Local Frames: Exploiting Inherited Origins to Bypass Content Blockers

Alisha Ukani
University of California, San Diego
San Diego, CA, USA
aukani@ucsd.edu

Hamed Haddadi
Imperial College London & Brave Software Inc
London, UK
h.haddadi@imperial.ac.uk

Alex C. Snoeren
University of California, San Diego
San Diego, CA, USA
snoeren@cs.ucsd.edu

Peter Snyder
Brave Software Inc
San Francisco, CA, USA
pes@brave.com

Abstract

We present a study of how local frames (i.e., iframes loading content like "about:blank") are mishandled by a wide range of popular Web security and privacy tools. As a result, users of these tools remain vulnerable to the very attack techniques against which they seek to protect themselves, including browser fingerprinting, cookie-based tracking, and data exfiltration. The tools we study are vulnerable in different ways, but all share a root cause: legacy Web functionality interacts with browser privacy boundaries in unexpected ways, leading to systemic vulnerabilities in tools developed, maintained, and recommended by privacy experts and activists.

We consider four core capabilities supported by most privacy tools and develop tests to determine whether each can be evaded through the use of local frames. We apply our tests to six popular Web privacy and security tools—identifying at least one vulnerability in each for a total of 19—and extract common patterns regarding their mishandling of local frames. Our measurement of popular websites finds that 56% employ local frames and that 73.7% of the requests made by these local frames should be blocked by popular filter lists but instead trigger the vulnerabilities we identify. From another perspective, 14.3% of all sites that we crawl make requests that should be blocked inside of local frames. We disclosed these vulnerabilities to the tool authors and discuss both our experiences working with them to patch their products and the implications of our findings for other privacy and security research.

CCS Concepts

• **Security and privacy → Privacy protections**.

Keywords

Content blockers; Adblocking; Anti-Adblocking; Filter lists

ACM Reference Format:
Alisha Ukani, Hamed Haddadi, Alex C. Snoeren, and Peter Snyder. 2025. Local Frames: Exploiting Inherited Origins to Bypass Content Blockers. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3744843

CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3744843

1 Introduction

Content-blocking tools are used by millions of people in order to protect their privacy by blocking tracking scripts, stay safe from scammers by blocking malware, save money by using less data, and enjoy a more pleasant browser experience by hiding ads. In order to provide these benefits, most content blockers maintain filter lists, which are lists of URLs that determine which network requests should be blocked or allowed. Modern tools also implement more sophisticated defenses such as resource replacement (e.g., loading benign scripts instead of privacy-invasive ones), scriptlet injection (i.e., employing custom JavaScript code to remove cookies or block trackers), and cosmetic filtering to hide undesirable page elements.

Yet, we find that several popular content blockers are vulnerable to evasion of one or more of these capabilities—and the majority of popular websites are currently doing such evasion. Specifically, our work shows that content blockers frequently mishandle a class of iframes we call *local frames* (an iframe with a non-URL source like "about:blank"), allowing content loaded within local frames to bypass blocker protections. Local frames initially load an empty HTML document, but content can be added to these frames dynamically. Local frames are popular with Web developers for a number of reasons, chief among them the fact that a local frame creates a clean JavaScript environment that can still access the main page. We find that content blockers fail to properly implement protections in local frames, allowing websites to include tracking scripts that should be blocked and show ads that should be hidden—all by wrapping their existing code in a local frame.

Content blockers use a frame's origin—the combination of a URL's protocol (e.g. "https"), hostname, and port—to determine how to handle its content. For example, many content blockers provide the option to block network requests made in third-party contexts; in order to determine if a request is made in a first- or third-party context, the content blocker must know the origin of the request's source and destination. While some tools (e.g., Brave on iOS) simply fail to provide any protection in local frames, we find the most common reason why local frames evade content blockers is that content blockers mis-attribute their origin and, thus, do not associate local frames with the site that creates them. The

```
1  <body>
2    <iframe src="about:blank">
3      <!-- Local frame for firstparty.com. Origin should be
         https://firstparty.com -->
4    </iframe>
5
6    <iframe src="https://thirdparty.com">
7      <iframe src="about:blank">
8        <!-- Local frame for thirdparty.com. Origin
         should be https://thirdparty.com -->
9      </iframe>
10   </iframe>
11 </body>
```

Listing 1: HTML code at `https://firstparty.com` which creates two iframes: a local frame and a third-party iframe that embeds its own local frame.

origin of an iframe is usually extracted from the iframe's source, but according to the HTML specification, local frames are supposed to inherit the origin of the document that creates them [40]. For example, in Listing 1, the local frame on line 2 should inherit the origin of the main page (`https://firstparty.com` in this example), whereas the local frame on line 7 should inherit the origin `https://thirdparty.com`. However, some content blockers do not correctly determine the origin of local frames; some compute the origin of the third-party local frame as "about:blank" while others may even fall back to `https://firstparty.com`.

Local frames are highly prevalent on the Web, and evasion is currently taking place (although we cannot determine if this evasion is intentional). We conduct a measurement study of websites with varying degrees of popularity according to Tranco and find local frames on 56% of websites (12,234 of the 21,965 successfully crawled websites). 73.7% of the URLs requested by these local frames should be blocked by popular content blockers according to a combination of filter lists from EasyList [9], EasyPrivacy [10], and uBlock Origin [49]. Users browsing these sites with vulnerable tools, however, will be exposed to the content anyway. Said another way, 14.3% of the websites we study are evading content blockers by making requests that would otherwise be blocked, but succeed because they are made inside of local frames. We emphasize, however, that we do not know if publishers create the local frames with an explicit intent of evading content blockers.

We consider four distinct capabilities of popular blockers—request blocking, resource replacement, scriptlet injection, and cosmetic filtering—and find vulnerabilities in AdBlock Plus, AdGuard, uBlock Origin Lite, the Brave Browser, DuckDuckGo, and Safari Content Blocking (which is used by many iOS apps):

- We find that websites can employ local frames to completely bypass filter list-based protections in Brave's iOS browser.
- For AdGuard, we find that local frames within third-party iframes inherit the origin of the first-party website. This means that AdGuard users can not only be tracked by third parties, but they may also experience site breakage because rules are applied improperly in local frames.
- Scriptlet injection and cosmetic filtering implemented in Brave's browser, as well as AdGuard's browser extensions and iOS app, can be evaded, allowing websites to perform sophisticated tracking and show ads that should be hidden.
- We find that the cosmetic filtering in uBlock Origin Lite, the AdBlock Plus iOS app, and Safari Content Blocking can be evaded, allowing websites to show ads that should be hidden.
- We find that while DuckDuckGo properly blocks tracking requests made inside local frames, they do not report these actions to the user like they do for regular frames.

We disclosed each of the 19 vulnerabilities we found to the relevant organization. Brave, Safari, AdGuard, and DuckDuckGo have patched their tools, and our disclosure has been acknowledged by uBlock Origin Lite and AdBlock Plus. Our code and data is available at `https://osf.io/9yq57`.

2 Background and Related Work

In this section we provide a brief overview of local frames, their intended use cases, and their relevance to content blocking. We then introduce the four common content-blocking capabilities we study before summarizing related work.

2.1 Local Frames

We use the term *local frame* to refer to an iframe with a non-URL source, the most common of which is "about:blank". This specific URI is often the default URI for new iframes [40] and is intended to be used by browsers for blank pages [41]; the resulting local frames are uninitialized iframes. Local frames (like all iframes) have their own page document [63], which creates a new environment that is unaffected by the JavaScript or CSS rules scoped to the document of the main page. However, as we discuss in the next section, local frames are distinct from other iframes because they can still access the parent document (i.e., they are not fully isolated).

2.1.1 Inherited origins. An origin for a URL is defined as the URL's protocol, port, and hostname (excluding subdomains) [40]. Origins are used to isolate websites from potentially malicious embedded content through the *same-origin policy* feature. This feature prevents a website from accessing the content of other tabs in the user's browser, or reading the cookies set for other websites (which could then be used for a cookie-hijacking attack).

However, the URL-based definition of origins cannot be applied to other URI schemes because they lack a hostname and port. Instead, URIs like "about:blank" should inherit the origin of the document in which they are contained [40]; since this URI inherits the same origin as its parent document, the same-origin policy dictates that a local frame will have access to the parent document. There are other non-standard URIs in addition to "about:blank" for which content blockers may confuse the origin. The "about" prefix is intended to reference an application's internal resources, so other URIs starting with "about" besides "about:blank" (such as "about:srcdoc") could be misinterpreted [41]. There are also several other prefixes, such as "blob", "file", and "data" [63]; in particular, the "data" prefix is supposed to receive an empty security context [40]. However, we find (see Section 3.2.1) that these other URI prefixes make up only 0.5% of local frames created by the websites we study.

2.1.2 Common usage. The lack of isolation between local frames and the main page has made them attractive for serving ads for over 15 years. In particular, local frames have been used to serve "Rich Media" ads, which have features such as displaying videos,

expanding their size, and moving around the page [28, 30]. These features require the ad to have access to the main page, so in 2008 the Interactive Advertising Bureau (IAB) published a set of best practices recommending websites to load ads in "friendly iframes"—a local frame with source "about:self" [31]. The Google Ad Manager documentation also notes that these friendly iframes are better able to collect metrics like viewability compared to cross-domain iframes [27]. Given local frames' power to access the main page, online resources note that websites loading ads in these frames should have a trusted relationship with advertisers [17, 50].

Beyond ads, local frames are also used for tracking purposes. FingerprintJS [24], a popular open-source library for browser fingerprinting, creates a local frame when identifying the fonts installed on a given machine. The FingerprintJS documentation indicates that a local frame is used so that the fingerprinting script does not affect the main page and vice versa.

2.2 Capabilities of Content Blockers

We consider four distinct capabilities of popular content blockers: request blocking, resource replacement, scriptlet injection, and cosmetic filtering.

