Cookieless Monster: Exploring the Ecosystem of Web-based Device Fingerprinting

Nick Nikiforakis*, Alexandros Kapravelos†, Wouter Joosen*, Christopher Kruegel†, Frank Piessens*, Giovanni Vigna†
*iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{firstname.lastname}@cs.kuleuven.be
†University of California, Santa Barbara, CA, USA
{kapravel,chris,vigna}@cs.ucsb.edu

***Abstract*—The web has become an essential part of our society and is currently the main medium of information delivery. Billions of users browse the web on a daily basis, and there are single websites that have reached over one billion user accounts. In this environment, the ability to track users and their online habits can be very lucrative for advertising companies, yet very intrusive for the privacy of users.**

In this paper, we examine how web-based device fingerprinting currently works on the Internet. By analyzing the code of three popular browser-fingerprinting code providers, we reveal the techniques that allow websites to track users without the need of client-side identifiers. Among these techniques, we show how current commercial fingerprinting approaches use questionable practices, such as the circumvention of HTTP proxies to discover a user's real IP address and the installation of intrusive browser plugins.

At the same time, we show how fragile the browser ecosystem is against fingerprinting through the use of novel browser-identifying techniques. With so many different vendors involved in browser development, we demonstrate how one can use diversions in the browsers' implementation to distinguish successfully not only the browser-family, but also specific major and minor versions. Browser extensions that help users spoof the user-agent of their browsers are also evaluated. We show that current commercial approaches can bypass the extensions, and, in addition, take advantage of their shortcomings by using them as additional fingerprinting features.

I. Introduction

In 1994, Lou Montulli, while working for Netscape Communications, introduced the idea of cookies in the context of a web browser [1]. The cookie mechanism allows a web server to store a small amount of data on the computers of visiting users, which is then sent back to the web server upon subsequent requests. Using this mechanism, a website can build and maintain state over the otherwise stateless HTTP protocol. Cookies were quickly embraced by browser vendors and web developers. Today, they are one of the core technologies on which complex, stateful web applications are built.

Shortly after the introduction of cookies, abuses of their stateful nature were observed. Web pages are usually comprised of many different resources, such as HTML, images, JavaScript, and CSS, which can be located both on the web server hosting the main page as well as other third-party web servers. With every request toward a third-party website, that website has the ability to set and read previously-set cookies on a user's browser. For instance, suppose that a user browses to *travel.com*, whose homepage includes a remote image from *tracking.com*. Therefore, as part of the process of rendering *travel.com*'s homepage, the user's browser will request the image from *tracking.com*. The web server of *tracking.com* sends the image along with an HTTP Set-Cookie header, setting a cookie on the user's machine, under the *tracking.com* domain. Later, when the user browses to other websites affiliated with *tracking.com*, e.g., *buy.com*, the tracking website receives its previously-set cookies, recognizes the user, and creates a profile of the user's browsing habits. These *third-party cookies*, due to the adverse effects on a user's privacy and their direct connection with online behavioral advertising, captured the attention of both the research community [2], [3], [4] and the popular media outlets [5] and, ever since, cause the public's discomfort [6], [7].

The user community responded to this privacy threat in multiple ways. A recent cookie-retention study by comScore [8] showed that approximately one in three users delete both first-party and third-party cookies within a month after their visit to a website. Multiple browser-extensions are available that reveal third-party tracking [9], as well as the "hidden" third-party affiliations between sites [10]. In addition, modern browsers now have native support for the rejection of all third-party cookies and some even enable it by default. Lastly, a browser's "Private Mode" is also available to assist users to visit a set of sites without leaving traces of their visit on their machine.

This general unavailability of cookies motivated advertisers and trackers to find new ways of linking users to their browsing histories. Mayer in 2009 [11] and Eckersley in 2010 [12] both showed that the features of a browser and its plugins can be fingerprinted and used to track users without the need of cookies. Today, there is a small number of commercial companies that use such methods to provide *device identification* through web-based fingerprinting. Following the classification of Mowery et al. [13], fingerprinting can be used either constructively or destructively. Constructively,

a correctly identified device can be used to combat fraud, e.g., by detecting that a user who is trying to login to a site is likely an attacker who stole a user's credentials or cookies, rather than the legitimate user. Destructively, device identification through fingerprinting can be used to track users between sites, without their knowledge and without a simple way of opting-out. Additionally, device identification can be used by attackers in order to deliver exploits, tailored for specific combinations of browsers, plugins and operating systems [14]. The line between the constructive and destructive use is, however, largely artificial, because the same technology is used in both cases.

Interestingly, companies were offering fingerprinting services as early as 2009, and experts were already voicing concerns over their impact on user privacy [15]. Even when fingerprinting companies honor the recently-proposed "Do Not Track" (DNT) header, the user is still fingerprinted for fraud detection, but the companies *promise* not to use the information for advertising purposes [16]. Note that since the fingerprinting scripts will execute regardless of the DNT value, the verification of this promise is much harder than verifying the effect of DNT on stateful tracking, where the effects are visible at the client-side, in a user's cookies [17].

In this paper, we perform a four-pronged analysis of device identification through web-based fingerprinting. First, we analyze the fingerprinting code of three large, commercial companies. We focus on the differences of their code in comparison to Panopticlick [12], Eckersley's "open-source" implementation of browser fingerprinting. We identify the heavy use of Adobe Flash as a way of retrieving more sensitive information from a client, including the ability to detect HTTP proxies, and the existence of intrusive fingerprinting plugins that users may unknowingly host in their browsers. Second, we measure the adoption of fingerprinting on the Internet and show that, in many cases, sites of dubious nature fingerprint their users, for a variety of purposes. Third, we investigate special JavaScript-accessible browser objects, such as `navigator` and `screen`, and describe novel fingerprinting techniques that can accurately identify a browser even down to its minor version. These techniques involve the ordering of methods and properties, detection of vendor-specific methods, HTML/CSS functionality as well as minor but fingerprintable implementation choices. Lastly, we examine and test browser extensions that are available for users who wish to spoof the identity of their browser and show that, unfortunately *all* fail to completely hide the browser's true identity. This incomplete coverage not only voids the extensions but, ironically, also allows fingerprinting companies to detect the fact that user is attempting to hide, adding extra fingerprintable information.

Our main contributions are:

- We shed light into the current practices of device identification through web-based fingerprinting and propose a taxonomy of fingerprintable information.
- We measure the adoption of fingerprinting on the web.
- We introduce novel browser-fingerprinting techniques that can, in milliseconds, uncover a browser's family and version.
- We demonstrate how over 800,000 users, who are currently utilizing user-agent-spoofing extensions, are more fingerprintable than users who do not attempt to hide their browser's identity, and challenge the advice given by prior research on the use of such extensions as a way of increasing one's privacy [18].

II. Commercial Fingerprinting

While Eckersley showed the principle possibility of fingerprinting a user's browser in order to track users without the need of client-side stateful identifiers [12], we wanted to investigate popular, real-world implementations of fingerprinting and explore their workings. To this end, we analyzed the fingerprinting libraries of three large, commercial companies: BlueCava[1], Iovation[2] and ThreatMetrix[3]. Two of these companies were chosen due to them being mentioned in the web-tracking survey of Mayer and Mitchell [19], while the third one was chosen due to its high ranking on a popular search engine. Given the commercial nature of the companies, in order to analyze the fingerprinting scripts we first needed to discover websites that make use of them. We used Ghostery [9], a browser-extension which lists known third-party tracking libraries on websites, to obtain the list of domains which the three code providers use to serve their fingerprinting scripts. Subsequently, we crawled popular Internet websites, in search for code inclusions, originating from these fingerprinting-owned domains. Once these web sites were discovered, we isolated the fingerprinting code, extracted all individual features, and grouped similar features of each company together.

