings. These files are advertised to enhance the browsing experience by saving settings such as a session or user ID, so that upon an ensuing visit, the user is spared the process of re-authentication. Another benign use of cookies is to provide personalized content in the form of advertisements tailored toward a user’s needs.

Nevertheless, the same local machine parameters contained within a cookie may also be utilized to track users across websites. If an individual accesses two or more websites hosting the same third-party tracking scripts from the same advertising network, the scripts can check for a local cookie set by one of the network scripts, and log the visit to its server.

Browser fingerprinting gathers data that can be used to identify an individual's web surfing behaviors even if the user takes basic precautions, such as deleting cookies, clearing the browser cache, and using a browser's stealth mode (e.g. Firefox's "private window"). The fingerprints are created using data that websites can gather from one's browser APIs, which is then stored on a server, unlike cookies that are stored locally on client machines. As such, the stealthily gathered information is untouchable by the user. This fingerprinting mechanism is "stateless", in that the information is collected via a JavaScript [1]. The JavaScript may be part of the website being visited, in which case it is considered a "first-party" tracker. However, these websites may load "third-party" trackers from another domain, whose scripts may also access the browser's APIs to garner data on its properties. Aside from traditional HTTP header request information used for fingerprint attribute collection, the recent proliferation of HTML5 allows for Canvas APIs to divulge even more specifying information such as font metrics, network IP addresses, and audio signal processing output [2].

Cookie and fingerprint-based surveillance obtains client identity through implementations of JavaScript. Most third-parties use both cookies and fingerprinting to monitor web behaviors, often in a synergistic manner. The simplest application combining the two is cross-referencing the user ID in a cookie with the hash value of all the information cumulated from a JavaScript running in the background [3]. Browser fingerprinting is more effective than cookie-based identification, if only because it does not require permission to store a local file. There are thousands of different fingerprinting scripts that are effective in fingerprinting the popular browsers to the tune of 83% while measuring only eight attributes [1], to 90% with the values of seventeen attributes [2]. The percentage of success has remained high due to successively advanced iterations of tracking applications that detect anti-tracking extensions, and work synergistically with cookies to verify identity, regenerate deleted cookies [4], and mask their own processes to appear benign [5].

The discussion of browser fingerprinting starts with a description of fingerprinting attributes, many of which are DOM (Document-Object Model) element values that are extracted via DOM APIs. The DOM is a logic tree structure made of node objects that may be containers, functions, and fields. The “navigator” and “screen” objects within the DOM yield useful information such as screen resolution, browser version, OS version, and the installed browser fonts.

Browser plugins and extensions are also integral to fingerprinting algorithms, as they provide a different attack-surface for enumeration of potential fingerprint fields. While HTML5 did away with the requirement of Flash Player plugin installation within a browser, it is still widely used, and the most exploited plugin for fingerprinting. Eckersley used Flash to show the installed fonts using a Flash API that reveals this information [1]. Other Flash APIs give OS and screen info, independent of the navigator DOM object. Plugins are similar to extensions in that they extend browser functionality. Plugins differ because they aren’t enumerable using browser APIs; yet, their presence can be detected secondary to their side-effects. In other words, if they are anti-trackers, they can be revealed according to the information they mask.

The evolution of browser fingerprinting is only inferred from experts who have raised concerns as the web has morphed into its current incarnation. The dearth of mechanistic information concerning fingerprinting is consequent to commercial fingerprinting scripts being proprietary technology, thus preventing development of specific countermeasures. Nevertheless, enough literature exists to highlight advances in the technology, as seen in figure 1.

In this work, we present a novel project aimed at identifying new browser parameters that have not been used to identify clients during browsing sessions. The unveiling of these variables will be useful in devising a proactive defense against future third-party tracking applications. We first discuss, in section 2, the background of fingerprinting techniques. We show various fingerprinting techniques by exploring previous works that delineate fingerprinting methods, and the countermeasures devised to defeat them.

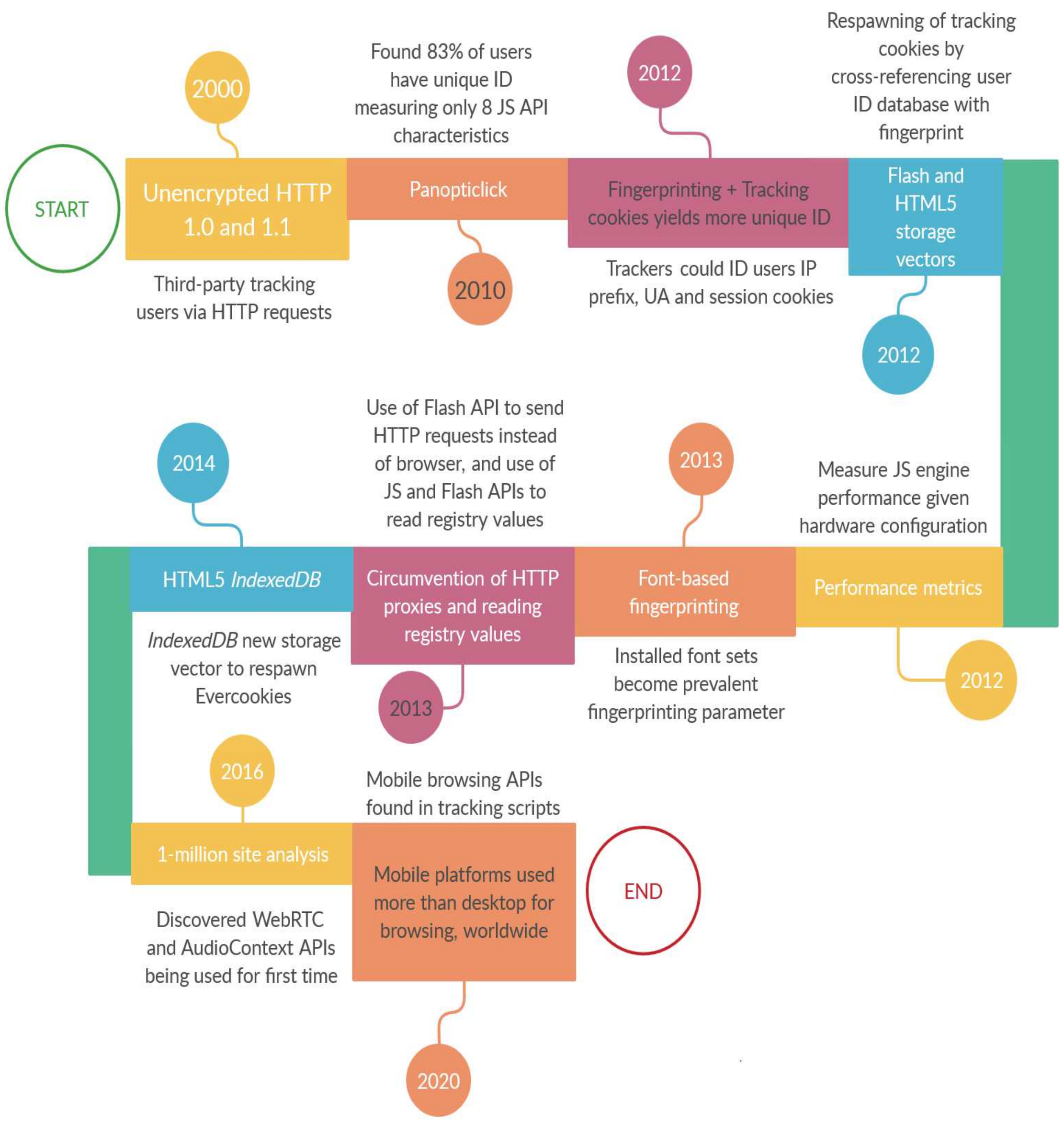

START
2000
Unencrypted HTTP 1.0 and 1.1
Third-party tracking users via HTTP requests
Found 83% of users have unique ID measuring only 8 JS API characteristics
Panopticlick
2010
2012
Fingerprinting + Tracking cookies yields more unique ID
Trackers could ID users IP prefix, UA and session cookies
Respawning of tracking cookies by cross-referencing user ID database with fingerprint
Flash and HTML5 storage vectors
2012
2014
HTML5 *IndexedDB*
*IndexedDB* new storage vector to respawn Evercookies
Use of Flash API to send HTTP requests instead of browser, and use of JS and Flash APIs to read registry values
Circumvention of HTTP proxies and reading registry values
2013
2013
Font-based fingerprinting
Installed font sets become prevalent fingerprinting parameter
Measure JS engine performance given hardware configuration
Performance metrics
2012
2016
1-million site analysis
Discovered WebRTC and AudioContext APIs being used for first time
Mobile browsing APIs found in tracking scripts
Mobile platforms used more than desktop for browsing, worldwide
2020
END

Figure 1. Flow diagram demonstrating the timeline for browser fingerprint advancements.

## 2. Background

In this section we present a synopsis of the works that detail fingerprinting techniques from its earliest detection within browser requests, through its incorporation of more complex methods such as cross-site scripting and re-spawning cookies.  The significance of fingerprint evolution lies within use of protocols that are integral to the network and application layers, which can’t be disabled.  Therefore, an examination of known fingerprinting utilizations is critical to development of preventative tools discussed at the end of this section.

## *2.1 Previous Work*

The earliest publication on privacy issues discussed the information available to servers via HTTP v1.0 and v1.1 requests [6], well before coining of the term, “browser fingerprinting”.  The article described how entities could track user history via their HTTP cache control requests.  It wasn’t until 2009 that research into the prevalence of web trackers was conducted; Krishnamurthy and Willis found that between 2005 and 2008, the incidence of third-party trackers in the top Alexa sites increased from 40% to 70% [7].

The first seminal research paper referencing the mechanism of HTTP request fingerprinting investigated the effectiveness of a fingerprinting algorithm that only measured eight browser characteristics [1].  They constructed an algorithm based on fingerprinting algorithms to “quantify how much of a privacy problem fingerprinting may pose”, and found that only 17% of individuals visiting their Panopticlick site could *not* be fingerprinted.  They demonstrated that API requests to Adobe Flash and the Java Virtual Machine, which the major browsers had installed by default, were culpable.

The combination of fingerprinting and tracking cookies reinforce the process of surveilling browsing behavior, as shown by Yen et al [8].  They collected a data set of millions, and concluded that trackers could identify users using the IP prefix with a user agent string to the tune of 80% accuracy, which increased to 88% in addition to one-time session cookies.

Roesner, Kohno and Wetherall exhibited the concept of spying with stateless fingerprinting and stateful cookies one step further, by classifying trackers into five categories [9]. One of those types utilized novel mechanisms by making use of third-party cookies that are updated according to cookies stored locally, via HTML5s *Local Storage* API and Flash LSO objects. The latter allowed for re-spawning of tracking cookies through Flash APIs that cross-referenced the server user ID database with the user's stateless fingerprint. Another technique was to use cross-site trackers that forced user browsers to redirect to a different domain via a popup, so that the third-party domain would become a first-party domain, and thus be able to set its own cookie. Yet another category of trackers made use of the two aforementioned approaches.

Some trackers took a different approach to stateless tracking by invoking a JavaScript algorithm to measure performance of the JS engine in performing certain APIs, which yielded consistent hardware metrics [10]. They were capable of accurately detecting the browser and its version; the operating system, and the CPU only with performance metrics.

Acar et al emphasized the importance of font-based fingerprinting being an integral part of tracking JavaScripts, as represented by prevalence of BlueCava scripts [11]. Their research showed 13 different font-probing scripts, of which BlueCava's was present in 250 of the top 500 Alexa sites. BlueCava queried font sets based on retrieval of the user's OS, and was able to dynamically inject itself into the host page, and then remove itself after collecting a fingerprint. The injection and removal of the script was a novel technique to evade detection.

Concurrently, Nikiforakis et al released a study on the top three commercial fingerprinting vendors relating novel capabilities, including code providing a backup font metric measurement, in case Flash was not present due to being phased-out by HTML5 and Microsoft's Silverlight [12]. The backup font code block used CSS to attempt rendering of fonts. The second novelty lay in the ability of these apps to execute Flash APIs to send an HTTP request instead of the browser, which circumvented any HTTP proxies meant to hide the user IP address. Finally, they exposed two vendors, BlueCava and ThreatMetrix, who were leveraging the capacity of Java and Flash browser plugins to read Windows registry values for OS type and installation date. These registry value strings alone were enough to provide a strong fingerprint.