2.2.1 Request blocking. The basic function of content blockers is to block network requests to undesirable targets such as ads and privacy-invasive tracking scripts. Network requests can either be created in the standard way (i.e., creating an HTML script or image element with the source set to the desired resource), or through asynchronous JavaScript APIs (e.g., `fetch` and `XMLHttpRequest` calls). Sometimes outright blocking these requests can lead to site breakage. So, many content blockers have the option to block these requests only if they originate from a third-party context. Another way to prevent breakage is to create fine-grained entries in the filter list that target specific scripts or paths; in this scenario, content blockers may allow a third-party iframe created by a third-party tracker to load, but block specific content within that iframe. As of August 17, 2024 EasyList contains 2,294 rules that use a third-party modifier to define the context in which these rules apply, and EasyPrivacy contains 4,151 rules with the modifier [9, 10].

2.2.2 Resource replacement. Even if content blockers attempt to block only requests occurring in a third-party context, they can still trigger site breakage. In particular, some websites will expect to see the side effects of their scripts, such as defining certain functions and variables. If a content blocker prevents that script from executing, the website will not see the expected side effects and may throw errors. Alternatively, the website could detect a "bait" resource being blocked (without having to check for side effects) and thus determine that the user is using a content blocker [47]. So, many popular content blockers support redirecting scripts to benign versions that define the expected side effects while avoiding privacy-invasive behavior. For example, uBlock Origin and AdGuard create benign versions of specific scripts that define expected objects (e.g., `window.google.ima` in the case of Google Ad Manager scripts[1]).

2.2.3 Scriptlet injection. Beyond blocking and redirecting requests, some content blockers inject their own JavaScript into loaded webpages to provide more extensive protection from tracking. For example, content blockers cannot modify cookies through blocking network requests. Scriptlets, however, can be used to modify cookies and perform other tasks like disabling access to certain JavaScript APIs. For example, uBlock Origin uses scriptlets to remove telemetry cookies from Bing[2], prevent some websites from saving browser fingerprints[3], and disguise the use of the content blocker on streaming sites like `cbs.com` and `paramountplus.com`[4].

2.2.4 Cosmetic filtering. Finally, sometimes ads cannot be blocked by preventing network requests. Websites may display ads through inline HTML, meaning no network requests are used to render the ads. Alternatively, websites may use network requests to display ads, but include benign, functional code in the same script [6]; if content blockers were to block these scripts, they could break website functionality. Instead, blockers can hide ads through cosmetic filters (also known as "element hiding") that identify unwanted content through HTML and/or CSS selectors.

2.3 Content Blocker Limitations

There is a long-running arms race between website publishers who display ads to generate revenue, and content blockers that attempt to hide these ads and block tracking scripts to improve user privacy and user experience. Many publishers try to detect the use of content blockers and change their websites to discourage the use of these tools, a well-studied practice known as anti-adblocking [34, 44, 68]. Studies disagree on the prevalence of anti-adblocking, reporting rates ranging from 0.7% [44] to 30.5% [68].

Websites also try to evade content blockers by exploiting a widely acknowledged limitation of popular content blockers: the reliance on matching URLs to handcrafted filter lists. In 2016 Wang *et al.* proposed a system to allow Web publishers to evade content blockers by automatically randomizing URLs and HTML attributes [62]. While it is unclear if this particular system is used in practice, websites do change the way they host tracking content for evasion. A 2020 analysis of 10K websites found 1,612 instances of a blocked script being hosted on a new domain, as well as other techniques to change where tracking scripts are hosted [54]. Our work similarly investigates the fragility of content blockers matching URLs against filter lists; however, we find evasion can take place without changing how tracking resources are hosted, by instead wrapping the request (or the requested content) inside a local frame.

Given the brittle nature of filter lists, alternative systems have been proposed to identify tracking resources. The AdGraph system generates a model of websites as graphs and feeds the graph context into an ML model to determine if resources are tracking or non-tracking [35]. In 2021, Chen *et al.* proposed using JavaScript event-loop signatures to identify tracking code [15]. They found 12.5% of websites were able to evade filter lists by including scripts that contain tracking behavior but were not previously included in filter

[1] https://github.com/gorhill/uBlock/blob/2c60b331e39e96114386e568d028240b37cdefc/src/web_accessible_resources/google-ima.js#L854

[2] https://github.com/uBlockOrigin/uAssets/blob/e19390098b75344eaeaefcf33d599b840b75663d/filters/privacy.txt#L649

[3] https://github.com/uBlockOrigin/uAssets/blob/b38c34e9ff6f6467144954293100b166e36033e8/filters/filters-2024.txt#L423

[4] https://github.com/uBlockOrigin/uAssets/blob/b38c34e9ff6f6467144954293100b166e36033e8/filters/filters-2024.txt#L92

lists. However, a system like the one Chen *et al.* propose would not solve the issues we identify with local frames, as we find tools fail to apply filter list rules to local frames, thus creating a vector for evasion.

Finally, content blockers implemented as browser extensions inherit the limitations of the browsers upon which they rely. In 2016, Bashir *et al.* measured the impact of a known bug in the Chrome WebSocket API that allowed websites to evade content blockers [11]. They found 2% of websites used the WebSocket API, and 60% of those sites were opening WebSockets to advertising and analytics companies.

3 Local Frame Usage

We motivate our study of vulnerable tools with a measurement of how frequently local frames are used on the Web currently. We also measure how often websites use local frames to carry out the kinds of behaviors that content-blocking tools target. We note, however, that our measurement is not able to infer the intent of website authors. Our results only show how often websites use local frames in ways that may circumvent existing privacy tools; we are not suggesting that website authors are using local frames with the sole intent of circumventing privacy tools.

We find that local frames are widely used on the Web, appearing on more than half (56%) of all websites we study. Furthermore, websites frequently use local frames to fingerprint users and make requests to URLs defined to be privacy harming (or otherwise unwanted) by popular blocklists—behaviors that privacy and content-filtering Web tools target and modify.

3.1 Measurement Methodology

We measure how, and how often, local frames are used on popular websites in multiple steps.

3.1.1 Website selection. We use the Tranco top-sites list to select websites with varying degrees of popularity at time of measurement. We collect data for 21,965 sites in total, including 11,965 of the Tranco top 15K, 5,000 websites uniformly sampled between the ranks of 15K to 100K, and 5,000 websites between ranks 100K and 1M. Our crawl attempts fail on 23% of websites, either because the domains are not used to serve websites (e.g., CDN domains used to serve page assets), the domains are geo-restricted, or, in a small number of cases, because of compatibility issues with our crawling tools. Hence, successfully crawling 5K sites required trying 6,362 sites between 15K–100K and 7,100 among the ranks 100K–1M.

3.1.2 Tools and vantage points. We visit each of the selected sites using the PageGraph crawler [12], a Web-measurement tool based on a current fork of Chromium. The crawler records 1) salient events that occur during the loading, rendering, and executing of a website (e.g., network requests issued; HTML elements created, modified, or inserted in the page; WebAPIs executed by JavaScript, etc.), and 2) attribution information for each of these events (e.g., the HTML element or JavaScript code unit responsible for a network request, whether an HTML element was created as a result of parsing an HTML text or by a script, which script called which WebAPI, etc.). The crawler outputs this log of events and actors into a graph, yielding one graph per measured website.

Rank Interval	Sites Crawled	Local Frame Prevalence		
		1p	3p	Either
[1–5K)	3,815	52.9%	20.0%	56.3%
[5K–10K)	4,239	54.8%	25.2%	59.8%
[10K–15K)	3,911	47.6%	19.9%	51.0%
[15K–100K)	5,000	58.3%	23.6%	60.8%
[100K–1M)	5,000	46.8%	19.8%	50.3%
Overall	21,965	52.2%	21.7%	55.7%

Table 1: Prevalence of local frames at various Tranco ranks.

We conduct our crawl from AWS EC2 servers in Amazon's `us-west-2` region. We configure the crawler to appear identical to a typical Chromium-based browser by running the browser in "head-full" mode, removing JavaScript properties that indicate to a website that a browser is a crawler (e.g., `window.webdriver`), and taking additional, similar steps to avoid signals that prior research has identified can influence Web measurement results [36]. For each website, the crawler visits the root page for each domain (i.e., either `https://domain.example` or `http://domain.example`), waits 30 seconds, and then records the resulting event graph.

3.2 Results

We post-process our crawl data to extract the first-party and third-party local frames from each website and analyze their behaviors.

3.2.1 Local-frame prevalence. In each website graph, we identify frames that are local during the entire page execution. This is an important filter because internally Chromium initializes all child frames (e.g., `<iframe>`, `<frame>`, `<object>`) as a local frame, even if that frame is then navigated to another URL. We find that "about:blank" is by far the most common local-frame URI, accounting for 95.8% of all local frames in our dataset; "about:srcdoc" (the only other URI with the "about" prefix we see) accounts for 3.7% of local frames, followed by "blob" at 0.4% and "data" at 0.1%. The prevalence of each prefix is consistent across website ranks. Results reported in the remainder of this section pertain only to "about:blank" and "about:srcdoc" local frames.

We find 74,263 local frames on 12,234 distinct sites, and, as shown in Table 1, there is no obvious trend in local-frame prevalence across website popularity. The vast majority of websites using local frames contain at least one first-party local frame; third-party local frames are significantly less common at all popularity ranks.

3.2.2 Privacy-relevant events. From the local frames we discover, we extract instances of the following behaviors to understand how often local frames are used for the kinds of activities that are targeted by privacy tools:

(1) Fingerprinting-related API calls including canvas, navigator, screen, and WebGL (the full list is in Appendix A),
(2) Requests made within a local frame (e.g., images requested, `fetch` and `XMLHttpRequest` calls, etc.),
(3) Calls to privacy-relevant WebAPIs and JavaScript built-ins (e.g., calls to `performance.now()`, Canvas APIs, etc.), and

	1p	3p	Fingerprinting API Calls	Requests	JS/API Calls	HTML
# Sites	7,942	4,776	4,280	5,168	6,596	4,629
Mean	2.07	1.31	1,900.97	4.38	5,257.19	32.52
Median	1	0	0	0	0	0
Max	98	76	327,600	431	636,724	7,432
Total	45,451	28,812	40,220,690	96,262	111,231,650	733,813

Table 2: Statistics regarding the prevalence of various privacy-relevant behaviors occurring inside of local frames created by the 21,965 crawled websites.

