Methodological Analysis of Client Recognition Mechanisms

Jyoti¹, Er. Bharti Grover²

¹Research Scholar, M.Tech Student (CSE), ²HOD, CSE

¹²Gurukul Vidyapeeth institute of Engg &Tech

Abstract: The purpose of correlating future visits from the same person or machine with historical data. Some uses of such tracking techniques are well established and ordinary. For example, they are commonly employed to tell real users from malicious bots, to make it harder for attackers to gain access to compromised accounts, or to store user preferences on a website. In the same vein, the online advertising industry has used cookies as the primary client identification technology since the mid-1990s. Other practices may be less known, may not essentially map to presented browser controls, and may be impossible or difficult to detect. Many of them - in particular, various methods of client fingerprinting - have garnered concerns from software vendors, standards bodies, and the media. To guide us in improving the range of existing browser controls and to highlight the potential pitfalls when designing new web APIs, we decided to prepare a technical overview of known tracking and fingerprinting vectors available in the browser. Note that we describe these vectors, but do not wish this document to be interpreted as a broad invitation to their use. Website owners should keep in mind that any single tracking technique may be conceivably seen as inappropriate, depending on user expectations and other complex factors beyond the scope of this doc. We divided the methods discussed on this page into several categories: explicitly assigned client-side identifiers, such as HTTP cookies; inherent client device characteristics that identify a particular machine; and measurable user behaviors and preferences that may reveal the identity of the person behind the keyboard. After reviewing the known tracking and fingerprinting techniques, we also discuss potential directions for future work and summarize some of the challenges that browser and other software vendors would face trying to detect or prevent such behaviors on the Web.

I. INTRODUCTION

The canonical advance to identifying clients across HTTP needs is to store an exclusive, long-lived token on the client and to programatically retrieve it on subsequent visits. Modern browsers offer a multitude of ways to achieve this goal, including but not limited to:

- Plain old HTTP cookies,
- Cookie-equivalent plugin features - most notably, Flash Local Shared Objects and Silverlight Isolated Storage,
- HTML5 client storage mechanisms, including localStorage, File, and IndexedDB APIs,
- Unique markers stored within locally cached resources or in cache metadata - e.g., Last-Modified and ETag,
- Fingerprints derived from browser-generated Origin-Bound Certificates for SSL connections,
- Bits encoded in HTTP Strict Transport Security pin lists across several attacker-controlled host names,
- Data encoded in SDCH compression dictionaries and dictionary metadata.

II. METHODOLOGY

We believe that the availability of any one of these mechanisms is sufficient to reliably tag clients and identify them later on; in addition to this, many such identifiers can be deployed in a manner that conceals the uniqueness of the ID assigned to a particular client. On the flip side, browsers provide users with some degree of control over the behavior of at least some of these APIs, and with several exceptions discussed later on, the identifiers assigned in this fashion do not propagate to other browser profiles or to private browsing sessions [1]. The remainder of this section provides a more in-depth overview of several notable examples of client tagging schemes that are within the reach of web apps.

A. HTTP cookies

HTTP cookies are the most familiar and best-understood method for persisting data on the client. In essence, any web server may issue unique identifiers to first-time visitors as a part of a HTTP response, and have the browser play back the stored values on all future requests to a particular site. All major browsers have for years been equipped with UIs for managing cookies; a large number of third-party cookie management and blocking software is available, too. In practice, however, external research has implied that only a minority of users regularly reviews or purge browser cookies. The reasons for this are probably complex, but one of them may be that the removal of cookies tends to be disruptive: contemporary browsers do not provide any heuristics to distinguish between the session cookies that are needed to access the sites the user is logged in, and the rest. Some browsers offer user-configurable restrictions on the ability for websites to set “third-party” cookies (that is, cookies coming from a domain other than the one currently displayed in the address bar - a behavior most commonly employed to serve online ads or other embedded content). It should be noted that the existing implementations of this setting will assign the “first-party” label to any cookies set by documents intentionally navigated to by the user, as well as to ones issued by content loaded by the browser as a part of full-page interstitials, HTTP redirects, or click-triggered pop-ups. Compared to most other mechanisms discussed below, overt use of HTTP cookies is fairly transparent to the user. That said, the mechanism may be used to tag clients without the use of cookie values that obviously resemble unique IDs. For example, client identifiers could be encoded as a combination of several seemingly innocuous and reasonable cookie names, or could be stored...
in metadata such as paths, domains, or cookie expiration times. Because of this, we are not aware of any means for a browser to reliably flag HTTP cookies employed to identify a specific client in this manner. Just as interestingly, the abundance of cookies means that an actor could even conceivably rely on the values set by others, rather than on any newly-issued identifiers that could be tracked directly to the party in question. We have seen this employed for some rich content ads, which are usually hosted in a single origin shared by all advertisers - or, less safely, are executed directly in the context of the page.

