Pool-Party: Exploiting Browser Resource Pools for Web Tracking

Peter Snyder
Brave Software

Soroush Karami
University of Illinois at Chicago

Arthur Edelstein
Brave Software

Benjamin Livshits
Imperial College London

Hamed Haddadi
Brave Software, Imperial College London

Abstract

We identify class of covert channels in browsers that are not mitigated by current defenses, which we call "pool-party" attacks. Pool-party attacks allow sites to create covert channels by manipulating limited-but-unpartitioned resource pools. This class of attacks have been known to exist; in this work we show that they are more prevalent, more practical for exploitation, and allow exploitation in more ways, than previously identified. These covert channels have sufficient bandwidth to pass cookies and identifiers across site boundaries under practical and real-world conditions. We identify *pool-party* attacks in all popular browsers, and show they are practical cross-site tracking techniques (i.e., attacks take 0.6s in Chrome and Edge, and 7s in Firefox and Tor Browser).

In this paper we make the following contributions: first, we describe *pool-party* covert channel attacks that exploit limits in application-layer resource pools in browsers. Second, we demonstrate that *pool-party* attacks are practical, and can be used to track users in all popular browsers; we also share open source implementations of the attack. Third, we show that in Gecko based-browsers (including the Tor Browser) *pool-party* attacks can also be used for *cross-profile* tracking (e.g., linking user behavior across normal and private browsing sessions). Finally, we discuss possible defenses.

1 Introduction

Browser vendors are increasingly developing and deploying new features to protect privacy on the Web. These new privacy features address the most common ways users are tracked on the Web: partitioning DOM storage to prevent tracking from third-party state, randomization or entropy reduction to combat browser fingerprinting, network state partitioning to prevent cache-based tracking, etc.

However, research has documented other ways Web users can be tracked, though in ways that may be difficult to conduct under realistic browsing conditions. Significantly among these are covert-channels that can be constructed through timing signals, or other side channels. These covert-channels allow sites to communicate with each other—or even other applications—in ways not intended by browsers. Such covert-channels can be used to reintroduce the kinds of cross-site tracking attacks the above-discussed browser protections were designed to prevent.

Browser vendors have responded to covert-channels in a variety of ways. Some covert-channels (e.g. timing signals from abusing HTTP cache state) have been addressed through platform wide improvements like network state partitioning. Other covert-channels have been addressed—or at least mitigated—through other protections, like isolating sites in their own OS processes. Others attacks have been left unaddressed, because browser vendors judge them to be impractical to execute in realistic browsing scenarios.

In this work we demonstrate that current browser protections are insufficient to prevent sites from using covert-channels to circumvent anti-tracking protections in browsers, including the protections deployed by the most privacy-focused browsers. We demonstrate this by defining a new category of techniques for constructing covert-channels, by exploiting the state of limited-but-unpartitioned resource pools in the browser. Because such covert-channels are exploited by two parties colluding in the same resource pool, we call this category of covert-channel "pool-party" attacks.

"Pool-party" attacks create covert-channels out of browser-imposed limits on pools of resources. When resource pools are limited (i.e. the browser only allows pages to access resources up to some hard limit, after which requests for more resources fail) and unpartitioned (i.e. different sites consume resources from a shared pool), sites can consume and release resources to leak information across security boundaries. Examples of such boundaries include site boundaries (e.g. the browser intends to prevent site A from communicating directly with site B) and profile boundaries (e.g. the browser intends sites visited in a "standard" browsing session to not be able to learn about sites visited in an "incognito" browsing session). More generally, attackers can use these covert-channels to conduct the kinds of cross-site tracking that the recent

browser features were intended to prevent.

We identify practical "pool-party" attacks in all popular browsers, both in browsers' default configurations, and non-standard, hardened configurations. We demonstrate "pool-party" attacks through three resource pools: WebSockets, Server-Sent Events, and Web Workers, and find that all browsers were vulnerable to at least one form of attack. We further identify other limited-but-unpartitioned resource pools in browsers that could be leveraged for "pool-party" attacks. Examples of such pools include certain kinds of resource handle (e.g. Web Speech API), or limits on how many network requests (distinct from network connections) can be in flight at once (e.g. DNS resolution), among others. Finally, we demonstrate that "pool-party" attacks are not just *theoretical* threats to user privacy, but *practical* threats that can be used to track users across sites. We show that in Gecko-based browsers (including the Tor Browser), "pool-party" attacks can create covert-channels *across profiles*, allowing sites to link behaviors in "private browsing" modes with standard, long term browser identities.

These findings are important for the development of browser partitioning. All browser engines support some forms of partitioning: WebKit partitions DOM storage and some kinds of network state, Gecko partitions DOM storage and network state, and Chromium partitions network state[1]. Brave has extended Chromium to also partition DOM storage.

1.1 Contributions

This work makes the following contributions:

- We **define a new category of technique for creating covert-channels in browsers**. We call this category of covert-channel "pool-party" attacks, and describe how the approach differs from the kinds of privacy attacks browsers currently aim to defend against;
- We **evaluate deployed browsers**, and find all popular browsers and browser engines are vulnerable "pool-party" attacks;
- We provide three **open-source, proof-of-concept** implementations of our attack that work in all browsers[2];
- We perform a **performance measurements** to evaluate the bandwidth and practicality of "pool-party" attacks, and find that "pool-party" attacks are a practical basis carrying out cross-site tracking attacks;
- We **discuss potential mitigation strategies** for how browsers could defend against "pool-party" attacks.

[1] At time of writing, network state partitioning is deployed for a portion of Chrome and Edge users, as part of the "NetworkIsolationKey" feature

[2] https://github.com/brave-experiments/pool-party-artifact/blob/master/static/inner.js

1.2 Responsible Disclosure

We have presented our findings to the following browser vendors (in alphabetical order): Apple, Brave, Google, Microsoft, Mozilla, Opera, Tor Project. All reports were made over 90 days in advance of this submission.

Microsoft and Opera responded that since the discussed vulnerabilities were in Chromium, they would wait for Google to address the problem. The Tor Project similarly said they would rely on Mozilla to address the vulnerabilities[3].

Some vendors have shipped fixes for the vulnerabilities identified in this work. Safari fixed the SSE event vulnerability in version 15.2[4], and Brave has released fixes for the WebSocket[5] and SSE[6] vulnerabilities.

Google[7] and Mozilla[8] also plan to address these vulnerabilities, though have not done so yet. These organizations are focusing on a mixture of browser-wide fixes (i.e. comprehensively partitioning all resource pools, not only the resource pools discussed in this work) and updates to Web standards (e.g. defining limits and the scope of connection pools).

2 "Pool-party": Definition and Background

This section provides context for how "pool-party" attacks relate to other Web tracking techniques, and how browser vendors' privacy models and goals have changed. This section also describes *why* existing browser protections fail to protect users against "pool-party" attacks.

This section first defines "Web tracking", followed by discussing how privacy models in Web browsers have improved, and why "pool-party" attacks allow trackers to violate intended privacy boundaries in all popular browsers. Next, we describe how "pool-party" attacks relate to both i) other covert-channels in browsers, and ii) conventional Web tracking techniques. We then explain why existing browser protections do not protect users against "pool-party" attacks, and conclude by describing how this work relates to a category of attack previously known to be possible, but not thought to be practical.

2.1 Web Tracking and Cross-Site Tracking

This sub-section gives a working definition of Web tracking. Our goal is not to provide a formal, unambiguous definition (the phrase "Web tracking" is used too broadly to likely allow for one), but instead to give a practical definition to build on through the rest of this work.