	Rank [1–15K)		Rank [15K–100K)		Rank [100K–1M)		Total	
# Requests in dataset	1,377,219	100.0%	579,966	100.0%	461,240	100.0%	2,418,425	100.0%
⌞ in a local frame	56,280	4.1%	26,884	4.6%	13,098	2.8%	96,262	4.0%
⌞ that should be blocked	42,111	74.8%	19,679	73.2%	9,148	69.8%	70,938	73.7%
# Sites crawled	11,965	100.0%	5,000	100.0%	5,000	100.0%	21,965	100.0%
⌞ making >= 1 request	10,863	90.8%	4,962	99.2%	4,971	99.4%	20,796	94.7%
⌞ in a local frame	2,833	26.1%	1,306	26.3%	939	18.9%	5,168	24.4%
⌞ that should be blocked	1,887	66.6%	778	59.6%	477	50.8%	3,142	61.9%

Table 3: Requests that sites make in local frames, including requests that popular Web security and privacy tools intend to block, but which are not blocked in some tools because of mishandling of frame origins.

Rank [1–15K)			Rank [15K–100K)			Rank [100K–1M)		
Entity	# Sites	# Frames	Entity	# Sites	# Frames	Entity	# Sites	# Frames
Google	1903	6369	Google	720	2531	Google	679	2541
PubMatic	673	4314	adtrafficquality.google	556	691	adtrafficquality.google	374	398
Unity	232	351	PubMatic	199	1049	Cloudflare	85	2124
Cloudflare	213	1058	Cloudflare	108	2453	PubMatic	78	413
Amazon	172	303	SeedTag	42	219	Amadeus	19	20
Vidoomy	52	113	AdYouLike	32	37	SeedTag	12	79
Datadome	42	51	admatic.de	20	47	Jivox	8	16
NextMillennium	36	89	Amadeus	16	19	Yandex	8	74
ConnectAdRealtime	35	83	Amazon	15	21	Chaturbate	8	26
Piano	33	136	ConnectAdRealtime	12	43	AdYouLike	6	6

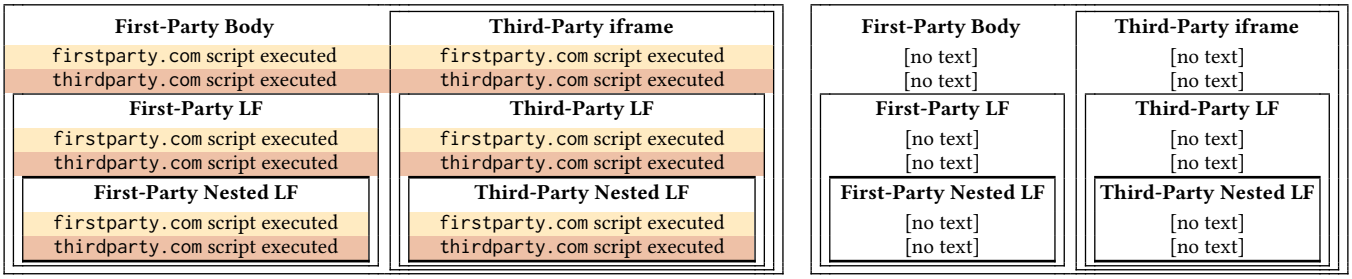
Table 4: The top-10 entities of content loaded into third-party local frames sorted by the number of sites on which they appear.

(4) Non-default HTML elements included (i.e., all HTML elements inserted into the frame except those created automatically such as <html>, <head>, <body>, etc.).

Table 2 shows that many sites use local frames to perform fingerprinting, make network requests, execute JavaScript, and present HTML elements to users. Approximately a third (4,280) of all websites containing local frames use them to perform fingerprinting, a clear privacy concern. Users may wish to manage the remaining behaviors with privacy tools as well, and these tools may be failing users by mishandling local frames.

3.2.3 Privacy-suspect events. One particularly suspicious class of events are requests made by local frames that would be blocked by popular filter lists. We extract the requests made from local frames and use the adblock-rs library [7] to check them against EasyList [9], EasyPrivacy [10], and the additional lists maintained by the uBlock Origin project [49]. Table 3 presents the results of this analysis. We find that in a significant number of cases, sites use local frames to conduct specific behaviors that are very likely to be targeted for blocking or modification by privacy tools. We observe, for example, that 61.9% of sites making requests inside of local frames attempt to request content that should be blocked by popular filter lists. We find that the rate of privacy-suspect requests seems to increase slightly with popularity, which is consistent with prior work that reports popular websites perform more tracking [23, 33].

3.2.4 Third-party local frames. Third-party local frames are particularly intriguing, as they likely are not under direct control of the website publisher—who may not even be aware of their use. We find 28,812 third-party local frames hosted on 4,776 unique websites. We map the security origin of the third-party local frame (i.e., the origin of the content loaded into the local frame) to its owning organization using the Disconnect entity list [32]; if there is no entity for a given URL, we use the URL as the entity. For each



(a) Structure of the test website when no filter list rules are applied, so all requests succeed.

(b) Expected behavior with filter list rules to block all requests to the resources.

Figure 1: Structure of the test website for blocking requests and the expected behavior for RQ1. Cells highlighted in yellow indicate successful script requests from `firstparty.com` and cells highlighted in red indicate successful script requests from `thirdparty.com`.

popularity rank we report the top-10 entities that create third-party local frames by the number of sites in Table 4 (see Appendix B for the corresponding eTLD+1s). Almost all entities are advertising and analytics companies. Unsurprisingly, large analytic companies like Google, PubMatic, and Cloudflare appear in the top four entities for each popularity rank. After that the entities become more varied, though the medium and unpopular ranks have high overlap.

4 Vulnerable Content Blockers

In this section, we show that websites can exploit local frames to bypass the intended behavior of popular content blockers. We start by designing tests to determine whether each capability is executed correctly; we say a capability can be bypassed if a rule that is intended to be applied to a local frame is not applied. We also check if a capability is misapplied by checking if a rule that is not intended to be applied to a local frame is applied. We then discuss the tools we choose to study, and finally present our results. We find five major products (the Brave Browser, Safari Content Blocking, uBlock Origin Lite, AdBlock Plus, and AdGuard) have vulnerabilities that expose users to privacy-invasive tracking and visible ads, both of which are intended to be blocked. One additional tool—DuckDuckGo—does not expose users but contains a vulnerability in its accounting functionality.

4.1 Designing Tests for Capabilities

For each capability we design test pages to determine if local frames can evade it. Each test uses two websites, one first-party and one third-party (referred to as `firstparty.com` and `thirdparty.com`), and we check whether the capability can be evaded in a first-party local frame and/or a third-party local frame. For ease of implementation, our tests employ "about:blank" local frames; it is possible that more extensive testing with other local-frame URIs might uncover additional vulnerabilities. Our tests are not meant to be comprehensive for all functionality provided by content blockers, nor comprehensive of all possible code paths that implement each capability. However, even our limited set of tests reveal mishandling of local frames in every tool we study. Our test pages, and the filter list rules used for each tool, are publicly available at `https://osf.io/9yq57/files`.

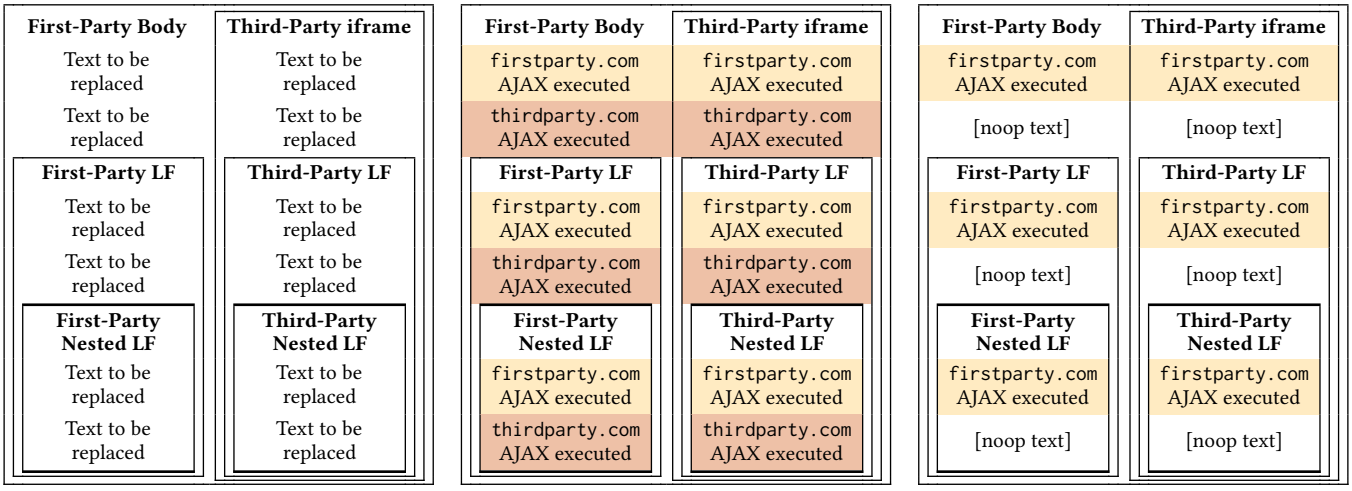
```
<body>
  <!-- Main frame for firstparty.com -->
  <iframe src="about:blank">
    <!-- Local frame for firstparty.com -->
    <iframe src="about:blank">
      <!-- Nested local frame for firstparty.com -->
    </iframe>
  </iframe>

  <iframe src="https://thirdparty.com">
    <!-- Main frame for thirdparty.com -->
    <iframe src="about:blank">
      <!-- Local frame for thirdparty.com -->
      <iframe src="about:blank">
       <!-- Nested local frame for thirdparty.com -->
      </iframe>
    </iframe>
  </iframe>
</body>
```

Listing 2: Structure of a website `firstparty.com` with a first-party local frame, a nested first-party local frame, a third-party local frame, and a nested third-party local frame.

4.1.1 Request blocking. For this test, we create a test website `firstparty.com` following the structure in Listing 2, where all six (local and non-local) frames request two JavaScript files: one from `firstparty.com` and one from `thirdparty.com`. Both scripts create a text element inside the frames in which they are requested; the script from `firstparty.com` adds the text "firstparty.com script executed" while the script from `thirdparty.com` adds the text "thirdparty.com script executed". A representation of the resulting page (without any filter list rules applied) is shown in Figure 1a.