B. Flash LSOs

Local Shared Objects are the canonical way to store client-side data within Adobe Flash. The mechanism is designed to be a direct counterpart to HTTP cookies, offering a convenient way to maintain session identifiers and other application state on a per-origin basis. In contrast to cookies, LSOs can be also used for structured storage of data other than short snippets of text, making such objects more difficult to inspect and analyze in a streamlined way.

In the past, the behavior of LSOs within the Flash plugin had to be configured separately from any browser privacy settings: by visiting a lesser-known Flash Settings Manager UI hosted on macromedia.com (standalone installs of Flash 10.3 and above supplanted this with a Control Panel / System Preferences dialog available locally on the machine). Today, most browsers offer a degree of integration: for example, clearing cookies and other site data will generally also remove LSOs. On the flip side, more nuanced controls may not be synchronized: say, the specific setting for third-party cookies in the browser is not always reflected by the behavior of LSOs. From a purely technical standpoint, the use of Local Shared Objects in a manner similar to HTTP cookies is within the apparent design parameters for this API - but the reliance on LSOs to recreate deleted cookies or bypass browser cookie preferences has been subject to public scrutiny [11].

C. Silverlight Isolated Storage

Microsoft Silverlight is a widely-deployed applet framework bearing many similarities to Adobe Flash. The Silverlight equivalent of Flash LSOs is known as Isolated Storage. The privacy settings in Silverlight are typically not coupled to the underlying browser. In our testing, values stored in Isolated Storage survive clearing cache and site data in Chrome, Internet Explorer and Firefox. Perhaps more surprisingly, Isolated Storage also appears to be shared between all non-incognito browser windows and browser profiles installed on the same machine; this may have consequences for users who rely on separate browser instances to maintain distinct online identities. As with LSOs, reliance on Isolated Storage to store session identifiers and similar state information does not present issues from a purely technical standpoint. That said, given that the mechanism is not currently managed via browser controls, its use of for client identification is not commonplace and thus may be viewed as less transparent than standard cookies.

D. HTML5 client-side storage mechanisms

HTML5 introduces a range of structured data storage mechanisms on the client; this includes the File API. Although semantically different from each other, all of them are designed to allow persistent storage of arbitrary blobs of binary data tied to a particular web origin. In contrast to cookies and LSOs, there are no significant size restrictions on the data stored with these APIs. In modern browsers, HTML5 storage is usually purged alongside other site data, but the mapping to browser settings isn’t necessarily obvious. For example, Firefox will retain local Storage data unless the user selects “offline website data” or “site preferences” in the deletion dialog and specifies the time range as “everything” (this is not the default). Another idiosyncrasy is the behavior of Internet Explorer, where the data is retained for the lifetime of a tab for any sites that are open at the time the operation takes place. Beyond that, the mechanisms do not always appear to follow the restrictions on persistence that applies to HTTP cookies. For example, in our testing, in Firefox, local Storage can be written and read in cross-domain frames even if third-party cookies are disabled. Due to the similarity of the design goals of these APIs, the authors expect that the perception and the caveats of using HTML5 storage for storing session identifiers would be similar to the situation with Flash and Silverlight.