[3] https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/41381

[4] https://support.apple.com/en-us/HT212982

[5] https://github.com/brave/brave-core/pull/11609

[6] https://github.com/brave/brave-core/pull/16882

[7] https://bugs.chromium.org/p/chromium/issues/detail?id=1249658

[8] https://bugzilla.mozilla.org/show_bug.cgi?id=1730797

We use "Web tracking" to refer to a user being re-identified across conceptual contexts, without the user's expectation or consent. We use "context" to refer to a grouping of activities that the user expects to be separate from, and not accessible to, other similar contexts. This definition is similar to the W3C's proposed privacy principals[9].

Contexts might be divided by *time* (e.g. a site re-identifying a user revisiting the same site a week after first visiting, despite the user clearing browsing data), *application* (e.g. a site re-identifying a user visiting a site in Safari as the same user who previously visited the site in Chrome), *profile* (e.g. a site identifying that the visiting in an private/incognito browser session is the same user visiting the site in a standard browser session), or *site* (e.g. two sites colluding to learn that browser sessions occurring on each site belong to the person). The commonality is the users' reasonable expectation that things that happen in one context are not readily known and available to other contexts.

2.2 First-Party Site as Privacy Boundary

Browser vendors are converging on the first-party site as the Web's privacy boundary. All browsers include features intended to prevent sites from communicating across first-party site boundaries. Some browsers enforce this boundary by default; others only do so with opt-in "privacy" modes, but all browsers include such features.

Using the first-party site as a privacy boundary means that a third-party embedded under two different first-parties should not be able to confidently know it was the same person visiting each site, unless the user intentionally re-identifies themselves to the third-party.

The rest of this subsection documents that, and in what configurations, each browser uses the first-party site as their privacy boundary. In all the discussed configurations, browsers intended to communication across first-party site boundaries, and in all cases attackers can circumvent the intended privacy boundary through "pool-party" attacks.

Gecko Browsers. Both Firefox (as of version 103) and Tor Browser enforce the first-party site as the privacy boundary by default. In Tor Browser, the protection is sometimes called "first-party isolation" or "cross-origin identifier unlinkability"[10]. In Firefox, the feature is called "Total Cookie Protection"[11].

WebKit Browsers. Safari uses the first-party site as the privacy boundary by default. The WebKit documentation makes this privacy boundary explicit in their documentation, which mentions that they intended to protect against cross-site communication through covert-channels[12].

Chromium Browsers. Chromium *does not* enforce the first-party site as a privacy boundary by default. However, Chromium allows for configurations that do, by a combination of i) disabling third-party cookies (to prevent DOM storage communication across site boundaries) and ii) enabling Chromium's `"NetworkIsolationKey"` (NIK)[13] features (which partition caches and other network state by first-party).

Neither Chrome or Edge disable third-party storage by default, but both do enable NIK features for most users. We note though that even when Chrome and Edge are configured to use the first-party site as the privacy boundary, those browsers are vulnerable to "pool-party" attacks.

The Brave Browser uses a modified version of Chromium that, by default uses the first-party site as a privacy boundary. It does this by partitioning third-party DOM storage by first-party[14], and by enabling (and extending) Chromium's NIK system for all users[15].

2.3 Description of "Pool-party" Attack

"Pool-party" attacks manipulating pools of browser resources which are limited (i.e. the browser restricts how many of the resource can be used at one time) and unpartitioned (i.e. different contexts consume resources from the same pool). While the examples focused on in this work utilize either limited-but-unpartitioned pools of i) network connections or ii) thread handles, browsers include many other limited-but-unpartitioned resource pools that could be similarly exploited, such as pools of file handles, subprocesses, or other resource handles.

A "pool-party" attack occurs when parties operating in distinct contexts (contexts the user expects to be distinct and blinded from each other) intentionally consume and query the availability of the limited resources in a resource pool, to create a cross-context communication channel. Each context can then use the communication channel to pass an identifier, allowing each party to link the user's behavior across the two contexts. We note again that most commonly the two contexts considered here are two different websites running in the same browser profile, but could also be the same (or different) websites running in different browser profiles.

Algorithm 1 presents a simple-though-limited technique for conducting a "pool-party" attack, where sites can trivially transform this optimization choice into a cross-site tracking mechanism.

[9] https://w3ctag.github.io/privacy-principles/

[10] https://2019.www.torproject.org/projects/torbrowser/design/#identifier-linkability

[11] https://blog.mozilla.org/security/2021/02/23/total-cookie-protection/

[12] https://webkit.org/tracking-prevention-policy/

[13] https://github.com/shivanigithub/http-cache-partitioning#choosing-the-partitioning-key

[14] https://brave.com/privacy-updates/7-ephemeral-storage/

[15] https://brave.com/privacy-updates/14-partitioning-network-state/

Algorithm 1 Toy example of a "pool-party" attack.

Site A: $I_a \leftarrow$ *random N bits*
Site B: $I_b \leftarrow$ *empty string*
while $i \leftarrow I_a$ **do**
  Site A: Stop any playing videos
  if $I_a[i] = 1$ **then**
    Site A: Play a video
  end if
  Wait 5 seconds
  if Site B is able to play a video **then**
    Site B: $I_b[i] = 1$
  else
    Site B: $I_b[i] = 0$
  end if
end while

For this toy example, assume a browser vendor wants to improve performance by only allowing one video element to be loaded at a time, across all sites. If a video is currently playing on any page, the site will receive an error if it tries to play a new video. An attacker use this implementation choice to "send" a bite across site-boundaries by playing (or not) a video on one site, and checking on another site whether there is a video playing. An arbitrarily large message can be sent by repeating this process. A more realistic and efficient technique is presented in Section 3.

2.4 Relationship to Other Covert-Channels

"Pool-party" attacks differ from other covert-channel attacks by targeting intentional, application-imposed limits. This differs from many other covert-channel attacks in two ways, both of which increase the practicality of "pool-party" attacks.

First, "pool-party" attacks target application-level resources, while many other covert-channels target parts of the system below, or at least distinct from, the application (e.g. hardware restrictions like CPU caches, OS details like interrupt schedules or memory management, or language runtime features like garbage collection). This is significant because, the lower in the stack the attacker targets, the more likely the resource is (all other things being equal) to be shared with other actors on the system. This means that lower-level covert-channels are more likely to be noisy, and so more difficult to communicate over.

Second, related but distinct, "pool-party" attacks target browser-managed resources, resources that are, in most cases, intentionally shielded from other applications on the system. This again reduces the chance that colluding parties will have to contend with a noisy, unpredictable covert-channel.

2.5 Relationship to Other Tracking Methods

"Pool-party" attacks do not fall neatly into the categories usually used to describe browser tracking techniques. This subsection briefly describes the rough-taxonomy used in online-tracking research, and why "pool-party" does not cleanly fall into existing categories.

Browser	DOM Storage	Network State
Brave	⊕	⊕
Chrome	×	⊖
Edge	×	⊖
Firefox	⊕	⊕
Safari	⊕	⊕
Tor Browser	⊕	⊕

Table 1: State partitioning features in popular browsers (in alphabetical order). ⊕, ⊖ and × indicate the feature being available by default for all, some, or no users, respectively.

Stateful Tracking. "Stateful tracking" most commonly refers to websites using explicit storage APIs in the Web API (e.g. cookies, localStorage, indexedDB) to assign identifiers to browser users, and then read those identifiers back in a different context, to link the identity (or, browser behavior) across those contexts.

Stateful-tracking also describes other ways websites can set and read identifiers, by using APIs and browser capabilities not intended for such purposes. Examples of such techniques include exploiting the browser HTTP cache, DNS cache or other ways of setting long term state (e.g. HSTS instructions [37], favicon caches [35], or, ironically, storage intended to prevent tracking [13]).

