DNS Cache-Based User Tracking

Amit Klein
Bar-Ilan University
aksecurity@gmail.com

Benny Pinkas
Bar-Ilan University
benny@pinkas.com

***Abstract*—We describe a novel user tracking technique that is based on assigning statistically unique DNS records per user. This new tracking technique is unique in being able to distinguish between machines that have identical hardware and software, and track users even if they use "privacy mode" browsing, or use multiple browsers (on the same machine).**

The technique overcomes issues related to the caching of DNS answers in resolvers, and utilizes per-device caching of DNS answers at the client. We experimentally demonstrate that it covers the technologies used by a very large fraction of Internet users (in terms of browsers, operating systems, and DNS resolution platforms).

Our technique can track users for up to a day (typically), and therefore works best when combined with other, narrower yet longer-lived techniques such as regular cookies - we briefly explain how to combine such techniques.

We suggest mitigations to this tracking technique but note that it is not easily mitigated. There are possible workarounds, yet these are not without setup overhead, performance overhead or convenience overhead. A complete mitigation requires software modifications in both browsers and resolver software.

I. Introduction

Online browser-based user tracking is prevalent. Tracking is used to identify users and track them across many sessions and websites on the Internet. Tracking is often performed in order to personalize advertisements or for surveillance purposes. It can either be done by sites that are visited by users, or by third-party companies which track users across multiple web sites and applications.

Existing tracking mechanisms are usually based on either *tagging* or *fingerprinting*. With tagging, the tracking party stores at the user's device some information, such as a cookie, which can later be tracked. Modern web standards and norms, however, enable users to opt-out from tagging. Furthermore, tagging is often specific for one application or browser, and therefore a tag that was stored in one browser cannot be identified when the user is using a different browser on the same machine, or when the user uses the private browsing feature of the browser (usage estimates varies: 20.1% [14] and 46% [12]). Fingerprinting is implemented by having the tracking party measure features of the user's machine (for example the set of installed fonts). Corporates, however, often install a single "golden image" (standard package of software installations) on many *identical* (hardware-wise) employee machines, and therefore it is hard to obtain fingerprints that can separate these machines from one another. For example, according to [25], for Windows 10 migration, 73% of organizations use imaging software or MDT tools - this, combined with identical hardware, forms a "golden image" scenario.

In this work we present a new tracking mechanism which is based on caching DNS answers. It is the first tracking technique that is able to cross the private browsing boundary (i.e. compute the same tracking ID for a private mode tab/window of a browser as for a regular tab/window of the browser), and in many cases - also the inter-browser gap, and simultaneously address the "golden image" problem, all this while maintaining a very good coverage of the platforms involved. To our knowledge, no other tracking method (or combination thereof) achieves all these goals simultaneously, i.e. can detect different users using the same "golden image", while detecting the same user using private browsing mode and across different browsers.

The new tracking mechanism stops working when the user machine is restarted, moves to a different network, or when reaching the TTL limit (which oftentimes is one day). Therefore, following the concept of Evercookie [18], tracking might work best in combination with other tracking techniques which are more persistent but have less coverage. Assume for example that tracking with the new technique is done using a "DNS ID" (explained later in the text) that has coverage over different browsers and privacy modes, but a relatively short TTL, and using cookies that have low coverage but long persistence. In this setting, a browser B_1 might visit site X and receive both a long-term tracking cookie C_1, and a DNS ID i_1. After an hour, B_1 visits site Y in privacy mode. The cookie C_1 is not sent, but our method still tags the visit with the same DNS ID i_1, and thus associates the session with the same user. Later the user uses a different browser B_2 to visit site Z. The system sends B_2 a new tracking cookie, C_2, and associates it with the same user due to the identical DNS ID i_1. Afterwards, all activities by either B_1 or B_2 are associated with the same user. At a later time the DNS ID might change (due to TTL issues). However, visits by either B_1 or B_2 are still associated with the user, and the user is given a new DNS ID which is associated with the same identity, and enables to track the user in private browsing or when using other browsers. To sum up, the DNS ID enables tracking to cross over private-browsing or multiple-browser gaps, while cookies and similar tagging

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23186
www.ndss-symposium.org

cross the TTL gap. Together they enable comprehensive and long term tracking of users.

The tracking mechanism we devised is not easily mitigated, especially not systematically. To fully address it, changes must be introduced to the way browsers, stub resolvers and DNS resolution platforms work. See full discussion in Section VI.

We exploit the DNS caching mechanism employed by the stub resolver (part of the operating system) on the user's device. Caching at large is a well known technique to expedite access to remote resources (e.g. DNS records stored on a DNS server) when these are used repetitively. The essence of caching is having the resource fetched once, then stored locally, in a dedicated cache storage, so that whenever this resource is needed later it can be fetched from the local cache storage, rather than from the remote storage.

A. Tracking and why it is important

Browser-based tracking of users or devices is a topic of much research over the last decade or two. This topic is important since the vast majority of online user activity is conducted via web browsing (at least in desktops/laptops), and websites have vast economic incentives to track users as they navigate from site to site. [6] specifically lists motivations for web-based fingerprinting as "fraud detection, protection against account hijacking, anti-bot and anti-scraping services, enterprise security management, protection against DDOS attacks, real-time targeted marketing, campaign measurement, reaching customers across devices, and limiting number of access to services". Tracking might be even more lucrative when private browsing is involved – since users that engage in private browsing clearly intend to protect their privacy, and thus compromising their privacy is probably more valuable. (For example, an insurance company might wish to identify potential customers whose browsing history associates them with a greater risk.)

We stress that we do not argue in this paper about the morality or legality of tracking. We present the new tracking technique in order to make users and manufacturers aware of it, and deploy mitigations against it.

Tracking can be used for more consensual purposes. For example, an anti-fraud functionality at a financial web site would benefit from user tracking as this can be used to better assess the risk of identity theft for a given session, and reduce the amount of extended authentication procedures needed (which can annoy innocent users). In such a scheme, the financial web site uses extended authentication for the user upon the first access to the website, and sets a tracking ID for the user. Upon a subsequent access to the website, even from a different browser or via privacy mode, the user is identified via the tracking ID, and only an abbreviated version of the authentication procedure is employed, as the risk for identity theft is much lower.

There are some obvious general requirements from a device tracking mechanism, such as the need to be browser-borne with the ability to be embedded in pages of arbitrary websites, to require zero interaction with the user, and to offer maximal longevity (ideally working across browser restarts). These are achieved by many known techniques. However, the ideal user/device tracking mechanism should also exhibit the following properties:

- Support a considerable (but not necessarily full) market share of the underlying (non-obsolete) technology. For example, the tracking method can assume browser support for HTML5 and Javascript, but cannot assume the availability of Java, Flash, Silverlight or other extensions. Particularly, it must not be limited to a particular operating system or a particular browser.
- Work across the "regular browser" / "privacy mode" boundary (the "privacy mode boundary"), and ideally across browsers.
- Uniquely identify devices even if they have the same hardware and software image (the "golden image" challenge).

B. The basic idea

The main concept of DNS cache-based user tracking is to place a statistically unique combination of DNS data in the user's stub resolver DNS cache. This data can then be used to tag the user (device) as long as the data remains in the DNS cache of the device. For practicality, this procedure should be carried out from the browser, by a 3rd-party HTML+Javascript code (the *tracking snippet*). Specifically, the tracking data is a set of DNS A/AAAA records which are the DNS resolutions for a set of host names in the domain (or domains) controlled by the tracker.

To make the example concrete, the tracking snippet can be implemented as a series of HTTP requests from hosts in the tracker's domain, say $\texttt{x}_i\texttt{.anonymity.fail}$. During the DNS resolution which precedes the HTTP access, the stub resolver sends DNS queries for those hosts, and receives for each host a set of tracker-owned IP addresses, in random order. The stub resolver caches the sets, retaining the order of each set. The browser then uses the first IP address in each set to send the HTTP request to. Each web server operated by the tracker returns a different answer (marking the individual web server accessed), and thus the tracking snippet can collect these values and form a tracking ID which can be communicated to the web site whose page is rendered by the browser. Note that since the order of the IP address list returned with each DNS answer is randomized, each stub resolver gets a different series of lists, thereby achieving statistical uniqueness of the tracking ID. On the other hand, all browsers on the user's device that use the stub resolver will report the same tracking ID. A more detailed explanation of the technique, as well as details regarding how to force the stub resolver to receive sufficiently random lists, are provided in section III.

C. Our contributions

A new tracking method: Our main contribution is a new user tracking method, which exhibits sufficient coverage, crosses the privacy mode boundary, and distinguishes among devices that were generated from the same "golden image".

The detailed technical analysis of the new tracking method, provided two additional contributions, listed below.

The first detailed analysis of DNS cache behavior in different platforms (stub resolver × browser combinations). We studied the specific behavior of browsers, operating systems and resolvers, in order to evaluate the effectiveness of our technique. To the best of our knowledge this is the first study that details some specific (and sometimes peculiar) DNS cache behaviors of browsers, stub resolvers and caching resolvers.