The creation of "Evercookies" had already been unearthed by analysis of client Flash storage vectors after clearing browser cookies, who then revisiting sites with trackers that used Flash APIs to restore Flash  tracking cookies [13]. A new storage vector, HTML5s *IndexedDB,* was identified as another reservoir for respawning these cookies. The *IndexedDB* vector was used to cross-reference third-party databases to respawn Flash cookies, which in-turn was used to restore HTTP cookies [4]. An added complexity was seen when third-party vendors synchronized their user-ID databases to help one-another respawn these Evercookies, and thus share their collective data.

In 2016, Englehardt and Naranayan utilized a measurement analysis of tracking on one-million sites and found two APIs used by the top ten scripts that had not previously been reported [2]. The first API, WebRTC, is a Javascript peer-to-peer function that collects the IP addresses of all available peers in a target's network.  The second was the AudioContext interface, which returned a digital signal processing metric to be incorporated into the fingerprint.

Applications for recording browser behavior also extend into the sphere of mobile devices.  According to Statcounter's website, mobile web browsing is more commonly used than the desktop platform, as of the time of this writing [14].  Approximately 56% of trackers incorporate code to operate on mobile and desktop platforms, and do so by exploiting mobile-specific APIs, such as a phone's accelerometer and gyroscope [15].  These APIs are combined with typical JavaScript APIs and HTTP cookies to make mobile fingerprints even more unique than their desktop counterparts.  The implication is that mobile tracking may be more deleterious to user privacy for this reason, and also due to mobile devices affording the geolocation of individuals.

The latest approach heretofore unseen is the use of CNAME cloaking. CNAME refers to the canonical name record for the pages of domains within DNS.  Third-party servers rotate the CNAMEs of their subdomains so as to circumvent blacklists created by anti-tracking plugins, as one mode of masking. However, another more nefarious procedure is to mask their subdomains as one of those of the first-party domain the user is visiting, so as to be recognized as a first-party tracker.  While CNAME cloaking is does not fall under the subjectmatter of this work, it is worth mentioning because the elevation of third-party tracking scripts to first-party privilege allows for greater flexibility in collecting client browser and hardware attributes.

## *2.2 Evolution of Fingerprinting*

Fingerprinting continues to evolve as browsers become more complex, and as the variety of plugins and extensions grows, so will the methods for creating unique fingerprints, and surveyance of an individual's behavior. While our experimental focus is on fingerprinting, one cannot ignore its synergy with stateful (cookie) based tracking, and any discussion herein of its erosive effects on user privacy and security implicitly accounts for both modalities. Approximately five-billion people world-wide use the internet, with the highest penetration occurring within North America and Europe [14]. Benign uses for fingerprinting exist such as confirming a user's identity to prevent fraud, or to tailor ads towards a person's tastes. Yet, the financial incentive to advance this technology stems from the market for PII that extends into the realms of advertising, product sales, local law enforcement, and nation-state surveillance [12].

The malicious ramifications to the user include divulgence of browser history; the collection of the user's browser and hardware state, and the possible exploitation of older browser version vulnerabilities. The first item is an infringement upon one's privacy; an individual may not want her or his browsing history to be known for myriad reasons. The last two can be employed by hackers to write scripts that compromise local clients and even networks, as seen in a Google Chrome vulnerability patched in 2019, which allowed the hacking group “WizardOpium” to embed a zero-day exploit to gain elevated privileges on vulnerable machines [16]. A different vulnerability allowed hackers to exploit a flaw in cookie-handling, consequently opening secure HTTPS connections to man-in-the-middle attacks [17]. Such vulnerabilities have been found, as seen in the Common Vulnerabilities and Exposures (CVE) database [18]. We do not wish to expound upon every possible repercussion to an individual, but only to reinforce potential harm with regards to user privacy and security.

In order to systemically apply mitigations to fingerprinting algorithms, researchers have contrived to detect their modes of operation, especially over the past decade. Although there are many significant contributions within this time period, three key works have added greatly to the understanding of fingerprinting through building custom-coded browser extensions and autonomous platforms. These stem from Roesner et al, who made browser extensions to classify third-party trackers[9]; Acar et al who wrote the FPDetective script to detect fingerprinting heuristically [11], and Englehardt and Naranayan, who made OpenWPM site surveying script [2].

### *2.3 Tracker Classification:*

Roesner’s group focused on detecting tracking mechanisms that assigned unique identifiers, which was important because previous research explored trackers that only inferred client identity[9]. Their extension revealed how real tracking code interacted with the browser state. This allowed them to sort tracking behaviors into five categories: 1) Trackers that perform third-party analytics for a site, and can only track user within a site; 2) Trackers that use third-party storage for track across sites; 3) Cross-site trackers that force users to its domain via redirects and popups; 4) Trackers that rely on other trackers in categories 1,2 or 5, to receive data, and 5) Cross-site trackers visited directly by the user. They found only behaviors 2 and 5 are found stand-alone while most trackers displayed a combination of behaviors. The classification system provided a structure for future studies to investigate these third-party scripts. The FPDetective framework was made to detect trackers without the aid of a whitelist [11]; in other words it detected changes in browser function outputs to ascertain their presence. It revealed new scripts and Flash objects used in the top five-hundred Alexa sites, and was able to detect scripts injecting and then removing themselves, to evade detection. The work provided a reliable heuristic tool for others to detect malicious scripts that collect stateless data, as well as a means to reverse-engineer those scripts based on the probable function calls made to JavaScript APIs.

The OpenWPM markedly built upon FPDetective by supporting stateful and stateless measurements from tracking scripts while allowing researchers to implement their own scripts in a modular fashion to monitor APIs of their interest. OpenWPM also provided a plugin for automated crawling, easing the process of data collection by eliminating the need for additional code. Finally, OpenWPM integrated two other measurement points aside from that provided by the browser extension, which included a network proxy, and a monitor for the hard drive state. Engelhardt and Naranayan's OpenWPM is an open source tool they provided to developers of privacy applications, and has often been used as a platform for such research [4].

*2.4 Prevention:*

Prevention of fingerprinting is an endeavor requiring user diligence and the innovation of anti-tracking software developers. Most users learn to clear cookies, yet even that may be an identifier that enhances a fingerprint when compared to the majority of individuals who do not; regardless, respawning Evercookies renders it a futile undertaking. Another step is to use a proxy, but in the face of JavaScript APIs to Flash and HTML5 queries regarding network IPs, proxies are ineffective [8]. VPN services have multiple servers (with different IPs), however, tracking scripts may also account for them. Any alteration to the browser hat returns a false static value is still fingerprintable.

Individuals must take comprehensive measures to maintain web browsing privacy. Disabling Flash is an important step thereto, because it would prevent cross-checking of object values such as the user agent and IP address against browser API call results.

Laperdrix at al estimated if Flash were absent, it would reduce the number of unique fingerprints by 13% [3]. They also calculated an additional 8% reduction if HTTP headers were standardized across browsers, since each browser vendor and version has header peculiarities.

Randomization, standardization, and blocking tools are also a necessity. For instance, an extension that randomized one's user agent string with each web crawl would reduce the linkability of that person’s current browsing session, with previous sessions [4].  The randomization would have to be applied to numerous HTML objects to increase the likelihood of trackers recognizing a “new user” upon a subsequent crawl.

Masking the location of local storage vectors for HTML5 and Flash (if present) is suggested to prevent Evercookies from reappearing [3]. Blocking scripts is more difficult because it requires code tailored towards functions executed within the script; that knowledge is usually proprietary. Standardization is an attractive option because a generic response for fingerprinting attributes would render user IDs useless. Unfortunately, this would require browser vendor collaboration and agreement upon a protocol, and would also break many websites that rely on fingerprinting to augment browsing. It would also affect ad revenue.

One of the earliest defenses against fingerprinting involved manipulation of HTTP request structures through HTTPOS, which obfuscated the TCP MSS and window resolution parameters, thus altering packet sizes [19]. HTTPOS split HTTP requests into partial requests and used a pipelining technique to execute incoming requests in a concurrent manner. Cai et al improved upon this by implementing a modified BUFLO algorithm that reduced overhead, responded to HTTP flow requests, and obscured application thread execution times metrics [20].

Roesner et al constructed ShareMeNot, a Firefox add-on that blocks cookies from third-party requests to trackers that committed forced redirects [9].

This allowed for users to still interact with widgets linking Facebook profiles to a given web page when a user clicked on the “Like” button, while disallowing tracking, otherwise.

Fifield and Egelman proposed a defense against font-based fingerprinting via shipping a standard set of fonts with each browser, without allowance for any future modification to the set [21]. They calculated this would reduce halve the entropy given by font-based fingerprinting metrics.

However, since browsers are unlikely to be standardized across vendors, others have continued to develop applications that block or deceive fingerprinting apps. PriVaricator was created to defeat fingerprinting by feeding false telemetry to trackers [22]. PriVaricator is instrumented on the browser, and subtly alters DOM elements to introduce a degree of uncertainty so that third-party fingerprinters can’t link client IDs. Similarly, FP-Block prevents cross-site tracking by generating a different “web identity” for a user, for each domain upon different visits [23]. It alters elements of the browser, such as the user agent, browser name and vendor; the system and user languages; the OS and CPU classes; the screen resolution and color depth, and the time zone.

Taking the results of these works as a whole, obfuscation at the network, transport, link, and application layers are crucial to avert tracking. Other works emphasize masking IPs, TLS session cookie IDs, and MAC addresses because each of them tend to provide high entropy values, thus raising the chance for a unique fingerprint [3], [5].

# 3 Prior Approaches to Fingerprinting

Our work builds upon previous methods by including a wider and more diverse set of collected elements for more precise and accurate fingerprinting. We also identify new elements that could potentially be used by third-parties to track user behaviors. The following sections elaborate established techniques with subsequent discussion of how our project improves upon them.

*3.1 Previous techniques to identify key fingerprint elements:*

The first large-scale examination of fingerprinting elements was conducted by Eckersley, who looked at the effectiveness of browser fingerprinting algorithms through sampling 470,000 browsers of informed clients to the Panopticlick website [1]. He found that 83.6% of visitors had a unique fingerprint, with an increase to 94.2% for those with Adobe Flash or Java Virtual Machine. The algorithm collected results of HTTP and AJAX requests that included the User Agent, Cookies Enabled, screen resolution, plugin lists, and font lists. The results of which were concatenated into a fingerprint that also included a hash of the visiting IP address.

The results of Eckersley were seminal because they demonstrated the stability in identifying return visitors when combining the fingerprint with a locally stored cookie, even if one of the identifiers (e.g. screen resolution) had changed since the previous visit. Eckersley also showed that each updated version of a plugin or extension produced a different output string, which significantly contributed to the uniqueness of a fingerprint.

Eckersley's work was continued by a group who analyzed the amount of information given by commonly queried DOM elements [8]. They used data sets comprised hundreds of millions of users from Hotmail and Bing searches, which showed that HTTP user-agent strings identified 60%-70% of hosts within a given dataset, and up to 80% when IP prefixes were included. The authors also calculated 88% of users were accurately marked upon repeat visits with the use of one-time cookies, despite having 33% of them clear their cookies, or browse in private mode.

Mowery et al delved further into browser fingerprinting through delineation of two techniques that involve use of JavaScript [10]. They built upon Eckersley's Panopticlick project by adding descriptors yielding the performance measures for operations within the JavaScript interpreter, through execution of a set of 39 functions via a customized version of V8 and SunSpider benchmark platforms. They found the results could not only distinguish between browser versions, but also hardware architecture and installed components, such as the CPU and graphics unit. The other technique probed for entries into NoScript whitelist; NoScript was a Firefox extension that blocked certain webpages from running scripts within the browser. Ironically, the whitelist provided a fingerprint of the client's list of visited websites, and thus enhanced the profile of browsing behavior.