In this section, we present the results of our analysis, in the form of a taxonomy of possible features that can be acquired through a fingerprinting library. This taxonomy covers all the features described in Panopticlick [12] as well as the features used by the three studied fingerprinting companies. Table I lists all our categories and discovered features, together with the method used to acquire each feature. The categories proposed in our taxonomy resulted by viewing a user's fingerprintable surface as belonging to a layered system, where the "application layer" is the browser and any fingerprintable in-browser information. At the top of this taxonomy, scripts seek to fingerprint and identify any browser customizations that the user has directly or indirectly performed. In lower levels, the scripts target user-specific information around the browser, the operating system and even the hardware and network of a user's

[1] `http://www.bluecava.com`
[2] `http://www.iovation.com`
[3] `http://www.threatmetrix.com`

Fingerprinting Category	Panopticlick	BlueCava	Iovation ReputationManager	ThreatMetrix
Browser customizations	Plugin enumeration$_{(JS)}$ Mime-type enumeration$_{(JS)}$ ActiveX + 8 CLSIDs$_{(JS)}$	Plugin enumeration$_{(JS)}$ ActiveX + 53 CLSIDs$_{(JS)}$ Google Gears Detection$_{(JS)}$		Plugin enumeration$_{(JS)}$ Mime-type enumeration$_{(JS)}$ ActiveX + 6 CLSIDs$_{(JS)}$ Flash Manufacturer$_{(FLASH)}$
Browser-level user configurations	Cookies enabled$_{(HTTP)}$ Timezone$_{(JS)}$ Flash enabled$_{(JS)}$	System/Browser/User Language$_{(JS)}$ Timezone$_{(JS)}$ Flash enabled$_{(JS)}$ Do-Not-Track User Choice$_{(JS)}$ MSIE Security Policy$_{(JS)}$	Browser Language$_{(HTTP, JS)}$ Timezone$_{(JS)}$ Flash enabled$_{(JS)}$ Date & time$_{(JS)}$ Proxy Detection$_{(FLASH)}$	Browser Language$_{(FLASH)}$ Timezone$_{(JS, FLASH)}$ Flash enabled$_{(JS)}$ Proxy Detection$_{(FLASH)}$
Browser family & version	User-agent$_{(HTTP)}$ ACCEPT-Header$_{(HTTP)}$ Partial S.Cookie test$_{(JS)}$	User-agent$_{(JS)}$ Math constants$_{(JS)}$ AJAX Implementation$_{(JS)}$	User-agent$_{(HTTP, JS)}$	User-agent$_{(JS)}$
Operating System & Applications	User-agent$_{(HTTP)}$ Font Detection$_{(FLASH, JAVA)}$	User-agent$_{(JS)}$ Font Detection$_{(JS, FLASH)}$ Windows Registry$_{(SFP)}$	User-agent$_{(HTTP, JS)}$ Windows Registry$_{(SFP)}$ MSIE Product key$_{(SFP)}$	User-agent$_{(JS)}$ Font Detection$_{(FLASH)}$ OS+Kernel version$_{(FLASH)}$
Hardware & Network	Screen Resolution$_{(JS)}$	Screen Resolution$_{(JS)}$ Driver Enumeration$_{(SFP)}$ IP Address$_{(HTTP)}$ TCP/IP Parameters$_{(SFP)}$	Screen Resolution$_{(JS)}$ Device Identifiers$_{(SFP)}$ TCP/IP Parameters$_{(SFP)}$	Screen Resolution$_{(JS, FLASH)}$

Table I
TAXONOMY OF ALL FEATURES USED BY PANOPTICLICK AND THE STUDIED FINGERPRINTING PROVIDERS - SHADED FEATURES ARE, IN COMPARISON TO PANOPTICLICK, EITHER SUFFICIENTLY EXTENDED, OR ACQUIRED THROUGH A DIFFERENT METHOD, OR ENTIRELY NEW

machine. In the rest of this section, we focus on all the non-trivial techniques used by the studied fingerprinting providers that were not previously described in Eckersley's Panopticlick [12].

A. Fingerprinting through popular plugins

As one can see in Table I, all companies use Flash, in addition to JavaScript, to fingerprint a user's environment. Adobe Flash is a proprietary browser plug-in that has enjoyed wide adoption among users, since it provided ways of delivering rich media content that could not traditionally be displayed using HTML. Despite the fact that Flash has been criticized for poor performance, lack of stability, and that newer technologies, like HTML5, can potentially deliver what used to be possible only through Flash, it is still available on the vast majority of desktops.

We were surprised to discover that although Flash reimplements certain APIs existing in the browser and accessible through JavaScript, its APIs do not always provide the same results compared to the browser-equivalent functions. For instance, for a Linux user running Firefox on a 64-bit machine, when querying a browser about the platform of execution, Firefox reports "Linux x86_64". Flash, on the other hand, provides the full kernel version, e.g., Linux 3.2.0-26-generic. This additional information is not only undesirable from a privacy perspective, but also from a security perspective, since a malicious web-server could launch an attack tailored not only to a browser and architecture but to a specific kernel. Another API call that behaves differently is the one that reports the user's screen resolution. In the Linux implementations of the Flash plugin (both Adobe's and Google's), when a user utilizes a dual-monitor setup, Flash reports as the width of a screen the sum of the two individual screens. This value, when combined with the browser's response (which lists the resolution of the monitor were the browser-window is located), allows a fingerprinting service to detect the presence of multiple-monitor setups.

Somewhat surprisingly, none of the three studied fingerprinting companies utilized Java. One of them had some dead code that revealed that in the past it probably did make use of Java, however, the function was not called anymore and the applet was no longer present on the hard-coded location listed in the script. This is an interesting deviation from Panopticlick, which did use Java as an alternate way of obtaining system fonts. We consider it likely that the companies abandoned Java due to its low market penetration in browsers. This, in turn, is most likely caused by the fact that many have advised the removal of the Java plugin from a user's browser [20], [21] due to the plethora of serious Java vulnerabilities that were discovered and exploited over the last few years.

B. Vendor-specific fingerprinting

Another significant difference between the code we analyzed and Panopticlick is that, the fingerprinting companies were not trying to operate in the same way across all browsers. For instance, when recognizing a browser as Internet Explorer, they would extensively fingerprint Internet-Explorer-specific properties, such as `navigator.securityPolicy` and `navigator.systemLanguage`. At the same time, the code accounted for the browser's "short-comings," such as using a lengthy list of predefined CLSIDs for Browser-Helper-Objects (BHOs) due to Internet Explorer's unwillingness to enumerate its plugins.

Listing 1 Side-channel inference of the presence or absence of a font

```
function get_text_dimensions(font){

  h = document.getElementsByTagName("BODY")[0];
  d = document.createElement("DIV");
  s = document.createElement("SPAN");

  d.appendChild(s);
  d.style.fontFamily = font;
  s.style.fontFamily = font;
  s.style.fontSize = "72px";
  s.innerHTML = "font_detection";
  h.appendChild(d);

  textWidth = s.offsetWidth;
  textHeight = s.offsetHeight;
  h.removeChild(d);

  return [textWidth, textHeight];
}
```

C. Detection of fonts

The system's list of fonts can serve as part of a user's unique fingerprint [12]. While a browser does not directly provide that list, one can acquire it using either a browser plugin that willingly provides this information or using a side-channel that indirectly reveals the presence or absence of any given font.

1) Plugin-based detection: ActionScript, the scripting language of Flash, provides APIs that include methods for discovering the list of fonts installed on a running system. While this traditionally was meant to be used as a way of ensuring the correct appearance of text by the plugin, it can also be used to fingerprint the system. Two out of the three studied companies were utilizing Flash as a way of discovering which fonts were installed on a user's computer. Interestingly, only one of the companies was preserving the order of the font-list, which points, most likely, to the fact that the other is unaware that the order of fonts is stable and machine-specific (and can thus be used as an extra fingerprinting feature).

2) Side-channel inference: The JavaScript code of one of the three fingerprinting companies included a fall-back method for font-detection, in the cases where the Flash plugin was unavailable. By analyzing that method, we discovered that they were using a technique, similar to the CSS history stealing technique [22], to identify the presence or absence of any given font - see Listing 1.

More precisely, the code first creates a `<div>` element. Inside this element, the code then creates a `<span>` element with a predetermined text string and size, using a provided font family. Using the `offsetWidth` and `offsetHeight` methods of HTML elements, the script discovers the layout width and height of the element. This code is first called with a "sans" parameter, the font typically used by browsers as a fall-back, when another requested font is unavailable on a user's system. Once the height and text for "sans" are discovered, another script goes over a predefined list of fonts, calling the `get_text_dimensions` function for each one. For any given font, if the current width or height values are different from the ones obtained through the original "sans" measurement, this means that the font does exist and was used to render the predefined text. The text and its size are always kept constant, so that if its width or height change, this change will only be due to the different font. Figure 1 shows three renderings of the same text, with the same font-size but different font faces in Google Chrome. In order to capitalize as much as possible on small differences between fonts, the font-size is always large, so that even the smallest of details in each individual letter will add up to measurable total difference in the text's height and width. If the height and width are identical to the original measurement, this means that the requested font did not exist on the current system and thus, the browser has selected the sans fall-back font. All of the above process, happens in an invisible iframe created and controlled by the fingerprinting script and thus completely hidden from the user.