Load balancing strategies: To the best of our knowledge this is the first research of DNS load balancing strategies. In addition to the importance of this analysis to DNS based tracking, this analysis can help an attacker optimize a DNS cache poisoning attack against the DNS resolution platform, e.g. by forcing DNS queries to be routed to a specific resolver, thus attacking one DNS resolver at a time.

II. Related work

Many tracking techniques were suggested in prior research. At large, proposals can be categorized by their passive/active nature. We use the terminology defined in [31]:

- A *fingerprinting* technique measures properties already existing in the browser (or operating system), ideally collecting a combination of data that uniquely identifies the browser/device without altering the state of the browser.
- A *tagging* technique, in contrast, stores data in the browser/device that identifies the browser/device. Further access to the browser can "read" the data and identify the device.

Of course, fingerprinting techniques have advantages over tagging techniques, since storing data on the device is more easily monitored and evaded.

A comprehensive collection of tracking methods and how they work can be found in Google Chromium's web page "Technical analysis of client identification mechanisms" ([17]).

A. Fingerprinting

There is a major drawback to fingerprinting techniques, which is that they cannot guarantee the uniqueness of the device ID. This problem becomes acute when considering organizations wherein desktops and laptops are cloned from "golden images", thus making those devices practically indistinguishable for passive techniques. (Furthermore, since fingerprinting techniques are known and understood nowadays, countermeasures are already deployed against some of these techniques.) For example, font-based fingerprinting, User-Agent header fingerprinting, WebGL (canvas) fingerprinting, browser plugin/extension fingerprinting, CPU/GPU performance fingerprinting are all methods that cannot distinguish between systems that are based on the exact same hardware and software.

There are some proposals for DNS-based fingerprinting methods. It was suggested in [8] to use the DNS resolver IP address. However, in an enterprise (or an ISP, or a campus), multitude of clients use the same resolver, and as such, this DNS-based fingerprinting method does not contribute toward distinguishing among these clients.

Internal IP address disclosure methods: The WebRTC (STUN) -based technique [28] has limited coverage (doesn't work with IE/Edge on Windows, doesn't work on mac OS Safari and on iOS), and can be turned off in Windows browsers [26]. Forcing FTP PORT command ([20]) works only with IE (and maybe Edge) browsers, so has very limited coverage.

Using the TCP timestamp to detect clock skew: [24] describes how to remotely measure an endpoint's clock skew. However, nowadays the risk of enabling TCP timestamps is well understood, and in Windows 10, this feature is disabled by default.

Sequential (per-host) IP ID: Using IP ID is proposed in [10] to detect multiple devices behind a NAT. However, nowadays macOS and iOS randomize their IP ID field, and Android (and Linux) increments the IP ID per host in a non sequential manner. Thus this technique has low coverage.

Using the Javascript Math.random() seed: This attack ([19]) was addressed by browser vendors in 2008-2010 and is no longer effective.

Ephemeral source ports in outgoing requests: This technique does not work behind a firewall/NAT, as the firewall/-NAT typically replaces the original client source port with a port from its own pool.

Accelerometer (in mobile devices): According to [11], it is possible to tag mobile devices in the browser by measuring the deviations in their accelerometer readouts. But his method does not cover non-mobile devices (i.e. desktops, laptops) at all, and as such its coverage is insufficient. Furthermore, it requires the mobile device not to move while the measurement takes place.

User action history: Except for DNS (see below), privacy mode renders this technique ineffective.

B. Tagging

In general, tagging methods are a well understood privacy threat. Therefore, one of the goals of the privacy modes that are implemented in major browsers is to make the tagging methods identify a private browsing session as a distinct instance, which has a different tag than that of the "regular" browser. Private browsing sessions also typically clear their residual data upon termination and start with an empty set of data when launched.

A summary of the privacy mode boundary-crossing status of many tagging techniques appears in [13, Table IV]. As is depicted in that table, almost all tagging techniques do not cross the private browsing boundary for most browsers. In general, nowadays tagging attempts are blocked by the relevant software vendors. For example, Flash cookies do not cross the privacy mode boundary ([5]), and anyway, Flash nowadays requires user interaction in order to run.

There are some advanced tagging techniques that are not covered in [13], and yet do not cross the privacy mode boundary: The TLS token binding protocol specifically requires browsers to separate privacy mode tokens from the regular browser tokens ([27, Section 7.3]). Firefox provides

a separation between the regular browser and privacy mode with respect to TLS session identifiers and session tickets ([2]), and likewise Chrome ([3]). HSTS data does not cross the privacy mode boundary in Chrome[17], Firefox[30] and Safari[30]. As for tagging techniques that do cross the privacy mode boundary:

DNS cache fingerprinting/tagging (timing based): A DNS-based fingerprinting method is proposed in [16], which can reveal elements of the user's browsing history. This fingerprinting method could in theory be converted to a tagging method. However the "read tag" operation is destructive as it changes the data (the tag).

HTTPS Public Key Pinning (HPKP): [32] describes a tagging technique based on HPKP. However, HPKP is being deprecated - it is only supported nowadays by Firefox.

C. The DNSCookie Method

Our technique is based on the same foundations as Daniel Dent's DNSCookie [15], but is superior to it. In the DNSCookie technique, the authoritative name server serves a single IP address (drawn at random from two IP addresses owned by the tracker). Thus, in an enterprise (or a campus, or ISP) whose resolution platform comprises a single DNS resolver, all clients (stub resolvers on different devices) will get the same tracking ID, due to caching of the single IP address in the single resolver. In contrast, our method handles well a resolution platform comprising a single DNS resolver (for BIND 9.x, Microsoft DNS Server and MaraDNS Deadwood 3.2.x DNS resolvers). This is an important distinction, as single resolvers are involved in roughly 40% of the resolution platforms, thus the DNSCookie technique covers no more than 60% of the population. (We explain in Appendix A how we came up with this estimation of the prevalence of single resolution platforms.) Our technique covers both multiple resolvers and single resolvers, and thus covers over 93% of the population (as fixed-order single resolver is <7% [21]).

The analysis in [17] mentions a technique which is de-facto identical to DNScookie (and predates it), based on the browser's DNS cache (but using the stub cache if it exists). They write that a serious limitation of their proposal is that "the value of this approach is limited by [...] the potential conflicts with resolver caching on ISP level" (which is also the major shortcoming of DNScookie). Our tracking technique addresses exactly this issue.

D. Misc. Tracking Methods Related to DNS

A DNS based "tracking method" was suggested in [29], which amounts to the server redirecting the browser to a uniquely-named subdomain. This technique resembles HTTP session cookies, rather than HTTP permanent cookies. As such, this technique does not constitute "user tracking" in the sense used in this paper.

E. DNS research and measurement

The recent analysis in [22] and [23] measures various DNS caching resolver platform attributes, such as the number of caching resolvers and their software. It does not cover DNS stub resolver caches, browser DNS caches, and DNS load balancing strategies.

III. The basic method

A. The Setting

To describe the technique, we first enumerate the parties involved (please refer to Fig. 1, where the tracker components are colored in red):

The browser – renders HTML pages and Javascript dynamic code from websites, received over HTTP/HTTPS. Invokes the stub resolver whenever DNS resolution is needed.

The operating system DNS stub resolver (with a DNS cache) is a component within the operating system that handles DNS resolution via API calls (e.g. `getaddrinfo()`). It sends recursive DNS queries over the network to the system-configured DNS resolver, receives answers from the resolver, optionally stores the answer in its cache, and finally returns the answer to the API call.

DNS resolution platform – this is a server (or servers) typically operated by a corporate, university or ISP. It handles DNS queries received over the network from clients, performing the actual DNS resolution by querying authoritative DNS servers down the DNS hierarchy until reaching an answer, which is optionally cached, and finally returned to the client.

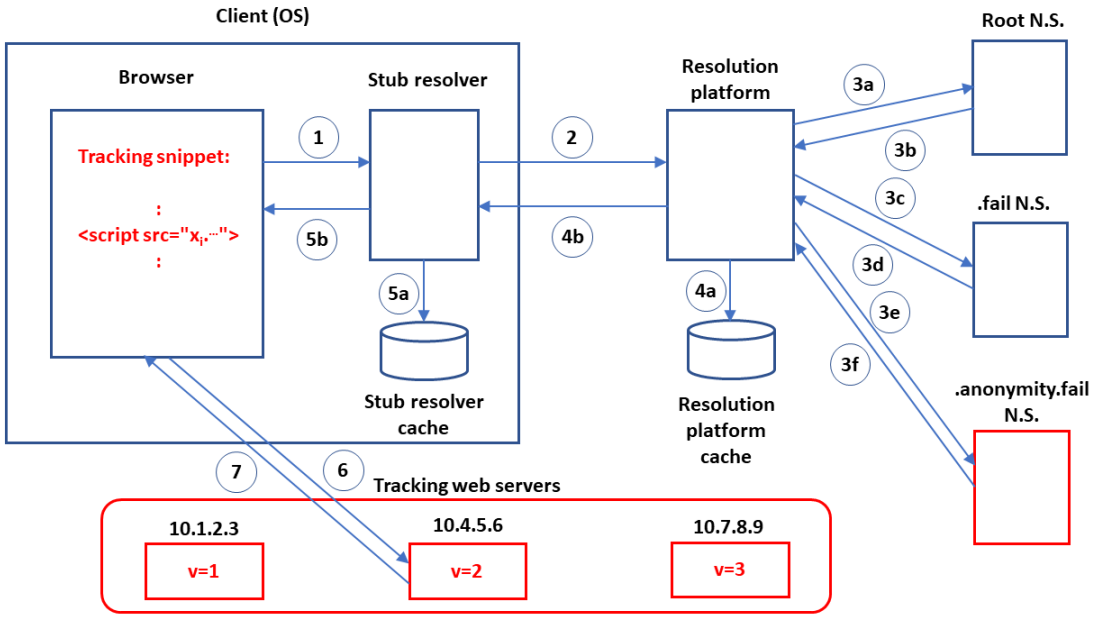
Authoritative DNS server (tracker owned/controlled) – in general this server answers queries about a set of DNS domains that it is responsible for. Specifically, in our setting the tracker has an authoritative DNS server responsible for the `anonymity.fail` domain (and all its subdomains), which returns a randomly ordered RRset[1] comprising the set of IP addresses e.g. {10.1.2.3, 10.4.5.6, 10.7.8.9} in answer to queries for $\texttt{x}_i\texttt{.anonymity.fail}$.