Cai et al constructed a web page fingerprinting attack that circumvented browser extensions masking HTTP packet header information [20]. They were even able to identify web pages loaded through an SSH tunnel, with 90%

accuracy, and furthermore identified web pages loaded through TOR with 80% accuracy. This was possible even when packet size information was removed within the header.

Users may employ HTTP proxies to hide their true IP address, which aids in cloaking their identities from being fingerprinted. Nikiforakis et al analyzed three third-party browser fingerprinting scripts that circumvented such proxies to reveal an IP address [12]. The ActionScript tracker defeated the proxies by querying Flash to contact the third-party host directly. In addition, these scripts had fall-back mechanisms to detect installed fonts in case Flash was absent, and did so in a browser-specific manner, since the most popular apps (e.g. Chrome, IE, and Firefox) have specific DOM function calls to query available fonts. The scripts attempted to delete, add, and modify custom DOM containers with *navigator* and *screen* objects to identify applications and hardware installed on a client machine. Execution of these scripts returned metrics indicating even more information about the browser and hardware versus if Flash were present on the client machine.

When above techniques are combined with cookie synching, even periodic clearance of local cookies doesn’t defeat re-spawning of individual tracking identifiers. As Acar et al found, IDs are re-spawned by different tracking domains that communicate IDs to one-another so that even after cookies are cleared, these domains can merge records of pre and post-clearance browsing logs [4]. They further found that HTTP cookies can resurrect Flash cookies, and vice-versa.

Laperdrix, Rudametkin and Baudry used the AmIUnique.org fingerprinting site to collect 118,934 fingerprints, and found 17 attributes could accurately identify each of them, and developed a JavaScript using them to fingerprint [3]. Their results demonstrated its effectiveness for mobile platform fingerprinting with a unique identifier rate of 81%, despite mobile devices lacking plugins and font-sets within their browsers.

Some HTTP and DOM query returns may change between initialization of browser instances for the same client. FP-STALKER is an implementation of an algorithm that can link instances from the same user, even if those instances yield slightly different fingerprints [5]. Vastel et al collected 98,598 fingerprints from 1905 browser instances, and found that FP-STALKER is able, on average, to link browser fingerprints from the same user, for 51 days.

### *3.2 Detection of Browser Fingerprinting*

Detecting browser fingerprinting is important in the evolution of browser defenses that rely upon modular execution, so as to reduce computational overhead. Conventional defenses include third-party cookie blocking add-ons, Do Not Track, client-side browser state clearance, pop-up blockers and private mode browsing. None of these are secure against trackers on their own, or in combination. An example of why is shown with Roesner et al, who developed a client-side algorithm called TrackingTracker that was outfitted within browsers, for detecting third-party trackers based on how they change browser parameters [9]. They detected over 500 unique trackers and discovered the top Alexa pages all had multiple trackers. The behaviors of these trackers fell into a combination of five categories involving analytics, storage, forced redirection, referrals (from one third-party tracker to another), and personal, which is a tracking site directly visited by a client.

Acar et al made FPDetective, a framework to detect and analyze fingerprinters [11]. It was outfitted on Chrome and PhantomJS browsers, which were also modified to automatically crawl certain sites. The results were presented in the form of logs that revealed which browser and device properties were accessed during the crawl, with an emphasis on JavaScript-based font detection. In all cases, they found there were no visible effects of fingerprinting, thus leaving the user unaware of it. Fifield and Egelman also highlighted font-based fingerprinting in a different way; they showed that font-rendering techniques can be used to distinguish users through the results of drawing a glyph from a given font-set, within a box [21]. The slight difference in pixel output reveals not only the installed font sets, but the graphics card and the browser version.

Yang and Yue developed the WTPatrol platform to determine tracking behaviors on 23,310 websites with both mobile and desktop page versions [15]. They further broke-down results to trackers using JavaScript API calls or HTTP cookies. Their results yielded 5835 unique JavaScript trackers, with 13.1% of those specific to mobile sites and 30.6% to desktop sites. They identified 5574 HTTP cookie trackers with 12.5% and 27.6% being mobile and desktop specific, respectively. WTPatrol is a critical new measurement platform because it gives researchers the ability to monitor browser fingerprinting in-the-wild, and to discover novel trackers.

Englehardt and Naryanan measured 15 elements that were part of stateful and stateless tracking mechanisms via the Open WPM tool, for over one-million websites [2]. They found 81,000 third-party trackers on two first-party domains: Googleanalytics and Google. They further found that Google, Twitter, Facebook, and AdNexus were the only third-party trackers present on approximately 10% of the websites. The 81000 trackers were distinct scripts that had overlap with regards to fingerprint variables attained via HTTP and DOM function requests, however, a troubling finding was that 460 of the top 1000 most frequently found tracking scripts communicated with one-another on the back-end to enable cookie-synching, which defeats most user interventions to maintain web-browsing privacy.

# 4 A New Method

Fingerprinting intrinsically requires measurements of client hardware performance metrics and browser element values. We wrote a Javascript program to collect values from hundreds of Chrome and Firefox browser attributes, with subsequent statistical analysis to determine the smallest subset needed to yield the most accurate fingerprint for individual users.

Our methods included construction of a second Javascript program that enabled organization of those elements into a tree structure. The structure itself was necessary for identification of repeat visitors through their fingerprints.

## *4.1 Browser Fingerprinting*

We constructed fingerprints with hundreds of measurements from the browser Windows and Navigator objects of Chrome and Firefox. Then, we attempted to determine the smallest set of variables from within those objects that yielded a unique user identity, which was linkable among browsing sessions from the same user. This strategy will have two desired effects within the tracking and anti-tracking communities. First, it will allow vendors to create scripts that use a small number of measurements to serve the more benign uses of trackers, while preserving “more” of the individual’s privacy in context of what data is revealed about one’s machine specifications, browsing history, and location. Second, it will impart elements that have not been used before by vendors, and allow anti-tracking research to gain a head start on developing counter-methods.

We achieved our ends by writing a recursive JavaScript function that enumerates through client browser information on over 200 DOM elements from the Windows and Navigator objects. The script separates each element as being a variable or function, and hierarchically assigns its characteristics and values to a node within an object tree, for which the root node is the Windows object. The Navigator object has the most numerous useful elements to fingerprint, outside of Windows. Navigator can be queried from Windows, which is why we choose Windows as our root. The script was hosted on a LAMP (**L**inux, **A**pache, **M**ySQL, **P**HP) server, and the webpage was advertised to University of Nebraska at Omaha students and faculty to garner their fingerprints upon visiting the page. LAMP was chosen due to its accessibility as an open-source tool, since it uses the well-documented Apache server. The server runs on the Linux OS backbone, and uses PHP script for source code interpretation; the PHP accesses MySQL databases for data manipulation.

The results were collected as a JavaScript Object Notation (JSON) object that was stored on the server. JSON objects are strings formatted in a manner such that each object holds one or more key to value pairings, each of which is separated by a colon. This format is heavily used to transmit data between client and server, but in this context, it also allows for its simple excerption according to key or value.

Next, the result files were input into another JavaScript to sort the acquired elements for subsequent statistical analysis, to find which measurements were most significant in generating a unique fingerprint. The significance was quantified in bits of entropy, from which we calculated a set with the fewest elements possible that could accurately fingerprint an individual, and link a user identity across multiple visits to our webpage.

## *4.2 Browser Attribute Collection*

There are three main algorithms necessary to complete our project. The first is visualized in figure 2, denoting the steps for browser attribute collection, which must fulfill several requirements, starting with acquisition of HTML DOM element values that do not require custom function calls. We choose to extract the Window object browser properties. To do this, the algorithm must iterate between two functions, the first of which evaluates the type of a given Window property. The type may be a string, number, Boolean, function, or object. If the property is an object, a recursive call is made to find the property types within the nested object. If the Window property is a function, a call is made to the second main function that determines the function identity and executes custom code to evaluate the function's input variables from the browser environment. The algorithm must also construct a hierarchical object tree representing the location and rank of an object within the results of the collection phase.

The attribute collection algorithm additionally collects user information in the least intrusive manner for Firefox and Chrome, which are the two most popular browsers in the world. This is achieved through construction of a LAMP server running a JavaScript that only prompts the user to allow for obtainment of geolocation. This is followed by a text box indicating completion of data procurement of browser window objects. The latter is effected by means of coding for browser-specific functions for a given Window property. The overall flow of the method is detailed in Figure 2.

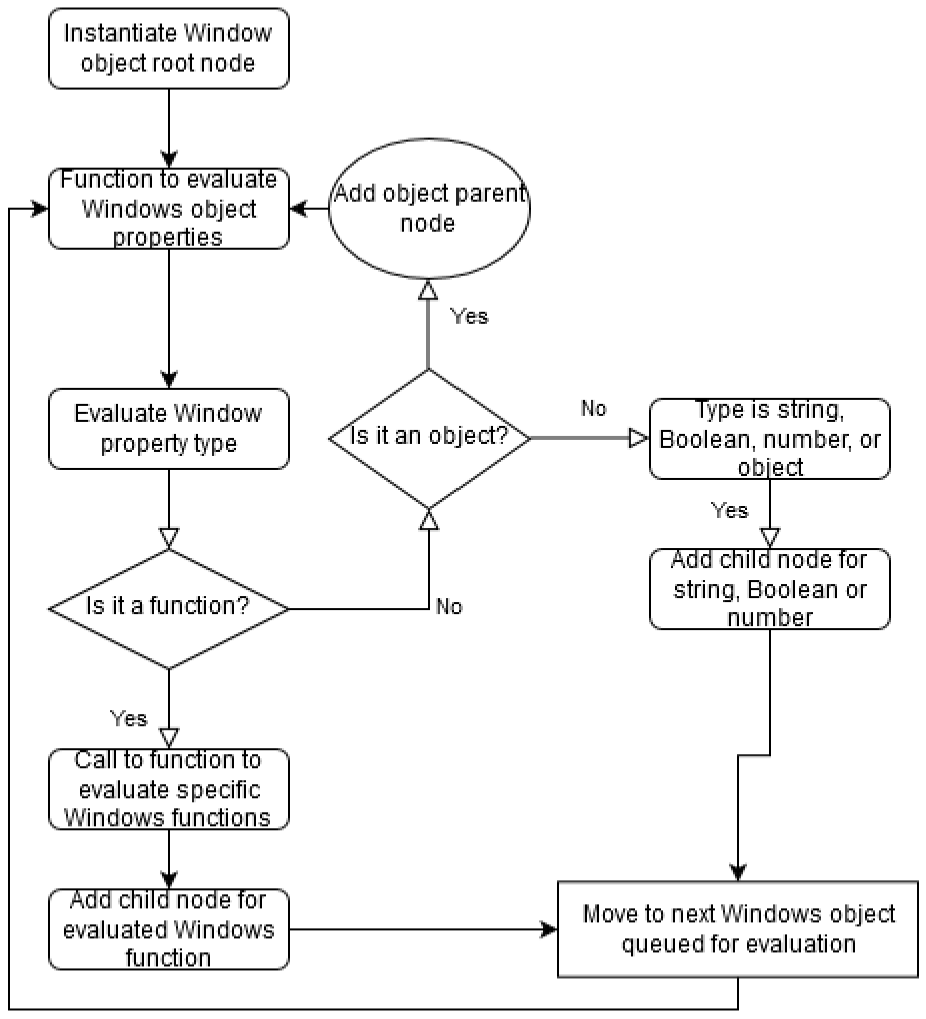
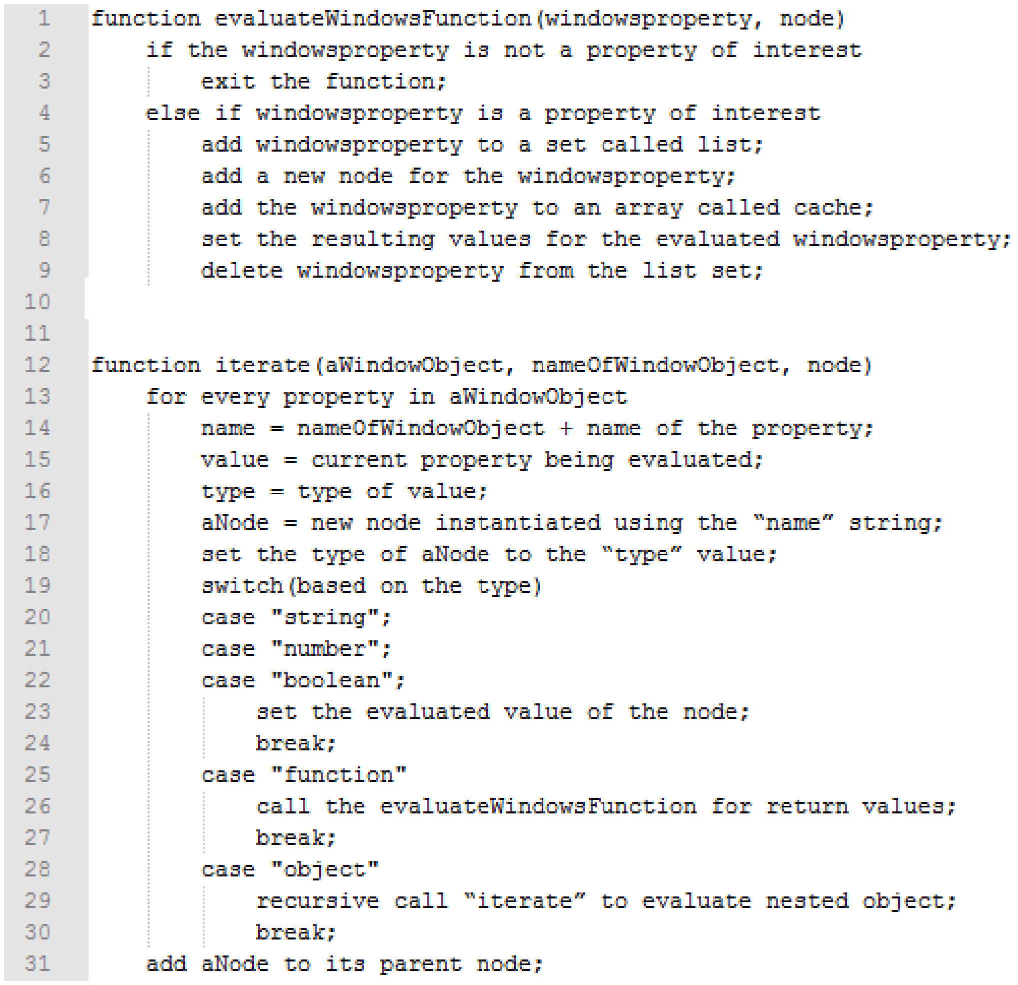

