Overloading Ad Blockers: Exploiting Filtering Logic for Selective Denial of Service

Hyunjoon Lee*
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
hjpaul@kaist.ac.kr

Suyoung Lee*
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
suyoung.lee@kaist.ac.kr

Heiko Kiesel
Technical University of Darmstadt
Darmstadt, Germany
hkiesel@seemoo.de

Sooel Son
Korea Advanced Institute of Science and Technology
Daejeon, Republic of Korea
sl.son@kaist.ac.kr

Abstract

We present the first denial-of-service attack against ad blockers, called the Overload attack. A malicious ad publisher or provider can launch our attack to selectively impose substantial page-loading delays and computational overhead specifically for users with ad blockers, thereby incentivizing them to disable their ad blockers. We introduce three variants of the Overload attack, each exploiting distinct vulnerabilities within popular ad blockers. All the attack variants operate by injecting adversarial HTML elements into a target webpage, thereby triggering excessive latency during page rendering. When applied to the Tranco Top-10K websites, our attack increases page-loading time by up to 76.94× and 146.34× on Chrome and Firefox, respectively, for users with ad blockers compared to those without. Finally, we propose mitigation strategies that efficiently neutralize the Overload attacks and can be seamlessly integrated into existing ad blockers.

CCS Concepts

• **Security and privacy → Domain-specific security and privacy architectures**; **Denial-of-service attacks**; *Privacy protections*; *Web application security*.

Keywords

Ad Blockers, Denial-of-Service Attacks, Filter Rule Matching

ACM Reference Format:
Hyunjoon Lee, Suyoung Lee, Heiko Kiesel, and Sooel Son. 2026. Overloading Ad Blockers: Exploiting Filtering Logic for Selective Denial of Service . In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832751

1 Introduction

As the global online ad market continues to grow—projected to reach USD 416.65 billion in 2026 [34]—advertising and tracking service (ATS) providers have pervasively tracked Internet users to deliver targeted ads [9, 16, 18, 20, 33, 38, 44, 45, 47, 48, 56, 62, 68]. To mitigate such tracking, ad blockers have emerged as a widely adopted tool, protecting users' privacy with minimal disruption to their browsing experience. As of 2023, 31.5% of global Internet users (i.e., approximately 912 million people) use ad blockers [58].

This widespread adoption has triggered an arms race between ad blockers, which aim to safeguard users' privacy, and ATS publishers and providers, who seek to maximize their revenue. To sustain their revenue, ATS publishers and providers have developed various anti-ad blocking techniques that primarily focus on detecting or bypassing ad blockers. For instance, to detect the presence of ad blockers, they deliberately insert bait HTML elements designed to be blocked [49]. To bypass blocking, advertisers also employ strategies such as randomly mutating HTML elements on webpages [63] or frequently changing their domains [11, 40, 50].

In this paper, we propose a novel class of anti-ad blocking techniques that directly target ad blockers through active attacks, rather than detecting or bypassing them. Prior anti-ad blocking research [11, 40, 49, 50, 63] has largely overlooked the possibility that ad blockers themselves may introduce new security weaknesses that ATS publishers or providers can exploit. Given the massive adoption of ad blockers, such vulnerabilities have critical implications for the security and trustworthiness of the Web.

Specifically, we ask the following question: *how can a malicious ATS publisher or provider selectively launch denial-of-service attacks only against ad blocker users by exploiting inherent weaknesses in ATS blocking algorithms?* This question is important from a security standpoint, as successful exploitation could not only degrade the browsing experience of ad blocker users but also lead users to disable their ad blockers.

Our contributions. To answer this research question, we present the first denial-of-service (DoS) attack that selectively imposes substantial page-loading latency and computational overhead specifically on users with ad blockers enabled. The objective of this attack is twofold: (1) to degrade the browsing experience of users with ad blockers to the point that they are incentivized to disable ad blockers, and (2) to minimize page-loading latency and overhead for users without ad blockers. We refer to this new attack as the *Overload* attack throughout the paper.

Ad blockers typically employ multiple ATS filtering mechanisms based on filter rules that define patterns of resources to be blocked.

*Both authors contributed equally to the paper.



This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.
CCS '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
https://doi.org/10.1145/3830454.3832751

We present *three* variants of the Overload attack, each targeting a distinct ATS filtering mechanism. ATS publishers or providers perform the Overload attack by injecting adversarial HTML elements into their webpages. The first variant injects HTML elements that initiate network requests to URLs that match an excessive number of filter rules, thereby imposing unnecessary latency during rule matching. The second and third variants introduce a large number of iframe instances. Each instance is designed to repeatedly trigger ATS-related cookie removal or content script injection, respectively, thereby resulting in substantial processing delays.

Designing an effective Overload attack entails two key technical challenges. (1) Injected adversarial HTML elements should not disrupt the browsing experience of users without ad blockers. That is, the attack should neither break webpage functionality nor introduce noticeable latency for these users. (2) Since the adversary is unable to know which specific ad blocker the user has installed, the attack should remain broadly effective across multiple ad blockers.

To address the first challenge, we conducted a comprehensive analysis of attack factors influencing page-loading time or functional correctness, and incorporated these insights into our attack design. Each attack factor was designed to maximize the attack's effectiveness while minimizing any negative impact on legitimate webpage functionality.