Concretely, we test **RQ1**: If a resource is supposed to be blocked, can it be loaded inside a local frame? If a content-blocking tool blocks both scripts from being requested by any domain, then neither script should get executed in any frame, and thus no text should be added to the website; this expected behavior is shown in Figure 1b. We consider more nuanced variants (i.e., if the script executes in only first- or third-party contexts) in Appendix C, but our tests reveal that tools either handle all cases correctly or none.

4.1.2 Resource replacement. There are many resources that popular content blockers can redirect; we focus our testing on AJAX requests as a representative example. (We suspect the content-blocker code paths that compute request origins are likely the same

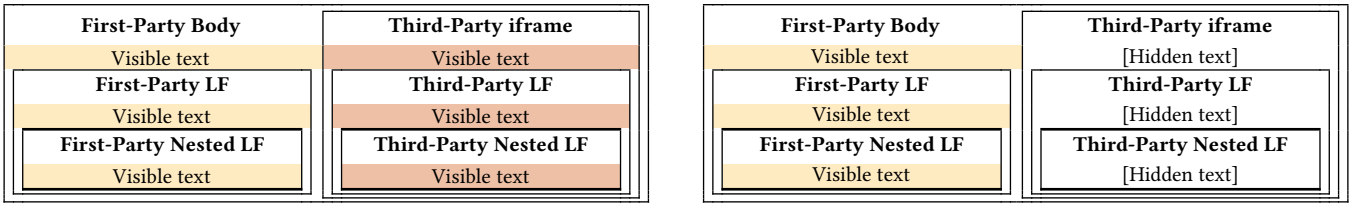


(a) Timestamp 0: Structure of the test website before any AJAX requests have been executed, regardless of the presence of filter lists.

(b) Timestamp 1a: Structure of the test website *without* filter list rules after all AJAX requests have been executed.

(c) Timestamp 1b: Structure of the test website *with* filter list rules redirecting AJAX requests for `thirdparty.com`.

Figure 2: Structure of the test website for resource replacement (a) before and (b) after AJAX request execution, and (c) the expected behavior of our test for RQ2. Cells highlighted in yellow indicate successful AJAX requests from `firstparty.com`, and cells highlighted in red indicate successful AJAX requests from `thirdparty.com`.



(a) No filter list rules applied.

(b) Expected behavior for hiding elements on `thirdparty.com`.

Figure 3: Representation of (a) our test website for cosmetic filtering and (b) the expected behavior for RQ4. Cells highlighted in yellow indicate elements created by `firstparty.com`, and cells highlighted in red indicate elements from `thirdparty.com`.

no matter which type of resource is redirected.) AJAX requests are asynchronous `XMLHttpRequests` made to another webserver. On our webpage, we include two `h1` (header) elements in every frame. Each frame also includes JavaScript code that replaces the first `h1` element with the contents of a text file retrieved from `firstparty.com` (which reads "firstparty.com AJAX executed"), and replaces the second `h1` element with the contents of a text file from `thirdparty.com` ("thirdparty.com AJAX executed").

We show a representation of our test website without any AJAX requests made in Figure 2a, and with all AJAX requests completed in Figure 2b. In our test, we consider **RQ2**: If we define a filter list rule to redirect one of the text files fetched in the AJAX request to an empty file, does the original AJAX request still occur? In Figure 2c, we show an example of the expected behavior for redirecting requests to the text file hosted on `thirdparty.com`. We also test redirecting requests for `firstparty.com` (not shown).

4.1.3 Scriptlet injection. To test if scriptlet injection can be evaded by local frames, we create two filter-list rules, each employing a distinct "set-constant" scriptlet. The "set-constant" scriptlets define a new property, `scriptletvalue`, of the `window` object for a given domain. Both of the scriptlets set values for the `window.scriptletvalue` object, but to different constants: 1 for first-party and 42 for third-party websites. By using different values, we can check if a third-party local frame is confused for a first-party frame or vice versa. Concretely, we evaluate **RQ3**: For each local frame, is the correct scriptlet injected? In our test website, each frame creates a text element on the page reporting the value of `window.scriptletvalue`.

4.1.4 Cosmetic filtering. Our test for cosmetic filtering checks if elements can be selected and hidden properly. In our test website we create an `h1` element with the class `cosmetic-filter` in every frame, as shown in Figure 3a. We then test **RQ4**: If we define a rule to hide `h1` elements with the `cosmetic-filter` class, are they hidden? Figure 3b shows the expected behavior for hiding these elements on `thirdparty.com`; we also test the first-party website.

4.2 Tools Studied

Users seeking to protect their privacy typically install browser extensions that can intercept requests to and from trackers, or use browsers with strong privacy protections. We test popular browsers

and browser extensions from six different organizations. We focus on tools that employ filter lists because their behavior is easy to manipulate—i.e., by modifying the contents of the filter list—but we expect the unintuitive nature of local frames is equally likely to confound developers of any tool that needs to determine the party responsible for the content and behavior of a frame, regardless of the tool's blocking strategy.

4.2.1 Extensions. For browser extensions, we consider the most-popular Chrome and Firefox extensions. Firefox recommends certain add-ons in the "Privacy & Security" category [42]. We select the top three (by download count) that rely on URL-based filter list rules: uBlock Origin, AdGuard, and DuckDuckGo. We do not test the two other tools in the top five; we exclude Ghostery because it blocks at the granularity of URL parameters and PrivacyBadger because it uses heuristics to determine what to block instead of a fixed set of filter list rules. The same three extensions appear in the top five in the "Privacy and Security" category of the Chrome Web Store [59] using the default sorting method of "Most relevant". The other two tools in the top five are OnlineSecurity, a malware-defense system and Authenticator, a multifactor-authentication app. We also test AdBlockPlus, which, while not recommended by Firefox, has over 3 million users—second only to uBlock Origin. (The Chrome Web Store categorizes AdBlock Plus as a "Workflow & Planning" extension; it does not appear in "Privacy and Security".) We test these four extensions on both Firefox and Chome, as well as their iOS, Android, and Desktop apps where available.

We also checked the list of most-popular extensions in the Firefox Public Data Report, which reports metrics from Firefox desktop users each week [43]. As of December 30, 2024, this report had three of the content blockers we study in the top-10 most-popular Add-ons, as well as one content blocker we do not study. The report lists uBlock Origin as the most popular extension, with a usage of 8.49%. The next-most-popular content blocker (and the 4th-most-popular Add-on) was AdBlocker Ultimate, which primarily uses filter lists based on EasyList/Easyprivacy and AdGuard. The next-most-popular content blockers (ranked 7th and 8th respectively) are AdBlockPlus and DuckDuckGo. (PrivacyBadger is ranked 9th overall, but as noted, uses heuristics instead of a filter list.)

We run browser extensions on Firefox version 129.0.1 or Chromium version 126.0.6478.182. We test Android apps on an emulated Pixel 8 Pro phone running API 35, and iOS apps on an iPhone 15 Pro running iOS version 17.5.1.

uBlock Origin is a widely used browser extension, with over 35 million downloads from the Chrome Web Store [60]. uBlock Origin supports all four capabilities and offers both a Firefox and Chrome extension (we test version 1.59.0 of both).

Despite uBlock Origin's popularity, its Chrome extension is built with the Chrome Manifest V2 (MV2) framework, which Chrome has deprecated as of June 2025 [39]; the details of MV2 or its successor, Manifest V3 (MV3), are not important for understanding this work, so we omit them. Though uBlock Origin is no longer supported due to MV2 deprecation, the maintainers now offer the uBlock Origin Lite extension, which is built with the new MV3 framework; we test version 2024.8.12.902. As of January 2025, uBlock Origin Lite had 1 million users on the Chrome Web Store [61].

```
<body>
  <iframe src="about:blank">
    <!-- Local frame for firstparty.com. Origin should be
     firstparty.com -->
    <script src="http://firstparty.com/should_be_allowed.
    js"></script>
    <script src="http://thirdparty.com/should_be_blocked.
    js"></script>
  </iframe>

  <iframe src="https://intermediate.com">
    <iframe src="about:blank">
      <!-- Local frame for intermediate.com. Origin
     should be intermediate.com -->
      <script src="http://firstparty.com/
     should_be_allowed.js"></script>
      <script src="http://thirdparty.com/
     should_be_blocked.js"></script>
    </iframe>
  </iframe>
</body>
```

Listing 3: Structure of a website `firstparty.com` that embeds `intermediate.com` instead of directly embedding `thirdparty.com`. For clarity, we do not show nested local frames or requests made outside of local frames.

AdGuard is another popular browser extension with over 14 million downloads from the Chrome Web Store as of January 2025 [58]. We test the AdGuard Chrome extension (version 4.3.53) and Firefox extension (version 4.3.64), both of which offer all four capabilities. AdGuard also offers an iOS app that changes the user's browsing experience in the Safari mobile browser. At the time of writing, AdGuard claims to offer all capabilities except resource replacement for their iOS app [3]. However, we are unable to get scriptlet injection rules working in the iOS app, so we only test request blocking and cosmetic filters.

DuckDuckGo is primarily known as a privacy-protecting search engine, but also offers browser extensions that block requests to trackers, as well as a standalone browser. All of the extensions and browser versions offer two of the capabilities outlined in Section 2.2: request blocking and resource replacement. In particular, the latter is implemented through "surrogate" rules, which redirect specific JavaScript files into benign or no-op versions. We test the Chrome and Firefox extensions (versions 2024.7.10 and 2024.7.24 respectively) as well as the MacOS browser (version 1.101.0), iOS browser (version 7.134.0.0), and Android browser (version 5.210.2).

One challenge we encounter is that we cannot properly test RQ1 using the setup in Listing 2. DuckDuckGo does not allow users to define custom filter list rules, so (as we explain later in Section 4.3) we spoof the DNS response for `thirdparty.com` to match a common tracker. However, DuckDuckGo then blocks the entire third-party iframe for matching a known tracker, and so we cannot test the behavior of requests made inside the third-party iframe. As a workaround, we create another website, `intermediate.com`, that makes requests to `thirdparty.com` and change the source of the third-party iframe on `firstparty.com` from `thirdparty.com` to `intermediate.com`; a simplified version of the resulting page is shown in Listing 3. This approach enables us to make third-party requests to a resource that should be blocked or redirected.