Instantiate Window object root node
Function to evaluate Windows object properties
Add object parent node
Yes
Evaluate Window property type
Is it an object?
No
Type is string, Boolean, number, or object
Yes
Add child node for string, Boolean or number
Is it a function?
No
Yes
Call to function to evaluate specific Windows functions
Add child node for evaluated Windows function
Move to next Windows object queued for evaluation

Figure 2. Attribute collection flowchart

The pseudocode for attribute collection relays a concrete description of the corresponding JavaScript. Lines 1-11 of figure 3 represent the process for evaluating a function within a Windows object container. We only wish to evaluate Windows functions that yield the most information for user identity; lines 2-3 whitelist for such functions. Lines 5-6 add the name of the windows property to a set, and create a child-node for the property, respectively. The “list” set in line 5 is to store the names of asynchronous functions that will not resolve in the order they are placed on the stack. These functions must be periodically checked for return values, after which they are deleted from the set, in line 9. Line 7 adds the name of the property to the “cache” array, while line 8 sets the evaluated Windows property values to the instantiated node from line 6. The cache array is used to set the property name and then associate it with the appropriate node, such that the evaluation of the function may be associated with the proper node.



```
 1  function evaluateWindowsFunction(windowsproperty, node)
 2      if the windowsproperty is not a property of interest
 3          exit the function;
 4      else if windowsproperty is a property of interest
 5          add windowsproperty to a set called list;
 6          add a new node for the windowsproperty;
 7          add the windowsproperty to an array called cache;
 8          set the resulting values for the evaluated windowsproperty;
 9          delete windowsproperty from the list set;
10
11
12  function iterate(aWindowObject, nameOfWindowObject, node)
13      for every property in aWindowObject
14          name = nameOfWindowObject + name of the property;
15          value = current property being evaluated;
16          type = type of value;
17          aNode = new node instantiated using the “name” string;
18          set the type of aNode to the “type” value;
19          switch(based on the type)
20          case "string";
21          case "number";
22          case "boolean";
23              set the evaluated value of the node;
24              break;
25          case "function"
26              call the evaluateWindowsFunction for return values;
27              break;
28          case "object"
29              recursive call “iterate” to evaluate nested object;
30              break;
31      add aNode to its parent node;
```

Figure 3: The pseudocode for browser attribute collection

The recursive “iterate” function beginning on line 12 iterates through all objects within a passed “aWindowNode” window object. Lines 14 sets a string denoting the object name and property; line 15 computes the value of the current property object, and line 16 assesses the object type, to be used in the following switch statement. Lines 17-18 create a new node object set to the determined name and type from line 14 and 16. The switch statement starting at line 19 will set the node value to the one from line 15 if the type is string, Boolean or number. However, if the type is a function, the switch will call the evaluateWindowsFunction function to receive output values to be written to the node. If the type is an object, a recursive call is made to iterate through all of the nested objects therein. Finally, at line 31, “aNode” is added to its parent node once the value (i.e. a function, string, number or Boolean type) is returned from the switch statement.

The results of the browser attribute collection are stored as a cookie that is named via a JavaScript random character generator. The cookie values for the user Windows DOM element results are extracted via the second main algorithm. This script parses fields from the cookie to yield data, as seen in figure 4. The program starts by taking all of the fingerprint files from the first algorithm and for each, garnering an ID (represented by a hash), the visitor IP address, and the contents of the fingerprint itself. Parsing creates three arrays for the following: 1) storage of the total number of user identities with each ID associated with its set of Window attribute collection, 2) storage of the total number of elements collected across the global set of users, associated with all values for each element, from the global set of users, and 3) storage of all elements mapped to incidence of each element, from the global set. In order to construct the first two arrays, the nodes from each JSON tree fingerprint object are “flattened”, meaning that the information from each leaf is taken and allocated to its appropriate category within the array. The information includes object name, type, index, and value. The data for these arrays are then written to a comma separated value file for the third algorithm to process, as shown in figure 4.

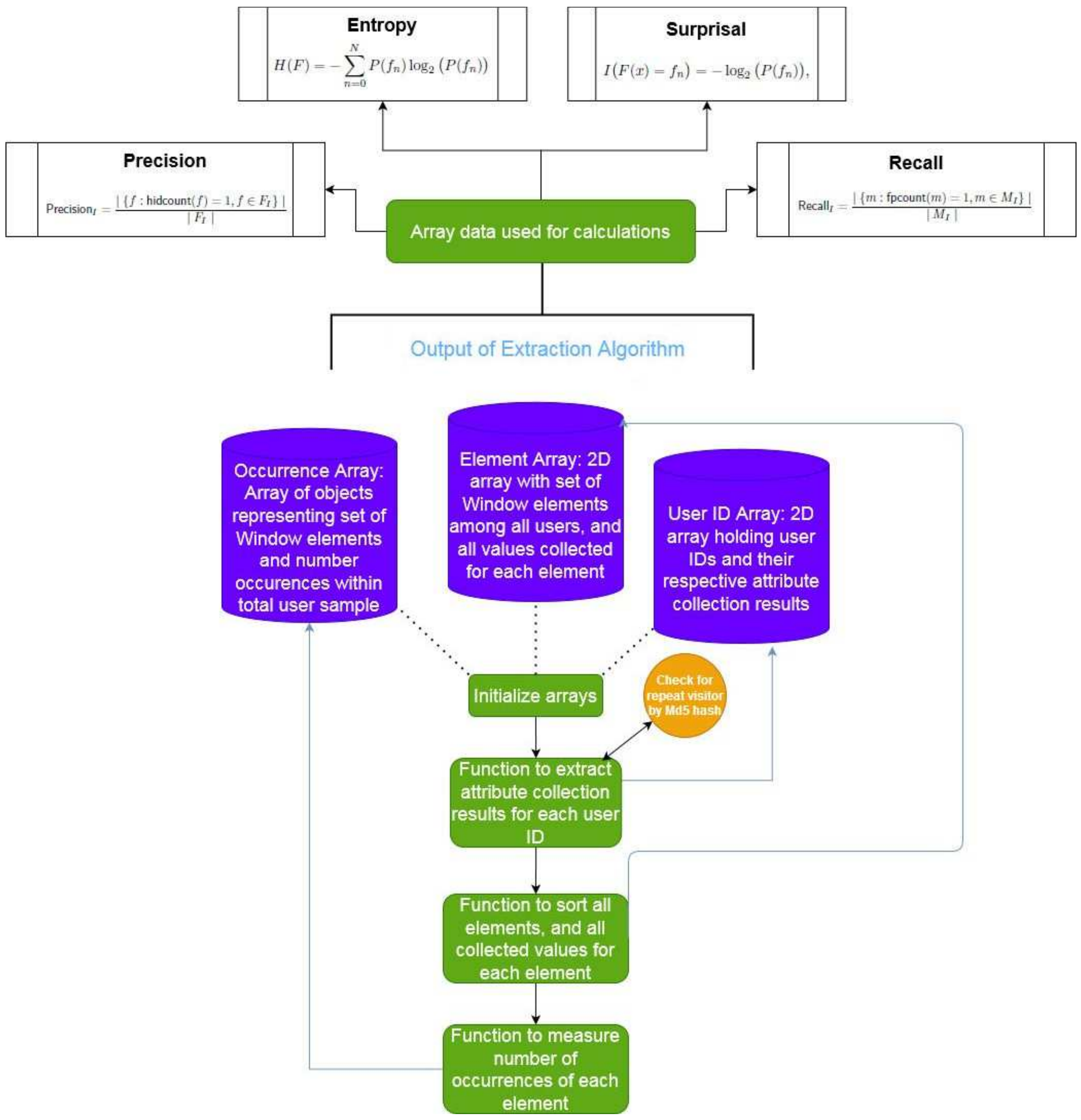

Entropy
$H(F) = -\sum_{n=0}^{N} P(f_n)\log_2(P(f_n))$
Surprisal
$I(F(x) = f_n) = -\log_2(P(f_n)),$
Precision
$\text{Precision}_I = \frac{|\{f : \text{hidcount}(f) = 1, f \in F_I\}|}{|F_I|}$
Recall
$\text{Recall}_I = \frac{|\{m : \text{fpcount}(m) = 1, m \in M_I\}|}{|M_I|}$
Array data used for calculations
Output of Extraction Algorithm
Occurrence Array: Array of objects representing set of Window elements and number occurences within total user sample
Element Array: 2D array with set of Window elements among all users, and all values collected for each element
User ID Array: 2D array holding user IDs and their respective attribute collection results
Initialize arrays
Check for repeat visitor by Md5 hash
Function to extract attribute collection results for each user ID
Function to sort all elements, and all collected values for each element
Function to measure number of occurrences of each element

Figure 4: Extraction of fingerprint elements for use in statistical analysis

The third algorithm calculates the entropy and surprisal, or “uniqueness” of the collected elements. The surprisal, *I*, is a measurement in bits for each element collected within a set for a single user, and indicates the amount of information about a user's identity. It is dependent on the $\log_2$ of a discrete probability function $P(f_n)$, which represents the probability of a set of outputs comprising a fingerprint [1]. The entropy of $I$ is represented by $H$, and is the expected surprisal value over all browser attribute sets collected. Elements that are unique to a given browser, or which occur infrequently within a dataset have an inverse correlation with their bits of entropy. When values of entropy and uniqueness of each element are combined, and such is performed among all elements, a resulting fingerprint is created that may be measured for accuracy. The accuracy of an element is expressed by computing precision and recall [8]. Precision relates to how well an identifier can correctly correlate fingerprint to a single user ID, while recall measures how effective an identifier is for tracking a host across repeat visits. The equation for precision takes the client ID count as a function of how many IDs can be mapped to a single fingerprint($f$), and the equation for recall requires the fingerprint count (fpcount) denoting the fingerprints belonging to which a hardware ID($m$). The data will then be normalized using a technique unseen in other literature. We will take the hex value of each attribute index (i.e. node position it falls within the JSON tree) and value, add them together, and then take a rounded $\log_{10}$ to avoid excessively long decimal fractions. These values will be used for calculation of each attribute surprisal; the entropy for each attribute, precision, and recall. Entropy, surprisal, recall, and precision all give values between 0 and 1. Accuracy will reflect the efficacy of our fingerprinting application through the calculation of f-measure, which is the weighted mean of precision and recall.