Font Family	String	Width x Height
Sans	font_detection	519x84
Arial	font_detection	452x83
Calibri	font_detection	416x83

Figure 1. The same string, rendered with different fonts, and its effects on the string's width and height, as reported by the Google Chrome browser

Using this method, a fingerprinting script can rapidly discover, even for a long list of fonts, those that are present on the operating system. The downside of this approach is that less popular fonts may not be detected, and that the font-order is no longer a fingerprintable feature.

D. Detection of HTTP Proxies

One of the features that are the hardest to spoof for a client is its IP address. Given the nature of the TCP protocol, a host cannot pretend to be listening at an IP address from which it cannot reliably send and receive packets. Thus, to hide a user's IP address, another networked machine (a proxy) is typically employed that relays packets between the user that wishes to remain hidden and a third-party. In the context of browsers, the most common type of proxies are HTTP proxies, through which users configure their browsers to send all requests. In addition to manual configuration, browser plugins are also available that allow for a more controlled use of remote proxies, such as the automatic routing of different requests to different proxies based on

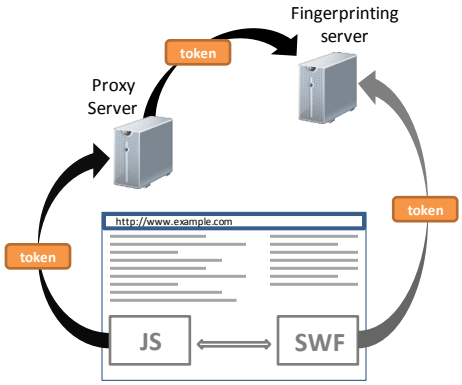

Figure 2. Fingerprinting libraries take advantage of Flash's ability to ignore browser-defined HTTP proxies to detect the real IP address of a user

pattern matching of each request[4], or the cycling of proxies from a proxy list at user-defined intervals[5].

From the point of view of device identification through fingerprinting, a specific IP address is an important feature. Assuming the use of fingerprinting for the detection of fraudulent activities, the distinction between a user who is situated in a specific country and one that *pretends* to be situated in that country, is crucial. Thus, it is in the interest of the fingerprint provider to detect a user's real IP address or, at least, discover that the user is utilizing a proxy server.

When analyzing the ActionScript code embedded in the SWF files of two of the three fingerprinting companies, we found evidence that the code was circumventing the user-set proxies at the level of the browser, i.e., the loaded Flash application was contacting a remote host directly, disregarding any browser-set HTTP proxies. We verified this behavior by employing both an HTTP proxy and a packet-capturing application, and noticing that certain requests were captured by the latter but were never received by the former. In the code of both of the fingerprinting companies, certain long alphanumerical tokens were exchanged between JavaScript and Flash and then used in their communication to the server. While we do not have access to the server-side code of the fingerprinting providers, we assume that the identifiers are used to correlate two possibly different IP addresses. In essence, as shown in Figure 2, if a JavaScript-originating request contains the same token as a Flash-originating request from a different source IP address, the server can be certain that the user is utilizing an HTTP proxy.

Flash's ability to circumvent HTTP proxies is a somewhat known issue among privacy-conscious users that has lead to the disabling of Flash in anonymity-providing applications, like TorButton [23]. Our analysis shows that it is actively exploited to identify and bypass web proxies.

[4]FoxyProxy - `http://getfoxyproxy.org/`
[5]ProxySwitcher - `http://www.proxyswitcher.com/`

E. System-fingerprinting plugins

Previous research on fingerprinting a user's browser focused on the use of popular browser plugins, such as Flash and Java, and utilized as much of their API surface as possible to obtain user-specific data [11], [12]. However, while analyzing the plugin-detection code of the studied fingerprinting providers, we noticed that two out of the three were searching a user's browser for the presence of a special plugin, which, if detected, would be loaded and then invoked. We were able to identify that the plugins were essentially native fingerprinting libraries, which are distributed as CAB files for Internet Explorer and eventually load as DLLs inside the browser. These plugins can reach a user's system, either by a user accepting their installation through an ActiveX dialogue, or bundled with applications that users download on their machines. DLLs are triggered by JavaScript through ActiveX, but they run natively on the user's machine, and thus can gather as much information as the Internet Explorer process.

We downloaded both plugins, wrapped each DLL into an executable that simply hands-off control to the main routine in the DLL and submitted both executables to Anubis [24], a dynamic malware analysis platform that executes submitted binaries in a controlled environment. We focused on the Windows registry values that were read by the plugin, since the registry is a rich environment for fingerprinting. The submitted fingerprinting DLLs were reading a plethora of system-specific values, such as the hard disk's identifier, TCP/IP parameters, the computer's name, Internet Explorer's product identifier, the installation date of Windows, the Windows Digital Product Id and the installed system drivers – entries marked with SFP in Table I.

All of these values combined provide a much stronger fingerprint than what JavaScript or Flash could ever construct. It is also worthwhile mentioning that one of the two plugins was misleadingly identifying itself as "Reputation-Shield" when asking the user whether she wants to accept its installation. Moreover, none of 44 antivirus engines of VirusTotal [25] identified the two DLLs as malicious, even though they clearly belong to the *spyware* category. Using identifiers found within one DLL, we were also able to locate a Patent Application for Iovation's fingerprinting plugin that provides further information on the fingerprinting process and the gathered data [26].

F. Fingerprint Delivery Mechanism

In the fingerprinting experiments of Mayer [11] and Eckersley [12], there was a 1-to-1 relationship between the page conducting the fingerprinting and the backend storing the results. For commercial fingerprinting, however, there is a N-to-1 relationship, since each company provides fingerprinting services to many websites (through the inclusion of third-party scripts) and needs to obtain user fingerprints from each of these sites. Thus, the way that the fingerprint and

the information about it are delivered is inherently different from the two aforementioned experiments.

Through our code analysis, we found two different scenarios of fingerprinting. In the first scenario, the first-party site was not involved in the fingerprinting process. The fingerprinting code was delivered by an advertising syndicator, and the resulting fingerprint was sent back to the fingerprinting company. This was most likely done to combat click-fraud, and it is unclear whether the first-party site is even aware of the fact that its users are being fingerprinted.

In the second scenario, where the first-party website is the one requesting the fingerprint, we saw that two out of the three companies were adding the final fingerprint of the user into the DOM of the hosting page. For instance, `www.imvu.com` is using BlueCava for device fingerprinting by including remote scripts hosted on BlueCava's servers. When BlueCava's scripts combine all features into a single fingerprint, the fingerprint is DES-encrypted (DES keys generated on the fly and then encrypted with a public key), concatenated with the encrypted keys and finally converted to Base64 encoding. The resulting string is added into the DOM of `www.imvu.com`; more precisely, as a new hidden input element in IMVU's login form. In this way, when the user submits her username and password, the fingerprint is also sent to IMVU's web servers. Note, however, that IMVU cannot decrypt the fingerprint and must thus submit it back to BlueCava, which will then reply with a "trustworthiness" score and other device information. This architecture allows BlueCava to hide the implementation details from its clients and to correlate user profiles across its entire client-base. Iovation's fingerprinting scripts operate in a similar manner.

Constrastingly, ThreatMetrix delivers information about users in a different way. The including site, i.e., a customer of ThreatMetrix, creates a session identifier that it places into a `<div>` element with a predefined identifier. ThreatMetrix's scripts, upon loading, read this session identifier and append it to all requests towards the ThreatMetrix servers. This means that the including site never gets access to a user's fingerprint, but only information about the user by querying ThreatMetrix for specific session identifiers.

G. Analysis Limitations

In the previous sections we analyzed the workings of the fingerprinting libraries of three popular commercial companies. The analysis was a mostly manual, time-consuming process, where each piece of code was gradually deobfuscated until the purpose of all functions was clear. Given the time required to fully reverse-engineer each library, we had to limit ourselves to analyze the script of each fingerprinting company as it was seen through two different sites (that is, two different clients of each company). However, we cannot exclude the possibility of additional scripts that are present on the companies' web servers that would perform more operations than the ones we encountered.

III. Adoption of fingerprinting

In Section II, we analyzed the workings of three commercial fingerprinting companies and focused on the differences of their implementations when compared to Panopticlick [12]. In this section, we study the fingerprinting ecosystem, from the point of view of websites that leverage fingerprinting.