E. Cached objects

For performance reasons, all mainstream web browsers maintain a global cache of previously retrieved HTTP resources. Although this mechanism is not explicitly designed as a random-access storage mechanism, it can be easily leveraged as such. To accomplish this, a cooperating server may return, say, a JavaScript document with a unique identifier embedded in its body, and set Expires / max-age= headers to a date set in the distant future. Once this unique identifier is stored within a script sub resource in the browser cache, the ID can be read back on any page on the Internet simply by loading the script from a known URL and monitoring the agreed-upon local variable or setting up a predefined callback function in JavaScript[9]. The browser will periodically check for newer copies of the script by issuing a conditional request to the originating server with a suitable If-Modified-Since header; but if the server consistently responds to such check with HTTP code 304 (“Not modified”), the old copy will continue to be reused indefinitely. There is no concept of blocking “third-party” cache objects in any browser known to the authors of this document, and no simple way to prevent cache objects from being stored without dramatically degrading performance of everyday browsing. Automated detection of such behaviors is extremely difficult owing to the sheer volume and complexity of cached JavaScript documents encountered on the modern Web. All browsers expose the option to manually clear the document cache. That said, because clearing the cache requires specific action on the part of the user, it is unlikely to be done regularly, if at all. Leveraging the browser cache to store session identifiers is very distinct from using HTTP cookies; the authors are unsure if and how the cookie settings - the convenient abstraction layer used for most of the other mechanisms discussed to date - could map to the semantics of browser caches.

F. Cache metadata: ETag and Last-Modified

To make implicit browser-level document caching work properly, servers must have a way to notify browsers that a newer version of a particular document is available for
related site data; the tags can be destroyed only by purging mechanisms are unaffected by the deletion of cookies and through cache objects, both of these “metadata” most browsers don’t even require the string to be a well-formed date scheme, tracking date back at least to 2000. The other versioning observation, and its potential applications for browser installations and in private browsing modes, which may be probed using a variety of techniques. In top of the fingerprinting mechanisms associated with HTTP caching and with the purpose-built APIs available to JavaScript programs and plugin-executed code, modern browsers provide several network-level features that offer an opportunity for storing unique identifiers on the browser, re-navigation to any participating domain will allow the window-bound token to be retrieved and the new session to be associated with the previously collected data.

### A. Lower-level protocol identifiers

On top of the fingerprinting mechanisms associated with HTTP caching and with the purpose-built APIs available to JavaScript programs and plugin-executed code, modern browsers provide several network-level features that offer an opportunity to store or retrieve unique identifiers:

- **Origin Bound Certificates (aka ChannellID)** are persistent self-signed certificates identifying the client to an HTTPS server, envisioned as the future of session management on the web[8]. A separate certificate is generated for every newly encountered domain and reused for all connections initiated later on[8]. By design, OBCs function as unique and stable client fingerprints, essentially replicating the operation of authentication cookies; they are treated as “site and plug-in data” in Chrome, and can be removed along with cookies. Uncharacteristically, sites can leverage OBC for user tracking without performing any actions that would be visible to the client: the ID can be derived simply by taking note of the cryptographic hash of the certificate automatically supplied by the client as a part of a legitimate SSL handshake. Channel ID is currently suppressed in Chrome in “third-party” scenarios (e.g., for different-domain frames).

- **HTTP Strict Transport Security** is a security mechanism that allows servers to demand that all future connections to a particular host name need to happen exclusively over HTTPS, even if the

### III. SCRIPT-ACCESSIBLE STORAGE

Several other more limited techniques make it possible for JavaScript or other active content running in the browser to maintain and query client state, sometimes in a fashion that can survive attempts to delete all browsing and site data. For example, it is possible to use `window.name` or `sessionStorage` to store persistent identifiers for a given window: if a user deletes all client state but does not close a tab that at some point in the past displayed a site determined to track the browser, re-navigation to any participating domain will allow the window-bound token to be retrieved and the new session to be associated with the previously collected data.

### G. HTML5 AppCache

Application Caches allow website authors to specify that portions of their websites should be stored on the disk and made available even if the user is offline. The mechanism is controlled by cache manifests that outline the rules for storing and retrieving cache items within the app. Similarly to implicit browser caching, AppCaches make it possible to store unique, user-dependent data - be it inside the cache manifest itself, or inside the resources it requests. The resources are retained indefinitely and not subject to the browser’s usual cache eviction policies, AppCache appears to occupy a netherworld between HTML5 storage mechanisms and the implicit browser cache. In some browsers, it is purged along with cookies and stored website data; in others, it is discarded only if the user opts to delete the browsing history and all cached documents.