AdBlock Plus is one of the most-popular privacy-focused browser extensions, with over 41 million downloads from the

Chrome Web Store as of January 2025 [57]. AdBlock Plus supports all four of the capabilities outlined in Section 2.2. We test the AdBlock Plus Chrome extension (version 4.10.1), Firefox extension (version 4.5), and iOS app (version 2.2.16).

4.2.2 Browsers and APIs. Among privacy-focused browsers, we test the Brave Browser as it has 75.9 million monthly active users as of November 2024 [55], and the Safari Content Blocker API because it powers the iOS filtering system for at least two of the tools we study (AdGuard and Brave), and its API may be relied upon by the billions of people who use Safari [26].

Safari Content Blocker. Apple allows developers to create content-blocking extensions that modify Safari on macOS and iOS [8]. Safari Content Blockers support only two of the capabilities in Section 2.2: request blocking (with third-party modifiers) and cosmetic filtering. To test this native content-blocking functionality, we implement our own content-blocking extension and test it on Safari (version 17.5) on MacOS 14.5; the code for these extensions is publicly available at `https://osf.io/9yq57/files`.

A Safari Content Blocker extension can only support 150,000 rules [48]; while this number may seem large, the combined EasyList and EasyPrivacy lists (as of August 17, 2024) have 123,931 rules [9, 10], and so any iOS apps wishing to include their own rules in addition to these standard lists are running out of space. This limitation impacts any iOS apps that implement request blocking and cosmetic filtering through the Safari content-blocking system, including the AdGuard and Brave iOS apps (see Section 6).

Brave Browser. Brave is a browser with many privacy-enhancing features and supports all of the four capabilities previously outlined in Section 2.2. We test the Brave MacOS app (version 1.68.141), iOS app (version 1.68), and Android app (version 1.68.137).

Through source-code analysis, we discover that there are two separate code paths for request blocking. Based on this discovery, we create two versions of our request-blocking test setup. The first version is exactly as described in Section 4.1.1. The second version makes AJAX requests instead of directly including the resources. While similar to our resource-replacement test, it seeks to test whether filter-list rules can block these requests, not (just) redirect them. In this modified version of the test we rename the requested resource to match one of Brave's existing filter-list rules: we request `thirdparty.com/ads/index.js` to match the existing rule for `/ads/index`.

4.3 Invoking Tests

Our tests, as described in Section 4.1, require custom filter-list rules to be added to content blockers. However, not all content blockers allow users to add these custom rules. Concretely, we cannot add custom filter list rules to the AdBlock Plus iOS app (which loads EasyList rules [1]) and all DuckDuckGo browsers and browser extensions (which load a custom, publicly available set of blocklists [18–21]). For these tools, we instead modify our tests to match the tools' existing filter list rules. In order to match existing rules, we spoof DNS responses to map the hosts in those rules to our test websites. For browsers and browser extensions (i.e. DuckDuckGo), we modify our local `/etc/hosts` file. For iOS apps (i.e. AdBlock Plus and DuckDuckGo), we set up an HTTP proxy with the Charles Web-proxy tool [14].

Specifically, for request-blocking tests, we map our third-party server to `doubleclick.net`, since this domain is blocked by all of the aforementioned filter lists. To test resource redirection for DuckDuckGo, we replace our AJAX requests with standard Web requests to a particular script (`npttech.com/advertising.js`) that DuckDuckGo redirects to an empty script on all domains. Finally, to test cosmetic filters in the AdBlock Plus iOS tool, we change the HTML class of the `h1` elements from `cosmetic-filter` to `ADBAR`.

In total, we create seven unique tests; one for each research question, as well as two variants of RQ1 (described in Appendix C) and the tool-specific tests for Brave's alternative request-blocking code path and DuckDuckGo.

4.4 Results

In this section we discuss the results of our testing; Table 5 presents a summary of the vulnerabilities we discover. In some instances our initial findings led us to analyze the tools' source code.

4.4.1 AdBlock Plus. We find that the Chrome and Firefox extensions of AdBlock Plus are not deterministically vulnerable to any of our tests. However, we discover that there is a race condition in the Chrome extension where local frames can load before the extension has an opportunity to inject scriptlets into them. (Fortunately, when a scriptlet wins the race and is injected, we find that the correct scriptlet is used and the local frames display the expected behavior.)

We also find that the AdBlock Plus iOS app does not inject cosmetic filters into local frames, which allows websites to show ads and unwanted content despite AdBlock Plus intending to block this content. It is possible that this is because AdBlock Plus implements cosmetic filtering through the Safari content blocking tool (see Section 4.4.6), but we are unable to verify this as the source code to AdBlock Plus is not publicly accessible.

4.4.2 uBlock Origin. Like AdBlock Plus, the uBlock Origin Chrome extension inconsistently applies scriptlets. When scriptlets are injected, we find that local frames display the expected behavior.

uBlock Origin Lite, on the other hand, does not inject any cosmetic filters into local frames. This allows websites to display ads that should otherwise be hidden, simply by putting them into a local frame. The maintainers do not plan to fix this issue for now, as they believe that patching the issue could incur high performance overhead [29]. However, during our disclosure process, the maintainers of uBlock Origin Lite discovered that scriptlets were also not being injected into local frames and patched the issue.

4.4.3 AdGuard. Across all tested platforms, we find that AdGuard incorrectly computes the origin of third-party local frames for scriptlet injection and cosmetic filters. Specifically, AdGuard determines the origin of these frames to be the origin of the first-party website. In our tests, we find that scriptlet and cosmetic rules intended for the first-party website are applied to third-party local frames. Conversely, the scriptlet and cosmetic rules intended for the third-party website are not applied to third-party local frames. Because of this issue, third-party content can easily evade scriptlet injection and cosmetic filtering. As described in Section 2.2, scriptlets and cosmetic filters are used to block tracking scripts, disguise the use of content blocking tools, and hide ads. AdGuard users are subject to all of these consequences for third-party content.

Tool	Platform	Request Blocking (RQ1)	Resource Replacement (RQ2)	Scriptlet Injection (RQ3)	Cosmetic Filters (RQ4)
AdBlock Plus	Chrome Extension	○	○	◐	○
	Firefox Extension	○	○	○	○
	iOS	○	N/A	N/A	●
uBlock Origin	Chrome Extension	○	○	◐	○
	Firefox Extension	○	○	○	○
	Chrome MV3 (uBlock Origin Lite)	○	○	○*	●
AdGuard	Chrome Extension	○	○	●	●
	Firefox Extension	○	○	●	●
	iOS	○	N/A	N/A	●
Brave Browser	Desktop	○	○	●	●
	iOS	●	●	●	●
	Android	○	○	●	●
DuckDuckGo	Chrome Extension	○	○	N/A	N/A
	Firefox Extension	○	○	N/A	N/A
	Desktop	●†	○	N/A	N/A
	iOS	○	○	N/A	N/A
	Android	○	○	N/A	N/A
Safari Content Blocker	MacOS	○	N/A	N/A	●

Table 5: Results showing which tools can be evaded for each capability. ● indicates the tool is vulnerable for that capability, ○ indicates the tool is not vulnerable for that capability and ◐ indicates that the capability is inconsistently effective. *uBlock Origin Lite was vulnerable before commit 520f81f; the issue was patched during the disclosure process for other uBlock Origin issues. †We find DuckDuckGo's request blocking cannot be evaded, but websites can evade the privacy harms of their website being reported to users.

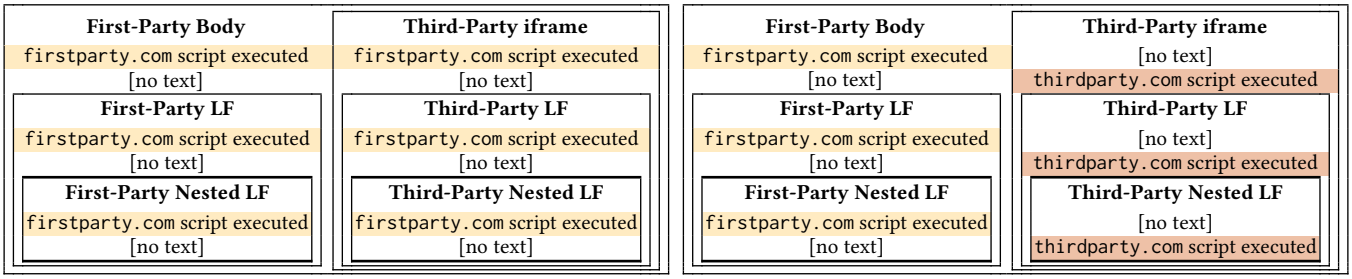
Another consequence of mis-attributing the origin of local frames is that AdGuard can introduce site breakage by applying the rules for the first-party website to the third-party local frames. Scriptlets that modify the JavaScript API may have unintended consequences. For example, AdGuard's scriptlets can prevent websites from making network requests via the `fetch` API, or prevent event listeners from being added to elements, both of which are standard tools for Web development [2]. Disabling these features could break the behavior of third-party content.

4.4.4 Brave. We find some of Brave's tools to be vulnerable to evasion of all four capabilities outlined in Section 2.2. Across all tested platforms, scriptlets and cosmetic filters are not injected into local frames. The Brave iOS app can be evaded in another way, as resource replacement rules are not injected inside local frames. We find that AJAX requests made inside local frames are not redirected to empty text responses. While we only test AJAX requests, resource replacement is often used to redirect privacy-invasive scripts to benign versions. Users of Brave's iOS app can be subject to these invasive tracking scripts if websites simply make these requests from inside local frames.