Note that the use of a single domain (`anonymity.fail`) throughout the paper is for simplicity and clarity. The technique can be implemented with multiple domains. Likewise, the repetition of IP addresses in each subdomain is for clarity's sake, and different IP address sets per different subdomains are possible. Also note that the authoritative name server is "dumb" - it does not collect/store any information (hence it is not a scalability bottleneck), and it does not care about the IP origin of the queries it serves.

Web (HTTP/HTTPS) servers (tracker owned/controlled) – in general such web servers listen on an IP address for incoming HTTP/HTTPS connections (on TCP port 80/443, respectively), and return data over HTTP, for example an HTML page or Javascript code. Specifically the tracker owns several IP addresses (e.g. 10.1.2.3, 10.4.5.6, 10.7.8.9), with a web server listening on each IP address, with each web server (bound to a different IP address) set to return a different value. Thus even when the servers are accessed by the same hostname, the answers from each web server will be different.

[1] the term "set" in "RRset" is a misnomer – the order of individual records is significant.

Figure 1: The basic method.



Note that for simplicity we assume a host per IP address, but it is possible to listen on several IP addresses on a single host.

The tracking snippet (tracker controlled) – an HTML+Javascript snippet that sets/gets the tracking ID. This snippet can be embedded in a bigger HTML page, and can be served from any URL (i.e. any domain) and from any IP address. The snippet's main functionality is running Javascript code loaded from the tracker's `x`$_i$`.anonymity.fail` hosts. The Javascript code provides a different answer per different IP address that is accessed. The snippet collects these answers from the multiple host names (for i from 1 to N), assembling these values to a single tracking ID number. The snippet can report the ID to other part of the HTML page in which it is embedded, or to communicate it to any HTTP/HTTPS server via Javascript code.

Note that the tracker only controls the authoritative DNS server, the web servers, and the tracking snippet. The browser, the operating system DNS stub resolver and the DNS resolution platform are out of the tracker's control.

B. The Basic Method

The basic operation combines the ID setting and ID querying into a single flow (referring to Fig. 1):

(a) The browser renders the tracking snippet.

(b) In Step 1, the browser requests resolution from the operating system's DNS stub resolver for `x`$_i$`.anonymity.fail`.

(c) The operating system DNS stub resolver resolves the hostnames into IP addresses, typically by (Step 2) forwarding the query to the resolution platform configured for the operating system.
The resolution platform (or an upstream resolver it uses) resolves the hostname (Step 3) concluding with the reception of an answer (Step 3f) from the tracker's authoritative name server. The answer is a randomly ordered RRset of A records containing tracker-owned IP addresses.
The resolution platform then caches (Step 4a) the result and answers the stub resolver (Step 4b) with a randomly ordered RRset (it shuffles the RRset received in Step 3f).

(d) The DNS stub resolver caches this data (RRset per query) in Step 5a, and returns it as-is in Step 5b to the browser (one RRset per query).

(e) The browser sends HTTP requests (Step 6) to the tracker's web servers designated by the IP addresses returned by the operating system (typically the browser sends the HTTP request to the first IP address in the RRset).

(f) The servers respond differently (Step 7) depending on the IP address they listen on (for example, serving Javascript code that sets a Javascript variable to a different value for different IP address). So for example, if the set of IP addresses used in the RRsets are 10.1.2.3, 10.4.5.6 and 10.7.8.9, then the web server listening at 10.1.2.3 may respond with the Javascript code `v[0]=1`, the web server listening at 10.4.5.6 may respond with `v[0]=2`, and the web server listening at 10.7.8.9 may respond with `v[0]=3`.

(g) The Javascript code running in the browser collects the data values returned from the servers and assembles them into an ID. In our example, this amounts to enumerating over the N array cells, and encoding them into an ID using base 3 encoding.

Suppose that there are K IP addresses for each web server used by the tracker (in our example $K = 3$). Assuming a

uniformly random order for the RRset, the values for the IP addresses of the servers are uniformly distributed. When using N host names there are K^N possible ID values, uniformly distributed. So even with a minimal K ($K = 2$), a large enough N enables the tracker to assign statistically unique IDs on an Internet-wide basis. (The choice of values for K and N is discussed in Sec. IV-H.) Note the importance of Step 5a – this is the step that provides persistence of the tracking ID. Without a *caching* stub resolver, there is no tracking ID persistence. As it happens, all common platforms (Windows, macOS, Android, iOS) do have a caching stub resolver.

Our technique relies on obtaining RRset in a random order, to ensure statistical uniqueness among clients of the same resolver platform. We show in Section IV that we can typically ensure that the stub resolver receives the RRset in random order. This happens in two possible (not necessarily mutually exclusive) scenarios: either (a) the queried resolver server (in the resolution platform) returns the RRset in random order; or (b) the resolution platform is load-balanced randomly per-query (see Sec. IV-C). It is also possible that the resolution platform comprises a single server which provides the RRset in a round-robin order. In that case it is still possible to randomize the order of the RRset using an auxiliary technique (see Sec. IV-G), so that the first IP address is uniformly random and the tracking method is still effective.

C. Benefits of the Technique

The tracking technique that we described demonstrates multiple advantages over existing tracking methods (the details are explained in Section IV):

- The tracking technique is orthogonal to other tracking techniques, and thus can be combined with them to attain synthesis in which overall coverage is increased as each technique bridges over the gaps of the other technique (à-la evercookie [18]).
- The tracking code can be embedded in HTML pages of arbitrary websites, and is supported by any **vanilla browser** (since it only relies on standard HTML and Javascript).
- The tracking method survives **browser restart** (except on Google Chrome on macOS), and also survives browser **"history cleanup"** procedures (except on Google Chrome on macOS). It also does not rely on "traditional" storage (and is thus less susceptible to vendor/user interference/restrictions).
- Tracking does not rely on having the tracking website being **continuously rendered**.
- Tracking works against **browser privacy mode** [7].
- The tracking method can uniquely identify machines even if they have the **same hardware and software** image.
- Tracking works **across browsers** (with some exceptions).
- Working through a **VPN** does not affect tracking, because the stub resolver of the operating system is the entity that emits and caches DNS queries (although it uses the DNS resolver configured for the VPN network).
- Using a **SOCKS4 proxy** does not affect the technique, since the client still performs the DNS queries. Ditto for SOCKS4a and SOCKS5 if the client elects to perform DNS queries (this was the case with some browsers).
- The technique can be used with **IPv6**, and/or **DNSSEC**.
- The technique can be used with **HTTPS**.
- When used with HTTPS, the technique can overcome **HTTP transparent proxies**, since these typically only intercept HTTP requests and resolve the hostname in their HTTP `Host` header [9].
- The technique can be used in a **Javascript-less** browser (except for Algorithm 1).[2]

IV. Technical Details

We encountered several technical challenges applying the technique to various browser×OS×resolver combinations. Below is a discussion of the various technical issues, and how we addressed them. We managed to develop our basic method into a comprehensive technique that covers the vast majority of browser×OS×resolver combinations in most cases.

The technical issues we discuss include the various conditions wherein cached DNS records can be evicted/flushed (this is not a common condition); we discuss how DNS TTL affects our technique (it may limit the life span of the ID to anywhere between weeks to hours); we describe the various load balancing techniques we uncovered (most of them provide sufficient randomness for our technique); we analyze common resolver chaining scenarios, and determine that the vast majority of the combinations support our technique; we explain why Tor and shared forward HTTP proxies are incompatible with our technique; we discuss the types of stub resolvers we encountered (this affects the RRset order randomization algorithm); we explain the limits of UDP packet size and how it affects the prime-size RRset technique; we elaborate on RRset ordering policies and how they affect our technique (with some additional improvements, it covers most popular resolver software); and finally discuss the subjects of future proofing and sensitivity to changes in the DNS landscape.