To tackle the second challenge, we examined the mechanisms shared by ad blockers to enforce ATS filtering rules. For example, we found that ad blockers employ keyword-based optimization algorithms to accelerate rule matching for each network request URL. However, following the enforcement of Manifest V3 [22–24], all ad blockers installed on Chrome delegate rule matching to the browser. In this case, Chrome itself implements a similar keyword-based optimization algorithm. Our attack therefore exploits a vulnerability in this shared ATS filtering mechanism, which is common across different ad blockers, including those installed on Chrome.

We evaluated the Overload attack against five widely used ad blockers—AdBlock, Adblock Plus, AdGuard, uBlock Origin (Lite), and AdBlocker Ultimate—on two web browsers, Chrome and Firefox. We first deployed the Overload attack against real-world websites selected from the Tranco Top-10K list in a local network environment. Without ad blockers, the attack incurred less than 1–2 seconds of page-loading latency. In contrast, when ad blockers are enabled, the attack increased main page load time by up to 76.94× (146.34×) and raised power consumption by up to 33.07× (43.48×) on Chrome (Firefox). Our manual breakage analysis showed that 99.22% of the evaluated webpages functioned normally without major disruption. We further assessed the attack in a live setting using a proxy on the same set of websites, where it introduced up to 12.61× additional loading delay on average.

To mitigate this threat, we propose defenses for each variant of the Overload attack. For our first attack, we recommend applying a compressibility-based filtering step prior to rule evaluation to reduce unnecessary filter rule matching. For the second and third attacks, we suggest postponing cookie removal and limiting the excessive use of blank iframes, which are exploited by our attack. We implemented these mitigation strategies within ad blockers, including those integrated with Chrome, and evaluated their effectiveness. With our mitigation in place, performance overhead was reduced by 32.60×, while no legitimate resources were blocked.

To facilitate future research, we release the implementation of the Overload attack and its demonstration videos at https://github.com/WSP-LAB/Overload.

2 Background

2.1 Ad Blockers

An ad blocker refers to a browser extension or mobile app that blocks web requests to ATS vendors, removes ATS-related resources, or hides ATS-related HTML elements. AdBlock [2], Adblock Plus [29], and uBlock Origin [6] are representative ad blockers, each with 18M+ users, underscoring their prevalence among Internet users.

These ad blockers rely on crowdsourced filter lists [3, 4], each comprising a set of filter rules that specify which web resources (e.g., network requests, HTML elements, and cookies) should be blocked, hidden, or removed during page rendering. Filter rules are generally categorized into four types: blocking, exception, snippet, and element hiding filters [28, 30].

Blocking filters. Blocking filters are basic network-level rules that prevent requests from being sent to URLs known to serve ATS resources. Each filter rule consists of a pattern and an optional specification. The *pattern* defines the matching conditions, such as URL substrings, while the *optional specification* imposes additional constraints, including the type of a requesting HTML element (e.g., scripts or images). For example, consider the rule `/ad_side.$~xmlhttprequest`, where `$` is a separator. This rule instructs ad blockers to block any network request that (1) contains `/ad_side.` in the URL path and (2) is *not* initiated via the `XMLHttpRequest` object.

Exception filters. Exception filters share a format similar to blocking filters but serve the opposite purpose: they allow network requests that would otherwise be blocked by one or more blocking filters. Accordingly, exception filters are evaluated only after a resource has matched at least one blocking filter.

Snippet filters. Snippet filters instruct ad blockers to execute short JavaScript (JS) snippets on webpages to block specific ATS-related behaviors. A snippet filter is applied when the fully qualified domain name (FQDN) of the current page ends with the domain specified in the rule. For example, consider the following rule: `news.com#$#cookie-remover adblock`, where `#$#` serves as a separator. This rule tells ad blockers to execute a JS snippet named `cookie-remover`, passing `adblock` as an argument when loading the page from `news.com`. This snippet is designed to remove cookies whose names contain `adblock` for this domain, which are likely used for anti-ad blocking.

Element-hiding filters. Element-hiding filters consist of a domain name and an (extended) CSS selector. These rules instruct ad blockers to hide elements that match the specified CSS selector. For example, consider the rule: `news.com##div:has-text(Sponsor)`. When a user visits `news.com`, this rule directs the browser to hide all `div` elements whose text content, including that of their descendants, contains the string "Sponsor".

On most browsers, ad blockers enforce filter rules by directly inspecting network requests or HTML elements. For blocking and exceptions filters, ad blockers monitor outgoing network requests and block (or allow) those that match the filter rules. For snippet and element-hiding filters, ad blockers inject *content scripts* [21]

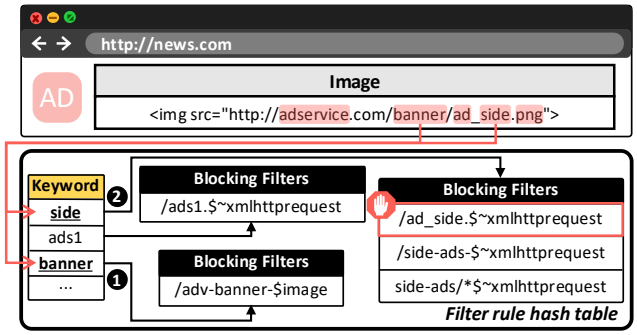


Figure 1: Overview of the keyword-based filter rule matching algorithm implemented by AdBlock and Adblock Plus.

into webpages; these scripts execute matching JS snippets or hide matching HTML elements.

In Chrome, however, extensions can no longer directly access network requests due to the enforcement of Manifest V3 [22–24]. Consequently, Chrome ad blockers provide their blocking and exception rules to the browser via the `chrome.declarativeNetRequest` API [25], and the browser itself monitors and blocks web requests. In other words, all ad blockers installed on Chrome rely on the *same* underlying algorithm to apply blocking and exception filters.