More critically, we find that Brave's iOS app is vulnerable to evasion in request blocking. As described above, Brave implements two different code paths for request blocking, and we design a version of the request blocking test that makes AJAX requests and matches Brave's default filter list rules (i.e. not added by the user). This version of our test shows that local frames are able to make requests that should otherwise be blocked. The reason that Brave has two different code paths is because of the aforementioned limitations on the number of rules that can be loaded in a Safari Content Blocking extension. Brave compiles a subset of EasyList and EasyPrivacy, called the "slim list", that are loaded into the Safari Content Blocking extension [56]. However, Brave still attempts to block a larger set of requests than this subset. They cannot block any more standard Web requests because iOS restricts access to these requests [8]. But apps still have access to asynchronous requests made through JavaScript APIs, such as AJAX requests and the `fetch` API. Brave attempts to block these asynchronous requests, but incorrectly computes the origin of the request originator[5]. This mistake allows local frames to make requests to trackers (that are not in the aforementioned slim list).

4.4.5 DuckDuckGo. We find that DuckDuckGo correctly blocks requests and redirects resources due to its block-by-default behavior; requests to trackers are blocked unless an exception is defined in an allowlist. However, source-code analysis reveals a minor bug that demonstrates how common it is for security/privacy tools to mishandle local frames: The bug causes DuckDuckGo to mis-report how many trackers have been blocked from a given website. This error may mislead DuckDuckGo users attempting to understand the privacy-harming behaviors of the websites they visit. Specifically, we find that requests blocked inside nested local frames are not properly accounted (see Appendix C).

[5]https://github.com/brave/brave-core/blob/094608dbd95704ae314acbb9e05080c566afa5ad/ios/brave-ios/Sources/Brave/Frontend/UserContent/UserScripts/Scripts_Dynamic/Scripts/Paged/RequestBlockingScript.js#L25



(a) Third-party request blocking using Safari Content Blocker. **(b) Expected behavior for blocking third-party requests.**

Figure 4: Structure of our test website for blocking requests. We find (a) Safari Content Blocker's interpretation of third-party requests differs from (b) all other non-vulnerable tools, which matches the expected behavior.

Browser/Tool	Issue	Report Date	Fix Date	Report URL
Brave	Scriptlets not injected	8/24/24	12/3/24	[13]
Brave	Cosmetic filters not applied	8/24/24	12/3/24	[13]
Brave	Resource redirection not working on iOS	8/24/24	12/3/24	[13]
Brave	Incorrect request blocking on iOS	8/24/24	12/3/24	[13]
DuckDuckGo	Accounting of blocked requests	9/20/24	10/16/24	[22]
AdGuard	Origin miscomputation for scriptlets	8/17/24	10/4/24	[5]
AdGuard	Origin miscomputation for cosmetic filters	8/17/24	10/4/24	[5]
Apple	Cosmetic filters not applied	8/17/24	3/31/25	N/A
Apple	Third-party definition does not match others	8/17/24	N/A	N/A
AdBlock Plus	Scriptlets not injected	8/17/24	N/A	N/A
AdBlock Plus	Cosmetic filters not applied	8/17/24	N/A	N/A
uBlock Origin	Scriptlets applied inconsistently	8/17/24	N/A	N/A
uBlock Origin Lite	Cosmetic filters not applied	8/20/24	N/A	N/A

Table 6: Summary of our responsible disclosure.

4.4.6 Safari Content Blocker. We find that Safari Content Blockers do not inject cosmetic filters into any local frames, meaning Safari users may see ads that should be blocked. We emphasize that this is not just an issue for Safari users, but also for users of iOS apps that rely on Safari's content blocking functionality.

In addition, we find a larger discrepancy (though not a vulnerability) between how request blocking works on Safari and how request blocking is implemented by all the other content-blocking tools we study. In particular, the discrepancy is in computing if a request is made in a third-party context. Most tools have the behavior shown in Figure 4b; in particular, these tools allow requests to `thirdparty.com` from the third-party iframe. These tools consider these requests to be executing in a first-party context, because the origin of the request is the same as the origin of the document in which the request is made. Safari, on the other hand, has the behavior shown in Figure 4a. Safari blocks the requests to `thirdparty.com` from the third-party iframe, meaning they compare all requests to the origin of the base webpage.

This mismatch in computing "partyness" does not result in a vulnerability in this instance, but can lead to unexpected behavior. In particular, if the iOS apps that use Safari's content blocking functionality use the same filter lists used on other platforms, they will see different behavior. We disclosed this finding to Apple.

5 Ethics and Disclosure

We disclosed all 19 vulnerabilities found to the six affected parties through their preferred channels (Table 6), and the standard 90-day disclosure period has passed. At time of writing, our findings were acknowledged by all organizations (AdBlock Plus, AdGuard, Apple, Brave, DuckDuckGo and uBlock Origin). Moreover, Brave, Apple, AdGuard, and DuckDuckGo have issued patches.

For AdGuard, we determined that introducing a flag in the extension's manifest file solves some—but not all—of their issues. The `match_origin_as_fallback` flag sets the origin of a local frame to the frame that created it [25]. Adding this flag to AdGuard's manifest addresses the scriptlet injection bypass, but AdGuard is still vulnerable to the cosmetic filtering bypass because they still attribute the local frame's origin incorrectly. This also highlights how content blockers may have multiple code paths for handling inherited origins, which can further increase the complexity of their codebases and make it harder to fix these errors. Ultimately, AdGuard's patch identifies local frames and uses the URL of the top-level parent of the local frame to determine whether to inject scriptlets and cosmetic filters [5].

For uBlock Origin, our disclosure process led the maintainers to discover local-frame bypasses with another product. We found that scriptlets are inconsistently injected into local frames for the uBlock Origin Chrome extension. Upon disclosing this to the maintainer of

uBlock Origin, he then tested uBlock Origin Lite (which we had not yet tested). The maintainer found that scriptlets are not injected in local frames and patched this immediately.[6] The patch for scriptlets involved checking if the current document's (non-inherited) origin is null, and if so, finding the first non-null origin of a parent frame. The maintainer subsequently helped us set up an environment to subject uBlock Origin Lite to our suite of tests. We confirmed that scriptlets could be evaded before the patch, and that the issue was fixed after the patch. We also found that cosmetic filters are not injected into local frames; we then disclosed this to the maintainer, who does not plan to fix this until receiving user complaints, as he believes the patch will incur high performance overhead [29].

6 Discussion

We discuss implications of our work for existing research and other tools that may present similar vulnerabilities.

6.1 Web Complexity

As the Web has evolved to provide more capabilities and rich functionality, its subtleties have also increased. Local frames and their inherited origins are just one example of the unexpected behaviors that Web developers must anticipate. It is difficult for any single person (or organization) to correctly parse every nuance of the standards governing the Web.

Furthermore, Web toolkits that are supposed to aid developers in reducing complexity can actually increase code complexity—and introduce avenues for evasion—as developers try to work around the API to achieve their desired functionality. Concretely, Safari Content Blocking is limited to 150,000 rules per extension [48]. Developers at both AdGuard and Brave found ways to apply larger sets of rules, but their approaches each have downsides. AdGuard instantiates several Safari Content Blocker extensions (each of which has an separate 150,000-rule limit), leading to a tedious setup process where users must enable each extension individually [4]. More critically, Brave implements an alternative code path in order to block additional requests, and this code path is vulnerable to evasion by local frames. Hence, while Safari Content Blocking provides a handy primitive, its inability to scale to the requirements of modern content blockers creates some of the same issues it tries to address.

6.2 Impact on Research

Many areas of Web research (both general measurement studies and those focused on security and privacy topics) often need to accurately distinguish between first and third parties on the Web. For example, any research that involves filter lists, either directly (e.g., the maintenance and exception policies of filter lists [54]), or indirectly (e.g., as a set of ground-truth labels for broken websites [38, 46, 52]) requires correctly replicating how browsers determine the "party-ness" of local frames. Even more broadly, correctly determining "party-ness" is important for topics like emulating the security and privacy policies of Web browsers, correctly measuring and attributing behaviors on Web pages (e.g., for browser fingerprinting measurement [23, 33]), and understanding what parties are reading and writing cookies (e.g., [37]).

[6] https://github.com/gorhill/uBlock/commit/520f81f

To understand the implications of our findings on existing research, we sampled a small number of papers from top security and privacy conferences focusing on filter list rules or ad measurement [6, 16, 38, 51, 64–66]. Of the seven papers, only five [6, 16, 38, 64, 66] provided code or pointers to the tools they used. Of these five, we did not find any that correctly computed "party-ness" or correctly handled local frames, either because 1) the provided code determines a frame's party (or security origin) incorrectly, or 2) the provided code logs information for future analysis, and the logging code does not capture the needed information to correctly determine "party-ness" (i.e., the frame's security origin). As one example, we find that the Adscraper tool [67] (and its use in one recent paper [64]) incorrectly checks for "party-ness" against the top-level document instead of the containing frame.

We do not claim that our analysis of a small sample of papers is comprehensive or representative, and conducting a comprehensive study would both be far beyond the scope of this work and require resources not available to us (e.g., source code, data sets, measurement raw data). Nevertheless, our sampling of existing research suggests that tools used to conduct many prior Web security and privacy research studies may contain non-trivial bugs leading to incorrect results, stemming from understandable—though important—misunderstandings in subtle aspects of browser security and privacy policies.

6.3 Other Possible Vulnerabilities

Beyond content blockers, there are many other classes of privacy-focused Web tools that could similarly mishandle local frames. We discuss two particularly sensitive ones below.

6.3.1 Password managers. We investigate the possibility that local frames could be exploited to exfiltrate user credentials by tricking password autofill tools into submitting credentials for a different website into a login form in a local frame. In particular, we consider a third-party local frame containing a login form, and check if autofill tools fill in the user's credentials for the first-party website.

Fortunately, in our tests of five browsers' native autofill features and two Chrome extensions, none are vulnerable. Several browsers (Chrome, Brave, Firefox, and Safari) and both Chrome extensions (1Password and LastPass) do not support autofilling credentials into any iframes—obviating concerns about local frames or any other type of iframe. The DuckDuckGo browser, on the other hand, supports autofilling credentials in iframes and correctly determines the origin of the local frame. (We note that 1Password's debugger correctly identifies the origin of the local frame as a third-party origin, suggesting that even if it were to autofill in iframes, it would not be vulnerable to evasion. We are unable to access similar debugging information for the other password managers.)