Our algorithms must be able to collect Window object state values and calculate a fingerprint. The entropy and uniqueness of each element are also determined. Windows.timing objects are excluded from fingerprints due to their arbitrary values based on timestamps, which could lead to drastic increase in false negatives.

Data Collection is executed by having University of Nebraska at Omaha students and faculty visit our LAMP server, and happened over the span of several weeks. Upon visiting the page hosting our script, a user's identity is ascertained as being "first-time", or "repeat" by checking for an existing cookie that contains the user's string ID. Delineation between desktop and mobile users is made by checking for return values of Window properties specific to mobile platforms, such as screen orientation and vibration.

# 5 Results

The raw results of the algorithms are given in two parts. The first is a filename string integrating a random ID of twenty characters, an IP address, and a timestamp for a client visitor. The second component encapsulates the contents of the fingerprint, which are a collection of Window objects represented within a JSON file.

| **TP** | **FP** | **TN** | **FN** | **Precision (P) TP/(TP+FP)** | **Recall (R) TP/(TP+FN)** | **F-Measure 2*(P*R/(P+R))** |
|---|---|---|---|---|---|---|
| 42 | 3 | 63 | 33 | 0.93 | 0.560 | 0.68 |

Table 1. Statistical scores characterizing accuracy of fingerprint algorithm in identifying repeat visitors. TP, FP, TN and FN are true positives, false positives, true negatives and false negatives, respectively.

A total of 141 fingerprints gathered from visitors using computers at University of Nebraska Omaha were analyzed, and all were from Windows desktop machines. In table 1, true positives are those clients categorized as being repeat visitors by our algorithm due to having fingerprints matched with corresponding IDs. False positives are clients whose fingerprints exist within our database, but who had different IDs, while false negatives have existing IDs but new fingerprints. True negatives are visitors whose IDs and fingerprints are not within our database, and thus new visitors. There were 42 (TP) repeat and 63 (TN) new visitors detected. The TNs correspond exactly upon manual inspection of the fingerprint file names, since there are 63 unique filename IDs among the 141 total fingerprints.

Recall was measured at 56%, and is a function of true positives versus all detected positives within the data. Precision gives an estimate of confidence for a true positive actually being a positive, and was calculated to be 93%. Our F-measure or F-score gauged the accuracy of our model as a function of precision and recall, at 68%.

| Window Object Name | Entropy | Std Dev |
|---|---|---|
| window.performance.getEntries()[0].responseStart | 1 | 3.40577 |
| window.performance.now | 0.961756 | 0.79614 |
| window.performance.timing.unloadEventStart | 0.721317 | 7.00896 |
| window.performance.timing.unloadEventEnd | 0.721317 | 7.00896 |
| window.close | 0.480878 | 1.60833 |
| window.navigator.appVersion | 0.463325 | 41.27527 |

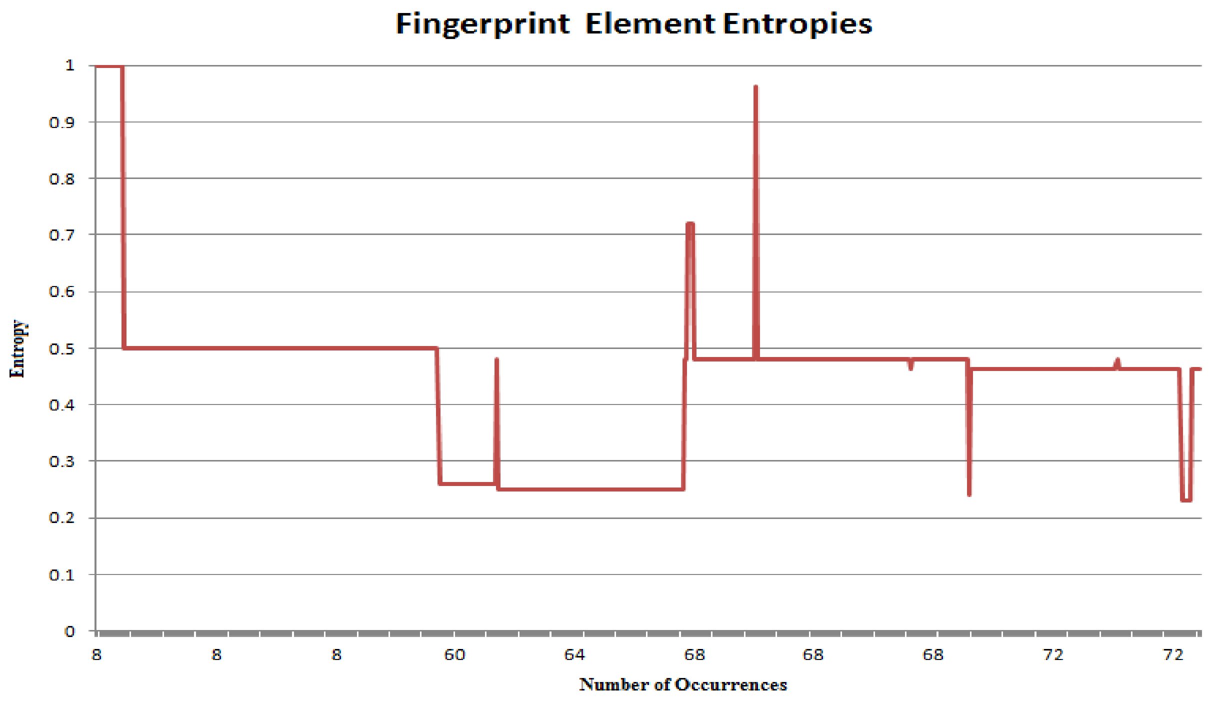
Table 2. A sampling of window attributes with high combination of entropies and standard deviations.

The complete list is comprised of 665 different window objects across the set of fingerprints. Table 2 displays a few attributes with a combination of high entropy and standard deviation; both of these values have a positive correlation with probability of identifying a unique fingerprint. The first four entries are those yielding the time in milliseconds, to complete a task. For example, the" window.performance.now" function returns the time to execute a function.

The "window.clientinformation", "window.application", "window.navigator.mimeTypes", and "window.navigator.plugins" nodes all scored among the lowest in entropy (h=0.25) and standard deviation (stddev=0), returning the exact same scores.

There were 105 window nodes dependent on the presence of JavaScript, with a wide range of entropy and standard deviations. The top four of these had entropies of 1.0, and standard deviations of 20.43459, while 73 had the entropy of 0.5 and standard deviation of 0. The greatest standard deviation was for window.navigator.appVersion, at 41.27527, which had an entropy of 0.463325. The average entropy for all nodes was 0.444395. In figure 5, we see that many elements with entropies between 0.25 and 1.0 appeared in eight fingerprints. A larger subset that was in 68 or more fingerprints had the same entropy range, but note that 489 fingerprints within this subset were between 0.46-0.50.

The standard deviation average was 0.463616, with 554 of 665 elements yielding a value from 0 to 0.02121, with the remainder falling within 0.76-1.00. Figure 5 depicts standard deviation with a near flat-line at the bottom, and visible spikes for those nodes with a product well above the average.



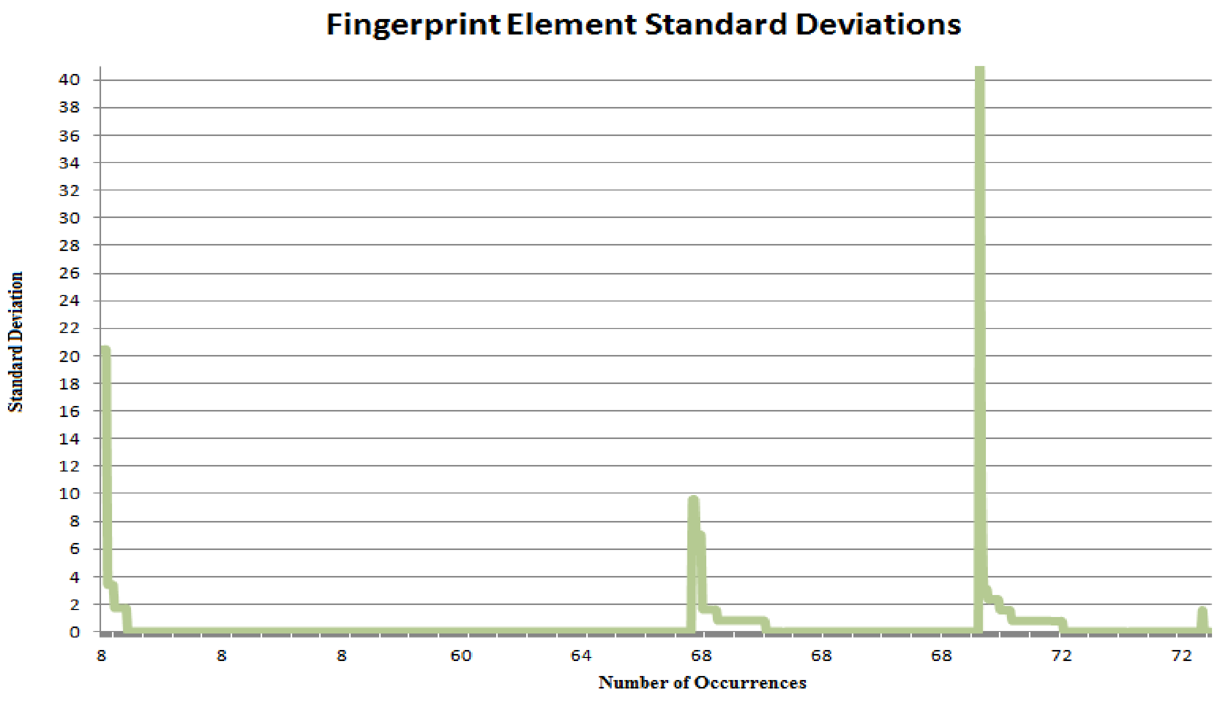


Figure 5. Charts displaying entropies and standard deviations of collected elements across all fingerprints. The average entropy for all nodes was 0.444395, and the average standard deviation was 0.463616.

# 6 Discussion

The results suggest our fingerprinting methodology is sound, and will be explained in several sections that cover the basic statistics and the most useful attributes. A comparison of their significance to that of select previous works will follow, with a final section interpreting the method's contributions to the science of fingerprinting, and its limitations.

## *6.1 Evaluation of the Raw Statistics*

The effectiveness of our fingerprint method is most reflected in the true-negative and false-positive count, which is a direct measure of how accurately it identifies repeat visitors. In this experiment, there were 66 visitors contributing to the 141 fingerprints. Of those, 63 were labeled as true-negatives, or completely new visitors. Another 3 were given as false-positives, which refer to those who have their fingerprint in our database, but somehow have a different generated ID. This is plausible because Firefox and Chrome have settings to clear browser cookies upon closing the browser windows. Upon revisiting the fingerprint page, a new random 20-character ID would be generated and be in the filename of the same fingerprint taken previously for the respective machine. Precision incorporates TP and FP to relay the algorithms' ability for positive prediction. The experimental score was 93%; while the figure is impressive, giving confidence towards its efficacy on a larger scale.