2.2 Evaluating Filter Rules

When a web resource is encountered, ad blockers determine whether to block its loading or rendering. Given that ad blockers typically maintain 87K+ filter rules, exhaustively checking every rule against each resource would be computationally expensive. To minimize page-loading latency, ad blockers (1) identify a minimal subset of relevant filter rules for each web resource and (2) evaluate those rules. Here, we define *filter rule evaluation* as checking whether each rule matches a web resource and, if so, performing the corresponding action (e.g., blocking a network request). Throughout this paper, we use the term *filter rule matching* to describe these two processing steps.

For snippet and element-hiding rules, identifying a minimal subset of relevant filter rules is straightforward. Since these rules explicitly specify the domains to which they apply, ad blockers can construct such a subset by selecting only those whose domain names match the current domain.

In contrast, constructing such a minimal set is challenging for blocking and exception rules, as these rules specify matching conditions using patterns (e.g., URL substrings) rather than an explicit domain name. To address this challenge, ad blockers commonly employ hash tables. In the following, we explain how ad blockers construct hash tables and use them to accelerate blocking and exception filter rule matching.

Hash table construction. Ad blockers typically construct a hash table from filter rules in filter lists such as EasyList [3] and EasyPrivacy [4]. Recall that each blocking or exception filter contains a pattern. Ad blockers extract keywords from these patterns, with each keyword serving as a key in the hash table.

For example, AdBlock and Adblock Plus identify keywords by extracting all alphanumeric substrings of at least two characters, excluding common predefined terms (e.g., `http` or `com`). Given the blocking filter `/ad_side.$~xmlhttprequest`, they extract `ad` and `side` as keywords. Among these, they select the keyword with the fewest filter rules and assign the current filter rule to that keyword.

Keyword-based rule matching. Figure 1 illustrates how ad blockers leverage hash tables to accelerate filter rule matching. The figure depicts a webpage from `news.com` that displays an ad image served by `adservice.com`. When AdBlock encounters this image request, it extracts alphanumeric keywords of at least two characters from the request URL, as shown in the figure. These keywords are then used to query the hash table and construct a minimal subset of filter rules for evaluation.

In this example, the hash table only contains `banner` and `side` as relevant keys, resulting in a subset of four blocking filters associated with these keywords. AdBlock then evaluates the candidate rules sequentially in the order in which their keywords appear in the URL. The rule associated with `banner` does not match the request URL, and thus AdBlock proceeds to the rules associated with `side`. Because the URL contains `/ad_side.`, and the request is not initiated via the `XMLHttpRequest` object, it matches the first filter associated with `side`. Consequently, the ad blocker stops after evaluating two rules and blocks the request.

This keyword-based rule matching mechanism enables ad blockers to efficiently construct a small set of candidate rules for each network request, thereby significantly reducing the overhead of filter rule evaluation. Other ad blockers, including Chrome, employ more or less the same keyword-based hash table and rule matching algorithm, but differ in how keywords are extracted. For example, Chrome extracts 5-grams as keywords.

3 Motivation

3.1 Threat Model

Goal. In this paper, we present three variants of the Overload attack. For all three attack variants, we assume our main adversary is an *ATS publisher* (i.e., first-party) who monetizes website traffic by rendering ads. We additionally consider a malicious *ATS provider* (i.e., third-party) for our first and third attack variants.

Since ad blockers reduce their revenue, the adversary seeks to induce their webpage visitors to disable ad blockers. To this end, our attack is designed to (1) impose substantial page-loading latency on users with ad blockers enabled, (2) while minimizing any negative impact on the browsing experience of users without ad blockers. Specifically, when ad blockers are enabled, our attack either stalls the entire page-rendering process and displays a blank page for several minutes, or gradually renders HTML elements by delaying the loading of randomly selected elements.

However, merely introducing page-loading latency may not clearly signal to ad blocker users that disabling ad blockers would improve page-loading performance. To address this issue, the adversary displays a message indicating that the observed performance degradation results from enabling ad blockers, prior to initiating the ad blocker overload.

Capabilities. We assume that the adversary is capable of injecting arbitrary HTML elements into webpages. For example, a malicious publisher can directly control the content and structure of their own

webpages. Similarly, a malicious ATS provider can embed JS snippets that manipulate webpage content within ad-serving scripts. This adversary model assumes that the provider can deliver at least one attack script that bypasses ad blockers, which can be achieved via CNAME cloaking [26, 54]. Once a single script is successfully delivered, our attack enables the provider to deliver subsequent ad resources by inducing users to disable their ad blockers, without requiring additional effort to evade filtering rules. Finally, we assume that the adversary does not know whether a user has enabled an ad blocker or which specific ad blocker is in use.

Differences from existing anti-ad blocking. Existing anti-ad blocking techniques typically attempt to detect the presence of ad blockers and, upon detection, display warnings that prompt users to disable them [10, 49]. However, most modern ad blockers, including Adblock Plus and uBlock Origin, counter such attempts using snippet filters—for example, by injecting JS snippets that overwrite ad blocker detection code.

In contrast, we propose a novel class of anti-ad blocking techniques that directly target ad blockers through active attacks, rather than attempting to detect them. Our approach exploits the intrinsic functionality of ad blockers themselves, deliberately triggering substantial computational overhead and thereby inducing users to disable ad blockers.