Browsers increasingly protect users from stateful tracking by partitioning storage by context, mostly commonly be the effective-top level domain (i.e. `eTLD+1`) of the website. Giving each context a unique storage area prevents trackers from reading the same identifier across multiple contexts, and so prevents the tracker from linking browsing behaviors in different contexts. Partitioning explicit storage APIs is often referred to as **DOM Storage** partitioning. Partitioning caches and other "incidental" ways sites can store values is often called **network state partitioning**. Table 1 provides a summary of state partitioning in popular browsers.

Browser state partitioning strategies fail to defend against "pool-party" attacks because "pool-party" attacks do not rely on setting or retrieving browser state (at least not in the way state is generally discussed in this context, meaning the ways that sites can write state to the users profile). "Pool-party" attacks instead rely on implementation details of browser architecture, where device resources are limited-but-unpartitioned. While partitioning strategies could also be used to defend against "pool-party" attacks, as discussed in Section 5, certain aspects of these attacks make partitioning approaches difficult in practice.

Stateless Tracking. "Stateless tracking," (also often called "browser fingerprinting") refers to the category of Web track-

Browser	Coordination	Stability	Uniqueness
Brave	⊕	⊕	⊕
Chrome	×	×	×
Edge	×	×	×
Firefox	⊕	×	⊖
Safari	×	×	⊕
Tor Browser	×	×	⊕

Table 2: Stateless tracking protections in popular browsers (in alphabetical order). ⊕, ⊖ and × indicate the defense is available by default, off by default, or not available, respectively.

ing techniques whereby the attacker constructs a unique identifier for the user by combining a large number of semi-distinguishing browser and environmental attributes into a stable, unique identifier. Examples of such semi-distinguishing features include the operating system the browser is running on, the browser version, the names and the number of plugins or hardware devices available, and the details around the graphics and audio hardware present, among many others [19].

In contrast to "stateful" tracking techniques, "stateless" techniques do not require sites to be able to set and read identifiers across context boundaries, and so are robust to storage partitioning defenses. Stateless attacks instead rely on three conditions to be successful:

- **Coordination:** code running in different contexts must know to query the same (or at least sufficiently large intersection of) browser attributes.
- **Stability:** the browser must present the same values for the same semi-distinguishing attributes across contexts (otherwise the browser will yield different fingerprints in different contexts, preventing the attacker from matching the two fingerprints).
- **Uniqueness:** the browser must present enough semi-distinguishing attributes to allow the site to accurately differentiate between users (otherwise the attack will confuse two different users as the same person)

Browsers defend against "stateless" trackers by attacking any of these three requirements. A browser might prevent **coordination** by blocking fingerprinting code on sites, or prevent **stability** by making the browser present different attributes to different sites (such as in [18, 25]), or prevent **uniqueness** by reducing the entropy provided by each attribute. Table 2 provides a summary of deployed "stateless" defenses in popular browsers.

Browser defenses against "stateless" tracking techniques fail to defend against "pool-party" attacks because of differences in the nature of the attack. "Stateless" techniques target stable semi-distinguishing browser characteristics which are set by a page's execution environment. "Pool-party" attacks, in contrast, are enabled by sites consuming and reading the availability of limited resources in the browser across execution contexts. Therefore, unsurprisingly, browser defenses against "stateless" tracking provide no protection against "pool-party" attacks.

XS (Cross-Site) Leaks. "Pool-party" attacks are most similar to a category of attack loosely called "XSLeaks"[16], a broad collection of ways sites can send signals to each other in ways generally unintended by browser vendors. However, we note that in contrast to "stateful", "stateless," "pool-party" attacks, XSLeaks do not have a common cause or remedy; instead, XSLeaks can be largely thought of as a catchall for cross-site (or cross-context) techniques that do not not fit in another category. Examples of XSLeaks include timing channels stemming from a variety of causes, unintended side effects of experimental browser features[17], or misuse of other browser APIs[18].

The lack of a common cause of XSLeaks makes it impossible to generalize about defensive strategies or deployed browser defenses. Recent work in this area has identified ways sites can leak information across browser-imposed boundaries, including through unintended side effects in how browsers handle errors, implement cross-origin opener-policy (COOP), cross-origin resource policy (CORP), and cross-origin read blocking (CORB) policies, or limit the length of redirection chains, among many other signals [16].

We note though that "pool-party" attacks are most common to the "connection pool" attacks identified by the XSLeaks project[19]. This work makes the following contributions beyond the issues documented by the XSLeaks project, and the related work done by Kinttel et al. [16].

1. This work defines a larger category of attack than XSLeaks, where any limited-but-unpartitioned resource pool can be transformed into a covert-channel. The attack documented by the XSLeaks project is a subset of the larger category of attack discussed in this work. Network connection pools can be abused to conduct "pool-party" attacks, but other kinds of resource pool can too. For example, the Web Workers pool in Firefox thecan be exploited to conduct "pool-party" attacks. Section 5.3 identifies additionalm non-network-connection pools that can be exploited.

2. We demonstrate that attacks of this type are not just theoretically possible, but are practical, and are real-world threats to Web privacy.

3. The "connection pool" attack identified by the XSLeaks project relies on abusing the connection pool to create timing channels, while "pool-party" attacks utilize the

[16] https://xsleaks.dev/

[17] e.g. https://xsleaks.dev/docs/attacks/experiments/scroll-to-text-fragment/

[18] e.g. https://xsleaks.dev/docs/attacks/window-references/

[19] https://xsleaks.dev/docs/attacks/timing-attacks/connection-pool/

number of available resources in the pool to create the communication channel. This small difference is significant. Using the amount of available of resources in the pool as the communication channel makes the attack more robust to noise introduced from other sites, and mitigations against one form of the attack may not apply to the other.

3 Generic "Pool-party" Attack Algorithm

In the previous section we presented the category of "pool-party" attack in the abstract, explained why "pool-party" attacks are different from other attacks discussed previously in the literature, and why current browser defenses fail to protect against "pool-party" attacks. In this section we present a generic algorithm for conducting "pool-party" attacks, which can then be applied to any resource pool in a browser where the following conditions are met.

1. The resource pool is **limited**, meaning that sites can request resources from the pool until a global limit is hit, after which sites are prevented from accessing more resources, in a manner the site can detect.

2. The resource pool is **unpartitioned**, meaning that different contexts (e.g. sites, profiles, etc.) all draw from the same global resource pool. Put differently, the attack will fail if each context gets a distinct resource pool.

3. Sites can **consume** resources from the pool without restrictions, as long as the pool is not already exhausted.

4. After consuming resources, sites can **release** any number of those resources back into the pool.

Any resource pool where the above four criteria are met can be transformed into a covert communication channel between any two parties sharing the resource pool.

We have identified resource pools matching the above criteria in current versions of all popular browsers, even browsers that particularly emphasize their privacy features (e.g. Brave Browser, Tor Browser), and even when browsers are "hardened" by the enabling of non-default, privacy-focused features (as discussed in Section 2.5).

3.1 "Pool-party" Algorithm

We present a generic protocol for conducting a "pool-party" attack over limited-but-unpartitioned resource pools in all browsers. This protocol is presented as Algorithm 2, and provides a generic way a site can use a limited-but-unpartitioned resource pool to track users.

Protocol Inputs. The algorithm takes several inputs. First, the algorithm takes which resource pool will be exploited to conduct the attack, which also determines the size of the pool.

Algorithm 2 General algorithm for a "pool-party" attack.