The conclusion is that our tracking technique works for the vast majority of users, and is in effect until the user machine is restarted or switches a network connection, or until our DNS records reach the TTL limit. The tracking method is therefore robust for at least a few hours. Consequently, it works best when combined with other known tracking techniques which might be more persistent but are not able to identify multiple browsers or private browsing sessions of the same user, or identify different users who use different machines with the same image. It should be noted that our analysis covers the most common DNS scenarios, which we believe are relevant for the vast majority of Internet users. Analysis of complex

[2]This can be done as follows: the snippet needs to be dynamically generated at the server side, where for each invocation the server generates a unique provisional session token *token*, and returns a snippet comprising a list of `IMG` tags pointing at URLs (e.g., x_i`.anonymity.fail/?t=`*token*). The tracking web servers then collect hits and associate them with the client via the provisional session token in the URL. The device ID is calculated at the server side according to which web server (IP address) was hit per URL.

and uncommon DNS scenarios requires extensive additional research assets. It should also be noted that most applications of tracking, for example for targeted advertising or as an anti-fraud measure, do not depend on perfect tracking of all users all of the time. Namely, utility from tracking is not an all-or-nothing game. Rather, there is some utility from each user who is being tracked, and the overall utility is linear in the fraction of users who are successfully tracked.

A. DNS cache flush/eviction

Clearly, flushing the DNS cache of the stub resolver deletes the DNS tracking ID (with subsequent rendering of the tracking snippet generating a new tracking ID for the device). Such flushing occurs in either operating system restart, network switching (e.g. moving from one WiFi network to another), or manual cache flush (e.g. in Windows, via the command line `ipconfig /dnsflush`). Additionally, the tracker DNS entries may get evicted from the cache if the cache volume is limited, to make room for more used records. However, under normal browsing conditions cache eviction of our tracking DNS records should be rare as typically most other DNS records consumed by the browser have TTL of several minutes (only), and thus the resolver cache is unlikely to become full.

Overall, eviction/flushing of DNS records is an unlikely scenario as long as the user is connected to the same network and keeps the machine running. This holds even if the user generates heavy and diverse network activity. To verify this claim, we conducted an experiment wherein 13 individuals with varied hardware, software and networks (ISPs), were asked to navigate on their machines to a page containing a tracking snippet, then continue working on their machine for 3-5 hours (with normal to heavy workload), and finally browse the snippet page again. We chose the test duration to be 3-5 hours to demonstrate several hours longevity while avoiding potential interference from the TTL cap imposed by Google Public DNS. The experiment tested whether the snippet generated the same tracking ID, demonstrating that the ID is long lasting and its DNS records are not evicted from the stub resolver cache. In all 13 test cases, the DNS ID generated by the tracking snippet survived the test duration.

B. Time To Live (TTL)

The Time To Live (TTL) value of the DNS records cached by the operating system determines the life expectancy of the DNS tracking ID, up to DNS flushing (see above). The impact of time expiration (due to TTL) of the tracking records is identical to DNS cache flushing (see above). Thus the tracker's authoritative DNS servers should emit maximal TTL records, so that DNS tracking IDs would have maximal life expectancy.

Typically, operating system stub resolvers cap the TTL of incoming DNS records, so that even if the original record has its TTL field set to the maximum possible value (2^{31}-1 seconds), the stub resolver cap will reduce it. We detail the TTL cap values of various operating system stub resolvers in Table III. Moreover, resolution platforms on the path may impose their own cap, further reducing the that TTL the stub resolver receives. For TTL cap values of various resolvers, see Table I. Even worse – resolution platforms may have the resource cached from a previous query, and when queried again at a later time, the TTL they provide with the RRset will be less than their TTL cap (or the record's original TTL – the smaller of the two) since some time has already passed.[3] The expected order of magnitude for cached DNS records at the stub resolver is therefore somewhere between a week (BIND resolver with macOS/iOS stub resolver) and several hours (Google Public DNS), which can suffice for many tracking goals (especially if combined with other tracking schemes that have less coverage but more persistence).

C. DNS load balancing

Oftentimes, the resolution platform comprises multiple resolvers. Even if the stub resolver uses a single IP address as its DNS resolver server, this address may merely be an entry IP address of a load-balancer, which actually "hides" multiple DNS resolvers. Also, at least one DNS resolver software - PowerDNS - fires up two resolution processes acting de-facto as two separate DNS resolvers behind a single IP address.

There are many ways to implement load balancing. We refer to the balancing strategy as *random load balancing* when the clients are referred to each resolver with uniform probability (i.e., given N resolvers, each query has a probability of $\frac{1}{N}$ to access a specific resolver), regardless of the query and of the client identity.

The DNS tracking technique takes advantage of random load balancing by offering a differently ordered RRset every time the authoritative server is queried. Thus, if there are (say) two resolvers in the resolution platform, it is likely that each of them will get a differently ordered RRset. In that case, the technique works in an ideal manner. Each client will be assigned a random resolver *for each query*, and thus will get a random tracking ID (statistically unique, given enough queries).

We tested several popular DNS resolution platforms (we reviewed the PowerDNS source code as well as tested it in the lab, and tested Google Public DNS Service and OpenDNS in various client configurations). We have observed the following load balancing strategies of popular DNS servers, which are based on specific combinations of query data fields (namely, the combination determines which DNS resolver the query is routed to):

- (*source IP*, *source port*) (used by OpenDNS). This is random load balancing as the *source port* value (which changes between queries) is enough to randomize the selection of resolver.

[3] For example, if a resolution platform cached a record at $t = 1AM$, with cap of (say) 24 hours, then when queried again for the same resource at $t = 11PM$ on the same day (22 hours after the record was retrieved and cached), it will report a TTL of only 2 hours (much less than its TTL cap of 24 hours).

- (*qname*) (used by PowerDNS Recursor 3.6 and above). This is *not* random load balancing, as each query for the same *qname* will be routed to the same resolver.
- (*source IP*, *source port*, *qname*) (used by Google Public DNS Service). This is random load balancing as the *source port* value (which changes between queries) is enough to randomize the selection of resolver.

We also observed process-based load balancing (provided natively by the operating system) in PowerDNS Recursor 3.0-3.5.x, which is quite likely a random load balancing.

Other than sticky by qname (which is only employed by PowerDNS 3.6 and above), the load balancing techniques we encountered are random and as such support our technique.

D. DNS resolver chaining

In reality, there may be "chains" of DNS resolvers. A typical such chain is an internal DNS resolver (actually, "forwarder"), which forwards all queries for non-enterprise names to an external resolution platform (e.g. the ISP resolver, or Google public DNS service), and caches the answers. For simplicity, we focus on chains of length 2 (intuitively these are the majority of the chains) - a *forwarder* and an *upstream resolver*. In such case, it can be easily shown that the technique will fail only if there is a single Unbound 1.6.x or PowerDNS 3.6+ forwarder, or multiple (load balanced) Unbound 1.6.x or PowerDNS 3.6 forwarders with the upstream resolver being a *single* Unbound 1.6.x or PowerDNS 3.6+. Clearly these are uncommon combinations.

E. HTTP forward proxy and Tor

The tracking technique does not work when the browser is configured to use a shared HTTP forward proxy, because DNS resolution in such case is performed on the HTTP forward proxy device, and as such the same ID is assigned to all the proxy clients. This is also the case with Tor.

F. Stub resolver cache types

We noticed two types of stub resolver DNS caches:

- A *record-based* cache – this cache behaves much like a resolver (server) cache. It caches the individual records received from the resolver, keyed by their resource name.
- A *query-based* cache – this cache stores the resolution data keyed by the original query.

The major difference between the cache types is their treatment of CNAME records. Consider a DNS query for `foo.example.com` sent by the client, which results in the following answer:

```
foo.example.com CNAME bar.example.com
bar.example.com A 10.2.3.4
```

A record-based cache will cache the two records separately, each under its own resource name. Thus if a subsequent query for `bar.example.com` is be needed, it will be answered from the local cache. A query-based cache, on the other hand, will cache the data under the original query (`foo.example.com`), either in its entirety or only the final result (i.e. `foo.example.com` $\rightarrow$ `10.2.3.4`). A subsequent query for `bar.example.com` will need to be resolved externally since `bar.example.com` does not exist in the global scope of cached query answers.

A query-based cache natively support Algorithm 1 of Section IV-G. A record-based cache needs to be explicitly tested to determine whether it supports the said algorithm.

G. RRset order

In case there is a single cache (resolver) in the resolution platform, load balancing is meaningless, and the tracking technique cannot rely on having multiple resolvers returning different results for the same query. However, DNS tracking can still be achieved, based on the RRset reordering behavior exhibited by some popular resolvers. When multiple records (an RRset) are returned by the authoritative server (or the upstream resolver), a resolver can employ several strategies to set the internal order of RRset records returned to clients:

- *Fixed* order - the records are returned in the same order for every subsequent query. Note that there may be an initial re-ordering (e.g. in PowerDNS), or the original order may be reserved (Unbound 1.6.x, Google DNS).
- *Round-robin* order – the records are returned in a cyclic round-robin fashion (Microsoft DNS Server, MaraDNS Deadwood 3.2.x).
- *Random* order – the records are returned in a random order per query (BIND 9.x).
- *Time-based round-robin* order – the records are returned in a round-robin fashion, but the order is not changed with every access (as in regular round-robin), but rather every second. This strategy is employed by OpenDNS.