3.2 Technical Challenges

In this paper, we present a novel attack, called the Overload attack, which incurs an excessive page-loading time only for users with ad blockers enabled. Designing our attack presents two unique technical challenges: (1) minimizing negative side effects and (2) ensuring general applicability across different ad blockers.

Minimizing side effects. Recall that the adversary aims to impose minimal impact on the browsing experience of users with ad blockers disabled. We define the browsing experience as computational overhead and the functional correctness of the target webpage. Accordingly, the attack should selectively introduce substantial loading delays for users with ad blockers, while having a negligible increase in page load time on those without ad blockers. Moreover, the attack must preserve the core functionality of the page.

To address this challenge, we decomposed our manipulation strategy into controllable factors, such as domain names and URL paths. We then conducted an empirical analysis to assess how each factor affects page load time and webpage functionality. We incorporated the insights from this analysis into our attack design to minimize adverse side effects for users who disabled ad blockers.

General applicability across ad blockers. In our threat model, the adversary exploits vulnerabilities in how ad blockers and Chrome perform filter rule matching. However, the adversary does not know in advance which specific ad blocker a user has installed. Therefore, the attack must target common weaknesses shared across popular ad blockers and the Chrome browser.

To achieve this, we analyzed how various ad blockers and Chrome enforce filter rules. Through this analysis, we identified three common vulnerabilities spanning different categories of filter rules. Based on these findings, we developed three variants of the Overload attack, each designed to exploit one of these vulnerabilities.

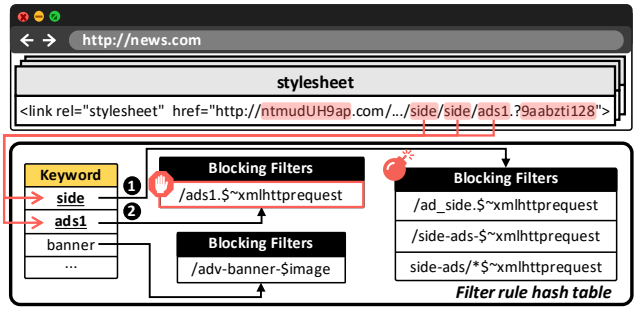


Figure 2: Triggering excessive rule evaluation.

4 Overload Attack

4.1 Overview

We present the Overload attack, the first selective DoS attack designed to induce excessive page-loading latency in client browsers when ad blockers are enabled. Given a target HTML webpage, the attack injects adversarial HTML elements crafted to trigger large computational overhead while enforcing filter rules. We present three attack variants, each targeting a different type of filter rule.

Triggering excessive rule evaluation. This attack exploits a vulnerability in how ad blockers and Chrome optimize filter rule matching for performance. For each outgoing network request, they first identify a subset of relevant blocking and exception rules and evaluate only those rules, rather than checking the entire filter rule set. Our attack undermines this optimization by forcing ad blockers and Chrome to evaluate an excessive number of filter rules for each request, causing substantial page-loading delays.

Invoking repeated cookie removal. The second attack targets a vulnerability in how snippet filters are enforced. Recall that ad blockers support snippet rules that remove cookies. An adversary can impose significant computational overhead by repeatedly triggering this cookie-removal JS snippet.

Repeatedly injecting content scripts. The last attack abuses how ad blockers handle blank iframes (i.e., iframes whose content is `about:blank`). Recall from §2.1 that ad blockers inject *content scripts* into webpages to enforce snippet and element-hiding filters. Because ATS publishers may attempt to load ads in blank iframes to bypass ad blockers, ad blockers execute content scripts not only in the main frame but also in each blank iframe. Consequently, an adversary can incur substantial computational overhead by repeatedly injecting blank iframes into a webpage.

4.2 Attack I: Triggering Excessive Rule Evaluation

In this scenario, the Overload attack exploits the keyword-based rule matching algorithm used to identify a subset of relevant blocking and exception filter rules for each web request. The attack has two objectives: (1) to overload ad blockers by generating request URLs of limited length that maximize the number of filter rules evaluated, and (2) to minimize additional page-loading latency and functional breakage for users without ad blockers, thus avoiding any negative impact on their browsing experience.

Algorithm 1: URL path construction

```
Input  : A maximum URL length (l_max).
         A keyword sampling strategy (is_gs).
         A set of the AdBlock (or uBlock Origin) keywords (K).
Output : A generated URL path (url_path).
1 function ConstructURLPath(l_max, is_gs, K)
2     url_path, l_cur, ← InitURLPath()
3     elem_type ← SelectElemType(is_gs)
4     while l_cur < l_max do
5         keyword ← SelectKeyword(K, elem_type, is_gs)
6         url_path ← AppendKeyword(url_path, keyword)
7         l_cur ← GetStrLength(url_path)
8     url_path ← AppendBlockPattern(url_path)
9     return url_path
```

Given an HTML webpage, the adversary mounts this attack by (1) crafting keyword-rich URLs and (2) injecting HTML elements that initiate requests to these URLs. Figure 2 illustrates how the injected HTML elements trigger excessive rule evaluation. In the following, we describe this workflow in detail, including the components involved in URL construction and the rationale behind their design. We then discuss the selection of HTML element types and attributes used to embed these URLs.

URL Domain. We designed the URL domains to minimize adverse effects on users who have disabled ad blockers. For users with ad blockers, the choice of FQDNs has minimal impact on page load time. Recall from §2 that ad blockers perform filter rule matching based on URL patterns before DNS resolution and block network requests at this stage.