Inputs.
POOL_SIZE ← size of resource pool
PKT_SIZE ← $\lfloor \log(POOL\_SIZE) \rfloor$
MSG ← binary string to transmit
NEGOTIATE_INTERVAL ←
time to choose sender and receiver roles
PULSE_INTERVAL ← time to transmit one chunk of data

1. Setup.
CHUNKS ← MSG split into packets of size PKT_SIZE
RECV_MSG ← *empty string*
START_TIME ← $\lceil$ NEGOTIATE_INTERVAL + len(CHUNKS) ∗ PULSE_INTERVAL $\rceil$

2. Determining Initial Sender and Receiver.
Both sites: sleep until START_TIME
Both sites: consume resources until pool is exhausted
if "Site A" is able consume over > 50% of pool **then**
    SENDER ← "Site A"
    RECEIVER ← "Site B"
else
    SENDER ← "Site B"
    RECEIVER ← "Site A"
end if
Sender: consumes 100% of pool resources
Receiver: releases all pool resources
Both sites: sleep until
START_TIME + NEGOTIATE_INTERVAL

3. Sending Data.
for $i \leftarrow 0..len$(CHUNKS) **do**
    Sleep until START_TIME + NEGOTIATE_INTERVAL + i ∗ PULSE_INTERVAL
    if SELF == SENDER **then**
        SEND_INT ← *binaryToDecimal*(CHUNK[i])
        Consume all unheld resources in pool
        Release SEND_INT resources in the pool
    else if SELF == RECEIVER **then**
        Sleep for 0.5 ∗ PULSE_INTERVAL
        Consume all unheld resources in pool
        RECV_INT ← number of consumed resources
        Release all held pool resources
        RECV_STR ← *decimalToBinary*(RECV_INT)
        RECV_MSG || = RECV_STR
    end if
end for
Release all held pool resources

Attackers can precompute the largest pool available for each browser. The size of the resource pool (i.e. the number of resources in the pool available) is stored as `POOL_SIZE`.

The second input is the message being sent over the channel, which is a binary string of arbitrary length. The binary string to be transmitted is stored as `MSG`.

Third, the algorithm takes two time intervals, stored as

NEGOTIATE_INTERVAL and PULSE_INTERVAL. These intervals could be fixed across all attack methods, and trade faster transmission time (smaller values) against higher reliability (lower values).

Step One: Setup. To begin, the sender splits MSG into PKT_SIZE sized chunks, yielding a vector of bit-strings each of size PKT_SIZE, and the receiver constructs an empty buffer, RECV_MSG, to accumulate the received message into one packet at a time. The sending and receiving sites must choose the same time to start communication: the shared START_TIME is set to the next integer ECMAScript epoch time (in seconds) that is greater than a multiple of the full negotiation and message transmission time.

Step Two: Determining Initial Sender and Receiver. Both parties synchronize by sleeping until the START_TIME, and then determine which site will be the initial *sender* and which the *receiver*. This negotiation is needed because, neither site initially knows what other colluding site(s) may be open and available to communicate with, and thus no way of assigning roles in the protocol.

Sites determine *sender* and *receiver* by racing to exhaust the resource pool. The site that is able to consume more than 50% of resource in the pool assigns itself the role of initial *sender*; the site that is prevented from requesting the $50\% + 1$ resource assigns itself as the initial *receiver*.

The *receiver* then releases the resources it holds, and the *sender* keeps consuming resources until the pool is exhausted.

Step Three: Sending Data. The third step of the protocol is where passing data across context (i.e. site) boundaries occurs. The *sender* and the *receiver* participate in this step of the protocol as follows.

The *sender* manipulates the state of the resource pool as follows for each c in their CHUNKS vector (recall that CHUNKS is a vector of binary strings, each of length PKT_SIZE). For each c, the *sender* first interprets the binary as positive integer representation (e.g. 0010010 becomes 18, etc), which is stored as SEND_INT. The *sender* then releases $\texttt{SEND\_INT} + 1$ resources from the pool and waits for a fixed period, PULSE_INTERVAL, to ensure that the the *receiver* has had time to read from the channel. Once the *sender* has finished sending their message, the *sender* releases all resources in the pool and proceeds to the next step in the protocol. Otherwise, the *sender* consumes all resources in the pool and repeats the current stage in the protocol to send the next c value.

Simultaneously, the *receiver* begins this stage of the protocol by waiting for PULSE_INTERVAL/2. Once that time has elapsed, the *receiver* tries to consume as many resources as possible, which will match the SEND_INT number of resources released by the *sender*, and stores this value (minus 1) as RECV_INT[20] Next, the *receiver* encodes RECV_INT in the inverse manner the *sender* used (e.g. 18 becomes 0010010), and concatenates the result onto the *receiver's* RECV_MSG. The *receiver* then releases all resources it holds, waits for the end of the pulse, and repeats the above process.

Step Four: Exchanging Roles. Finally, if desired, the two parties can exchange roles to pass data in the *receiver* to *sender*. This is trivially accomplished by each party assuming the opposite role, and continuing again from step 3. Otherwise, if there is no more data to transmit, both parties can abort the protocol. Note that the protocol itself does not provide a mechanism for the parties to indicate whether they wish to continue or end the protocol, though parties could easily signal such through the contents of the messages being passed.

4 Evaluation in Popular Browsers

In the previous section we presented a generic algorithm for turning limited-but-unpartitioned resource pools into cross-context communication channels, which in turn can be used to cookie-sync and track users across the Web. In this section we demonstrate three examples of such limited-but-unpartitioned resource pools in popular browsers, and measure how exploitable and practical they are for cross-site tracking.

Specifically, we show that practical forms of "pool-party" attacks can be carried out in popular browsers. We implemented three examples of "pool-party" attacks, using the WebSockets, Server Sent Events (SSE), and Web Workers APIs. Before this work, all Chromium browsers were vulnerable to the WebSockets and SSE attacks, Firefox was vulnerable to the Web Sockets and Web Workers attacks, Tor Browser was vulnerable to the WebSockets attack, and Safari was vulnerable to the SSE attack.

We assess the practicality of each of implemented "pool-party" attack through four measurements: **Availability**, the size of the relevant resource pool, and the kind of context-linking possible, **bandwidth**, or how long it takes to send a 35-bit identifier through the channel, **consistency**, or how often the identifier is sent correctly, and **background noise**, or how often sites on the Web use resources in each resource pool.

Finally, for all measurements of Chromium browsers, we configured each browser to enable all site-as-privacy-boundary features enabled (i.e. we enabled all browser features designed to allow communication across sites or site-partitions). Specifically, we disabled third-party cookies, and enabled all "Network Isolation Key" features to enforce cache and network-state partitioning (see Section 2.5 for more information on these "hardened" Chromium configurations see). We note the exception here was Brave, which partitions DOM storage and network state by default.

[20] Recall that, by construction, the *sender* is not able to obtain more than 2^{PKT_SIZE} resources, and so the *receiver* can be certain that values greater than the limit, or equal to zero, are not data the *sender* is attempting to transmit. If so, the *receiver* will exit the protocol.

4.1 Attack Availability

Methodology. We first checked the availability (and so, exploitability) of each example "pool-party" attack by experimenting with browsers and examining the source code of each browser engine to identify limited-but-unpartitioned resource pools in those browsers.

Specifically, we considered Web APIs that might hypothetically represent a finite pool of resources (network connections, threads, etc.). We then used the developer console in each browser to manually test whether each Web API would leak and whether that resource pool could be predictably exhausted. We conducted these tests as follows:

1. We opened a new tab,. visited a blank page, and checked if instantiating the candidate Web API repeatedly in a loop in the developer console caused errors after a finite and predictable number of calls.

2. Once that pool was exhausted, we then opened a second tab to a different site. Did we find that no more resources were available under the second site?

3. If we released a resource under the first tab, could a single resource now be consumed without error under the second tab?