### H. Flash resource cache

Flash maintains its own internal store of resource files, which can be probed using a variety of techniques. In particular, the internal repository includes an `asset cache`, relied upon to store Runtime Shared Libraries signed by Adobe to improve applet load times. There is also Adobe Flash Access, a mechanism to store automatically acquired licenses for DRM-protected content. As of this writing, these document caches do not appear to be coupled to any browser privacy settings and can only be deleted by making several independent configuration changes in the Flash Settings Manager UI on macromedia.com. We believe there is no global option to delete all cached resources or prevent them from being stored in the future. Browsers other than Chrome appear to share Flash asset data across all installations and in private browsing modes, which may have consequences for users who rely on separate browser instances to maintain distinct online identities [7].

### I. SDCH dictionaries

SDCH is a Google-developed compression algorithm that relies on the use of server-supplied, cacheable dictionaries to achieve compression rates considerably higher than what’s possible with methods such as gzip or deflate for several common classes of documents. The site-specific dictionary caching behavior at the core of SDCH inevitably offers an opportunity for storing unique identifiers on the client: both the dictionary IDs (echoed back by the client using the `Avail-Dictionary` header), and the contents of the dictionaries themselves, can be used for this purpose, in a manner very similar to the regular browser cache. In Chrome, the data does not persist across browser restarts; it was, however, shared between profiles and incognito modes and was not deleted with other site data when such an operation is requested by the user. Google addressed this in bug 327783[6].

---

**International Journal of Technology and Computing (IJTC)**

**ISSN-2455-099X**

**Volume 3, Issue 1 January 2017**
original URL nominally begins with “http://”. It follows that a fingerprinting server could set long-lived HSTS headers for a distinctive set of attacker-controlled host names for each newly encountered browser; this information could be then retrieved by loading faux (but possibly legitimately-looking) sub resources from all the designated host names and seeing which of the connections are automatically switched to HTTPS. In an attempt to balance security and privacy, any HSTS pins set during normal browsing are carried over to the incognito mode in Chrome; there is no propagation in the opposite direction, however. It is worth noting that leveraging HSTS for tracking purposes would require establishing \( \log(n) \) connections to uniquely identify \( n \) users, which makes it relatively unattractive, except for targeted uses; that said, creating a smaller number of buckets may be a valuable tool for refining other imprecise fingerprinting signals across a very large user base.\( \text{.st} \) but not least, virtually all modern browsers maintain internal DNS caches to speed up name resolution (and, in some implementations, to mitigate the risk of DNS rebinding attacks). Such caches can be easily leveraged to store small amounts of information for a configurable amount of time; for example, with 16 available IP addresses to choose from, around 8-9 cached host names would be sufficient to uniquely identify every computer on the Internet. On the flip side, the value of this approach is limited by the modest size of browser DNS caches and the potential conflicts with resolver caching on ISP level[12].

IV. MACHINE-SPECIFIC CHARACTERISTICS

With the notable exception of Origin-Bound Certificates, the techniques described in section 1 of the document rely on a third-party website explicitly placing a new unique identifier on the client system. Another, less obvious approach to web tracking relies on querying or indirectly measuring the inherent characteristics of the client system. Individually, each such signal will reveal just several bits of information - but when combined together, it seems probable that they may uniquely identify almost any computer on the Internet. In addition to being harder to detect or stop, such techniques could be used to cross-correlate user activity across various browser profiles or private browsing sessions.

V. NETWORK CONFIGURATION FINGERPRINTS

An interesting set of additional device characteristics is associated with the architecture of the local network and the configuration of lower-level network protocols; such signals are disclosed independently of the design of the web browser itself. These traits covered here are generally shared between all browsers on a given client and cannot be easily altered by common privacy-enhancing tools or practices; they include:

- The external client IP address. For IPv6 addresses, this vector is even more interesting: in some settings, the last octets may be derived from the device’s MAC address and preserved across networks.
- A broad range of TCP/IP and TLS stack fingerprints, obtained with passive tools such as \( \text{p0f} \). The information disclosed on this level is often surprisingly specific: for example, TCP/IP traffic will often reveal high-resolution system uptime data through TCP timestamps.
- Ephemeral source port numbers for outgoing TCP/IP connections, generally selected sequentially by most operating systems.
- The local network IP address for users behind network address translation or HTTP proxies (via WebRTC). Combined with the external client IP, internal NAT IP uniquely identifies most users, and is generally stable for desktop browsers (due to the tendency for DHCP clients and servers to cache leases).
- Information about proxies used by the client, as detected from the presence of extra HTTP headers (Via, \( \text{X-Forwarded-For} \)). This can be combined with the client’s actual IP address revealed when making proxy-bypassing connections using one of several available methods.