In contrast, for users without ad blockers, the choice of FQDNs directly affects page load time. Using existing FQDNs would result in successful DNS resolution and subsequent network requests, introducing unwanted latency for these users. We thus designed the Overload attack to use non-existent FQDNs in the crafted URLs, ensuring that the browser terminates the requests early during DNS resolution without sending actual network traffic. Specifically, we generate domains composed of 10 random alphanumeric characters followed by a valid top-level domain, such as `ntmudUH9ap.com`.

URL path. We design the URL path component to maximize the filtering workload imposed on ad blockers. For this, the adversary generates paths that embed a large number of keywords, each selected to be associated with a large number of filter rules.

The Overload attack constructs such URL paths using two keyword selection strategies: greedy sampling (GS) and random sampling (RS). GS involves repeatedly including the same keyword associated with the largest number of filter rules, while RS constructs URL paths with randomly selected keywords. If the adversary aims to maximize the attack's effectiveness, they would choose GS. However, this approach may increase the likelihood of detection due to its repeated use of the same keyword (see §6.1). Therefore, to bypass the detection, the adversary can instead opt for the RS strategy at the cost of imposing lower page-loading latencies.

Algorithm 1 describes our URL path generation algorithm. The `ConstructURLPath` function takes as input a maximum URL length, a keyword sampling strategy, and a set of keywords. The maximum URL length and keyword sampling strategy are user-configurable parameters. However, we assume that the adversary is unaware of the exact keyword set used by a given user, which may vary depending on the installed ad blocker.

To address this, we construct the keyword set by extracting keywords from EasyList [3], a widely adopted filter list shared among popular ad blockers. Specifically, we derive two different keyword sets from EasyList, each based on the keyword extraction methods used by the two most widely deployed ad blockers, AdBlock [2] and uBlock Origin [6]. This process yields 105 and 170 keywords for AdBlock and uBlock Origin, respectively.

When launching the attack, we mix these two keyword sets. For a fraction f_{ab} of the injected HTML elements, we craft URLs by passing the AdBlock keyword set to the `ConstructURLPath` function, and for the remaining elements, we use the uBlock Origin keyword set. Our intuition is that these keyword sets would substantially overlap with sets extracted using different keyword extraction methods; consequently, request URLs crafted using our keyword sets would also impose high overhead on other ad blockers. Unless otherwise stated, we set $f_{ab} = 0.5$.

The `ConstructURLPath` function begins by initializing the URL path as an empty string (Line 2) and selecting the HTML element type used for embedding the final URL (Line 3). It then iteratively selects and appends keywords to the URL path until the maximum length is reached (Lines 4–7). We elaborate later on why the attack selects the HTML type before constructing the URL path.

The `SelectKeyword` function in Line 5 samples a keyword from the keyword set according to the user-specified sampling strategy. Under the RS strategy, it randomly selects a keyword from the set. Under the GS strategy, the goal is to select the most efficient keyword—one that maximizes the number of rule evaluations relative to its character length. Among the keywords in the AdBlock and uBlock Origin keyword sets, `side` and `ads` satisfy this criterion, respectively. Accordingly, the function consistently selects these keywords under the GS strategy.

The selected keywords are appended to the current URL path via the `AppendKeyword` function in Line 6. Each keyword is prefixed with a `/` separator to ensure that ad blockers correctly extract the appended keyword from the resulting URL. For instance, when the keyword `side` is selected, the function appends `/side` to the current URL path. Assuming that the adversary uses the GS strategy with the AdBlock keyword set, the resulting URL path after the iteration in Lines 4–7 consists of repeated instances of `/side`, as shown in Figure 2.

However, while this URL path effectively triggers the evaluation of numerous blocking filters (e.g., rules associated with the keyword `side` in the figure), it is not designed to match any of them. As a result, it does not trigger the evaluation of exception filters, which is triggered only after a URL matches at least one blocking filter. To further exploit exception filters, the `AppendBlockPattern` function in Line 8 appends the string `/ads1.` to the URL path. As shown in Figure 2, this string is appended to the end of the URL path and matches one of the blocking rules in the filter list. This ensures that the URL is marked for blocking after all preceding blocking rules associated with `side` have been evaluated. As a result, the ad blocker proceeds to evaluate exception rules.

URL query string. We leverage URL query strings to bypass browser caching, thereby consistently imposing large page-loading latencies. If ad blockers block an initial request, the browser may

cache this result and reuse it for subsequent requests without matching them, which undermines the effectiveness of the attack. To avoid such browser caching, we append a random query string, which is composed of 10 alphanumeric characters, to the end of each crafted URL (see Figure 2). This ensures that each request appears unique, forcing ad blockers to evaluate every request individually.

To summarize the URL construction process, the adversary generates the final URL by combining the FQDN, URL path, and query string, following the design we have discussed so far. The adversary then generates HTML elements embedding these crafted URLs in their `src` or `href` attributes. For each generated HTML element (except stylesheets), the adversary randomly selects an element within the webpage body and inserts the new element as its sibling. In contrast, stylesheets are inserted into the head of the webpage.

HTML element type. Recall that the HTML element type is selected before constructing the URL path, as shown in Line 3 of Algorithm 1. This ordering is particularly important under the RS strategy. For example, consider the keyword `image`. While four filter rules are associated with this keyword, their optional specifications restrict these rules to apply only to image elements. If the attacker were to select the element type after constructing the URL, and the chosen type were not an image, the resulting URL would fail to trigger these rules, thus reducing the effectiveness of the attack. Therefore, we select the element type prior to sampling keywords for URL path construction.