A. Adoption on the popular web

To quantify the use of web-based fingerprinting on popular websites, we crawled up to 20 pages for each of the Alexa top 10,000 sites, searching for script inclusions and iframes originating from the domains that the three studied companies utilize to serve their fingerprinting code. To categorize the discovered domains, we made use of the publicly-available domain categorization service of TrendMicro [6], a popular anti-virus vendor.

Through this process, we discovered 40 sites (0.4% of the Alexa top 10,000) utilizing fingerprinting code from the three commercial providers. The most popular site making use of fingerprinting is *skype.com*, while the two most popular categories of sites are: "Pornography" (15%) and "Personals/Dating" (12.5%). For pornographic sites, a reasonable explanation is that fingerprinting is used to detect shared or stolen credentials of paying members, while for dating sites to ensure that attackers do not create multiple profiles for social-engineering purposes. Our findings show that fingerprinting is already part of some of the most popular sites of the Internet, and thus the hundreds of thousands of their visitors are fingerprinted on a daily basis.

Note that the aforementioned adoption numbers are lower bounds since our results do not include pages of the 10,000 sites that were not crawled, either because they were behind a registration wall, or because they were not in the set of 20 URLs for each crawled website. Moreover, some popular sites may be using their own fingerprinting algorithms for performing device identification and not rely on the three studied fingerprinting companies.

B. Adoption by other sites

To discover less popular sites making use of fingerprinting, we used a list of 3,804 domains of sites that, when analyzed by Wepawet [27], requested the previously identified fingerprinting scripts.

Each domain was submitted to TrendMicro's and McAfee's categorization services [7] which provided as output the domain's category and "safety" score. We used two categorizing services in an effort to reduce, as much as possible, the number of "untested" results, i.e., the number of websites not analyzed and not categorized. By examining the results, we extracted as many popular categories as possible

[6] TrendMicro - `http://global.sitesafety.trendmicro.com/`
[7] McAfee - `http://mcafee.com/threat-intelligence/domain/`

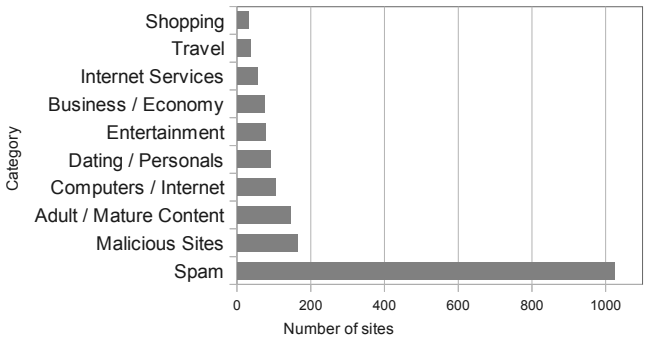

Figure 3. The top 10 categories of websites utilizing fingerprinting

and created aliases for names that were referring to the same category, such as "News / Media" versus "General News" and "Disease Vector" versus "Malicious Site". If a domain was characterized as "dangerous" by one, and "not dangerous" by the other, we accepted the categorization of the latter, so as to give the benefit of the doubt to legitimate websites that could have been compromised, when the former service categorized it.

Given the use of two domain-categorization services, a small number of domains (7.9%) was assigned conflicting categories, such as "Dating" versus "Adult/Mature" and "Business/Economy" versus "Software/Hardware." For these domains, we accepted the characterization of McAfee, which we observed to be more precise than TrendMicro's for less popular domains. Excluding 40.8% of domains which were reported as "untested" by both services, the results of this categorization are shown in Figure 3.

First, one can observe that eight out of the ten categories, include sites which operate with user subscriptions, many of which contain personal and possibly financial information. These sites are usually interested in identifying fraudulent activities and the hijacking of user accounts. The Adult/Mature category seems to make the most use of fingerprinting as was the case with the Alexa top 10,000 sites.

The top two categories are also the ones that were the least expected. 163 websites were identified as malicious, such as using exploits for vulnerable browsers, conducting phishing attacks or extracting private data from users, whereas 1,063 sites were categorized as "Spam" by the two categorizing engines. By visiting some sites belonging to these categories, we noticed that many of them are parked webpages, i.e., they do not hold any content except advertising the availability of the domain name, and thus do not currently include fingerprinting code. We were however able to locate many "quiz/survey" sites that are, at the time of this writing, including fingerprinting code from one of the three studied companies. Visitors of these sites are greeted with a "Congratulations" message, which informs them that they have won and asks them to proceed to receive their prize. At some later step, these sites extract a user's personal details and try to subscribe the user to expensive mobile services.

While our data-set is inherently skewed towards "maliciousness" due to its source, it is important to point out that all of these sites were found to include, at some point in time, fingerprinting code provided by the three studied providers. This observation, coupled with the fact that for all three companies, an interested client must set an appointment with a sales representative in order to acquire fingerprinting services, point to the possibility of fingerprinting companies working together with sites of dubious nature, possibly for the expansion of their fingerprint databases and the acquisition of more user data.

IV. Fingerprinting the behavior of special objects

In Section II, we studied how commercial companies perform their fingerprinting and created a taxonomy of fingerprintable information accessible through a user's browser. In Table I, one can notice that, while fingerprinting companies go to great lengths to discover information about a browser's plugins and the machine hosting the browser, they mostly rely on the browser to willingly reveal its true identity (as revealed through the `navigator.userAgent` property and the User-Agent HTTP header). A browser's user-agent is an important part of a system's fingerprint [18], and thus it may seem reasonable to assume that if users modify these default values, they will increase their privacy by hiding more effectively from these companies.

In this section, however, we demonstrate how fragile the browser ecosystem is against fingerprinting. Fundamental design choices and differences between browser types are used in an effort to show how difficult it can be to limit the exposure of a browser to fingerprinting. Even different versions of the same browser can have differences in the scripting environment that identify the browser's real family, version, and, occasionally, even the operating system. In the rest of this section we describe several novel browser-identifying techniques that: a) can complement current fingerprinting, and b) are difficult to eliminate given the current architecture of web browsers.

A. Experimental Fingerprinting Setup

Our novel fingerprinting techniques focus on the special, browser-populated JavaScript objects; more precisely, the `navigator` and `screen` objects. Contrary to objects created and queried by a page's JavaScript code, these objects contain vendor- and environment-specific methods and properties, and are thus the best candidates for uncovering vendor-specific behaviors.

To identify differences between browser-vendors and to explore whether these differences are consistent among installations of the same browser on multiple systems, we constructed a fingerprinting script that performed a series of

"everyday" operations on these two special objects (such as adding a new property to an object, or modifying an existing one) and reported the results to a server. In this and the following section, we describe the operations of our fingerprinting script and our results. Our constructed page included a JavaScript program that performed the following operations:

1) Enumerated the `navigator` and `screen` object, i.e., request the listing of all properties of the aforementioned objects.
2) Enumerated the `navigator` object again, to ensure that the order of enumeration does not change.
3) Created a custom object, populated it, and enumerated it. A custom, JavaScript-created object, allows us to compare the behavior of browser-populated objects (such as `navigator`) with the behavior of "classic" JavaScript objects.
4) Attempted to delete a property of the `navigator` object, the `screen` object, and the custom object.
5) Add the possibly-deleted properties back to their objects.
6) Attempted to modify an existing property of the `navigator` and `screen` objects.
7) If `Object.defineProperty` is implemented in the current browser, utilize it to make an existing property in the `navigator`, `screen`, and custom object non-enumerable.
8) Attempt to delete the `navigator` and `screen` objects.
9) Attempt to assign new custom objects to the `navigator` and `screen` variable names.

At each step, the objects involved were re-enumerated, and the resulting data was Base64-encoded and sent to our server for later processing. Thus, at the server side, we could detect whether a property was deleted or modified, by comparing the results of the original enumeration with the current one. The enumeration of each object was conducted through code that made use of the *prop in obj* construct, to avoid forcing a specific order of enumeration of the objects, allowing the engine to list object properties in the way of its choosing.

B. Results

By sharing the link to our fingerprinting site with friends and colleagues, we were able, within a week, to gather data from 68 different browsers installations, of popular browsers on all modern operating systems. While our data is small in comparison to previous studies [11], [12], we are not using it to draw conclusions that have statistical relevance but rather, as explained in the following sections, to find deviations between browsers and to establish the consistency of these deviations. We were able to identify the following novel ways of distinguishing between browsers:

Order of enumeration: Through the analysis of the output from the first three steps of our fingerprinting algorithm (Sec. IV-A), we discovered that the order of property-enumeration of special browser objects, like the `navigator` and `screen` objects, is consistently different between browser families, versions of each browser, and, in some cases, among deployments of the same version on different operating systems. While in the rest of this section we focus to the `navigator` object, the same principles apply to the `screen` object.