4. Were we able to find logic in the corresponding browser engine code that was imposing this resource pool limit?

If the answer to all four of these questions was yes, then we concluded this Web API was vulnerable to the pool party attack for the tested browser. We thus proceeded to implement attacks against each vulnerable browser based on the algorithm presented in Section 3, implemented in JavaScript[21]. We then examined whether we could use the relevant resource pool to create a covert-channel and communicate across site boundaries, across profile boundaries, or both. Importantly, we tested whether the attack technique can be used to communicate between a standard-browsing profile, and a "private browsing mode" profile[22].

Results. Our *availability* measurements yielded several significant findings, summarized in Table 3.

First, we were able to identify exploitable limited-but-unpartitioned resource pools in all major browsers, which we were successfully able to exploit through "pool-party" attacks (though Safari and Brave both fixed some vulnerabilities during the "responsible disclosure" process). As noted, the resource pools targeted in each browser engine differ. We were able to use the relatively large WebSockets connection pool in Chromium- and Gecko-based browsers to conduct "pool-party" attacks. Safari's WebSockets implementation was not exploitable, since WebKit does not restrict how many WebSocket connections can be opened simultaneously. Safari's implementation of the SSE API, though, was previously exploitable before they fixed it. (Gecko's implementation of the SSE API *was not* exploitable).

Firefox alone was vulnerable to the Web Workers form of the attack (a surprising finding given that Tor Browser uses the same Gecko engine).

Second, we found that Gecko-based browsers (i.e. Firefox and Tor Browser) were vulnerable to "pool-party" attacks in a way more concerning than other browser engines. While "pool-party" attacks can be used for cross-site tracking in all browsers, **in Gecko-based browsers "pool-party" attacks can be used to track users *across profiles***. Significantly, this means that, in Gecko-based browsers, sites can conduct "pool-party" attacks between private browsing sessions and standard browsing sessions. More concretely, a site running in a private browsing window can collude with a site running in a standard browsing window, and identify both sessions as belonging to the same person. This is particularly concerning since it violates the core promise of a private browsing session; that behaviors conducted using a private browsing are "ephemeral", and cannot be linked other accounts or behaviors a user maintains. Additionally, this vulnerability undermines the work and research that has been done to strengthen private browsing modes in browsers (for example, [4, 8, 21]).

4.2 Attack Bandwidth

Methodology. We measured the bandwidth of each attack by measuring how long each implemented "pool-party" attack took to transmit a 35-bit string across the site (or in the case of Firefox's WebSockets implementation, profile) boundaries.

We selected a 35-bit string for two reasons. First, because it is over 33-bits, or what is needed to uniquely identify the approximately 7.9 billion people on the planet, and two, because it aligns cleanly with the 5-bit packet size used in WebSocket and Web Worker experiments.

We conduct each measurement as follows. First, we manually open two tabs on a browser to two pages on two different sites we controlled. Each page includes an implementation of the relevant "pool-party" attack (Websockets and SSE in Chromium-based browsers, WebSockets and Web Workers in Gecko-based browsers, and SSE in Safari), implemented through JavaScript included in the page. We then experimentally varied the negotiation time and pulse time until we found the minimum times necessary to ensure that messages were passed accurately with a high success rate. We then conduct this measurement 100 times, using a clean browser profile for each measurement, and report the average.

Results. We report the results of our bandwidth measurements in Table 4. Times are reported in seconds, and the

[21] https://github.com/brave-experiments/pool-party-artifact/blob/master/static/inner.js

[22] This feature goes by different names in different browsers, but generically refers to the ability to run the browser in a way where stored values only last the lifetime of the browsing session.

Browser	Engine	Version	WebSockets	Web Workers	Server-Sent Events
Brave	Chromium	1.44.101	∗ 255	-	1,350
Chrome	Chromium	105.0.5195.125	255	-	1,350
Edge	Chromium	106.0.1370.42	255	-	1,350
Firefox	Gecko	105.0.1	† 200	512	-
Safari	WebKit	15.2	-	-	∗ 6
Tor Browser	Gecko	11.5.2	200	-	-

Table 3: **Attack Availability**: Size of each resource pool used to conduct each instance of a "pool-party" attack. "-" denotes that the browser was not vulnerable. "∗" indicates that the vulnerability was fixed before this work was submitted. "†" denotes that the resource-pool can be exploited to conduct cross-profile attacks (in addition to cross-site).

Browser	Method	Setup	Send	Total	Success
Brave	SSE	3.0	5.0	8.0	100%
Chrome	SSE	2.0	5.0	7.0	100%
Edge	SSE	2.0	5.0	7.0	100%
Chrome	WS	0.1	0.5	0.6	100%
Edge	WS	0.1	0.5	0.6	100%
Firefox	WS	2.0	5.0	7.0	71%
Tor Browser	WS	2.0	5.0	7.0	73%
Firefox	WW	1.5	7.5	9.0	95%

Table 4: **Attack Bandwidth**: Number of seconds to transmit a 35-bit string). Times are reported in seconds; all values are reported over 100 runs.

Web API	% page Loads	% of URLs Desktop	% of URLS Mobile
Web Worker	12.34%	12.29%	11.9%
WebSocket	9.55%	4.33%	3.72%
Server-Sent Events	0.79%	0.8%	0.06%

Table 5: **Attack background noise**: Web API metrics reported by the Chrome Platform Status service, as of August 9, 2022. Numbers reflect the % of page loads and % of URLs observed across all channels and platforms.

transmission success rate (discussed in the next subsection) is reported as a percentage.

We find that our example "pool-party" attacks are practical. Even the slowest forms of the attack complete in under ten seconds (far below the average page dwell time of slightly under a minute [23]). Each attack could further be carried out between pages that are left open for a moderate amount of time, either because they get lost in a browser users ever-growing collection of tabs, or because the site is intended to stay open for a long time (e.g. sites that function and email clients, instant messaging applications, video streaming sites, etc). Our example "pool-party" attacks are fast enough that the could be conducted multiple times during an average page view (again, assuming an average page dwell time of slightly under one minute), as a simple error handling technique to account for noisy channels.

The "Setup" column in Table 4 corresponds to the "Determining Initial Sender and Receiver" section of Algorithm 2; the "Send" column measures the "Sending Data" steps.

4.3 Attack Consistency

Methodology. We also evaluated how consistently each "pool-party" attack example completed successfully, in the absence of other sites running on the browser. This measurement provides an upper bound on how practical the attack could be, as having other sites running in parallel in the browser will in some cases further reduce the success rate.

We measured the consistency of each attack using the same methodology described in Section 4.2. We again ran the attack 100 times, on two different pages in a single instance of the browser, each time in a clean profile. We then report the percentage of times the 35-bit string was received correctly.

Results. The results of our consistency measurement is also reported in Table 4. We find that most forms of the attack are either perfectly consistent (i.e. all 100 evaluations executed correctly), or consistent enough to be practical (i.e. the Web Worker attack on Firefox). The WebSocket attack was less consistently successful in the Gecko browsers. We investigated the root cause and found that the pool size was not consistently enforced; Firefox occasionally allowed additional sockets to be created, resulting in a corrupted message.

4.4 Attack Background Noise

Methodology. Finally, we evaluated how noisy the communication channels used in our demonstrative "pool-party" attacks are in practice. We build on the intuition that resource pools that are infrequently used by sites "in the wild" for being purposes) are easier to convert into practical side-channels. Put differently, if sites are already consuming and releasing resources in a resource pool for benign purposes, then other

sites intending to use it as a covert communication channel have to contend with more noise and uncertainty, and thus communication will be more difficult.

We estimate an upper bound on the presence of background noise per tab that could interfere with our "pool-party" attacks by using HTML & JavaScript usage metrics reported by the Chrome Platform Status website [23] to look at how often Web sites use WebSocket, Web Worker, and/or SSE capabilities.