Under the RS strategy, the `SelectElemType` function in Line 3 randomly selects an element type capable of initiating network requests, such as images, scripts, stylesheets, audios, videos, or iframes. The `SelectKeyword` function in Line 5 then samples a keyword from a set of keywords associated with filter rules that apply to the selected element type.

In contrast, under the GS strategy, the adversary consistently crafts the same URL path by repeatedly sampling the keyword `side`. Notably, all filter rules associated with this keyword apply to any HTML element type. In other words, the choice of element types for this strategy does not impact the adversary's first goal of maximizing the overhead imposed on ad blockers. Instead, element type selection in this strategy is guided by the second goal: minimizing page-loading latency for users without ad blockers.

To evaluate the impact of element type on page-loading latency, we conducted a preliminary experiment in a local network environment. We injected 1,000 identical HTML tags, each embedding the GS-crafted URL, into an empty HTML page, varying the tag type across six HTML elements capable of initiating network requests. We then deployed the generated six webpages on a web server and loaded each page five times from a separate machine using a vanilla Chrome browser with no ad blockers enabled.

Figure 3 summarizes the result. As shown in the figure, the browser processes different element types with varying latencies. When the crafted URL is embedded as a stylesheet, the page loads within 503 ms on average. In contrast, embedding the URL as an iframe results in nearly 100 seconds of load time. Based on these findings, we configured the `SelectElemType` function to always select the stylesheet type for the GS strategy.

HTML element attribute. For users without ad blockers, injecting HTML elements into the page may disrupt its functionality. To minimize such potential disruption, all crafted URLs are directed

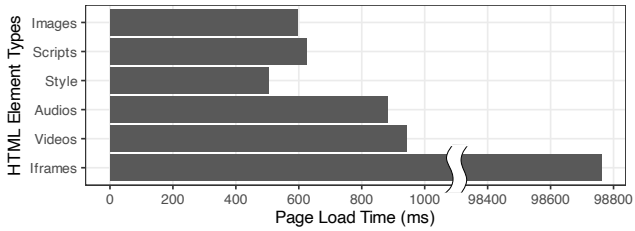


Figure 3: Average page-loading time (s) measured on Chrome after injecting 1,000 network-initiating HTML elements into an empty page while varying their types.

to non-existent endpoints. However, certain tags that affect page layout, such as `<img>`, `<video>`, `<audio>`, and `<iframe>`, can still cause apparent visual breakage, even if the associated network requests fail. To avoid this, we apply the CSS rule `display: none` to all such injected elements via the `style` attribute, ensuring that the browser hides them and preserves the page's visual integrity.

User-configurable parameters. The maximum URL length, the number of injected HTML elements, and the fraction f_{ab} of elements that embed URLs generated from the AdBlock keyword set are user-configurable parameters of this attack. The adversary can tune these parameters to balance the trade-off between the computational overhead imposed on ad blockers and the page-loading latency experienced by users without ad blockers. We evaluate the impact of varying these parameters in §5.3.

4.3 Attack II: Invoking Repeated Cookie Removal

The second attack exploits snippet filter rules designed to remove ATS-related cookies. Because cookie-removal snippet rules are domain-specific, this attack is feasible only if the filter list includes a snippet rule targeting the adversary's website. The adversary's goal is twofold: (1) to selectively impose large computational overhead on users with ad blockers by repeatedly invoking the cookie-removal JS code, and (2) to minimize negative impact on the browsing experience of users without ad blockers.

To achieve the first goal, the adversary injects multiple iframes, each loading an attack page that sets a large number of cookies with the targeted name. Upon loading, ad blockers attempt to repeatedly remove these cookies, incurring substantial overhead. Because the cookie-removal process occurs only when ad blockers are enabled, the attack selectively imposes overhead on users with ad blockers.

For the second goal, we designed our attack to use a limited number of iframe instances, ensuring that the web navigation experience for users without ad blockers remains unaffected. While injecting more iframe instances can increase the number of snippet rule evaluations and thus the overhead imposed on ad blockers, it also degrades the performance for users without ad blockers. We regard the number of injected iframes as a parameter and evaluate its impact in §5.3.

Recall that we only consider ATS publishers (i.e., first-party) as adversaries for this attack variant. For example, suppose the adversary controls the domain `news.com` and targets the snippet rule `news.com#$#cookie-remover adblock`, which may be found in publicly available filter lists [1, 5]. This snippet rule is evaluated each

time a page under `news.com` is loaded. To exploit this rule, the adversary injects multiple iframes into their webpage, each loading the same attack page under this domain (e.g., `news.com/set-cookie`).

We design this attack page as an empty page that sets a large number of cookies with the targeted name (e.g., `adblock`) via `Set-Cookie` headers, thus triggering repeated execution of the cookie-removal JS code. As a result, the injected iframes induce unnecessary page-loading delays by repeatedly evaluating the cookie-removal filters. When injecting each iframe element, it is hidden by setting its style attribute to `display: none` to minimize visual disruption. In the following, we detail how these adversarial cookies are set by the attack page.

Cookie name. To match the specified pattern, the attack cookies are named using the string specified by the snippet rule, followed by a numeric suffix. For instance, to exploit our example snippet rule, the page loaded into each iframe uses the following response header: `Set-Cookie: adblock1=1; path=/set-cookie;`. This header sets a cookie named `adblock1` with an arbitrary value. The page issues multiple instances of such headers while varying the cookie names (e.g., `adblock2`, `adblock3`, etc.). Because all cookie names include the substring `adblock`, the JS snippet attempts to remove each of them during its execution.