Our analysis was conducted in the following manner. After grouping the `navigator` objects and their enumerated properties based on browser families, we located the `navigator` object with the least number of properties. This version was consistently belonging to the oldest version of a browser, since newer versions add new properties which correspond to new browser features, such as the `navigator.doNotTrack` property in the newer versions of Mozilla Firefox. The order of the properties of this object, became our baseline to which we compared the `navigator` objects of all subsequent versions of the same browser family. To account for ordering changes due to the introduction of new properties in the `navigator` object, we simply excluded all properties that were not part of our original baseline object, without however changing the relative order of the rest of the properties. For instance, assume an ordered set of features B, where B_0 = {a, b, c, d} and B_1 = {a, b, e, c, d, f}. B_1 has two new elements in comparison with B_0, namely `e` and `f` which, however, can be removed from the set without disrupting the relative order of the rest. For every browser version within the same browser-family, we compared the `navigator` object to the baseline, by first recording and removing new features and then noting whether the order of the remaining features was different from the order of the baseline.

The results of this procedure are summarized in Table II. For each browser family, we compare the ordering of the `navigator` object among up to five different versions. The most current version is denoted as V_C. The first observation is that in almost 20 versions of browsers, no two were ever sharing the same order of properties in the `navigator` object. This feature by itself, is sufficient to categorize a browser to its correct family, regardless of any property-spoofing that the browser may be employing. Second, all browsers except Chrome maintain the ordering of `navigator` elements between versions. Even when new properties were introduced, these do not alter the relative order of all other properties. For instance, even though the newest version of Mozilla Firefox (V_C) has 7 extra features when compared to the oldest version (V_{C-4}), if we ignore these features then the ordering is the same with the original ordering (W).

Google Chrome was the only browser that did not exhibit this behavior. By analyzing our dataset, we discovered that

Browser	V_{c-4}	V_{c-3}	V_{c-2}	V_{c-1}	V_c
Mozilla Firefox	W	W+1	W+4	W+5	W+7
Microsoft IE	-	-	X	X	X+1
Opera	Y	Y+1	Y+1	Y+3	Y+5
Google Chrome	Z	Z	Z′+1	Z″+1	Z‴+1

Table II
DIFFERENCES IN THE ORDER OF NAVIGATOR OBJECTS BETWEEN VERSIONS OF THE SAME BROWSER

Chrome not only changed the order between subsequent versions of the browser, but also between deployments of the same browser on different operating systems. For instance, Google Chrome v.20.0.1132.57 installed on Mac OSX has a different order of elements than the same version installed on a Linux operating system. In Table II, we compare the order of properties of the `navigator` object when the underlying OS is Windows XP. While this changing order may initially appear to be less-problematic than the stable order of other browsers, in reality, the different orderings can be leveraged to detect a specific version of Google Chrome, and, in addition, the operating system on which the browser is running.

Overall, we discovered that the property ordering of special objects, such as the `navigator` object, is consistent among runs of the same browser and runs of the same version of browsers on different operating systems. Contrastingly, the order of properties of a custom script-created object (Step 3 in Section IV-A) was identical among all the studied browsers even though, according to the ECMAScript specification, objects are *unordered* collections of properties [28] and thus the exact ordering can be implementation-specific. More precisely, the property ordering of the custom objects was always the same with the order of property creation.

In general, the browser-specific, distinct property ordering of special objects can be directly used to create models of browsers and, thus, unmask the real identity of a browser. Our findings are in par with the "order-matters" observation made by previous research: Mayer discovered that the list of plugins as reported by browsers was ordered based on the installation time of each individual plugin [11]. Eckersley noticed that the list of fonts, as reported by Adobe Flash and Sun's Java VM, remained stable across visits of the same user [12].

Unique features: During the first browser wars in the mid-90s, browser vendors were constantly adding new features to their products, with the hope that developers would start using them. As a result, users would have to use a specific browser, effectively creating a browser lock-in [29]. The features ranged from new HTML tags to embedded scripting languages and third-party plugins. Signs of this "browser battle" are still visible in the contents of the user-agent string of modern browsers [30].

Browser	Unique methods & properties
Mozilla Firefox	screen.mozBrightness screen.mozEnabled navigator.mozSms + 10
Google Chrome	navigator.webkitStartActivity navigator.getStorageUpdates
Opera	navigator.browserLanguage navigator.getUserMedia
Microsoft IE	screen.logicalXDPI screen.fontSmoothingEnabled navigator.appMinorVersion +11

Table III
UNIQUE METHODS AND PROPERTIES OF THE NAVIGATOR AND SCREEN OBJECTS OF THE FOUR MAJOR BROWSER-FAMILIES

Today, even though the HTML standard is governed by the W3C committee and JavaScript by Ecma International, browser vendors still add new features that do not belong to any specific standard. While these features can be leveraged by web developers to provide users with a richer experience, they can also be used to differentiate a browser from another. Using the data gathered by our fingerprinting script, we isolated features that were available in only one family of browsers, but not in any other. These unique features are summarized in Table III. All browser families had at least two such features that were not shared by any other browser. In many cases, the names of the new features were starting with a vendor-specific prefix, such as `screen.mozBrightness` for Mozilla Firefox and `navigator.msDoNotTrack` for Microsoft Internet Explorer. This is because browser-vendors are typically allowed to use prefixes for features not belonging to a standard or not yet standardized [31]. In the context of fingerprinting, a script can query for the presence or absence of these unique features (e.g., typeof screen.mozBrightness != "undefined") to be certain of the identity of any given browser.

An interesting sidenote is that these unique features can be used to expose the real version of Mozilla Firefox browser, even when the user is using the Torbutton extension. Torbutton replaces the `navigator` and `screen` objects with its own versions, spoofing the values of certain properties, so as to protect the privacy of the user [32]. We installed Torbutton on Mozilla Firefox version 14 and, by enumerating the `navigator` object, we observed that, among others, the Torbutton had replaced the `navigator.userAgent` property with the equivalent of Mozilla Firefox version 10, and it was claiming that our platform was Windows instead of Linux. At the same time, however, special Firefox-specific properties that Mozilla introduced in versions 11 to 14 of Firefox (such as `navigator.mozBattery` and `navigator.mozSms`)

were still available in the `navigator` object. These discrepancies, combined with other weaknesses found in less thorough user-agent-spoofing extensions (see Section V), can uncover not only that the user is trying to hide, but also that she is using Torbutton to do so.

Mutability of special objects: In the two previous sections, we discussed the ability to exploit the enumeration-order and unique features of browsers for fingerprinting. In this section, we investigate whether each browser treats the `navigator` and `screen` objects like regular JavaScript objects. More precisely, we investigate whether these objects are mutable, i.e., whether a script can delete a specific property from them, replace a property with a new one, or delete the whole object. By comparing the outputs of steps four to nine from our fingerprinting algorithm, we made the following observations.

Among the four browser families, only Google Chrome allows a script to delete a property from the `navigator` object. In all other cases, while the "delete" call returns successfully and no exceptions are thrown, the properties remain present in the special object. When our script attempted to modify the value of a property of `navigator`, Google Chrome and Opera allowed it, while Mozilla Firefox and Internet Explorer ignored the request. In the same way, these two families were the only ones allowing a script to reassign `navigator` and `screen` to new objects. Interestingly, no browser allowed the script to simply delete the `navigator` or `screen` object. Finally, Mozilla Firefox behaved in a unique way when requested to make a certain property of the `navigator` object non-enumerable. Specifically, instead of just hiding the property, Firefox behaved as if it had actually deleted it, i.e., it was no longer accessible even when requested by name.

Evolution of functionality: Recently, we have seen a tremendous innovation in Web technologies. The competition is fierce in the browsers' scene, and vendors are trying hard to adopt new technologies and provide a better platform for web applications. Based on that observation, in this section, we examine if we can determine a browser's version based on the new functionality that it introduces. We chose Google Chrome as our testing browser and created a library in JavaScript that tests if specific functionality is implemented by the browser. The features that we selected to capture different functionality were inspired by web design compatibility tests (where web developers verify if their web application is compatible with a specific browser). In total, we chose 187 features to test in 202 different versions of Google Chrome, spanning from version *1.0.154.59* up to *22.0.1229.8*, which we downloaded from *oldapps.com* and which covered all 22 major versions of Chrome. We found that not all of the 187 features were useful; only 109 actually changed during Google Chrome's evolution. These browser versions covered not only releases from the stable channel of Google Chrome, but also from Beta and Dev channels. We refer to a major version as the first number of Google Chrome's versioning system, and to minor version as the full number of the version. We used a virtual machine with Windows XP to setup all browser versions, and used all versions to visit our functionality-fingerprinting page.