The case of random order is easiest for tracking. The authoritative server merely needs to return (per query) an RRset which includes *multiple* addresses. The resolver will provide a randomly ordered RRset to every client query. Therefore, even though the answers of the authoritative server are cached, each subsequent client which sends a set of queries effectively receives a statistically unique ID.

The case of round-robin order is more difficult, as using RRsets with K values yields only K possible IDs (as all RRsets are advanced together, and thus have a combined cycle length equal to K). One solution is to have the the lengths of RRsets that are returned as query answers be different prime numbers, to ensure that the combined cycle length is maximal. (For example, with RRsets of size 2,3,5,7,11,13,17,19 we get 9699690 possible IDs, and adding 23 and 29 yields over 6 billion different IDs.) If the host name queried is no longer than 31 bytes (including dots), then a single DNS answer can accommodate 29 A records for it. All resolver platforms tested and all stub resolvers (namely, operating systems) tested support such answers (we tested with 28 records due to technical constraints).

An alternative solution, which requires less IP addresses but is less generic, is to forcibly randomize the current position in the RRset that the resolver keeps, after each query. Such technique exists for Microsoft DNS Server, but

not for MaraDNS Deadwood 3.2.x. The idea is to force the resolver to conduct a random number of round-robin steps, thus making the first record (IP address) returned in the RRset of the next query random, which is enough for the tracking technique to be effective. So instead of simply querying for `x`$_i$`.anonymity.fail`, the tracking snippet would run the code of Algorithm 1.

```
// the query() function forces a DNS query
// (e.g. by inserting a SCRIPT tag
// to the DOM).
// The unique_random() function generates
// a DNS label which is statistically
// globally unique - e.g. using
// crypto.getRandomValues()

query("x_i.anonymity.fail");
// n is a random integer 0...(K-1)
var n=Math.floor(Math.random()*K);
for (var j=0;j<n;j++)
{
  query("bar_i."+unique_random()+
          ".anonymity.fail");
}
```

Algorithm 1: Randomize a round-robin RRset

The tracker's authoritative DNS server should answer queries starting with "`bar`$_i$`.`" with `CNAME` to `x`$_i$`.anonymity.fail`. The resolver then forwards the original CNAME record obtained from the tracker's authoritative DNS server, but also adds the RRset for `x`$_i$`.anonymity.fail` from its cache, advancing its order in round-robin fashion. Therefore, there are `n` (randomly chosen) round-robin steps of rotating RRset after the actual sampling (of `x`$_i$`.anonymity. fail`) takes place. Which means the first IP address in the RRset obtained by the *next* stub resolver for `x`$_i$`.anonymity.fail` will be random.

The case of a (single resolver) fixed RRset order is not addressed by our techniques. However, this is only problematic if the resolver does not use load balancing, since unique ID randomization is essentially implemented by load balancing (as described in Section IV-C). Of the resolvers which use a fixed RRset order, Google Public DNS uses random load balancing, and only Unbound and PowerDNS (which have a very small market share) do not use random load balancing. (See also Section V-A for the experimental evaluation of different resolvers.)

Finally, the case of time-based round robin is easily solved (though this issue is of theoretic interest only – since this case is only found in OpenDNS, and the OpenDNS resolution platform comprises multiple resolvers so it is covered by load-balanced randomization) by the client randomly choosing some queries to run at time t (now), and the remaining queries to run at time $t+1$. In this manner, it is easy to see that there are 2^N possible different results for N queries, thus enough to make the ID statistically unique given large enough N.

To sum up, With the support for round-robin RRsets, our technique covers a very large percentage of the resolution platform landscape (with only single-resolver Unbound and PowerDNS 3.6 and up as exceptions), see Table II.

NOTE: Stub resolvers (at least the ones we surveyed: in Windows, iOS and Android, and macOS in most of the time) always return the RRset in the original order it was received. Browsers (at least the ones we surveyed) always use the first IP address they receive from the stub resolver (as long as it is responsive). This is crucial for the effectiveness of the tracking technique – if the stub resolver would have returned a differently ordered RRset each time it was queried, or if the browser would have used different IP addresses from the RRset each time it needed a resolution, the tracking ID would have been unstable.

H. Multiple tracking systems

A single tracking system consumes multiple entries in the stub resolver cache - at least N DNS names and $N \cdot K$ A records, possibly more (see Sec. IV-G). Theoretically, if multiple such systems are employed simultaneously, the stub resolver's cache may get saturated, and start evicting DNS records that belong to those tracking systems, thus rendering them less reliable. However, the economy of Internet user tracking prefers centralized tracking services, and thus we expect that there will be few (and likley only one) such tracking system in wide deployment.

I. DNS ID uniqueness

A careful choice of K (the number of IP addresses per host) and N (the number of DNS hosts/queries) can reduce the likelihood of DNS ID collisions. The expected number of collisions for M IDs is $\binom{M}{2}/K^N$, therefore increasing K and/or N can keep this quantity arbitrarily small. Increasing K requires more IP addresses, while increasing N increases traffic and error potential (dropped packets). Throughout this paper, we use small K to simplify the discussion and due to logistic constraints (shortage of IP addresses).

J. Future proofing

The technique relies on several properties of many present-day resolvers, stub resolvers and browsers, namely:

- Resolvers: re-ordering the RRsets.
- Resolver platforms: non-sticky (by client) load balancing.
- Stub resolvers: maintaining a fixed-order RRsets.
- Browsers: using the first IP address in an RRset.

Our research into specific platforms and software provides a snapshot of their current behavior. Future versions can violate the requirements listed above, and thus reduce the coverage of the technique. This can happen overnight (e.g. in public DNS services), in a short period of time (e.g. with regards to stub resolvers and browsers, which get automatic, frequent updates), or over years (with regards to resolver software, especially when bundled with a server operating system distro). For example, during our research we noticed that Google Public DNS Service reduced its TTL cap from 86400 seconds (in June 2017) to 21600 seconds (as of October 2017).

V. Platforms and Experiments

Our lab setup is described in detail in Appendix B.

A. Resolution platforms

We conducted extensive studies of the behavior of popular resolution platforms, in order to evaluate the following issues which are relevant to effectiveness of tracking:

How many caches are there in the resolution platform? As we will see below, the tracking technique generally prefers multiple caches, and may have problems with some single cache resolution platforms. We used the technique described in [23] to detect the number of caches in the resolution platforms that we studied. We also verified this data with the software configuration and documentation whenever this was possible.

What is the TTL cap imposed by the resolution platform? The TTL cap directly affects the longevity of the tracking ID as the stub resolver respects the TTL it receives from the DNS resolution platform (and may apply its own cap on top of it). Obtaining the TTL cap was a simple matter of serving a record with maximal TTL from an authoritative DNS server through the platform, and observing the TTL field returned by the resolution platform.

What is the order of the records in the RRset returned? This is important for single cache platforms. The more random the order is, the better. For this test, we created an RRset in an authoritative name server, comprising 10 A records for different IP addresses. We then observed the order in which this RRset was returned.

If the RRset order is round-robin, can randomization be forced? If we can randomize the order to some degree, we can still use the tracking technique, even with a single cache in round-robin RRset order. For this test we ran Algorithm 1.

When multiple caches are present, how random is the load balancer? If the load balancing is random "enough", then the tracking technique is guaranteed to be effective. The test applied here calculates the likelihood of two consecutive queries to arrive at the same resolver (cache). This is done by repeatedly querying a random resource via a client, while the authoritative name server answers this query (from the resolution platform) with a CNAME to a fixed resource name, which in turn resolves into a unique RRset each time it is queried from the authoritative server. Eventually all resolvers in the resolution platform get the fixed resource cached, but each resolver gets a different (unique) RRset. By observing the RRsets received by the client we can tell whether two consecutive answers are from the same resolver or not.

If load balancing is used, is it sticky per query? Sticky (per query) load balancing among multiple caches effectively reduces the situation into (the more problematic) single cache scenario. We test this by sending repeated queries to the resolution platform for a fixed resource name. The first time this query arrives at the authoritative name server, the answer is a special RRset, and each subsequent query is answered with another (possibly random) RRset. At the client, we count the number of special RRsets received from the resolver. The quotient of the number of special RRset answers to the number of queries is an indication of how sticky the resolution platform is.

The raw results of these tests are summarized in Table I. Note that all BIND versions (9.9.0 to 9.11.2-P1) behaved identically in our tests, all PowerDNS 4.x installations behaved identically in our tests, all Unbound 1.6.x behaved identically, all MaraDNS Deadwood versions (3.2.0-3.2.11) behaved identically, and all Microsoft DNS Servers (6.1-10.0) behaved identically. Therefore we designate them BIND 9.x, PowerDNS 4.x Unbound 1.6.x, MaraDNS Deadwood 3.2.x and Microsoft DNS, respectively.