We note that we initially measured background use on the Web through an automated, Web-scale crawl, using browsers instrumented to count how often the Web Workers, WebSockets and Server-Sent Events APIs were used. However, we abandoned this approach on realizing that an automated crawl would potentially *under report* background noise, since in some cases sites would only use these "advanced" browser capabilities on user interaction. This realization lead us to instead look for measurements of browsers under real-world use, and thus to Chrome telemetry.

Results. We report how often the relevant browser APIs are used during real-world browser use (as reported by "Chrome Platform Status") in Table 5. Reported numbers are of August 9, 2022. As noted, no browser feature is used on most websites; one reported feature, SSE, is used on less than 1% of websites, and less than 1% of page loads. Put differently, the vast majority of sites do not use any of these browser features, meaning that in the common case sites could use the resource pool without any interference from other pages.

Notes and qualifications. In practice, sites colluding in a "pool-party" attack would need to contend with the union of all open sites accessing resources from the relevant resource pool. Browser with large numbers of tabs open are therefore more likely to present interference to the two attacking tabs. The numbers resulting from our methodology are a lower bound on how noisy the given resource pool would be, and attackers might need to implement ways of communicating over noisy channels.

Additionally, we note that resources used by a site during the duration of a "pool-party" attack will not effect correctness, only bandwidth. Held resources merely reduce the total limit on a resource pool, but our algorithm can still proceed to send messages. Only resources that go from unconsumed to consumed by a site (or vise versa) during the attack will affect correctness.

Further, we note that the faster an attack completes, the less susceptible the attack is to errors introduced by background noise. This is for two reasons. First, attacks that finish quickly are less likely to be interrupted with benign background resource use (simply because there is less opportunities for background resource use). And second, attacks that finish quickly can engage in simple error correcting techniques to account for possible background noise (for example, conducting the attack multiple times and taking the majority result).

[23] https://chromestatus.com/metrics/feature/popularity

Nonetheless, some percentage of sites are likely to be calling the Web APIs in a more dynamic manner, so it's useful to understand how often these features are used in the wild.

5 Discussion

5.1 Implementation Challenges

We have shown that, a limited-but-unpartitioned resource pool whose resources can be consumed and released by scripts in web pages are sufficient to allow cross-site communication. A few additional anomalies arose, however, in the implementation of our algorithm that would be necessary to consider for the development of a robust message-passing script.

5.2 Drifting Resource Pool Limit

Because the global limit on a resource pool is not a central feature of web browsers, it is possible that developers may overlook certain behaviors of the resource pool. For example, we found that the Firefox global WebSocket pool limit was not strictly fixed. While our script continuously created and destroyed WebSockets to manipulate the number of unheld WebSocket vacancies in the pool, we observed that occasionally the total number of WebSockets that could be created increased. That is: while the initial limit on WebSockets was 512, after a few cycles of the algorithm, the total number of resources that could be held was observed to be 513. That number continued to increase over time. We attribute this behavior to a likely race condition that meant the limit on the total number of allow WebSockets was not consistently enforced, but occasionally a WebSocket slipped through and was not counted.

If the limit changed during a message cycle, then at least one value passed from sender to receiver in our implementation would be incorrect. That would result in an incorrect 35-bit message being recorded by the receiver. To minimize the effects of this anomaly, we ensured that the sender repeatedly attempted to consume more resources than the expected limit, so that even if the limit silently increased during one cycle, the the algorithm would correctly pass the message in subsequent cycles.

Delayed feedback. A second anomaly we observed in the browser-JavaScript implementation of our algorithm was an inconsistent delay in feedback from the resource pool when the pool limit was hit. If a site tries to consume an n-th resource when only $n-1$ resources were available, the attack script must receive an indication that the limit has been reached for the algorithm to succeed.

For example, to consume a resource in the WebSocket pool, it is necessary for the script to call `new WebSocket(...)`. In all cases, whether or not a limit on the WebSocket pool has already been reached, the call returns a WebSocket object. To determine whether the limit has been reached, it's necessary

to find out whether the WebSocket object is in a valid state or not. The state can be ascertained by using the `onerror` callback property on the WebSocket object; however this callback is not fired after a consistent time interval. Instead the time interval varied by as much as tens of milliseconds in some cases. Therefore, it is necessary to introduce a delay before checking for the presence of an error state.

In order to avoid introducing too much delay when attempting to consume n resources, we first create all n resource objects in a tight loop, and then wait for success or failure for all of the resources in parallel.

5.3 Additional Attack Vectors

In this work we demonstrated that "pool-party" attacks are possible and practical in all popular browsers, by exploiting the limited-but-unpartitioned implementations of WebSockets, Web Workers, and Server-Sent Events. However, there are many other "pool-party" attack opportunities in current browsers. This section details some additional APIs and browser capabilities that can be converted into covert-channels though "pool-party" attacks. We do not intend this to be a comprehensive list; we expect that there are many more "pool-party" attack vectors in browsers. We identified the below browser capabilities and APIs as likely exploitable by "pool-party" attacks based on their implementations in Gecko, WebKit and / or Chromium, though we did not build attacks to test the exploit-ability of all listed APIs.

Chromium. Chromium's DNS resolver has a global, unpartitioned limit of 64 simultaneous requests, which could be exploited by an attacker that could control the response time of DNS queries. Second, when Chromium browsers are configured to use an HTTP proxy, they impose a global limit of 32 simultaneous network requests. Third, several APIs in Chromium are thin-wrappers around OS-provided systems, and maintain a single global handle to the system process, effectively creating limited-but-unpartitioned pool of size one (e.g. the Web Speech API).

Gecko. Browsers built on Gecko have many of the same additional attack vectors as Chromium based browsers; Gecko has a global limit on the number of DNS requests that can be in the air at the same time, and a lower limit on the number of requests that can be open when using a HTTP(S) proxy.

WebKit. We identified far fewer additional limited-but-unpartitioned resource pools in WebKit than in other browser engines. We did not identify additional attack vectors in Safari (the most popular WebKit browser) beyond SSEs. However, other, less popular browsers also use WebKit, and some of these introduce additional limited-but-unpartitioned resource pools. For example, the GTK-based version of WebKit[24] uses a DNS resolver with a limit of 8 requests at a time, and a pre-fetch cache with a limit of 64 hosts.

[24] https://trac.webkit.org/wiki/WebKitGTK

5.4 Defenses and Constraints

Defending against "pool-party" attacks in browsers is difficult since, at root, systems will always have limited resources, and thus some underlying limited-but-unpartitioned pool that can be exploited by a sufficiently motivated party. Currently browser resource limitations are mostly explicit and intentional, but even if browsers removed such limits (e.g. limits on WebSocket connections), the underlying system would necessarily have a global limit, either explicitly (e.g. OS imposed limitations on open network sockets) or implicitly (e.g. systems have a finite amount of memory, and so unavoidably can only maintain a finite number of network sockets).

However, even if "pool-party" attacks will fundamentally always be possible, browsers can still take steps to limit how practical such attacks might be.

One approach is to lift browser-imposed limits on resource pools where applicable, and require attackers to contend with much larger system-maintained resource pools. This approach would make attacks much more obvious to the user, who would notice their system slowing down or other applications on the system impacted while the attack was being carried out. Such detectability might deter attackers, at least in the common case. Relying on the OS or system level limits would also make attacks more difficult to carry out. For example, the system network connection pool will be much "noisier" than the browser's WebSocket pool, making it much more difficult to use the resource pool as a reliable covert channel.

Second, browsers could take the opposite approach, and instead of dramatically widening the size of resource pools, browsers could maintain existing resource caps but partition resource pools the same way browsers increasingly partition DOM storage and network state (see Section 2.5). If resource pools were partitioned by site, a site would not learn anything by exhausting its resource pool; the resource pools for other sites would be unaffected. A determined attacker could regain the ability to conduct a "pool-party" attack by controlling a large number of sites, and using them to collectively drain the resource pool. Nevertheless, partitioning resource pools by site would make "pool-party" attacks significantly more difficult for an attacker to carry out.