Our method determined the 66 client machines repeatedly visited our page 42 times, as reflected in the true-positive count, with other clients that were also repeats, falling with the false-negative category. The false-negatives yielded conflicting fingerprints for existing client IDs. The discrepancies are explained by any browser update or alteration that would change one of the 665 window attribute values. Even alteration of the browser window size, or acquisition of a new monitor with a different resolution would change the fingerprint, if only slightly. However, if the client-stored cookie remains, then our scripts allow proper determination of those as effective positives, which also illustrates the potential of these scripts to be used on a larger scale. The most important consideration is whether the TPs and FPs include repeat visitations, which is true within the bounds of our work upon manual inspection of the IDs and corresponding values within these fingerprints. The recall, which is calculation of our algorithms to predict how many true positives are predicted among all positives within the dataset, was 56%. The result seems poor until one considers the context within the scope of fingerprinting. The script logic is strict in that if even one of the 665 window elements has a slightly different value for a repeat visitor with the same ID, it is considered to be a negative, and upon verification of the same IDs with incongruent fingerprints, further subcategorized as a FN. We can also inductively determine the visitor with two slightly different fingerprints are using the same machine, by inferring that it is highly improbable a completely different visitor would navigate to our page and have the exact same random 20-character ID. The likelihood of such an occurrence for a dataset of 141 fingerprints is $3.96*10^{-17}$, using the formula for combination of elements with repetition: C'(n,r) = (r+n-1)!/(r! * (n-1)!) [27]. Here, C(n,r) is the total number of combinations, n is the number of selectable elements, which is 61 different characters, and r is the number of chosen elements per ID. Thus the low recall suggests a higher accuracy, since the rules for exclusion from being a TP are so strict that only those clients with unseen IDs and fingerprint values for all elements are considered TPs.

F-measure with the F-Beta modifier, is $F_\beta = (1+\beta^2)$ PR/ $(\beta^2$ P)+R), and is a reflection of accuracy as a function of precision and recall. Our F-measure was 0.7, due to the gulf between precision and recall. The score itself has no implication without understanding if FPs or FNs have more negative impact within our scope. For fingerprinting machines from a computer lab, a false-positive has the more detrimental effect since it is nearly impossible to verify if a visitor who has a different ID but the exact same fingerprint as another within our database, is that same person, or a new client who happens to have the same values for all 665 measured window nodes of said fingerprint. In models where false-positives are more impactful, the beta modifier is set to zero, giving more weight to precision. According to Simic, since our score of 0.7 is less than 1, recall is 0.7x less important than precision, which is more favorable to recognizing fingerprint values of repeat visitors who have cleared their locally stored cookies [24].

## *6.2 Which Elements are Useful for Fingerprinting?*

There are 206 of the 665 window objects measured among all fingerprints that are present for only 10 to 17 of the 141 fingerprints. However, of these, only 16 have non-zero standard deviation values, and all have entropies of either 0.5 or 1 due to each of these elements having a set of values comprised of the same integer value, or two different values. The greatest standard deviations for this group are for window.performance.toJSON().timing.unloadEventEnd, window.performance.toJSON().timing.unloadEventStart, window.performance.timing.unloadEventStart, and window.performance. timing.unloadEventEnd, all of which are equal to 20.43459 and occurred for 10 fingerprints. The entropies for all of them equal 1.0. These four elements would lend the most to the uniqueness of a fingerprint due to their combination of maximum entropy and high standard deviations, but are not attractive within the context of our experiment because they appeared in 7% of fingerprints.

On the other end, 310 attributes contribute to the bulk of the set of fingerprints, occurring for 64-76 of the 141, with entropies ranging from 0.25-0.721317. Of these, 109 have entropy values of 0.25 and standard deviations of zero, and are the least informative towards a unique fingerprint, despite their prevalence within the set. Nevertheless, within these 310, the top four elements are window.performance.timing.responseEnd (entropy = 0.480878, stddev = 9.59472); window.performance.timing.toJSON().responseEnd (entropy = 0.480878, stddev = 9.59468); window.navigator.vendor (entropy = 0.463325, stddev = 10.12415), and window.navigator.appVersion (entropy = 0.463325, stddev = 41.27527). These elements are fit to be within a subset of fingerprinting window elements due to their standard deviations, despite their entropies that lie slightly above the mean entropy for the set.

The concept of surprisal, and by association entropy, comes from mathematician Claude Shannon. In1948, Claude published an article in Bell

System Technical Journal discussing signals that rise above the “noise” (i.e. mean) in a “surprising” manner, and thus return more information [25]. Accordingly, in table 3, the top 20 window attributes according to their combination of entropy and standard deviation, and the number of times they appeared in a fingerprint, as a suitable subset of the 665 measured. These 20 may be used in a future experiment against the global set to ascertain whether it gives enough information to verify a unique fingerprint.

| **Window element** | **Entropy** | **Std Dev** | **# of occurrences** |
|---|---|---|---|
| window.screen.availLeft | 0.463325 | 2.33629 | 72 |
| window.screen.availTop | 0.463325 | 2.33629 | 72 |
| window.screenTop | 0.463325 | 2.3363 | 72 |
| window.screenY | 0.463325 | 2.33631 | 72 |
| window.navigator.language | 0.463325 | 2.33633 | 72 |
| window.navigator.languages[0] | 0.463325 | 2.33634 | 72 |
| window.navigator.platform | 0.463325 | 2.33634 | 72 |
| window.navigator.mediaCapabilities.decodingInfo | 0.463325 | 3.09389 | 72 |
| window.indexcount | 0.463325 | 3.10511 | 72 |
| window.navigator.userAgent | 0.463325 | 3.1151 | 72 |
| window.navigator.vendor | 0.463325 | 10.12415 | 72 |
| window.navigator.appVersion | 0.463325 | 41.27527 | 72 |
| window.timer | 0.463325 | 1.5659 | 76 |
| window.performance.timing.toJSON().responseEnd | 0.480878 | 9.59468 | 68 |
| window.performance.timing.responseEnd | 0.480878 | 9.59472 | 68 |
| window.performance.timing.toJSON().unloadEventEnd | 0.721317 | 7.00709 | 68 |
| window.performance.timing.toJSON().unloadEventStart | 0.721317 | 7.00709 | 68 |
| window.performance.timing.unloadEventEnd | 0.721317 | 7.00896 | 68 |
| window.performance.timing.unloadEventStart | 0.721317 | 7.00896 | 68 |
| window.performance.now | 0.961756 | 0.79614 | 68 |

Table 3. Subset containing top 20 window attributes with high combination of entropies and standard deviations, with emphasis on choosing those that appear in 68 or more fingerprints.

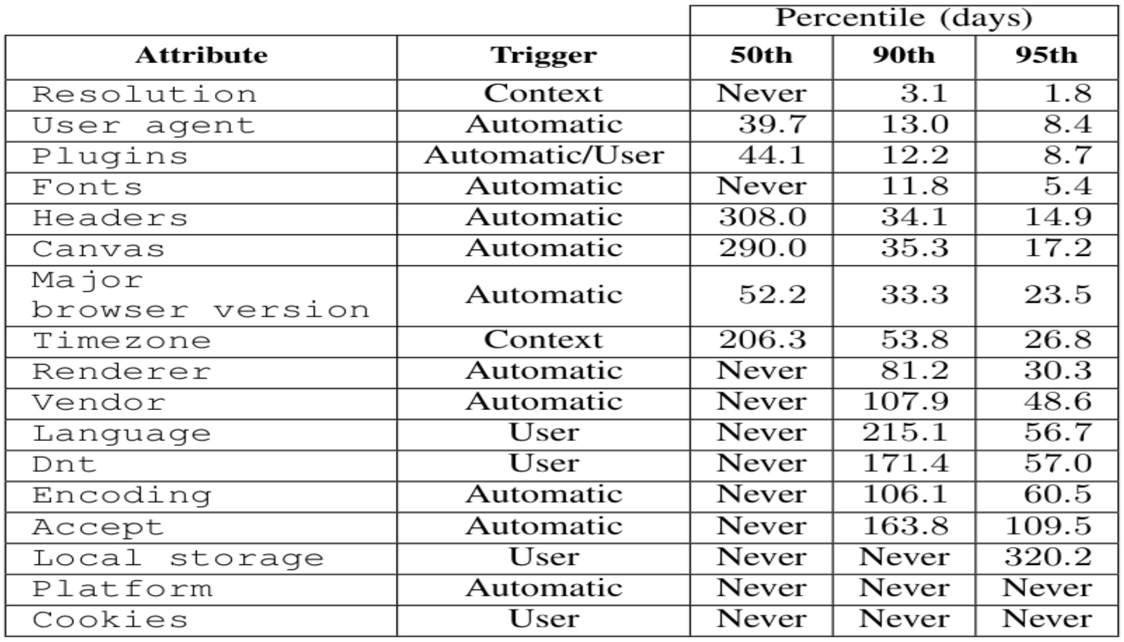
There were other window elements with high entropy or standard deviation, but they occurred in too few fingerprints. Conversely, there were many other elements nearly ubiquitously appearing within our set, but exhibited low entropy or standard deviation.

## *6.3 Comparison to Previous Works*

The method and results of this research will be compared to four seminal works with aim to delineate strengths of our work over their methods, and weaknesses that expose how we may improve our scripts.

Our algorithm looks for matches with respect to retrieved client cookie ID (if it exists) and the absolute value of fingerprint contents. It is heavily reliant on the locally stored cookie to persist, thus informing our algorithm of a repeat visitor whose browser attributes have changed. Therefore, it is naïve compared to other known algorithms, such as the seminal work of Eckersley's Panopticlick [1], which returns a revisit match if string comparison of eight different attributes is greater than 85% between two fingerprints. If there are too many fingerprints that match the reference in this manner, then it assigns a new ID. Their element set included user_agent, plugins, fonts, video, supercookies, http_accept, timezone, and cookies_enabled, with the following entropy values respectively (in bits): 10.0, 15.4, 13.9, 4.83, 2.12, 6.09, 3.04, and 0.353. They were able to label 83.6% of fingerprints as unique within their sample set of 470,161. They claimed their method to predict a revisit 65% of the time if Javascript were enabled, and correctly link it to a previous fingerprint at a 99.1% rate.

On the surface, our approach seems lacking compared to Eckersley's because we did not implement a prediction heuristic; their attributes have much higher entropies than most of ours, and because they have a much larger dataset to test. While implementation of a heuristic would be an important addendum, our logic is more effective if the stored cookie persists either on the client machine or our database, because we can detect 100% of FN as positives, and thus catch all positives. Furthermore, our method does not rely on JavaScript being enabled due to the much larger pool of measured window nodes that includes all of theirs except supercookies and fonts. Although the entropies of seven out of their eight attributes are much higher than those within our set, this is due to their larger sample size that contains many more values for their eight elements, thus calculating to increased and more accurate entropy values for those elements.