Note the following properties in the table:

- While MaraDNS Deadwood 3.2.x technically reorders RRset in a round-robin fashion, Algorithm 1 does not work for that system. The reason is that querying for $\texttt{bar}_i\texttt{.anonymity.fail}$ does not reorder the RRset for $\texttt{x}_i\texttt{.anonymity.fail}$ when $\texttt{x}_i\texttt{.anonymity.fail}$ is queried directly (which is the ultimate goal of the algorithm).
- PowerDNS Recursor 4.x is sticky per query, which means that practically, the tracking technique does not benefit from having two processes and caches in each installation (by default). That is, a single PowerDNS Recursor (actually starting from version 3.6) will behave from all practical aspects of the technique as a single cache.
- PowerDNS Recursor 3.5.3 (actually all 3.x versions until 3.6) is not sticky by query, but rather has a load balancer in front of two processes, with the load balancer based on process load (i.e. practically random, or at worst case flipping between the processes).
- In Google Public DNS and OpenDNS public DNS, the load balancing appears to be random (the probability of identical pairs and the probability of accessing the same server twice for the same query are both around $1/L$ where L is the number of caches, as expected from a random load balancer).

Summary: The applicability of the DNS tracking technique is derived from the raw results, and is summarized in Table II. In short, the DNS tracking technique is applicable to all resolvers that we tested, except for single-server Unbound 1.6.x and single-server PowerDNS 3.6+. It is important to note that in enterprise use, Unbound and PowerDNS represent a very small portion of the resolution platform software [21].

B. Stub resolvers

The study of stub resolvers should answer the following questions:

Cache type (query-based or record-based) – a query-based cache supports Algorithm 1. A record-based cache may or may not (an explicit test is needed). This is easily tested by querying a resolver for `foo.example.com` and getting a resolver response such as: `foo.example.com CNAME bar.example.com; bar.example.com A 10.2.3.4`.

Querying now for `bar.example.com` in a query-based cache yields another query from the stub resolver for `bar.example.com`, whereas in a record-based cache, the

Table I: Resolver behavior

	BIND 9.x	Unbound 1.6.x	PowerDNS Recursor 4.x	PowerDNS 3.3.2 Recursor 3.5.3	MaraDNS Deadwood 3.2.x	Microsoft DNS server	Google Public DNS	OpenDNS public DNS
Cache count	1	1	2	2	1	1	6[4]	14[4]
TTL cap [sec.]	604800	none	86400	86400	86400	86400	21600	604800
RRset order (*single* server)	random	fixed	fixed	fixed	round-robin	round-robin	fixed	cyclic (time based)
RRset order rand.	N/A	N/A	N/A	N/A	No	Yes	N/A	N/A
Identical pairs [prob.]	N/A	N/A	0.487	0.026	N/A	N/A	0.178	0.080
Stickyness [prob.]	N/A	N/A	1	0.45	N/A	N/A	0.1525	0.075

[4] The actual number of resolvers may vary according to the region, the ISP and the time of day.

Table II: Applicability of the DNS tracking techniques to resolution platforms

Resolver Platform	Applicable?	TTL Cap		Resolver Platform	Applicable?	TTL Cap
BIND 9.x, single server	Yes	604800		PowerDNS <3.6, multiple servers	Yes	86400
BIND 9.x, multiple servers	Yes	604800		MaraDNS Deadwood 3.2.x, single server	Yes[5]	86400
Unbound 1.6.x, single server	No	N/A		MaraDNS Deadwood 3.2.x, multiple servers	Yes	86400
Unbound 1.6.x, multiple servers	Yes	none		Microsoft DNS, single server	Yes[6]	86400
PowerDNS 3.6+, single server	No	N/A		Microsoft DNS, multiple servers	Yes	86400
PowerDNS 3.6+, multiple servers	Yes	86400		Google Public DNS service	Yes	21600
PowerDNS <3.6, single server	Yes	86400		OpenDNS Public DNS	Yes	604800

[5] Requires the tracker to control 20-40 different IP addresses.
[6] Requires the tracker to control 20-40 different IP addresses; alternatively use Algorithm 1.

answer will be served from the cache (without an additional query).

Cache size limit – in order for the technique to work, the cache should be able to contain at least several dozen entries, ideally much more.

TTL cap imposed – this limit directly affects the longevity of the tracking ID. Obtaining the TTL cap is not straight-forward as the stub resolver does not return it in `getaddrinfo()`. In some cases it may be possible to view the cache (e.g. in Windows and in macOS) and observe the TTL value directly. In other cases (iOS, Android), the value needs to be discovered by looking at the source code.

When does cache flushing occur? – particularly, does a network change flushes the cache? testing this is as simple as switching networks and observing the cache.

Is the order of IP addresses returned from `getaddrinfo()` preserved? – obviously if it is not preserved, then the technique fails completely for the stub resolver. Testing this is a simple matter of invoking `getaddrinfo()` multiple times for a resource whose RRset comprises multiple A records, and observing the order in which they are returned by `getaddrinfo()`.

A summary of the results can be found in Table III. As can be seen, all stub resolvers support the DNS tracking technique.

C. Browsers

The study of browsers should answer these questions:

Does the browser have an internal cache, and what is its expiration policy? An internal cache with generous expiration policy can extend the life of the tracking ID, and can also bridge the network-change gap. Obtaining this data can be via source code inspection (if the browser source code is available), documentation, or empiric experimentation.

Does the browser go through the operating system's stub resolver cache? Surprisingly, some browsers have a built-in stub resolver (in which case their internal cache is the "main" cache for the browser). Testing this can be done by introducing a record to the operating system's stub resolver cache (e.g. by pinging this host), and observing whether browsing to this host generates a DNS query.

Does the technique work across browser windows? browser tabs? privacy mode? Testing this is straightforward.

Does the technique survive clearing the browsing history? For browsers that use the stub resolver cache, this should be answered in the affirmative. Browsers which implement their own stub resolver may very well yield a negative result. Testing this is straight forward.

We ran these tests over the major browsers, and report the results in the following sub-sections.

D. Windows stub resolver and browsers

The Windows stub resolver (and cache) is implemented as the "DNS Client" service. We tested Windows 10, but the results should be applicable to Windows 8.x and Windows 7, and perhaps even for earlier versions. The Windows stub resolver is oblivious to network issues at the carrier protocols (e.g. TCP). It provides the same answer to `getaddrinfo()` regardless of whether the IPs in it are responsive or not. The Windows DNS cache exhibits a most peculiar property: there are actually two side-by-side DNS caches. One cache (which we will designate as W_0) is used for DNS resolution queries made with `getaddrinfo()` parameter "address

Table III: Stub resolvers

	Stub resolver component/SW	Cache type	Cache size limit	TTL cap [seconds]	Cache flush trigger	RRset order
Windows 10 ver. 1709 (build 16299.192)	DNS Client Service	query -based	4096 answers (est.)	86400	network change, OS restart	preserved
macOS High Sierra version 10.13.2	Bonjour mDNSResponder	record -based	(none)	1610612[7]	network change, OS restart	preserved
Google Chrome 63.0.3239.108 on macOS	Google Chrome	query -based	1000 answers	(none)	browser restart, clearing browsing history, network change(!), OS restart	preserved
iOS 11.2.5	Bonjour mDNSResponder	record -based	1MB (4310 "entries" [8])	1610612	network change, OS restart	preserved for RRset size <8, otherwise reversed
Android 8.1.0	Bionic libc	query -based	640 answers	(none)	network change, OS restart	preserved

[7] Equals to $60000000_{(16)}$ milliseconds.
[8] In an RRset, each record value consumes one "entry" (CacheEntry object - 232 bytes in 64bit distribution).

Table IV: Browser properties

	OS	Cache	Internal TTL (cumulative)	Works across privacy boundary	Works across browser restart	Works across history deletion
Microsoft Edge 41.16299.15.0 (EdgeHTML 16.16299)	Win 10	W_0	1800	Yes	Yes	Yes
Microsoft IE 11.192.16299.0	Win 10	W_0	1800	Yes	Yes	Yes
Google Chrome 64.0.3282.119	Win 10	W_2	60	Yes	Yes	Yes
Mozilla Firefox Quantum 58.0.1	Win 10	W_0	TTL from stub resolver	Yes	Yes	Yes
Opera 50.0.2762.67	Win 10	W_2	60	Yes	Yes	Yes
Safari 11.0.2 (13604.4.71.3)	macOS	N/A	0	Yes	Yes	Yes
Chrome 63.0.3239.108	macOS	N/A	N/A	Yes	No	No
Mozilla Firefox 57.0.1	macOS	N/A	60+60(grace)	Yes	Yes	Yes
Mobile Safari version 11.0	iOS	N/A	0	Yes	Yes	Yes
Chrome version 64.0.3282.112	iOS	N/A	0	Yes	Yes	Yes
Mozilla Firefox version 10.5b8741	iOS	N/A	0	Yes	Yes	Yes
Samsung Internet 6.2.01.12	Android	N/A	60	Yes	Yes	Yes
Google Chrome 64.0.3282.137	Android	N/A	60	Yes	Yes	Yes
Mozilla Firefox 58.0.1	Android	N/A	60+60(grace)	Yes	Yes	Yes

family" set to `AF_UNSPEC`, meaning that the address family is unspecified and it accepts both IPv4 and IPv6 addresses. Another cache (which we will designate as W_2) is used for DNS resolution queries that set the "address family" parameter of `getaddrinfo()` to `AF_INET`, meaning that only IPv4 addresses are required.