Third, browsers could combine these two approaches, and simultaneously remove global limits on the size of resource pools, but limit the number of resources each site or context and use. Such a hybrid approach would achieve some of the benefits of both of the above approaches.

5.5 Applicability to Mobile

As part of this work, we checked whether mobile versions of each of browser were also vulnerable to "pool-party" attacks. To do so, we checked that the availability of the WebSockets and SSE attacks on mobile versions of each browser matched the availability of each attack on the desktop version. We

found that the availability of each attack was the same; attacks that worked on the desktop version of a browser also worked on the browser's mobile browser, and attacks that did not work on the desktop version also did not work on mobile.

We did not measure the bandwidth or stability of the identified "pool-party" attack on mobile, both because i. of limited resources, and ii. it being somewhat more difficult to conduct automated measurements on mobile browsers (e.g. most automation tools target desktop versions of browsers, or rely on imperfect simulations of mobile environments). Assessing the practicality of "pool-party" attacks on mobile browsers is an important area for future work.

Identifying Resource Pools. The vulnerabilities discussed in this work were identified through a combination of domain expertise, source code review, and manually interaction with each API in each browser.

We began by generating a list of candidate APIs, based on our experience in both browser and Wbb-app development. Our list of candidate APIs focused on features that i. needed to be handled in parallel (e.g. threads, I/O operations), ii. use limited system resources (e.g. file handles) or iii. are exclusive by nature (e.g. only one voice can speak at a time when using the Web Speech API).

Next, once we had our set of candidate APIs, we examined the source code for API's implementation in each browser engine. We looked for explicit limits on resources, both in the source code and surrounding comments. Often (though not always), limits were relatively easy to find, since they were encoded in constants or runtime flags.

Finally, we manually evaluated whether we could use these browser limits to conduct pool-party attacks by constructing two different test pages on different sites and seeing if we could detect on one site when the relevant resources were being exhausted by the other site.

As noted, this process is entirely manual; it is likely-to-certain that there are more vulnerable resource pools in browsers. Developing a system for systematically or automating the detection of vulnerable resource pools would be a valuable area for future work.

6 Related Work

Online tracking through feature misuse. Our work exists alongside a large body of work on ways browser features can be (mis)used by online trackers, to track their users in ways unintended by the browser vendor.

The largest volume of work in this area is on browser fingerprinting. We highlight significant work in the area, specifically those that identified new fingerprinting vulnerabilities in browsers. Mowery et al. [24] famously demonstrated that differences in how browsers executed drawing (i.e. canvas and WebGL) operations could be used to identify individuals, and Acar et al. [1] showed that differences in what fonts users have installed could be similarly misused. Englehardt et al. [6], as part of a project to measure privacy violations on the 1m most popular websites, show the WebAudio and WebRTC APIs could be used to track users. Olejnik et al. [27] showed the Battery Status API could be used for fingerprinting, and Olejnik and Janc [28] demonstrated the Ambient Light API could be similarly misused. Zhang et al. [42] found that in mobile browsers, websites can use unpermissioned access to motion sensors to identify users, and Starov and Nikiforakis [36] demonstrated that what browser extensions the user has installed can make users more identifiable. Eckersley [5] documented that a screen resolution and display size were practical fingerprinting vectors, and Nikiforakis et al. [26] showed that other display details contributed to identifiability. Laperdrix et al. [20] found that, ironically, the presence of a content blocker could help fingerprinters distinguish users. Iqbal et al. [12] identified dditional browser APIs that fingerprinting methods misuse (e.g. proximity sensor APIs, media capabilities, the Presentation API) by examining the JavaScript source code of known fingerprinting scripts and identifying the additional APIs those scripts abuse.

Distinct from browser fingerprinting, researchers have found other ways of misusing browser features to construct u user identifiers. Solomos et al. [35] transformed the browser's "favicon" cache into a persistent tracking mechanism, Janc et al. [13] showed that Safari's "Intelligent Tracking Prevention"[25] features could be abused to re-identify users, and Syverson and Traudt [37] showed how the browsers "HTTP Strict-Transport-Security" system could be re-purposed to construct and assign unique identifiers.

A parallel body of work attempts to prevent browser fingerprinting. Nikiforakis et al. [25] found browsers could resist fingerprinting by manipulating the values common Web APIs return. Laperdrix et al. [18] extended this approach by introducing small amounts of noise into the Web Audio and Canvas APIs, changes large enough to cause fingerprinters to misidentify users, but small enough that benign uses of features would not be impacted. Snyder et al. [34] suggested disabling Web APIs whose cost to the users (including additional identifiability) was higher than the benefit to them (in terms of desirable page behaviors), based on prior work finding that most browser APIs are rarely used at all [33]. Smith et al. [32] suggest a strategy for rewriting malicious code to prevent fingerprinting scripts from accessing the underlying, identifying values. Other works aim to prevent attackers from abusing features for tracking purposes by removing the entropy added by OS or hardware differences. Wu et al. [41] presents a method for removing hardware-induced differences in WebGL operations, and though not targeting fingerprinting attacks, Andrysco et al. [2] proposed a similar "improve privacy by making dissimilar systems execute similarly" approach for floating-point based channels in browsers.

[25] https://webkit.org/blog/7675/intelligent-tracking-prevention/

Attacks on browser partitioning and sandboxing. Our work also builds on, and exists along side, a large body of work documenting ways browser partitioning efforts can be circumvented, whether those partitions are enforced directly by the application, through OS-based process isolation, or otherwise. Again, the breath of work in this area makes a comprehensive discussion here impossible, so we discuss papers that are particularly significant, novel and/or recent.

Using timing methods to circumvent browser partitioning (in this case the same-origin-policy) dates back at least as far as 2000, when Felten et al. [7] presented a way sites could determine what other sites the user had visited by probing the HTTP cache and measuring timing differences. Bortz et al. [3] published similar foundational work on how sites could exploit timing differences in how, and how quickly, cross-site requests and resources were loaded to to learn about users' state on other sites. Since then, researchers have demonstrated many ways sites can circumvent browser imposed restrictions on what sites can learn about user behavior on other sites, and how sites can communicate with each other.

Schwartz et al. [30] document ways sites can create high resolution timers, with descriptions for how such timers can form covert channels across application boundaries. Smith et al. [31] show that sites can circumvent browsers attempts to partition a users browsing history by exploiting cache state and side effects in painting behaviors. Kohlbrenner and Shacham [17] presented a way of creating a covert channel across site boundaries by exploiting floating-point related timing channels in how browsers render SVGs. Gruss et al. [10] extended the Rowhammer [15] attack, previously used to leak information across OS process boundaries, to be exploitable be through site-included JavaScript code, to violate browser imposed process isolation. Jin et al. [14] show how security-focused protections like isolating sites in their own OS processes can be exploited to learn what sites the user is, or has recently, visited. Lipp et al. [22] demonstrated how sites could puncture site isolation protections and infer what the user was typing on a different site (or different application) by observing timing patterns in JavaScript execution, caused by the OS responding to key presses issued to other contexts. Vila et al. [40] presented a related attack, where a site could transform contention in the browser's main event loop (distinct from the event loop presented to an executing JavaScript context) into a cross-context side channel. van Goethem et al. [39] showed how other browser capabilities like service workers and the (now deprecated) application cache can be transformed into timer-based covert channels as well. As part of a larger project of creating an automated system for detecting cross-site information leaks, Knittel et al. [16] identified how many other browser features that had side effects that could be detected across site boundaries.