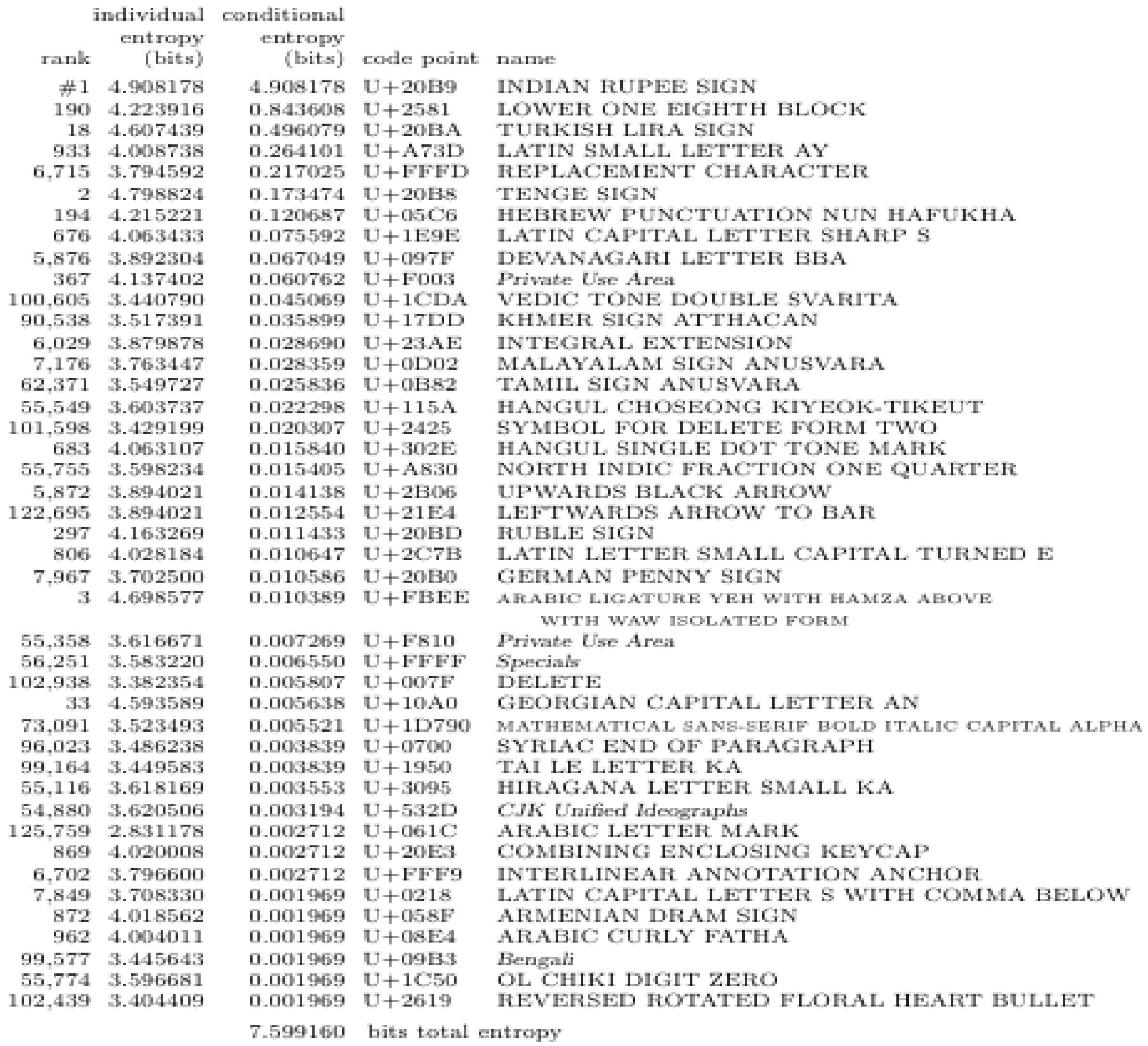
| | | Percentile (days) | | |
|---|---|---|---|---|
| **Attribute** | **Trigger** | **50th** | **90th** | **95th** |
| Resolution | Context | Never | 3.1 | 1.8 |
| User agent | Automatic | 39.7 | 13.0 | 8.4 |
| Plugins | Automatic/User | 44.1 | 12.2 | 8.7 |
| Fonts | Automatic | Never | 11.8 | 5.4 |
| Headers | Automatic | 308.0 | 34.1 | 14.9 |
| Canvas | Automatic | 290.0 | 35.3 | 17.2 |
| Major browser version | Automatic | 52.2 | 33.3 | 23.5 |
| Timezone | Context | 206.3 | 53.8 | 26.8 |
| Renderer | Automatic | Never | 81.2 | 30.3 |
| Vendor | Automatic | Never | 107.9 | 48.6 |
| Language | User | Never | 215.1 | 56.7 |
| Dnt | User | Never | 171.4 | 57.0 |
| Encoding | Automatic | Never | 106.1 | 60.5 |
| Accept | Automatic | Never | 163.8 | 109.5 |
| Local storage | User | Never | Never | 320.2 |
| Platform | Automatic | Never | Never | Never |
| Cookies | User | Never | Never | Never |

Table 4. Duration attributes measured in FP-Stalker article remained constant for the median, the 90th and the 96th percentile of days [5].

Vastel et al created an application called FP-Stalker to track evolution of fingerprints from visitors over time [5]. The aim of their work was more encompassing than ours; however they overlap with Eckersley in choosing a set of attributes to measure, and an algorithm for prediction. Unlike other works, they construed entropy indirectly as a function of attribute value stability over time, using machine learning heuristics.

Table 4 from Vastel's work informs that Local Storage, Platform and Cookies remain static indefinitely at all percentiles, while those elements above them vary in stability. Our set contains all but the "Canvas", "Font", and "Renderer" variables, although we do have values from screen height and width that give resolution. It is not possible to make a direct comparison between their method and ours, since Vastel et al used a hybrid machine learning algorithm to validate and track changes in the elements. Moreover, those elements with low entropy but long-term persistence are more important to their algorithm's ability in detecting repeat visitors. Nevertheless, their work suggests the installed fonts set (accessible via Canvas functions) and the graphics chip model with supported features (known by the WebGL API) would enhance the uniqueness of our fingerprints due to the volume of divulged information, lending towards greater entropies for each of those categories.

Font measurements augment fingerprinting techniques and may be used to distinguish between different browser instances on a given machine, which is more specific, as seen in Fifield and Egelman's work [21]. Therein, they analyze font glyphs by rendering characters from various sets within boxes of different sizes. The resultant data exposed more than installed browser fonts; it divulged different versions of a given font, minimum font size, and even rendering options such as anti-aliasing. They found that glyph rendering only 43 code point measurements were necessary to determine presence among a set of 125,766 glyphs that comprise all font sets, as seen in table 5.

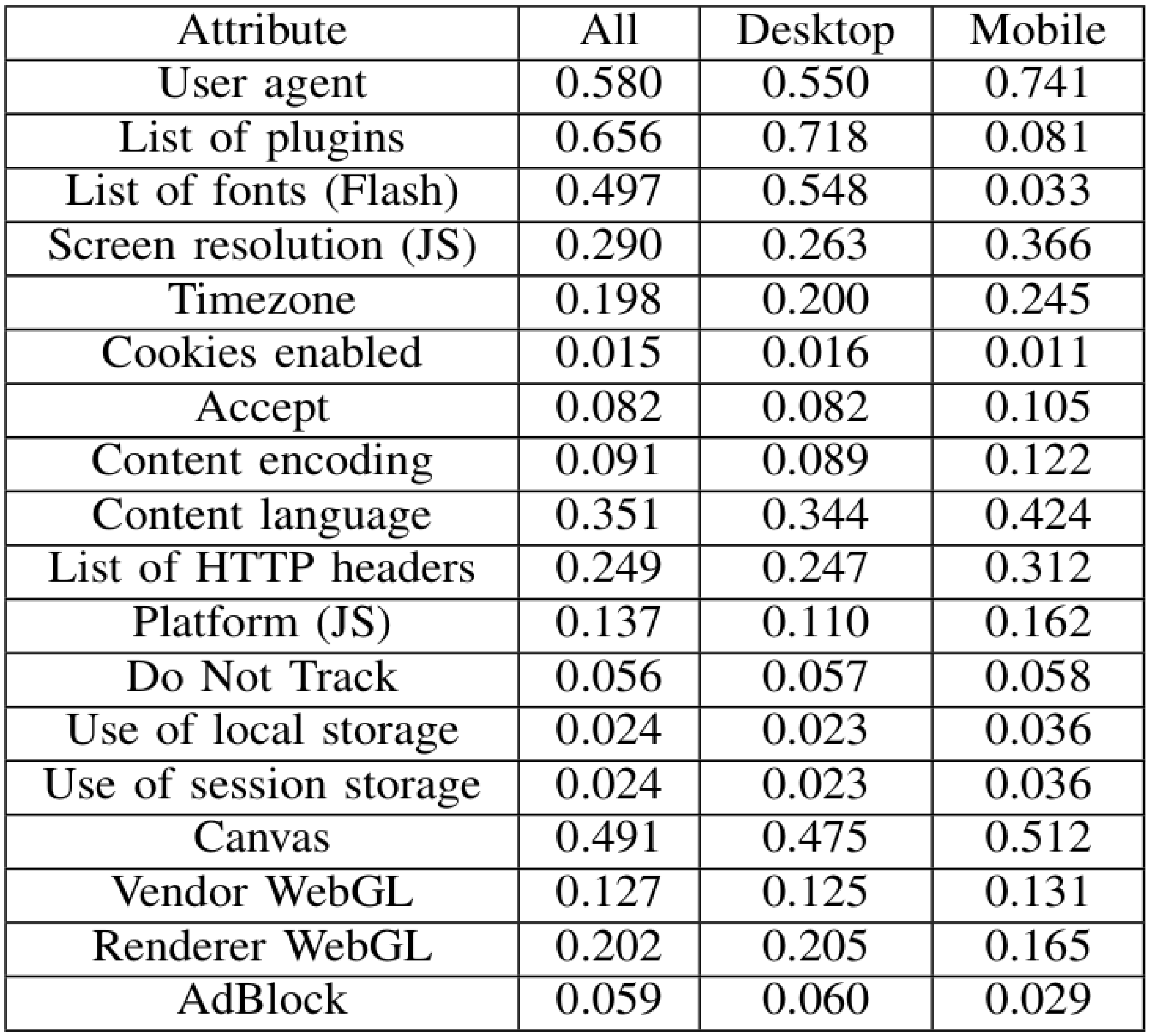


| rank | individual entropy (bits) | conditional entropy (bits) | code point | name |
|---|---|---|---|---|
| #1 | 4.908178 | 4.908178 | U+20B9 | INDIAN RUPEE SIGN |
| 190 | 4.223916 | 0.843608 | U+2581 | LOWER ONE EIGHTH BLOCK |
| 18 | 4.607439 | 0.496079 | U+20BA | TURKISH LIRA SIGN |
| 933 | 4.008738 | 0.264101 | U+A73D | LATIN SMALL LETTER AY |
| 6,715 | 3.794592 | 0.217025 | U+FFFD | REPLACEMENT CHARACTER |
| 2 | 4.798824 | 0.173474 | U+20B8 | TENGE SIGN |
| 194 | 4.215221 | 0.120687 | U+05C6 | HEBREW PUNCTUATION NUN HAFUKHA |
| 676 | 4.063433 | 0.075592 | U+1E9E | LATIN CAPITAL LETTER SHARP S |
| 5,876 | 3.892304 | 0.067049 | U+097F | DEVANAGARI LETTER BBA |
| 367 | 4.137402 | 0.060762 | U+F003 | *Private Use Area* |
| 100,605 | 3.440790 | 0.045069 | U+1CDA | VEDIC TONE DOUBLE SVARITA |
| 90,538 | 3.517391 | 0.035899 | U+17DD | KHMER SIGN ATTHACAN |
| 6,029 | 3.879878 | 0.028690 | U+23AE | INTEGRAL EXTENSION |
| 7,176 | 3.763447 | 0.028359 | U+0D02 | MALAYALAM SIGN ANUSVARA |
| 62,371 | 3.549727 | 0.025836 | U+0B82 | TAMIL SIGN ANUSVARA |
| 55,549 | 3.603737 | 0.022298 | U+115A | HANGUL CHOSEONG KIYEOK-TIKEUT |
| 101,598 | 3.429199 | 0.020307 | U+2425 | SYMBOL FOR DELETE FORM TWO |
| 683 | 4.063107 | 0.015840 | U+302E | HANGUL SINGLE DOT TONE MARK |
| 55,755 | 3.598234 | 0.015405 | U+A830 | NORTH INDIC FRACTION ONE QUARTER |
| 5,872 | 3.894021 | 0.014138 | U+2B06 | UPWARDS BLACK ARROW |
| 122,695 | 3.894021 | 0.012554 | U+21E4 | LEFTWARDS ARROW TO BAR |
| 297 | 4.163269 | 0.011433 | U+20BD | RUBLE SIGN |
| 806 | 4.028184 | 0.010647 | U+2C7B | LATIN LETTER SMALL CAPITAL TURNED E |
| 7,967 | 3.702500 | 0.010586 | U+20B0 | GERMAN PENNY SIGN |
| 3 | 4.698577 | 0.010389 | U+FBEE | ARABIC LIGATURE YEH WITH HAMZA ABOVE WITH WAW ISOLATED FORM |
| 55,358 | 3.616671 | 0.007269 | U+F810 | *Private Use Area* |
| 56,251 | 3.583220 | 0.006550 | U+FFFF | *Specials* |
| 102,938 | 3.382354 | 0.005807 | U+007F | DELETE |
| 33 | 4.593589 | 0.005638 | U+10A0 | GEORGIAN CAPITAL LETTER AN |
| 73,091 | 3.523493 | 0.005521 | U+1D790 | MATHEMATICAL SANS-SERIF BOLD ITALIC CAPITAL ALPHA |
| 96,023 | 3.486238 | 0.003839 | U+0700 | SYRIAC END OF PARAGRAPH |
| 99,164 | 3.449583 | 0.003839 | U+1950 | TAI LE LETTER KA |
| 55,116 | 3.618169 | 0.003553 | U+3095 | HIRAGANA LETTER SMALL KA |
| 54,880 | 3.620506 | 0.003194 | U+532D | *CJK Unified Ideographs* |
| 125,759 | 2.831178 | 0.002712 | U+061C | ARABIC LETTER MARK |
| 869 | 4.020008 | 0.002712 | U+20E3 | COMBINING ENCLOSING KEYCAP |
| 6,702 | 3.796600 | 0.002712 | U+FFF9 | INTERLINEAR ANNOTATION ANCHOR |
| 7,849 | 3.708330 | 0.001969 | U+0218 | LATIN CAPITAL LETTER S WITH COMMA BELOW |
| 872 | 4.018562 | 0.001969 | U+058F | ARMENIAN DRAM SIGN |
| 962 | 4.004011 | 0.001969 | U+08E4 | ARABIC CURLY FATHA |
| 99,577 | 3.445643 | 0.001969 | U+09B3 | *Bengali* |
| 55,774 | 3.596681 | 0.001969 | U+1C50 | OL CHIKI DIGIT ZERO |
| 102,439 | 3.404409 | 0.001969 | U+2619 | REVERSED ROTATED FLORAL HEART BULLET |
| | | 7.599160 | bits total entropy | |

Table 5. The 43 code points from Vastel et al for glyphs that yield the most information for a fingerprint [26].