Cookie path. Note from the `Set-Cookie` header in the example above that each cookie is set with the path `/set-cookie`. This design choice is intended to maximize the computational overhead imposed on ad blockers. Since the JS snippets triggered by the injected iframes run in parallel, a cookie removed by one iframe may no longer exist when processed by another, thereby reducing the overall overhead.

To avoid such a case, we exploit the behavior of cookie-removing snippets used by ad blockers. To expire a cookie with a matching name, such snippets typically overwrite the cookie by setting its expiration date to the Unix epoch (i.e., `Thu, 01 Jan 1970 00:00:00 GMT`), under the assumption that the cookie was set with the `/` path. Since a cookie is uniquely identified by its name, domain, and path according to RFC 6265 [14], only cookies with the default path `/` are removed, while cookies assigned to other paths remain intact. Consequently, the adversary is able to ensure that the expiration code runs without actually removing the cookies by setting the path of adversarial cookies to `/set-cookie`.

Number of cookies. The number of adversarial cookies impacts both the computational overhead imposed on ad blockers and the functionality of the webpage. While setting more adversarial cookies increases the workload for ad blockers, it also risks webpage functionality, as browsers enforce a limit on the number of cookies per domain. For example, Chrome allows a maximum of 180 cookies per eTLD+1 [13, 52]. Exceeding this limit causes the browser to overwrite existing cookies, thus disrupting webpage functionalities.

To determine how many malicious cookies can be safely set, we analyzed cookie usage on popular websites. We visited the main pages of the Tranco Top-10K websites and counted the number of cookies set by each page. Among these websites, we successfully accessed 8,449 websites without errors.

We found that 90% of the analyzed websites set fewer than 18 cookies, and 99% set fewer than 40 cookies. The maximum observed was 67 cookies. Based on this empirical analysis, we conclude that setting 113 attack cookies is unlikely to overwrite benign cookies when launching our attack on the main pages of the Tranco Top-10K websites. We therefore use this value in our evaluation in §5. We also note that the adversary controlling their own website is able to determine an optimal number of attack cookies, potentially exceeding 113, depending on available cookie slots.

4.4 Attack III: Repeatedly Injecting Content Scripts

The last Overload attack targets how ad blockers handle blank iframes (i.e., iframes whose content is `about:blank`). Since ATS publishers may attempt to exploit blank iframes to bypass ad blockers, ad blockers inject and execute content scripts in each blank iframe, thereby enforcing snippet and element-hiding rules in these iframes as well.

For example, uBlock Origin Lite employs default snippet rules that apply across all domains. To enforce these rules, it injects a predefined set of 34 JS snippets as content scripts into every blank iframe. Similarly, to enforce element-hiding rules, it injects content scripts that dynamically monitor changes to the DOM. For instance, consider the (extended) CSS selector `div:has-text(Sponsor)`. To enforce an element-hiding rule with this selector, ad blockers use the injected content script to monitor whether any `div` element whose text content contains the string "Sponsor" is generated.

To exploit this behavior, the attack injects a large number of blank iframe instances into a target webpage, thus incurring substantial page-loading delays as ad blockers repeatedly inject and execute content scripts within these iframes. In contrast, since blank iframes do not render any visible content, they impose negligible performance overhead on browsers without ad blockers.

The effectiveness of the attack scales with the number of injected iframes. However, the browser must still process the injected blank iframes, which inevitably introduces rendering overhead. Therefore, the adversary must carefully adjust the number of injected iframes to balance the trade-off between maximizing overhead on ad blockers and minimizing performance impact on browsers without ad blockers. We treat the number of injected iframes as an attack parameter and investigate its impact in §5.3.

5 Evaluation

5.1 Experimental Setup

Implementation. All experiments are conducted using five machines. To measure the performance degradation of ad blockers, we use four identical client machines running 64-bit Ubuntu 22.04.5 LTS, each equipped with an Intel i7-8700 CPU (12 cores) and 32 GB of RAM. A dedicated web server is hosted on a separate machine.

Our attack is implemented in Python 3.8 and consists of 10K+ LoC. We use the Selenium library [57] to automate webpage visits. For the web server, we employ Flask 2.0.3 for Attack II and the Python `http.server` library for the other attack variants.

Benchmarks. For Attacks I and III, we crawl the Tranco Top-10K websites [51] as of July 2025. However, since top-ranked websites often render ads from a small number of major ATS vendors, relying solely on them could introduce bias [47, 53, 54]. To mitigate this, we select all websites in the Tranco Top-100 and randomly sample an additional 900 websites ranked between 101 and 10K. From the landing pages of these 1,000 websites, we collect their HTML files to

Table 1: Performance overhead imposed by our attack on ad blockers measured in a local network environment. Additional page load time (PLT) and CPU usage are reported in seconds.