Browsers and other network tools may be using one of those caches or both of them, depending on how they invoke `getaddrinfo()`, or other DNS APIs. Browser behavior for Windows is summarized in the first part of Table IV. The operation of the Firefox browser is rather unique. We therefore highlight some observations about its behavior:

- Firefox uses W_0 cache for its DNS queries. However, Firefox retrieves the TTL for the DNS records (which is not available via `getaddrinfo()`) using Windows' `DNSQuery_A()` API which goes through the W_2 cache. Firefox emits two `DNSQuery_A()` calls: the first for A records, and the second for AAAA records. The TTL is the minimum of the TTLs in the A and AAAA records received via the `DNSQuery_A()` calls.
- Firefox has its own internal cache (on top of the stub resolver cache), in which it caches DNS records received from the stub resolver, with the TTL it retrieves per the above. This cache is shared among all tabs and windows.
- Firefox also keeps expired RRsets in its cache. When a new RRset is retrieved for a hostname, if an expired RRset for the same hostname exists in the Firefox cache with the exact same records as the new RRset, the old RRset will be retained (with its TTL updated from the new RRset). Thus Firefox may retain the *order* of the records within the RRset after it is expired and re-queried. This can extend the life span of the tracking technique beyond the nominal TTL.

Conclusion: The net result is that all four browsers tested on Windows support the tracking technique, with maximum TTL imposed by the Windows stub resolver (one day). Tracking is also effective across different browsers: Google Chrome and Opera comprise one set of cross-browser ID compatible browsers, and Microsoft Internet Explorer, Microsoft Edge and Mozilla Firefox comprise another such set.

E. macOS stub resolver and browsers

The macOS stub resolver is based on Apple's Bonjour mDNSResponder. The cache is record-based. The results for macOS browsers are summarized in Table IV.

Overcoming the stub resolver's IP order preference: The macOS stub resolver and `getaddrinfo()` apparently

access some per-host network connectivity/RTT, so while the stub resolver cache maintains the original RRset order, the order may change in the result of `getaddrinfo()`, such that less responsive (or higher RTT) IP addresses are pushed to the end of the list. Ultimately this may lead in some cases to situations where the first IP address (among the K cached) is identical across the N names. This is detrimental for the tracking technique since all devices that behave this way end up being assigned a tracking ID out of only K possible values. A revised technique can address this issue as following: choose K so that the entropy of random order over K, $\log_2(K!)$ is sufficient for a DNS ID, then have $N = \binom{K}{2}$ and for $i = 1...\binom{K}{2}$, resolve x_i`.anonymity.fail` into the i^{th} pair of different IP addresses from the pool of K addresses, served in random order. Assuming the macOS stub resolver imposes its preference order among the K addresses in a random fashion (since they all potentially go through the same routing and arrive at the same server), this order can be easily calculated from the destination IP addresses of the subsequent HTTP requests, and thus this order can serve as an alternative DNS ID, with sufficient entropy. The analysis of a "mixed" case in which the order induces equivalnce classes is omitted due to space considerations, but results in having at least $\log_2(K!)$ entropy.

Google Chrome: Surprisingly, Google Chrome on macOS does not use the macOS stub resolver at all. Instead, Google Chrome implements its own, internal, standalone DNS stub resolver (and cache). The cache is per-query, it respects the original TTL (does not cap it), and is shared among regular browsing and privacy mode browsing. It does get wiped when browsing history is cleared (and when the browser is restarted). Therefore, while Chrome on macOS does not share DNS ID with other browsers, it does share the DNS ID between regular mode and privacy mode ("incognito"), and thus the tracking technique is still effective in this use case.

Firefox: The observations about Firefox's cache RRset expiration in Microsoft Windows (Section V-D) holds for macOS as well.

Conclusion: The net result is that all three browsers tested on macOS support the tracking technique, with the maximum TTL imposed by the macOS stub resolver (18.6 days) for Safari and Firefox (and no TTL capping for Chrome), and with cross browser tracking ID between Safari and Firefox.

F. iOS stub resolver and browsers

The iOS stub resolver implements a record-based cache. It supports Algorithm 1 as long as the RRsets for x_i`.anonymity.fail` do not differ (up to record order) in the various answers. The original RRset for x_i`.anonymity.fail` is retained by the stub resolver, and thus the tracking ID is kept intact in the stub resolver, while the algorithm randomizes the RRset order of x_i`.anonymity.fail` in the resolution platform's cache.

The iOS stub resolver takes notice of host outages and RTT issues and imposes its order preference on `getaddrinfo()`, just like macOS. See the treatment in Section V-E.

Conclusion: The results for iOS browsers, summarized in Table IV, are that all three browsers tested on iOS support the tracking technique, with maximum TTL imposed by the iOS stub resolver (18.6 days), and with cross browser tracking ID.

G. Android stub resolver and browsers

The Android stub resolver is oblivious to network issues at the carrier protocols (e.g. TCP). It provides the same answer to `getaddrinfo()` regardless of whether the IPs in it are responsive or not.

The observation on Firefox's cache RRset expiration from Section V-D holds for Android as well.

Conclusion: The results for Android browsers, summarized in Table IV, are that all browsers we tested on Android support the tracking technique, with cross browser tracking ID.

H. Linux

The market share of desktop Linux is very low. As of Feb. 2018, [1] reports "Other [operating systems]" (which include desktop Linux) as 0.6% of all operating systems, and [4] reports "Linux" at 2.32% of the desktop operating systems. Thus, and due to time constraints, we decided not to analyze tracking of Linux users.

I. Connectivity issues at the TCP/HTTP level

The browsers we tested skip an IP address if it is not responsive, and move to the next address (in the IP address list returned by `getaddrinfo()`). Thus a temporary network loss/outage/slowness in one or several of the IP addresses may temporarily change a few bits in the tracking ID, but as soon as all IP addresses are responsive, the tracking ID will resume its original, correct value. The tracking technique can be made resilient against such temporary TCP/HTTP issues, for example by issuing each request for individual script (hostname) three times, and taking the majority answer.

It is also possible that network conditions prevent the browser from accessing the IP address of one tracking server during the ID generation process (including macOS/iOS stub resolver reordering of the returned IP address list from `getaddrinfo()`). This case can be detected by going over all N values obtained from the K tracking web servers, and observing whether each of the K web servers returned at least one value. If this is not the case then the ID is deemed invalid. By Boole's inequality, the probability of a false positive (i.e., choosing not to connect to a tracking server although all servers are alive) has an upper bound of $K(1 - \frac{1}{K})^N$, which can be made as low as needed by increasing N.

J. DNS tracking ID Experiment

We ran an experiment to validate some properties of the tracking technique in the wild. Specifically, we simulated multiple (100) different clients of the same network (caching resolver) obtaining tracking IDs, and checking whether their IDs are unique, as well as the TTL assigned to the DNS records by the caching resolver. This was implemented as a standalone C program which emits DNS queries directly to the

Table V: DNS Tracking ID Experiment

Network	Type	DNS resolver IP	DNS resolver owner	TTL cap [s]	Num. IDs	Unique IDs [%]
Cellular network 1	Cellular ISP	8.8.8.8	Google	21600	100	100
Celluler network 2	Cellular ISP	194.90.0.11	Cellular provider DNS	604800	100	100
Cafè	Café hotspot	192.115.106.35	local ISP	604800	100	100
Conference wifi	Conference hotspot	8.8.8.8	Google	21600	100	100
Home office 1	xDSL ISP	8.8.4.4	Google	21600	100	100
Home office 2	Fiber ISP	non-routable (modem/router)	(forwarding to Google)	21600	100	100
Home office 3	Cable ISP	non-routable (modem/router)	(forwarding to ISP)	none	100	100
University	University network	132.71.147.34	University DNS	604800	100	100

nameserver (bypassing the stub resolver cache), and repeatedly generating the DNS tracking ID (employing Algorithm 1). We tested this for 8 networks that represent a wide variety of network scenarios. The results were positive - in 8 out of 8 networks, the technique provided 100 unique IDs to 100 simulated clients (i.e. no ID collisions). Also in the these 8 networks, the TTL did not exhibit unusually low values - in fact the minimum value (21600 seconds) is a result of some networks using the Google public DNS service (which caps TTL to this value). The results are summarized in Table V.

VI. Remediation and Countermeasures

It is difficult to prevent our DNS-based tracking technique since DNS caching is crucial for performance. For example, implementing a straightforward solution of disabling the DNS cache and always sending fresh DNS lookup queries will result in a noticeable hit to performance. Detecting DNS-based tracking is not straightforward. For example, while the tracking uses longer than average TTL there are still many sites that use TTL as long as 1 day, thus long TTL cannot be used to distinguish tracking from legitimate DNS records.

Systematic solutions: We suggest two systematic solutions that limit the effectiveness of tracking:

Adding randomization to the client side: Clients (browsers) should randomize the choice of IP addresses within the RRset, prior to every new connection to the resolved hostname. This takes care of the randomized RRset tagging scenario.

Using a client-sticky load balancing: That is, when load balancing is used, the same client always arrives at the same resolver. This takes care of the scenario wherein load balancing is used by the tracking technique (with *one* IP address RRset). It effectively reduces the number of IDs assigned to clients to the number of actual resolvers (caches) in the resolution platform. Client-stickiness can be achieved for example via IP-address based load balancing.