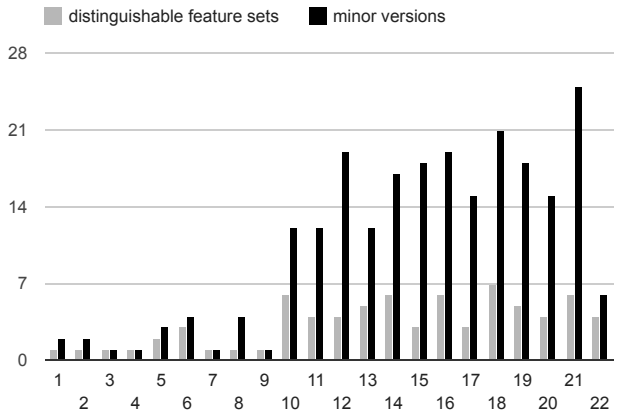

Figure 4. A comparison between how many distinguishable feature sets and minor Google Chrome versions we have per Google Chrome's major versions.

In total, we found 71 sets of features that can be used to identify a specific version of Google Chrome. Each feature set could identify versions that range from a single Google Chrome version up to 14 different versions. The 14 Chrome versions that were sharing the same feature set were all part of the *12.0.742.** releases. Among all 71 sets, there were only four cases where the same feature set was identifying more than a single major version of the browser. In all of these cases, the features overlapped with the first Dev release of the next major version, while subsequent releases from that point on had different features implemented. In Figure 4, we show how many minor versions of Chrome we examined per major version and how many distinct feature sets we found for each major version. The results show that we can not only identify the major version, but in most cases, we have several different feature sets on the same major version. This makes the identification of the exact browser version even more fine-grained.

In Figure 5, we show how one can distinguish all Google Chrome's major versions by checking for specific features. Every pair of major versions is separated by a feature that was introduced into the newer version and did not exist in the previous one. Thus, if anyone wants to distinguish between two consecutive versions, a check of a single feature is sufficient to do so. Notice that our results indicate that we can perform even more fine-grained version detection than the major version of Google Chrome (we had 71 distinct sets of enabled features compared to 22 versions of Chrome), but for simplicity we examined only

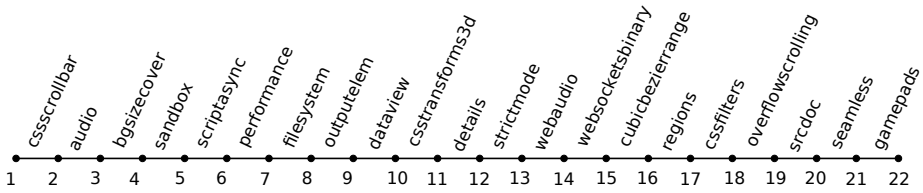

Figure 5. Feature-based fingerprinting to distinguish between Google Chrome major versions

the major version feature changes in detail.

Miscellaneous: In this section, we list additional browser-specific behaviors that were uncovered through our experiment but that do not fall in the previous categories.

Our enumeration of object-properties indirectly uses the method `toString()` for the examined objects. By comparing the formatted output of some specific properties and methods, we noticed that different browsers treated them in slightly different ways. For instance, when calling `toString()` on the natively implemented `navigator.javaEnabled` method, browsers simply state that it is a "native function." Although all the examined browser families print "function javaEnabled() { [native code] }," Firefox uses newline characters after the opening curly-bracket and before the closing one. Interestingly, Internet Explorer does not list the `navigator.javaEnabled` when requested to enumerate the `navigator` object, but still provides the "native function" print-out when asked specifically about the `javaEnabled` method. In the same spirit, when our scripts invoked the `toString()` method on the `navigator.plugins` object, Google Chrome reported "[object DOMPluginArray]," Internet Explorer reported "[object]," while both Mozilla Firefox and Opera reported "[object PluginArray]."

Lastly, while trying out our fingerprinting page with various browsers, we discovered that Internet Explorer lacks native support for Base64 encoding and decoding (`atob` and `btoa`, respectively) which our script used to encode data before sending them to the server.

C. Summary

Overall, one can see how various implementation choices, either major ones, such as the traversal algorithms for JavaScript objects and the development of new features, or minor ones, such as the presence or absence of a newline character, can reveal the true nature of a browser and its JavaScript engine.

V. Analysis of User-Agent-Spoofing Extensions

With the advent of browser add-ons, many developers have created extensions that can increase the security of users (e.g., extensions showing HTML forms with non-secure destinations) or their privacy (e.g., blocking known ads and web-tracking scripts).

Extension	#Installations	User Rating
Mozilla Firefox		
UserAgent Switcher	604,349	4/5
UserAgent RG	23,245	4/5
UAControl	11,044	4/5
UserAgentUpdater	5,648	3/5
Masking Agent	2,262	4/5
User Agent Quick Switch	2,157	5/5
randomUserAgent	1,657	4/5
Override User Agent	1,138	3/5
Google Chrome		
User-Agent Switcher for Chrome	123,133	4/5
User-Agent Switcher	21,108	3.5/5
Ultimate User Agent Switcher, URL sniffer	28,623	4/5

Table IV
List of user-agent-spoofing browser extensions

In the context of this paper, we were interested in studying the completeness and robustness of extensions that attempt to hide the true nature of a browser from an inspecting website. As shown in Table I, while the studied companies do attempt to fingerprint a user's browser customizations, they currently focus only on browser-plugins and do not attempt to discover any installed browser-extensions. Given however the sustained popularity of browser-extensions [33], we consider it likely that fingerprinting extensions will be the logical next step. Note that, unlike browser plugins, extensions are not enumerable through JavaScript and, thus, can only be detected through their side-effects. For instance, some sites currently detect the use of Adblock Plus [34] by searching for the absence of specific iframes and DOM elements that are normally created by advertising scripts.

Since a browser exposes its identity through the user-agent field (available both as an HTTP header and as a property of the JavaScript-accessible `navigator` object), we focused on extensions that advertised themselves as capable of spoofing a browser's user agent. These extensions usually serve two purposes. First, they allow users to surf to websites that impose strict browser requirements onto their visitors, without fulfilling these requirements. For instance, some sites are developed and tested using one specific browser and, due to the importance of the content loading correctly, refuse to load on other browsers. Using a user-agent-spoofing extension, a user can visit such a site, by pretending to use one of the white-listed browsers.

	Google Chrome	Mozilla Firefox	MSIE	Opera
navigator.product	Gecko	Gecko	N/A	N/A
navigator.appCodeName	Mozilla	Mozilla	Mozilla	Mozilla
navigator.appName	Netscape	Netscape	Microsoft Internet Explorer	Opera
navigator.platform	Linux i686	Linux x86_64	Win32	Linux
navigator.vendor	Google Inc.	(empty string)	N/A	N/A

Table V
STANDARD PROPERTIES OF THE NAVIGATOR OBJECT AND THEIR VALUES ACROSS DIFFERENT BROWSER FAMILIES

Another reason for using these extensions is to protect the privacy of a user. Eckeresly, while gathering data for the Panopticlick project, discovered that there were users whose browsers were reporting impossible configurations, for instance, a device was pretending to be an iPhone, but at the same time had Adobe Flash support. In that case, these were users who were obviously trying to get a non-unique browser fingerprint by Panopticlick. Since Eckersley's study showed the viability of using common browser features as parts of a unique fingerprint, it is reasonable to expect that legitimate users utilize such extensions to reduce the trackability of their online activities, even if the extensions' authors never anticipated such a use. Recently, Trusteer discovered in an "underground" forum a spoofing-guide that provided step-by-step instructions for cybercriminals who wished to fool fraud-detection mechanisms that used device-fingerprinting [35]. Among other advice, the reader was instructed to download an extension that changes the User-Agent of their browser to make their sessions appear as if they were originating by different computers with different browsers and operating systems.