6.3.2 Anti-fingerprinting tools. We also test whether browser extensions that spoof canvas fingerprints can be evaded by local frames. Our tests of three extensions considered in a recent study by Nguyen and Vadrevu [45] find that, while all successfully spoof fingerprints in local frames, one extension fails to achieve the stronger goal of creating consistent per-domain fingerprints. We test Canvas Fingerprint Defender (Chrome extension version 0.2.2), CanvasBlocker (Firefox extension version 1.11), and Canvas Blocker - Fingerprint

Protect (Chrome and Firefox extensions version 0.2.1). The latter extension fails to create consistent per-domain fingerprints on Firefox because their method for copying fingerprints from parent frames fails to account for Firefox's extension sandboxing model; we reported this bug to the developer along with a patch. (The Chrome extension successfully creates consistent fingerprints.)

6.4 Potential Breakage

There is always the potential that modifying website behavior—either within or outside of local frames—can cause undesirable user-visible impacts, referred to as "breakage". When acknowledging our findings, every content-blocker organization indicated that these were unintentional vulnerabilities, and not choices meant to avoid breaking websites. In general, when privacy tools cause website breakage, authors add custom-tailored exceptions to their filter lists to restore website functionality—which tools must properly handle within local frames as well. Hence, we consider whether proper local-frame handling may induce additional breakage.

We sample 50 of the 3,142 websites that make requests inside of local frames that should be blocked (i.e., the set of websites that may suffer additional breakage when implementing privacy protections within local frames). We employ a methodology to identify breakage proposed by prior work [6, 33, 53], asking two non-authors to interact with two Chrome browsers, one with the uBlock Origin Lite extension (which correctly implements request blocking, resource replacement, and scriptlet injection in local frames) and one without. The testers interact with each of the 50 websites in both browsers looking for obvious visual differences and the following types of breakage: non-functional search bar, menu, page navigation, comment sections, reviews, social media widgets, or icons. (Suppressed ads are not considered to be breakage.) Neither tester found any instances of breakage involving local frames.

Each tester found exactly one website that exhibited breakage, but the breakage was inconsistent, i.e., the website worked fine for the other tester—and was not related to local frames. One tester observed that `huffpost.com` failed to load videos embedded within articles when uBlock Origin Lite is installed, while the other reviewer found that the login page of `modbee.com` occasionally (but not always) caused the browser with uBlock Origin Lite installed to crash. The video requests on `huffpost.com` are made within the top-level frame, so any breakage is not related to local-frame handling. Similarly, while we cannot replicate the `modbee.com` crash, we confirm that the login page does not create any local frames.

7 Conclusion

Content-blocking tools aim to improve users' browsing experience and protect user privacy by blocking trackers and hiding ads, but we find that they can be easily evaded. Our work shows that many popular content blockers confuse the origin of local frames, and therefore do not correctly apply their filter-list rules to local frames. We find 19 vulnerabilities in the Brave Browser, AdBlock Plus, AdGuard, uBlock Origin Lite, DuckDuckGo, and Safari Content Blocking (a primitive used by many privacy-enhancing iOS apps). We also find that these vulnerabilities are being exploited by website publishers to evade content blockers (though we do not know if this evasion is intentional). Local frames are prevalent on more than half of popular websites and 14.3% of these popular websites make requests to resources that should be blocked according to popular filter lists. Based on our work, Brave, Safari, DuckDuckGo, and AdGuard have patched their systems.

Acknowledgments

We thank Stefan Savage, Miro Haller, Ali Ukani, Paul Chung, Anirudh Canumalla, Cindy Moore, and our anonymous reviewers. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-2038238. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] AdBlock. Introduction to Filter Lists. https://helpcenter.getadblock.com/hc/en-us/articles/9738523403027-Introduction-to-Filter-Lists. Accessed 2024-09-03.
[2] Available Scriptlets. https://github.com/AdguardTeam/Scriptlets/blob/master/wiki/about-scriptlets.md. Accessed 2024-08-21.
[3] How to create your own ad filters. https://adguard.com/kb/general/ad-filtering/create-own-filters. Accessed 2024-08-30.
[4] AdGuard. Safari Web extension. https://adguard.com/kb/adguard-for-ios/web-extension/. Setup instructions for the AdGuard iOS app. Accessed 2024-09-04.
[5] Commit 277790f. https://github.com/AdguardTeam/tsurlfilter/commit/277790f4786f017c0f4f552795c13627dbee1f66.
[6] Abdul Haddi Amjad, Danial Saleem, Muhammad Ali Gulzar, Zubair Shafiq, and Fareed Zaffar. TrackerSift: Untangling Mixed Tracking and Functional Web Resources. In *Proceedings of the 21st ACM Internet Measurement Conference*, 2021.
[7] Anton Lazarev, Brave Software. adblock-rs. https://github.com/brave/adblock-rust. Rust library for parsing and applying filter list rules. Accessed 2024-09-03.
[8] Apple. Creating a content blocker. https://developer.apple.com/documentation/safariservices/creating-a-content-blocker. Accessed 2024-08-20.
[9] EasyList Authors. EasyList. https://easylist.to/easylist/easylist.txt.
[10] EasyList Authors. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.
[11] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. How Tracking Companies Circumvented Ad Blockers Using WebSockets. In *Proceedings of the 18th ACM Internet Measurement Conference*, 2018.
[12] Brave. pagegraph-crawl. https://github.com/brave/pagegraph-crawl.
[13] fix(privacy): Issues with content filtering in local frames on iOS. https://github.com/brave/brave-core/pull/26622.
[14] Charles Web Debugging Proxy. https://www.charlesproxy.com/.
[15] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.
[16] Ha Dao and Kensuke Fukuda. Alternative to third-party cookies: Investigating persistent PII leakage-based web tracking. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, 2021.
[17] Mike Diaz. What I Learned at Work this Week: Friendly IFrames and Debounce. *Medium*, 2020. Accessed 2024-12-27.
[18] DuckDuckGo. android-tds.json. https://github.com/duckduckgo/tracker-blocklists/blob/3ea4fbb7821f4bb0065722dfae4f4c9d63a1266d/web/v5/android-tds.json. DuckDuckGo blocklist for the Android platform. Accessed 2024-09-03.
[19] DuckDuckGo. extension-tds.json. https://github.com/duckduckgo/tracker-blocklists/blob/3ea4fbb7821f4bb0065722dfae4f4c9d63a1266d/web/v6/extension-tds.json. DuckDuckGo blocklist for browser extensions. Accessed 2024-09-03.
[20] DuckDuckGo. ios-tds.json. https://github.com/duckduckgo/tracker-blocklists/blob/3ea4fbb7821f4bb0065722dfae4f4c9d63a1266d/web/v5/ios-tds.json. DuckDuckGo blocklist for the iOS platform. Accessed 2024-09-03.
[21] DuckDuckGo. macos-tds.json. https://github.com/duckduckgo/tracker-blocklists/blob/3ea4fbb7821f4bb0065722dfae4f4c9d63a1266d/web/v6/macos-tds.json. DuckDuckGo blocklist for the macOS platform. Accessed 2024-09-03.
[22] Change tab URL calculation for contentblockerrules.js and surrogates.js. https://github.com/duckduckgo/BrowserServicesKit/pull/1021.
[23] Steven Englehardt and Arvind Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, 2016.
[24] FingerprintJS. https://github.com/fingerprintjs/fingerprintjs.
[25] Chrome for Developers. Content scripts. https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts. Accessed 2024-08-30.