The cross-profile tracking attack against Gecko-based browsers described in this work build on other cross-profile attacks such as van Goethem and Joosen [38], which found that application efforts to isolate "incognito" browsing sessions from standard browsing sessions could be circumvented by using contention for disk and memory resources as a covert channel. Oren et al. [29] showed that contention in the CPU cache could be exploited by unprivileged, malicious JavaScript code to learn what sites a user was visiting in an "incognito" mode session. with JavaScript running on other sites, in other processes, or applications outside the browser. Gruss et al. [9] present a similar attack, though instead targeting timing channels stemming from memory deduplication.

Finally, Asankah [11] defines"ephemeral fingerprinting", where sites observe infrequent global events identify a user across contexts.

7 Conclusions

In this work we define a new category of practical privacy attack in popular Web browsers we call "pool-party" attacks. "pool-party" attacks allow sites to break out of the "contextual sandboxes" that browsers try to enforce, and so allow sites to circumvent privacy protections in even the most aggressively privacy-focused browsers. More alarming still, we find that "pool-party" techniques can be used to track users beyond cross-site tracking (specifically, that in Gecko-based browsers "pool-party" attacks can track users *across profiles*).

While some attacks in this category have been known to be *theoretically* possible, this work demonstrates that such attacks are *practical*, and must be dealt with as a real-world threat to the Web users' privacy. Further, we show that "pool-party" attacks can be carried out using a wider range of browser capabilities than previously documented, further emphasizing the severity of the risk to user privacy.

Web privacy has moved in two very different directions over the last two decades. *Privacy attacks* have moved in a dispiriting direction, with privacy violations becoming common place. This disappointing trend is reinforced by a combination of conflicting incentives from (some) browser vendors, backwards compatibility concerns, and user-harming financial incentives. *Privacy defenses* in browsers, though, have been recently moving in an encouraging direction. This is due to (in part) a combination of regulatory pressure, increasing user awareness, the tireless efforts of privacy-focused researchers and developers, and a virtuous competition between (some) browsers to own the "most private browser" title. We hope that this work helps the latter, to the determent of the former.

8 Acknowledgements

We'd like to thank Deian Stefan and Michael Smith from University of California, San Diego for contributing additional examples of limited-but-unpartitioned resource pools. We'd also like to thank Rainer Böhme from University of Innsbruck for refining and improving this work.

References

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. Fpdetective: dusting the web for fingerprinters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1129–1140, 2013.

[2] Marc Andrysco, Andres Nötzli, Fraser Brown, Ranjit Jhala, and Deian Stefan. Towards verified, constant-time floating point operations. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1369–1382, 2018.

[3] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *International Conference on the World Wide Web (WWW)*, pages 621–628, 2007.

[4] Gaurav Aggarwal Elie Burzstein, Collin Jackson, and Dan Boneh. An analysis of private browsing modes in modern browsers. In *USENIX Security Symposium*, 2010.

[5] Peter Eckersley. How unique is your web browser? In *International Symposium on Privacy Enhancing Technologies Symposium (PETS)*, pages 1–18. Springer, 2010.

[6] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1388–1401, 2016.

[7] Edward W Felten and Michael A Schneider. Timing attacks on web privacy. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 25–32, 2000.

[8] Xianyi Gao, Yulong Yang, Huiqing Fu, Janne Lindqvist, and Yang Wang. Private browsing: An inquiry on usability and privacy protection. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 97–106, 2014.

[9] Daniel Gruss, David Bidner, and Stefan Mangard. Practical memory deduplication attacks in sandboxed javascript. In *European Symposium on Research in Computer Security (ESORICS)*, pages 108–122. Springer, 2015.

[10] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer. js: A remote software-induced fault attack in javascript. In *International Conference on Detection of Intrusions, Malware, and Vulnerability Assessment*, pages 300–321. Springer, 2016.

[11] Asanka Herath. Ephemeral fingerprinting on the web. https://github.com/asankah/ephemeral-fingerprinting, 2020.

[12] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *IEEE Symposium on Security and Privacy (SP)*, pages 1143–1161. IEEE, 2021.

[13] Artur Janc, Krzysztof Kotowicz, Lukas Weichselbaum, and Roberto Clapis. Information leaks via safari's intelligent tracking prevention. 2020.

[14] Zihao Jin, Ziqiao Kong, Shuo Chen, and Haixin Duan. Site isolation enables timing-based cross-site browsing surveillance. In *IEEE Symposium on Security and Privacy (SP)*, 2022.

[15] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[16] Lukas Knittel, Christian Mainka, Marcus Niemietz, Dominik Trevor Noß, and Jörg Schwenk. Xsinator.com: From a formal model to the automatic evaluation of cross-site leaks in web browsers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1771–1788, 2021.

[17] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security Symposium*, pages 69–81, 2017.

[18] Pierre Laperdrix, Benoit Baudry, and Vikas Mishra. Fprandom: Randomizing core browser objects to break advanced device fingerprinting techniques. In *International Symposium on Engineering Secure Software and Systems*, pages 97–114. Springer, 2017.

[19] Pierre Laperdrix, Nataliia Bielova, Benoit Baudry, and Gildas Avoine. Browser fingerprinting: A survey. *ACM Transactions on the Web (TWEB)*, 14(2):1–33, 2020.

[20] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy (SP)*, pages 878–894. IEEE, 2016.

[21] Benjamin S Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, pages 57–74. Springer, 2013.

[22] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security (ESORICS)*, pages 191–209. Springer, 2017.

[23] Chao Liu, Ryen W White, and Susan Dumais. Understanding web browsing behaviors through weibull analysis of dwell time. In *ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 379–386, 2010.

[24] Keaton Mowery and Hovav Shacham. Pixel perfect: Fingerprinting canvas in html5. *W2SP*, pages 1–12, 2012.

[25] Nick Nikiforakis, Wouter Joosen, and Benjamin Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *International Conference on the World Wide Web (WWW)*, pages 820–830, 2015.

[26] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy (SP)*, pages 541–555. IEEE, 2013.

[27] Łukasz Olejnik, Gunes Acar, Claude Castelluccia, and Claudia Diaz. The leaking battery. In *Data Privacy Management, and Security Assurance*, pages 254–263. Springer, 2015.

[28] Lukasz Olejnik and A Janc. Stealing sensitive browser data with the w3c ambient light sensor api, 2017.

[29] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1406–1418, 2015.

[30] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *International Conference on Financial Cryptography and Data Security*, pages 247–267. Springer, 2017.

[31] Michael Smith, Craig Disselkoen, Shravan Narayan, Fraser Brown, and Deian Stefan. Browser history re: visited. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[32] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2844–2857, 2021.

[33] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Internet Measurement Conference (IMC)*, pages 97–110, 2016.

[34] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 179–194, 2017.

[35] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[36] Oleksii Starov and Nick Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[37] Paul Syverson and Matthew Traudt. HSTS supports targeted surveillance. In *USENIX Workshop on Free and Open Communications on the Internet (FOCI)*, 2018.

[38] Tom Van Goethem and Wouter Joosen. One side-channel to bring them all and in the darkness bind them: Associating isolated browsing sessions. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[39] Tom Van Goethem, Wouter Joosen, and Nick Nikiforakis. The clock is still ticking: Timing attacks in the modern web. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1382–1393, 2015.

[40] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, pages 849–864, 2017.

[41] Shujiang Wu, Song Li, Yinzhi Cao, and Ningfei Wang. Rendered private: Making GLSL execution uniform to prevent webgl-based browser fingerprinting. In *USENIX Security Symposium*, pages 1645–1660, 2019.

[42] Jiexin Zhang, Alastair R Beresford, and Ian Sheret. Sensorid: Sensor calibration fingerprinting for smartphones. In *IEEE Symposium on Security and Privacy (SP)*, pages 638–655. IEEE, 2019.