Table IV shows the Mozilla Firefox and Google Chrome extensions that we downloaded and tested, together with their user base (measured in July 2012) and the rating that their users had provided. The extensions were discovered by visiting each market, searching for "user-agent" and then downloading all the relevant extensions with a sufficiently large user base and an above-average rating. A high rating is important because it indicates the user's satisfaction in the extension fulfilling its purpose. Our testing consisted of listing the `navigator` and `screen` objects through JavaScript and inspecting the HTTP headers sent with browser requests, while the extensions were actively spoofing the identity of the browser. As in Section IV, we chose to focus on these two objects since they are the ones that are the most vendor-specific as well as the most probed by the fingerprinting libraries. Through our analysis, we discovered that, unfortunately, in *all* cases, the extensions were inadequately hiding the real identity of the browser, which could still be straightforwardly exposed through JavaScript. Apart from being vulnerable to every fingerprinting technique that we introduced in Section IV, each extension had one or more of the following issues:

- **Incomplete coverage of the navigator object.** In many cases, while an extension was modifying the `navigator.userAgent` property, it would leave intact other revealing properties of the navigator object, such as `appName`, `appVersion` and `vendor` - Table V. Moreover, the extensions usually left the `navigator.platform` property intact, which allowed for improbable scenarios, like a Microsoft Internet Explorer browser running on Linux.
- **Impossible configurations.** None of the studied extensions attempted to alter the `screen` object. Thus, users who were utilizing laptops or normal workstations and pretended to be mobile devices, were reporting impossible screen width and height (e.g., a reported 1920x1080 resolution for an iPhone).
- **Mismatch between User-agent values.** As discussed earlier, the user-agent of any given browser is accessible through the HTTP headers of a browser request and through the userAgent property of the `navigator` object. We found that some extensions would change the HTTP headers of the browser, but not of the `navigator` object. Two out of three Chrome extensions were presenting this behavior.

We want to stress that these extensions are not malicious in nature. They are legitimately-written software that unfortunately did not account for all possible ways of discovering the true identity of the browsers on which they are installed. The downside here is that, not only fingerprinting libraries can potentially detect the actual identity of a browser, thus, undermining the goals of the extension, but also that they can discover the discrepancies between the values reported by the extensions and the values reported by the browser, and then use these differences as extra features of their fingerprints. The discrepancies of each specific extension can be modeled and thus, as with Adblock Plus, used to uncover the presence of specific extensions, through their side-effects.

The presence of any user-agent-spoofing extension is a discriminatory feature, under the assumption that the majority of browsing users are not familiar enough with privacy threats (with the possible exception of cookies) to install such spoofing extensions. As a rough metric, consider that the most popular extension for Mozilla Firefox is Adblock Plus [34] that, at the time of this writing, is installed by fifteen million users, 25 times more users than UserAgent Switcher, the most popular extension in Table IV.

We characterize the extension-problem as an *iatrogenic* [8] one. The users who install these extensions in an effort

[8] *iatrogenic* - Of or relating to illness caused by medical examination or treatment.

to hide themselves in a crowd of popular browsers, install software that actually makes them more visible and more distinguishable from the rest of the users, who are using their browsers without modifications. As a result, we advice against the use of user-agent-spoofing extensions as a way of increasing one's privacy. Our findings come in direct antithesis with the advice given by Yen et al. [18], who suggest that user-agent-spoofing extensions *can* be used, as a way of making tracking harder. Even though their study focuses on common identifiers as reported by client-side HTTP headers and the client's IP address, a server capable of viewing these can respond with JavaScript code that will uncover the user-agent-spoofing extension, using any of the aforementioned techniques.

VI. Discussion

Given the intrusive nature of web-based device fingerprinting and the current inability of browser extensions to actually enhance a user's privacy, in this section, we first discuss possible ways of reducing a user's fingerprintable surface and then briefly describe alternative uses of fingerprinting which may become more prevalent in the future.

A. Reducing the fingerprintable surface

Flash. As described in Section II, Adobe Flash was utilized by all three fingerprinting libraries that we studied, due to its rich API that allow SWF files to access information not traditionally available through a browser's API. In all cases, the SWF file responsible for gathering information from the host was hidden from the user, by either setting the width and height of the `<object>` tag to zero, or placed into an iframe of zero height and width. In other words, there was no visible change on the web page that included the fingerprinting SWF files. This observation can be used as a first line of defense. All modern browsers have extensions that disallow Flash and Silverlight to be loaded until explicitly requested by the user (e.g., through a click on the object itself). These hidden files cannot be clicked on and thus, will never execute. While this is a straightforward solution that would effectively stop the Flash-part of the fingerprint of all three studied companies, a circumvention of this countermeasure is possible. By wrapping their fingerprinting code into an object of the first-party site and making that object desirable or necessary for the page's functionality, the fingerprinting companies can still execute their code. This, however, requires much more integration between a first-party website and a third-party fingerprinting company than the current model of "one-size-fits-all" JavaScript and Flash.

In the long run, the best solution against fingerprinting through Flash should come directly from Flash. In the past, researchers discovered that Flash's Local Shared Objects, i.e., Flash's equivalent of browser cookies, were not deleted when a user exited her browser's private mode or even when she used the "Clear Private Data" option of her browser's UI [36]. As a result, in the latest version of Flash, LSOs are not stored to disk but simply kept in memory when the browser's private mode is utilized [37]. Similarly, when a browser enters private mode, Flash could provide less system information, respect any browser-set HTTP proxies and possibly report only a standard subset of a system's fonts, to protect a user's environment from fingerprinting.

JavaScript. There are multiple vendors involved in the development of JavaScript engines, and every major browser is equipped with a different engine. To unify the behavior of JavaScript under different browsers, all vendors would need to agree not only on a single set of API calls to expose to the web applications, but also to internal implementation specifics. For example, hash table implementations may affect the order of objects in the exposed data structures of JavaScript, something that can be used to fingerprint the engine's type and version. Such a consensus is difficult to achieve among all browser vendors, and we have seen diversions in the exposed APIs of JavaScript even in the names of functions that offer the same functionality, e.g., `execScript` and `eval`. Also, based on the fact that the vendors *battle* for best performance of their JavaScript engines, they might be reluctant to follow specific design choices that might affect performance.

At the same time, however, browsers could agree to sacrifice performance when "private-mode" is enabled, where there could be an attempt to expose a unified interface.

B. Alternative uses of fingerprinting

Although, in this paper, we have mostly focused on fingerprinting as a fraud-detection and web-tracking mechanism, there is another aspect that requires attention. Drive-by downloads and web attacks in general use fingerprinting to understand if the browser that they are executing on is vulnerable to one of the multiple available exploits. This way, the attackers can decide, at the server-side, which exploit to *reveal* to the client, exposing as little as they can of their attack capabilities. There are three different architectures to detect drive-by downloads: low-interaction honeypots, high-interaction honeypots and honeyclients. In all three cases, the browser is either a specially crafted one, so that it can instrument the pages visited, or a browser installation that was never used by a real user. Given the precise, browser-revealing, fingerprinting techniques that we described in this paper, it is possible to see in the future these mechanisms being used by attackers to detect monitoring environments and circumvent detection.

VII. Related Work

To the best of our knowledge, this paper is the first that attempts to study the problem of web-based fingerprinting from the perspectives of all the players involved, i.e., from the perspective of the fingerprinting providers and their

fingerprinting methods, the sites utilizing fingerprinting, the users who employ privacy-preserving extensions to combat fingerprinting, and the browser's internals and how they relate to its identity.

Eckersley conducted the first large-scale study showing that various properties of a user's browser and plugins can be combined to form a unique fingerprint [12]. More precisely, Eckersley found that from about 500,000 users who visited `panopticlick.eff.org` and had Flash or Java enabled, 94.2% could be uniquely identified, i.e., there was no other user whose environment produced the same fingerprint. His study, and surprisingly accurate identification results, prompted us to investigate commercial fingerprinting companies and their approach. Yen et al. [18] performed a fingerprinting study, similar to Eckersley's, by analyzing month-long logs of Bing and Hotmail. Interestingly, the authors utilize a client's IP address as part of their tracking mechanism, which Eckersley explicitly avoids dismissing it as "not sufficiently stable." As a way of protecting oneself, the authors advocated the use of user-agent-spoofing extensions. As we discussed in Section V, this is actually counter-productive since it allows for more fingerprinting rather than less.

Mowery et al. [13] proposed the use of benchmark execution time as a way of fingerprinting JavaScript implementations, under the assumption that specific versions of JavaScript engines will perform in a consistent way. Each browser executes a set of predefined JavaScript benchmarks, and the completion-time of each benchmark forms a part of the browser's performance signature. While their method correctly detects a browser-family (e.g., Chrome) 98.2% of the time, it requires over three minutes to fully execute. According to a study conducted by Alenty [38], the average view-time of a web page is 33 seconds. This means that, with high likelihood, the benchmarks will not be able to completely execute and thus, a browser may be misclassified. Moreover, the reported detection rate of more specific attributes, such as the browser-version, operating system and architecture, is significantly less accurate.