Manual solutions: Other solutions require manual intervention at the client:

Using a shared HTTP forward proxy (or Tor). In this case, the DNS queries are emitted from the shared proxy server, thus assigning the same tracking ID to all the clients behind it. This solution may be non-trivial to implement, both for consumers (finding a stable open proxy, etc.), and for enterprise users.

Flushing the operating system DNS cache very often. This is both cumbersome, and detrimental to performance (as the DNS cache's raison d'être is improving performance). In the extreme – disable DNS caching altogether.

Enterprise level solutions: Enterprises and ISPs can instruct their clients to use a forward HTTP proxy, or alternatively set up a resolution platform with sticky-by-IP load balancing, and fixed RRset order (e.g. Unbound or PowerDNS). In the latter workaround, however, all users may end up using a single IP address per hostname, which is probably not a desirable outcome.

Non-generic solution (domain blacklisting): A generic domain-based detection approach is not applicable since many legitimate hosts resolve into multiple IP addresses, and the technique doesn't rely on using hosts in the same domain. It is possible to block the domain name(s) (e.g. `anonymity.fail`) and/or the full hostnames used by the tracking technique (e.g. $\texttt{x}_i\texttt{.anonymity.fail}$) from correctly resolving. This can be done at the resolution platform or at the stub resolver (and maybe the browser). For example at the stub resolver, $\texttt{x}_i\texttt{.anonymity.fail}$ can be forced to resolve to `127.0.0.1` by adding an entry in the `/etc/hosts` file. This approach requires prior knowledge of the domain names used by the tracker, and as such it is always a step behind the tracker (who has the timing advantage). Note that the technique can be implemented using multiple domains and non-intersecting sets of IP addresses, thus hindering automatic detection of the tracking domains.

VII. Summary

We developed a novel user tracking technique, based on per-user caching of statistically unique DNS records. This technique covers the technologies used by a very large fraction of Internet users (in terms of browsers, operating systems, and DNS resolution platforms). It can distinguish between machines that were cloned from a "golden image" (i.e., have identical hardware and software), can overcome the "privacy mode" boundary, and in most cases can track users across different browsers (on the same machine).

This tracking technique is not easily mitigated. There are possible workarounds, but these are not without setup overhead, performance overhead or convenience overhead. A complete mitigation of this tracking technique requires software modifications in both operating systems and resolver software. This tracking technique is orthogonal to existing tracking techniques, and as such can be combined with them

to provide enhanced stickiness and additional confidence in user identification.

ACKNOWLEDGEMENTS

We thank Yotam Harchol for his helpful comments.

REFERENCES

[1] analytics.usa.gov. https://analytics.usa.gov/.
[2] Bug 1101528 - Firefox uses the same TLS session ticket and/or ID between normal and private browsing. https://bugzilla.mozilla.org/show_bug.cgi?id=1101528.
[3] Maintain separate tls session caches per-profile. https://bugs.chromium.org/p/chromium/issues/detail?id=30877.
[4] Operating system market share. https://netmarketshare.com/operating-system-market-share.
[5] Private browsing in flash player 10.1. https://www.adobe.com/devnet/flashplayer/articles/privacy_mode_fp10_1.html.
[6] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *ACM CCS '13*, pages 1129–1140, 2013.
[7] G. Aggarwal, E. Bursztein, C. Jackson, and D. Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security'10*, pages 6–6, 2010.
[8] F. Alaca and P. C. van Oorschot. Device fingerprinting for augmenting web authentication: Classification and analysis of methods. In *ACSAC '16*, pages 289–301, 2016.
[9] R. Auger. Socket capable browser plugins result in transparent proxy abuse. http://www.thesecuritypractice.com/the_security_practice/TransparentProxyAbuse.pdf, 2009.
[10] S. M. Bellovin. A technique for counting Natted hosts. In *2nd SIGCOMM Workshop on Internet Measurement*, pages 267–272, 2002.
[11] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. Mobile device identification via sensor fingerprinting. *CoRR*, abs/1408.1416, 2014.
[12] C. Buckle. Almost 1 in 2 use private browsing windows. https://blog.globalwebindex.com/chart-of-the-day/almost-1-in-2-use-private-browsing-windows/, 2016.
[13] T. Bujlow, V. Carela-Español, J. Solé-Pareta, and P. Barlet-Ros. Web tracking: Mechanisms, implications, and defenses. *CoRR*, abs/1507.07872, 2015.
[14] E. Bursztein. Understanding how people use private browsing. https://elie.net/blog/privacy/understanding-how-people-use-private-browsing, 2017.
[15] D. Dent. DNS cookie demonstration. http://dnscookie.com/, October 2015.
[16] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *ACM CCS '00*, pages 25–32, 2000.
[17] A. Janc and M. Zalewski. Technical analysis of client identification mechanisms. https://www.chromium.org/Home/chromium-security/client-identification-mechanisms.
[18] S. Kamkar. Evercookie. https://samy.pl/evercookie/, 2010.
[19] A. Klein. Predictable javascript math.random and http multipart boundary string. http://www.securitygalore.com/site3/math_random_and_multipart_boundary.
[20] A. Klein. The localhosed attack – stealing internet explorer 11-7 cookies for hosts on the local machine (and leaking the ip address as a byproduct). http://www.securitygalore.com/files/localhosed.pdf, June 2015.
[21] A. Klein. Hijacking DNS like it's 2016, October 2016.
[22] A. Klein, V. Kravtsov, A. Perlmuter, H. Shulman, and M. Waidner. X-ray-DNS. https://github.com/alonperl/X-Ray-DNS/tree/master/Parent_Attacker_Victim/X-Ray_tool, 2017.
[23] A. Klein, H. Shulman, and M. Waidner. Counting in the dark: Caches discovery and enumeration in the internet. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2017.
[24] T. Kohno, A. Broido, and k. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, May 2005.
[25] J. Moskowitz. New survey conveys the challenges of the windows 10 migration. https://betanews.com/2018/07/18/new-survey-conveys-the-challenges-of-the-windows-10-migration/, 2018.
[26] T. Nakamura. Google Chrome WebRTC IP address leakage (Comment 67).
[27] A. Popov, M. Nystrom, D. Balfanz, A. Langley, and J. Hodges. The Token Binding Protocol Version 1.0. Internet-Draft draft-ietf-tokbind-protocol-16, IETF, Oct. 2017. Work in Progress.
[28] D. Roesler. Demo. https://diafygi.github.io/webrtc-ips/, 2015.
[29] K. Seifried. User tracking methods. https://seifried.org/security/www-auth/section-5.html, 2001.
[30] M. Stockley. Anatomy of a browser dilemma – how hsts 'supercookies' make you choose between privacy or security. https://nakedsecurity.sophos.com/2015/02/02 /anatomy-of-a-browser-dilemma-how-hsts-supercookies-make-you-choose-between-privacy n-or-security/.
[31] H. Wramner. Tracking users on the world wide web.
[32] Y. Zhu. Weird new tricks for browser fingerprinting. https://zyan.scripts.mit.edu/presentations/toorcon2015.pdf.

APPENDIX

A. Prevalence of Single DNS Resolver

Our estimation of the portion of single resolvers in enterprise networks is explained as following: according to [23, Fig. 5], in 28% of the enterprise networks, a single resolver is used as the upstream resolution platform. Additionally, according to [21], in 35% of the enterprise networks, Google public DNS is used as the upstream resolution platform, and single Google Public DNS Service employs multiple caches, these two populations are mutually exclusive. This means that of the non- Google Public DNS Service upstream platforms, some $\frac{28}{65} = 43\%$ are single resolvers. If we assume that of the enterprises that use the Google Public DNS server, the vast majority (say, 80%) of enterprises use an internal resolver forwarding to Google Public DNS Service (and not Google Public DNS Service directly), and that the distribution of single resolvers there remain the same as the non- Google Public DNS service (namely, 43% are single resolver), we then have that in $0.28 + 0.8 \cdot 0.35 \cdot 0.43\% \approx 40\%$ of the enterprise networks, a single resolver is involved in the DNS resolution.

This means that any method that aspires to achieve wide coverage cannot ignore the single resolver scenario. However, the DNSCookie proposal does not address this scenario, thus fails to fulfill a crucial requirement for a tracking technique.

B. Lab Setup

We set up a lab in Microsoft Azure cloud with an authoritative name server for the domain `anonymity.fail`, written in Perl, based on the DNS X-Ray Tool [22]. and modified to include the specific tests needed for this research. In addition, the lab included four web servers, with dedicated IP addresses, running Apache 2.4.18 and PHP 7.0.18, and serving a simple Javascript code indicating which server served it.

The resolution platforms that were evaluated in the lab are BIND 9.11.2-P1, 9.10.3-P4 (Ubuntu), and 9.9.5-3 (Ubuntu); Unbound 1.6.8 and 1.6.0; PowerDNS Recursor 4.1.1, 4.0.4-1, 3.5.3; MaraDNS Deadwood 3.2.11 and 3.2.09; Microsoft DNS Server 10.0.16299, 10.0.14393, 6.2.9200, and 6.1.7601; and a DNS client using a Perl script running Dig for customized DNS tests.

More details about the lab setup are described in the full version of the paper.