[26] Lauren Forristal. Report shows that Safari reaches one billion worldwide users, still behind Google Chrome. *TechCrunch*, 2022. Accessed 2025-01-06.
[27] Google. Viewability best practices. https://support.google.com/admanager/answer/6199883?hl=en. Accessed 2024-12-27.
[28] Google. What is rich media? https://support.google.com/richmedia/answer/2417545?hl=en. Accessed 2024-12-27.
[29] Raymond Hill. Private communication.
[30] Interactive Advertising Bureau (IAB). Rich Media Measurement Guidelines. https://www.iab.com/wp-content/uploads/2015/06/Rich_Media_Measurement_Guidelines_v2.pdf, 2007. Accessed 2024-08-29.
[31] Interactive Advertising Bureau (IAB). Best Practices for Rich Media Ads in Asynchronous Ad Environments. https://www.iab.com/wp-content/uploads/2015/09/rich_media_ajax_best_practices.pdf, oct 2008. Accessed 2024-08-29.
[32] Disconnect Inc. Entity List. https://github.com/mozilla-services/shavar-prod-lists/blob/02f6a2835a851fd92d3f996409cfd18c2d4b0a2b/disconnect-entitylist.json, 2024. Accessed 2025-05-03; last updated 2025-04-25.
[33] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.
[34] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the 17th ACM Internet Measurement Conference*, 2017.
[35] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.
[36] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Aidan Beggs, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. Towards Realistic and Reproducible Web Crawl Measurements. In *Proceedings of the 30th ACM Web Conference*, 2021.
[37] Jordan Jueckstock, Peter Snyder, Shaown Sarker, Alexandros Kapravelos, and Benjamin Livshits. Measuring the Privacy vs. Compatibility Trade-off in Preventing Third-Party Stateful Tracking. In *Proceedings of the 31st ACM Web Conference*, 2022.
[38] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. AutoFR: Automated Filter Rule Generation for Adblocking. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.
[39] David Li. Manifest V2 phase-out begins. *Chromium Blog*, May 2024. Accessed 2024-08-20.
[40] Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. Accessed 2024-08-25.
[41] S Moonesamy. The "about" URI Scheme. RFC 6694, August 2012.
[42] Mozilla. Extensions in Privacy & Security. https://addons.mozilla.org/en-US/firefox/extensions/category/privacy-security/?page=1&sort=recommended%2Cusers. Accessed 2025-01-08.
[43] Mozilla. Firefox Public Data Report. https://data.firefox.com/dashboard/usage-behavior. Accessed 2025-01-08.
[44] Muhammad Haris Mughees and Zhiyun Qian. Detecting Anti Ad-blockers in the Wild. In *Proceedings on Privacy Enhancing Technologies*, volume 3, 2017.
[45] Hoang Dai Nguyen and Phani Vadrevu. Breaking the Shield: Analyzing and Attacking Canvas Fingerprinting Defenses in the Wild. In *Proceedings of the 34th ACM Web Conference*, 2025.
[46] Alexandra Nisenoff, Arthur Borem, Madison Pickering, Grant Nakanishi, Maya Thumpasery, and Blase Ur. Defining "Broken": User Experiences and Remediation Tactics When Ad-Blocking or Tracking-Protection Tools Break a Website's User Experience. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.
[47] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J Murdoch. Adblocking and Counter-Blocking: A Slice of the Arms Race. In *Proceedings of the 6th USENIX Workshop on Free and Open Communications on the Internet*, 2016.
[48] Sofia Orlova. AdGuard v1.11 for Safari: Fight for filtering rule limits. *AdGuard Blog*, March 2022. Accessed 2024-09-03.
[49] Raymond Hill. uBlock Origin. https://github.com/gorhill/uBlock. An efficient blocker for Chromium and Firefox. Accessed 2024-09-03.
[50] Human Security. Types of iFrames and When to Use Them. https://www.humansecurity.com/learn/topics/types-of-iframes-and-when-to-use-them. Accessed 2024-12-27.
[51] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *Proceedings of the 31st USENIX Security Symposium*, 2022.
[52] Michael Smith, Peter Snyder, Moritz Haller, Benjamin Livshits, Deian Stefan, and Hamed Haddadi. Blocked or Broken? Automatically Detecting When Privacy Interventions Break Websites. In *Proceedings on Privacy Enhancing Technologies*, volume 4, 2022.
[53] Peter Snyder, Cynthia Taylor, and Chris Kanich. Most Websites Don't Need to Vibrate: A Cost–Benefit Approach to Improving Browser Security. In *Proceedings of the 24th ACM Conference on Computer and Communications Security*, 2017.
[54] Peter Snyder, Antoine Vastel, and Ben Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2), 2020.
[55] Brave Software. Platform Stats & Token Activity. https://brave.com/transparency/. Accessed 2025-01-06.
[56] Brave Software. Slim List System. https://github.com/brave/slim-list-lambda.
[57] Chrome Web Store. Adblock Plus - Chrome Web Store. https://chromewebstore.google.com/detail/adblock-plus-free-ad-bloc/cfhdojbkjhnklbpkdaibdccddilifddb. Accessed 2024-08-19.
[58] Chrome Web Store. AdGuard AdBlocker - Chrome Web Store. https://chromewebstore.google.com/detail/adguard-adblocker/bgnkhhnnamicmpeenaelnjfhikgbkllg. Accessed 2024-08-19.
[59] Chrome Web Store. Privacy & Security. https://chromewebstore.google.com/category/extensions/make_chrome_yours/privacy. Accessed 2025-01-08.
[60] Chrome Web Store. uBlock Origin - Chrome Web Store. https://chromewebstore.google.com/detail/ublock-origin/cjpalhdlnbpafiamejdnhcphjbkeiagm. Accessed 2024-08-19.
[61] Chrome Web Store. uBlock Origin Lite - Chrome Web Store. https://chromewebstore.google.com/detail/ublock-origin-lite/ddkjiahejlhfcafbddmgiahcphecmpfh. Accessed 2024-08-20.
[62] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.
[63] WHATWG. HTML Standard. https://html.spec.whatwg.org/commit-snapshots/c974c42b8bbc3b04cba372f0088fad503be3cc04/. Accessed 2024-08-29.
[64] Christina Yeung, Tadayoshi Kohno, and Franziska Roesner. Analyzing the (In)Accessibility of Online Advertisements. In *Proceedings of the 24th ACM Internet Measurement Conference*, 2024.
[65] Ahsan Zafar and Anupam Das. Comparative Privacy Analysis of Mobile Browsers. In *Proceedings of the 13th ACM Conference on Data and Application Security and Privacy*, 2023.
[66] Ahsan Zafar, Aafaq Sabir, Dilawer Ahmed, and Anupam Das. Understanding the Privacy Implications of Adblock Plus's Acceptable Ads. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2021.
[67] Eric Zeng. Adscraper: A Web Crawler for Measuring Online Ad Content.
[68] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Proceedings of the 25th Network and Distributed System Security Symposium*, 2018.

A Fingerprinting APIs

We classify a fingerprinting-related API call as any access to one of the following APIs:

- CanvasRenderingContext2D.measureText
- HTMLCanvasElement.toDataURL
- MediaDevices.enumerateDevices
- Navigator: appCodeName.get, appName.get, appVersion.get, bluetooth.get, brave.get, deviceMemory.get, doNotTrack.get, getBattery, globalPrivacyControl.get, hardwareConcurrency.get, language.get, languages.get, maxTouchPoints.get, mediaCapabilities.get, mediaDevices.get, plugins.get, productSub.get, usb.get, userAgent.get, userAgentData.get, vendor.get, vendorSub.get
- Screen: availHeight.get, availLeft.get, availTop.get, availWidth.get, colorDepth.get, height.get, isExtended.get, pixelDepth.get, width.get
- WebGL2RenderingContext: getExtension, getParameter
- WebGLRenderingContext: getExtension, getParameter, getShaderPrecisionFormat

B Third-Party Entities

Table 7 details the corresponding eTLD+1s for the third-party entities presented in Table 4. We report the top-10 entities (URLs are mapped to owning organization using the Disconnect entity

Entity	eTLD+1s
Ranks [1–15K)	
Google	[2mdn.net, doubleclick.net, google.com, googlesyndication.com, recaptcha.net, youtube-nocookie.com, youtube.com]
PubMatic	[pubmatic.com]
Unity	[yellowblue.io]
Cloudflare	[cloudflare.com, cloudflarestream.com]
Amazon	[amazon-adsystem.com, twitch.tv]
Vidoomy	[vidoomy.com]
Datadome	[captcha-delivery.com]
NextMillennium	[nextmillmedia.com]
ConnectAdRealtime	[connectad.io]
Piano	[piano.io, tinypass.com]
Ranks [15K–100K)	
Google	[2mdn.net, doubleclick.net, google.com, googlesyndication.com, recaptcha.net, youtube-nocookie.com, youtube.com]
adtrafficquality.google	[adtrafficquality.google]
PubMatic	[pubmatic.com]
Cloudflare	[cloudflare.com]
SeedTag	[seedtag.com]
AdYouLike	[omnitagjs.com]
admatic.de	[admatic.de]
Amadeus	[travelaudience.com]
Amazon	[amazon-adsystem.com]
ConnectAdRealtime	[connectad.io]
Ranks [100K–1M)	
Google	[2mdn.net, doubleclick.net, google.com, googlesyndication.com, recaptcha.net, youtube-nocookie.com, youtube.com]
adtrafficquality.google	[adtrafficquality.google]
Cloudflare	[cloudflare.com]
PubMatic	[pubmatic.com]
Amadeus	[travelaudience.com]
SeedTag	[seedtag.com]
Jivox	[jivox.com]
Yandex	[yandex.ru]
Chaturbate	[chaturbate.com]
AdYouLike	[omnitagjs.com]

Table 7: The eTLD+1s for the content loaded into third-party local frames by the top-10 entities as reported in Table 4.

list [32]) that are targeted by these privacy-suspect requests in Table 8, all of which are advertising and analytics companies. We find that Google is the most common entity, contacted by almost 5× more sites than the next-most-popular entity.

C Additional Testing Details

This section provides additional details that we consider in testing, but that do not affect the results of vulnerable tools. The nested local frames shown in Listing 2 test for the case wherein a tool checks for local frames by only considering the local frame's direct parent (not the local frame's non-local ancestor). If this were true, then a tool could be evaded by nested local frames, but not by regular local frames. We do not find that nested local frames can evade the capabilities outlined in Section 2.2. However, we find that nested

Entity	# Sites	# Requests
Google	1514	23644
Microsoft	350	1283
PubMatic	332	718
Integral Ad Science	332	1497
Criteo	329	1284
Magnite	292	602
Taboola	250	498
IndexExchange	233	349
LiveIntent	233	373
Nexxen	229	539

Table 8: The top-10 entities that receive privacy-suspect requests, i.e. requests from local frames that should be blocked.

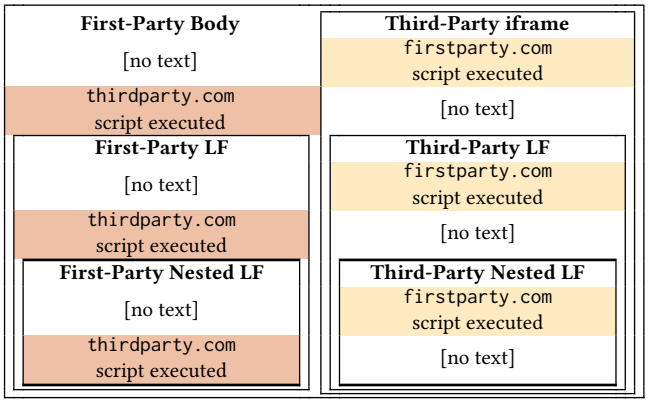


Figure 5: Expected behavior for blocking first-party requests.

local frames can trick DuckDuckGo into misreporting the privacy harms of a website (Section 4.4.5).

Concretely, in the main body of this work, we only present results for universally blocking requests. However, as noted in Section 2.2.1, content blockers sometimes block requests only when they are loaded in a third-party context. We test two variants of **RQ1** that check the context in which request are made. First, we consider **RQ1a**: If you block third-party requests to a resource, does the resource only load in a first-party context? This means the first-party frames should allow the script from `firstparty.com` and block the script from `thirdparty.com`. The third-party frames should allow the script from `thirdparty.com`—since the script is local with respect to the origin of the iframe—and block the script from `firstparty.com`. Expected behavior is shown in Figure 4b.

Second, we consider **RQ1b**: If you block first-party requests to a resource, does the resource only load in a third-party context? As shown in Figure 5, the first-party frames will only allow the scripts from `thirdparty.com`, and the third-party frames will only allow the scripts from `firstparty.com`.

We do not find any tools that are vulnerable to RQ1a or RQ1b, but not RQ1. However, we find that Safari computes the "party-ness" of requests differently than all other tools we study (Section 4.4.6).