The top glyph with respect to conditional entropy is the Indian Rupee, at 4.9 bits of information. Conditional entropy describes the remaining entropy once the other 42 glyphs have been evaluated in the order presented; there are different measurements performed for each glyph. As such, the Rupee glyph is the only one giving a result above 1.0, with 36 not even breaking 0.1. The smallest entropy value for our set is 0.23, by comparison. It is notable their method only reveals 34% of the fingerprints as unique, and must be augmented with calculations on other elements. Our technique is more crudely effective in that we identify all unique fingerprints, but would be enhanced by incorporating appraisal of glyph code points. We would be remiss to not mention that their application is proven on a much larger data set.

Laperdrix et al looked at the utility of 17 attributes within a set of 118934 fingerprints, using a script on the AmIUnique.org site, with results seen in table 6 [3]. An asset to their work is in covering mobile device fingerprints, although our method would have no problem evaluating such devices. Furthermore, they use HTML5s Canvas and WebGL APIs to render given 2D and 3D shapes, respectively, returning performance metrics and information on underlying hardware and according software drivers. Another element found in their set but not ours, is AdBlock, although the returned entropy for it seems inconsequential. Laperdrix et al uniquely identified 89.4% of the fingerprints and used more recent technologies in Canvas and WebGL, which is are major features our algorithms do not employ, but will be in future versions. Nevertheless, they omitted 16% of their initial data to exclude fingerprints without JavaScript, since HTTP headers would be the only attribute of the 17 elements in table 6 that do not require JavaScript. Our script measures nodes that require JavaScript, but also includes hundreds that do not, and thus are not dependent upon it for measurement of a unique fingerprint, or a repeat visitor. Notwithstanding, their work emphasizes the attractiveness of assimilating Canvas and WebGL APIs in our scripts, going forward.



| Attribute | All | Desktop | Mobile |
|---|---|---|---|
| User agent | 0.580 | 0.550 | 0.741 |
| List of plugins | 0.656 | 0.718 | 0.081 |
| List of fonts (Flash) | 0.497 | 0.548 | 0.033 |
| Screen resolution (JS) | 0.290 | 0.263 | 0.366 |
| Timezone | 0.198 | 0.200 | 0.245 |
| Cookies enabled | 0.015 | 0.016 | 0.011 |
| Accept | 0.082 | 0.082 | 0.105 |
| Content encoding | 0.091 | 0.089 | 0.122 |
| Content language | 0.351 | 0.344 | 0.424 |
| List of HTTP headers | 0.249 | 0.247 | 0.312 |
| Platform (JS) | 0.137 | 0.110 | 0.162 |
| Do Not Track | 0.056 | 0.057 | 0.058 |
| Use of local storage | 0.024 | 0.023 | 0.036 |
| Use of session storage | 0.024 | 0.023 | 0.036 |
| Canvas | 0.491 | 0.475 | 0.512 |
| Vendor WebGL | 0.127 | 0.125 | 0.131 |
| Renderer WebGL | 0.202 | 0.205 | 0.165 |
| AdBlock | 0.059 | 0.060 | 0.029 |

Table 6. Normalized entropy measurements of AmIUnique fingerprints [3].

*6.4 Contributions and Limitations*

The contributions to the greater body of browser fingerprinting applications begin with the algorithm design for this project. We collected more attributes for evaluation than seen in any other publication, and show timing based window DOM objects have high entropy and standard deviation, and therefore contribute strongly to unique fingerprints due to high variation in execution of their timed functions. In addition, if the locally stored client cookie is present, we can recognize re-visitors with 100% accuracy.

Perhaps the most innovative aspect is our unique method, is to combine each collected element’s node index and value, into a hex value, and then to standardize it for calculation of surprisals, and derived entropies. No other known work incorporates the index of an attribute within a JSON tree; the index itself is a key identification variable because it provides a reference point defining the two-dimensional structure of a fingerprint. Moreover, it allows for more dynamic identification repeat client visits through development of our prediction heuristic for a future version of our algorithm.

The limitations start with the small sample size of fingerprints. The consequences include being unable to apply proper parametric statistical analyses, especially since only 66 clients created them. Another issue involves too few values attained from all fingerprints, for each element, which skews standard deviation and entropy due to less dense and narrow probability distributions.

The other glaring limitation stems from the local client cookie serving as the only means to positively identify re-visitors. Since cookie clearance is a common behavior of end-users, in future versions we must have a heuristic to compare fingerprint similarities outside of looking for absolute equality between a visitor, and those fingerprints in our database.

# 7 Conclusion and Future Work

Our aim was to develop a method that contributed to the field of browser fingerprinting by revealing new properties that could be used to identify and track individuals, so that future counter-tracking mechanisms may be developed against them. We succeeded in this endeavor by highlighting the greater variations in window.performance objects, while reiterating the importance of those attributes used in previous works, such as User-Agent and installed font list. The novelty of leveraging element locations within a tree structure in synergy with the other element characteristics (name, type, and value) to enhance fingerprinting, cannot be overstated, as it adds a layer of specificity.

Nevertheless, the algorithms must be improved in the future by adding a prediction heuristic to increase the confidence of a revisit, especially considering the most obvious weakness of our method is that we can only do so with acuity if the client has not cleared local cookies. The newer HTML5 APIs within WebGL and Canvas that are used in other works should also be assimilated because they will lend to an increase in the average entropy of collected fingerprints. Finally, while there is no reason to doubt the potential efficacy of this work, we must prove the effectiveness of our method through mass data collection, for its next iteration.

# References

[1] P. Eckersley, “How Unique Is Your Web Browser?,” in *Privacy Enhancing Technologies*, 2010, pp. 1–18. doi: 10.1007/978-3-642-14527-8_1.

[2] S. Englehardt and A. Narayanan, “Online Tracking,” Oct. 2016. doi: 10.1145/2976749.2978313.

[3] P. Laperdrix, W. Rudametkin, and B. Baudry, “Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints,” May 2016. doi: 10.1109/sp.2016.57.

[4] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, “The Web Never Forgets,” presented at the ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, 2014. doi: 10.1145/2660267.2660347.

[5] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, “FP-STALKER: Tracking Browser Fingerprint Evolutions,” May 2018. doi: 10.1109/sp.2018.00008.

[6] M. Pool, “meantime: non-consensual http user tracking using caches,” *www.sourcefrog.net*, Mar. 29, 2000. https://www.sourcefrog.net/projects/meantime (accessed Nov. 29, 2022).

[7] B. Krishnamurthy and C. Wills, “Privacy diffusion on the web,” 2009. doi: 10.1145/1526709.1526782.

[8] T.-F. Yen, Y. Xie, F. Yu, R. (Peng) Yu, and M. Abadi, “Host Fingerprinting and Tracking on the Web:Privacy and Security Implications,” *www.microsoft.com*, Feb. 01, 2012. https://www.microsoft.com/en-us/research/publication/host-fingerprinting-and-tracking-on-the-webprivacy-and-security-implications/ (accessed Nov. 29, 2022).

[9] F. Roesner, T. Kohno, and D. Wetherall, “Detecting and Defending Against Third-Party Tracking on the Web,” presented at the USENIX conference on Networked Systems Design and Implementation, San Jose, CA, Apr. 2012.

[10] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, “Fingerprinting Information in JavaScript Implementations,” in *Proceedings of W2SP*, Oakland, CA, May 2011, vol. 2, no. 11. Accessed: Nov. 28, 2022. [Online]. Available: https://search.iczhiku.com/paper/hgdOSDNQ7g2K8zv8.pdf

[11] G. Acar *et al.*, “FPDetective: Dusting the Web for Fingerprinters,” 2013. doi: 10.1145/2508859.2516674.

[12] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, “Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting,”

presented at the IEEE Symposium on Security and Privacy, Berkeley, CA, May 2013. doi: 10.1109/sp.2013.43.

[13] M. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoofnagle, “Flash Cookies and Privacy II: Now with HTML5 and ETag Respawning,” *SSRN Electronic Journal*, 2011, doi: 10.2139/ssrn.1898390.

[14] “Desktop vs Mobile vs Tablet Market Share Worldwide | StatCounter Global Stats,” *StatCounter Global Stats*, 2019. https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet

[15] Z. Yang and C. Yue, “A Comparative Measurement Study of Web Tracking on Mobile and Desktop Environments,” *Proceedings on Privacy Enhancing Technologies*, vol. 2020, no. 2, pp. 24–44, Apr. 2020, doi: 10.2478/popets-2020-0016.

[16] S. Sista, “Stable Channel Update for Desktop,” *Chrome Releases*, Oct. 31, 2019. https://chromereleases.googleblog.com/2019/10/stable-channel-update-for-desktop_31.html (accessed Nov. 28, 2022).

[17] S. Khandelwal, “Exploiting Browser Cookies to Bypass HTTPS and Steal Private Information,” *The Hacker News*, Sep. 25, 2015. https://thehackernews.com/2015/09/https-cookies-hacking.html (accessed Nov. 28, 2022).

[18] “CVE - Common Vulnerabilities and Exposures (CVE),” *Mitre.org*, 2016. https://cve.mitre.org/index.html (accessed Nov. 28, 2022).

[19] X. Luo, P. Zhou, Edmond, W. Lee, Rocky, and R. Perdisci, “HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows,” 2011.

[20] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson, “Touching from a distance,” 2012. doi: 10.1145/2382196.2382260.

[21] D. Fifield and S. Egelman, “Fingerprinting Web Users Through Font Metrics,” *Financial Cryptography and Data Security*, pp. 107–124, 2015, doi: 10.1007/978-3-662-47854-7_7.

[22] N. Nikiforakis, W. Joosen, and B. Livshits, “PriVaricator,” presented at the International Conference on World Wide Web, Florence, Italy, May 2015. doi: 10.1145/2736277.2741090.

[23] C. F. Torres, H. Jonker, and S. Mauw, “FP-Block: Usable Web Privacy by Controlling Browser Fingerprinting,” in *Computer Security -- ESORICS 2015*, Vienna, Austria, 2015, pp. 3–19. doi: 10.1007/978-3-319-24177-7_1.

[24] M. Simic, "F-Beta Score | Baeldung on Computer Science," *www.baeldung.com*, Jun. 12, 2022. https://www.baeldung.com/cs/f-beta-score (accessed Nov. 19, 2022).

[25] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x.

[26] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies," presented at the Proceedings of the 27th USENIX Conference on Security Symposium, Baltimore, MD, Aug. 2018.

[27] B. Szyk and D. Czernia, "Combination Calculator (nCr) | Combinations Generator," *www.omnicalculator.com*. https://www.omnicalculator.com/statistics/combination