VI. USER-DEPENDENT BEHAVIORS & PREFERENCES

In addition to trying to uniquely identify the device used to browse the web, some parties may opt to examine characteristics that aren’t necessarily tied to the machine, but that are closely associated with specific users, their local preferences, and the online behaviors they exhibit. Similarly to the methods described in section 2, such patterns would persist across different browser sessions, profiles, and across the boundaries of private browsing modes. The following data is typically open to examination:

- Preferred language, default character encoding, and local time zone (sent in HTTP headers and visible to JavaScript).
- Data in the client cache and history. It is possible to detect items in the client’s cache by performing simple timing attacks; for any long-lived cache items associated with popular destinations on the Internet, a finger printer could detect their presence simply by measuring how quickly they load (and by aborting the navigation if the latency is greater than expected for local cache). long haul, particularly if they are attempted on the client (where the party seeking to avoid detection can reverse-engineer the checks and iterate until the behavior is no longer flagged as suspicious).
- The distinction between behaviors that may be acceptable to the user and ones that might not is hidden from view: for example, a cookie set for abuse detection looks the same as a cookie set to track online browsing habits. Without a way to distinguish between the two and properly classify the observed behaviors, tracking detection mechanisms may provide little real value to the user [10].

VII. POTENTIAL DIRECTIONS FOR FUTURE WORK

There may be no simple, universal, technical solutions to the problem of tracking on the Web by parties who are
intent on doing so with no regard for user controls. That said, the authors of this page see some theoretical room for improvement when it comes to building simpler and more intuitive privacy controls to provide a better framework for the bulk of interactions with responsible sites and parties on the Internet:

- The current browser privacy controls evolved almost exclusively around the notion of HTTP cookies and several other very specific concepts that do not necessarily map cleanly to many of the tracking and fingerprinting methods discussed in this document. In light of this, to better meet user expectations, it may be beneficial for in-browser privacy settings to focus on clearly explaining practical privacy outcomes, rather than continuing to build on top of narrowly-defined concepts such as “third-party cookies”[3].

- We worry that in some cases, interacting with browser privacy controls can degrade one’s browsing experience, discouraging the user from ever touching them. A canonical example of this is trying to delete cookies: reviewing them manually is generally impractical; while deleting all cookies will kick the user out of any sites he or she is logged into and frequents every day. Although fraught with some implementation challenges, it may be desirable to build better heuristics that distinguish and preserve site data specifically for the destinations that users frequently log into or meaningfully interact with[5].

- Even for extremely privacy-conscious users who are willing to put up with the inconvenience of deleting one’s cookies and purging other session data, resetting online fingerprints can be difficult and fail in unexpected ways. An example of this is discussed in section 1: if there are ads loaded on any of the currently open tabs, clearing all local data may not actually result in a clean slate. Investing in developing technologies that provide more robust and intuitive ways to maintain, manage, or compartmentalize one’s online footprints may be a noble goal.

- Today, some privacy-conscious users may resort to tweaking multiple settings and installing a broad range of extensions that together have the paradoxical effect of facilitating fingerprinting - simply by making their browsers considerably more distinctive, no matter where they go. There is a compelling case for improving the clarity and effect of a handful of well-defined privacy settings as to limit the probability of such outcomes.

We present these ideas for discussion within the community; at the same time, we recognize that although they may sound simple when expressed in a single paragraph, their technical underpinnings are elusive and may prove difficult or impossible to fully flesh out and implement in any browser [4].

REFERENCES:

[3]. https://www.nczonline.net/blog/2009/05/05/http-cookies-explained
[5]. https://support.google.com/chrome/answer/95647?co=GENIE.Platform%3DDesktop&hl=en-GB
[8]. https://www.bountysource.com/teams/dhowe-adnauseam/issues