Mowery and Shacham later proposed the use of rendering text and WebGL scenes to a `<canvas>` element as another way of fingerprinting browsers [39]. Different browsers will display text and graphics in a different way, which, however small, can be used to differentiate and track users between page loads. While this method is significantly faster than the execution of browser benchmarks, these technologies are only available in the latest versions of modern browsers, thus they cannot be used to track users with older versions. Contrastingly, the fingerprinting techniques introduced in Section IV can be used to differentiate browsers and their versions for any past version.

Olejnik et al. [40] show that web history can also be used as a way of fingerprinting without the need of additional client-side state. The authors make this observation by analyzing a corpus of data from when the CSS-visited history bug was still present in browsers. Today, however, all modern browsers have corrected this issue and thus, extraction of a user's history is not as straightforward, especially without user interaction [41]. Olejnik et al. claim that large script providers, like Google, can use their near-ubiquitous presence to extract a user's history. While this is true [42], most users have first-party relationships with Google, meaning that they can be tracked accurately, without the need of resorting to history-based fingerprinting.

VIII. Conclusion

In this paper, we first investigated the real-life implementations of fingerprinting libraries, as deployed by three popular commercial companies. We focused on their differences when compared to Panopticlick and discovered increased use of Flash, backup solutions for when Flash is absent, broad use of Internet Explorer's special features, and the existence of intrusive system-fingerprinting plugins.

Second, we created our own fingerprinting script, using multiple novel features that mainly focused on the differences between special objects, like the `navigator` and `screen`, as implemented and handled by different browsers. We identified that each browser deviated from all the rest in a consistent and measurable way, allowing scripts to almost instantaneously discover the true nature of a browser, regardless of a browser's attempts to hide it. To this end, we also analyzed eleven popular user-agent spoofing extensions and showed that, even without our newly proposed fingerprinting techniques, all of them fall short of properly hiding a browser's identity.

The purpose of our research was to demonstrate that when considering device identification through fingerprinting, user-privacy is currently on the losing side. Given the complexity of fully hiding the true nature of a browser, we believe that this can be efficiently done only by the browser vendors. Regardless of their complexity and sophistication, browser-plugins and extensions will never be able to control everything that a browser vendor can. At the same time, it is currently unclear whether browser vendors would desire to hide the nature of their browsers, thus the discussion of web-based device fingerprinting, its implications and possible countermeasures against it, must start at a policy-making level in the same way that stateful user-tracking is currently discussed.

Acknowledgments: We want to thank our shepherd and the anonymous reviewers for their valuable comments. For KU Leuven, this research was performed with the financial support of the Prevention against Crime Programme of the European Union (B-CCENTRE), the Research Fund KU Leuven, the EU FP7 projects NESSoS and WebSand, as well as the IWT project SPION. For UCSB, this work was supported by the Office of Naval Research (ONR)

under grant N000140911042, and by the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537, and in part by Secure Business Austria.

REFERENCES

[1] The New York Times - John Schwartz, "Giving the Web a Memory Cost Its Users Privacy," http://www.nytimes.com/2001/09/04/technology/04COOK.html.

[2] B. Krishnamurthy, "Privacy leakage on the Internet," presented at IETF 77, March 2010.

[3] B. Krishnamurthy and C. E. Wills, "Generating a privacy footprint on the Internet," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '06, New York, NY, USA, 2006, pp. 65–70.

[4] F. Roesner, T. Kohno, and D. Wetherall, "Detecting and defending against third-party tracking on the web," in *NSDI'12: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012, pp. 12–12.

[5] The Wall Street Journal, "What They Know," http://blogs.wsj.com/wtk/.

[6] J. Turow, J. King, C. J. Hoofnagle, A. Bleakley, and M. Hennessy, "Americans Reject Tailored Advertising and Three Activities that Enable It," 2009.

[7] B. Ur, P. G. Leon, L. F. Cranor, R. Shay, and Y. Wang, "Smart, useful, scary, creepy: perceptions of online behavioral advertising," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12. New York, NY, USA: ACM, 2012, pp. 4:1–4:15.

[8] comScore, "The Impact of Cookie Deletion on Site-Server and Ad-Server Metrics in Australia," January 2011.

[9] "Ghostery," http:wwww.ghostery.com.

[10] "Collusion: Discover who's tracking you online," http://www.mozilla.org/en-US/collusion/.

[11] J. R. Mayer, "Any person... a pamphleteer," Senior Thesis, Stanford University, 2009.

[12] P. Eckersley, "How Unique Is Your Browser?" in *Proceedings of the 10th Privacy Enhancing Technologies Symposium (PETS)*, 2010.

[13] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in JavaScript implementations," in *Proceedings of W2SP 2011*, H. Wang, Ed. IEEE Computer Society, May 2011.

[14] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *IEEE Symposium on Security and Privacy*, May 2012.

[15] E. Mills, "Device identification in online banking is privacy threat, expert says," CNET News (April 2009).

[16] "Opt out of being tracked," http://www.bluecava.com/preferences/.

[17] J. R. Mayer, "Tracking the Trackers: Early Results — Center for Internet and Society," http://cyberlaw.stanford.edu/node/6694.

[18] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi, "Host Fingerprinting and Tracking on the Web: Privacy and Security Implications," in *Proceddings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[19] J. R. Mayer and J. C. Mitchell, "Third-party web tracking: Policy and technology," in *IEEE Symposium on Security and Privacy*, 2012, pp. 413–427.

[20] G. Cluley, "How to turn off Java on your browser - and why you should do it now," http://nakedsecurity.sophos.com/2012/08/30/how-turn-off-java-browser/.

[21] B. Krebs, "How to Unplug Java from the Browser," http://krebsonsecurity.com/how-to-unplug-java-from-the-browser.

[22] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript Web applications," in *Proceedings of CCS 2010*, Oct. 2010.

[23] "Torbutton: I can't view videos on YouTube and other flash-based sites. Why?" https://www.torproject.org/torbutton/torbutton-faq.html.en#noflash.

[24] "Anubis: Analyzing Unknown Binaries," http://anubis.iseclab.org/.

[25] "VirusTotal - Free Online Virus, Malware and URL Scanner," https://www.virustotal.com/.

[26] G. Pierson and J. DeHaan, "Patent US20080040802 - NETWORK SECURITY AND FRAUD DETECTION SYSTEM AND METHOD."

[27] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web (WWW)*, 2010, pp. 281–290.

[28] "ECMAScript Language Specification, Standard ECMA-262, Third edition."

[29] M. Zalewski, *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2011.

[30] A. Andersen, "History of the browser user-agent string," http://webaim.org/blog/user-agent-string-history.

[31] "Web Tracking Protection," http://www.w3.org/Submission/2011/SUBM-web-tracking-protection-20110224/.

[32] P. Eckersley, "Panopticlick — Self-Defense," https://panopticlick.eff.org/self-defense.php.

[33] J. Scott, "How many Firefox users have add-ons installed? 85%!" https://blog.mozilla.org/addons/2011/06/21/firefox-4-add-on-users/.

[34] "Adblock plus - for annoyance-free web surfing," http://adblockplus.org.

[35] A. Klein, "How Fraudsters are Disguising PCs to Fool Device Fingerprinting," http://www.trusteer.com/blog/how-fraudsters-are-disguising-pcs-fool-device-fingerprinting.

[36] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle, "Flash Cookies and Privacy," in *SSRN preprint (August 2009)*.

[37] J. Xu and T. Nguyen, "Private browsing and Flash Player 10.1," http://www.adobe.com/devnet/flashplayer/articles/privacy_mode_fp10_1.html.

[38] J.-L. Gassée and F. Filloux, "Measuring Time Spent On A Web Page," http://www.cbsnews.com/2100-215_162-5037448.html.

[39] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in HTML5," in *Proceedings of W2SP 2012*, M. Fredrikson, Ed. IEEE Computer Society, May 2012.

[40] Ł. Olejnik, C. Castelluccia, and A. Janc, "Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns," in *the 5th workshop on Hot Topics in Privacy Enhancing Technologies (HOTPETS 2012)*.

[41] Z. Weinberg, E. Y. Chen, P. R. Jayaraman, and C. Jackson, "I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, ser. SP '11, 2011, pp. 147–161.

[42] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. V. Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.