	Chrome								Firefox							
	Attack I (GS)		Attack I (RS)		Attack II		Attack III		Attack I (GS)		Attack I (RS)		Attack II		Attack III	
	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU
AdBlock	66.94 (± 17.53)	99.86 (± 21.76)	2.35 (± 1.59)	6.28 (± 3.22)	1.63 (± 0.58)	5.54 (± 1.46)	4.97 (± 3.41)	15.01 (± 9.62)	111.55 (± 17.00)	128.27 (± 15.17)	1.73 (± 1.09)	4.03 (± 1.55)	3.67 (± 2.46)	9.72 (± 4.67)	7.14 (± 5.44)	14.97 (± 8.42)
uBO (uBO Lite)	44.12 (± 11.71)	50.36 (± 8.41)	2.39 (± 1.04)	4.06 (± 1.85)	3.64 (± 1.34)	11.47 (± 4.45)	3.09 (± 1.30)	5.68 (± 2.42)	15.99 (± 4.73)	24.54 (± 6.79)	0.77 (± 1.05)	3.10 (± 1.47)	3.85 (± 2.21)	9.23 (± 4.66)	1.34 (± 0.76)	3.16 (± 1.52)
ABP	66.55 (± 17.84)	98.96 (± 21.44)	2.18 (± 1.00)	5.86 (± 2.04)	1.60 (± 0.59)	5.42 (± 1.43)	5.10 (± 3.61)	15.09 (± 10.43)	112.54 (± 16.25)	129.33 (± 13.93)	1.76 (± 1.08)	4.07 (± 1.53)	3.63 (± 2.43)	9.80 (± 4.60)	5.37 (± 5.54)	11.38 (± 10.02)
AdGuard	42.84 (± 10.71)	51.40 (± 8.53)	2.12 (± 1.00)	5.30 (± 2.13)	5.15 (± 1.61)	17.66 (± 4.54)	4.89 (± 1.73)	11.29 (± 3.78)	3.47 (± 2.05)	12.53 (± 4.15)	3.72 (± 1.55)	10.27 (± 4.21)	6.55 (± 2.39)	18.20 (± 5.75)	4.06 (± 1.64)	12.55 (± 4.75)
ABU	43.90 (± 10.80)	50.99 (± 7.76)	2.69 (± 1.21)	5.95 (± 2.41)	1.99 (± 0.71)	7.98 (± 2.60)	7.05 (± 2.87)	20.25 (± 8.98)	> 300	> 300	21.06 (± 4.70)	26.98 (± 6.61)	1.89 (± 0.27)	6.19 (± 1.05)	1.74 (± 0.77)	3.37 (± 1.47)
w/o ad blockers	0.87 (± 1.49)	3.02 (± 2.19)	0.97 (± 0.79)	2.39 (± 1.89)	0.87 (± 0.26)	2.65 (± 0.56)	1.00 (± 0.70)	3.03 (± 1.59)	2.05 (± 1.85)	6.90 (± 2.56)	0.49 (± 1.07)	2.56 (± 1.53)	0.63 (± 0.26)	2.02 (± 0.77)	0.43 (± 0.51)	1.06 (± 0.78)

construct our benchmark dataset. We also manually verify that the selected websites are active, and their landing pages are accessible without user interaction. We then randomly sample 10% of the dataset to tune attack parameters (§5.3) and use the remaining 90% to evaluate our attacks.

For Attack II, we collect all cookie-removal snippet filters from the default filter lists used by our target ad blockers [1, 5] and crawl all websites that match these rules. This process yields 84 target websites, all of which we include in our experiments.

Target ad blockers. To demonstrate the effectiveness of our attack, we select five widely used ad blockers with 1M+ users on both Chrome and Firefox: AdBlock 6.18.0 [2], uBlock Origin (uBO) 1.63.2 [6], Adblock Plus (ABP) 4.19.0 [29], AdGuard 5.1.139 [43], and AdBlocker Ultimate (ABU) 4.0.95 for Chrome (3.8.44 for Firefox) [42]. Since uBO has been deprecated on Chrome and replaced by uBlock Origin Lite (uBO Lite), we evaluate uBO Lite 2025.4.13.1188 [7] instead of uBO on Chrome.

We install each ad blocker as a browser extension on Chrome 138.0.7204.49 and Firefox 132.0.2, using the default configuration in all cases. To avoid interference between different ad blockers, we create a separate browser profile for each ad blocker.

Metrics. We employ two primary metrics to evaluate our attacks: page load time (PLT) and CPU usage. To measure PLT, we leverage the `PerformanceNavigationTiming` interface, a JS API provided by web browsers [31]. Specifically, we compute PLT as the difference between the navigation start time (i.e., `startTime`) and the time at which all resources have been downloaded and the page has been fully parsed (i.e., `loadEventEnd`). To measure CPU usage, we record the CPU time consumed by the browser process responsible for rendering the webpage, using the `psutil` Python library.

To further evaluate the user's browsing experience, we additionally measure the first contentful paint time (FCPT) in §5.3 by leveraging the `PerformancePaintTiming` interface [32]. FCPT measures the time elapsed until the browser renders the first bit of DOM content, reflecting how quickly the page begins to display meaningful content to the user. To isolate the overhead introduced solely by injecting adversarial elements, we do not display warnings that prompt users to disable ad blockers in our experiments.

To mitigate the effect of network fluctuations in website response times, we visit each webpage five times and report the mean along with the standard deviation. We clear all caches and cookies only between visits to different websites, simulating a user's first visit followed by repeated visits to the adversary's webpage.

The adversary's goal is to minimize the overhead introduced by the attack when the ad blocker is disabled, while maximizing it when the ad blocker is enabled. To capture this trade-off, we measure performance and resource usage under three settings: (1) visiting the original page with the ad blocker disabled, (2) visiting the manipulated adversarial page with the ad blocker disabled, and (3) visiting the manipulated adversarial page with the ad blocker enabled. In other words, our attacks seek to minimize the performance difference between the first and second cases, while maximizing the difference between the first and third cases.

Environment. To minimize the impact of network fluctuations and isolate the performance degradation, we conduct the evaluation under a controlled local network environment in §5.2 and §5.3. Specifically, we visit the webpages of all websites in our benchmark dataset and save the pages along with embedded static resources. We then modify each page by launching the Overload attack and host the resulting pages on a local web server, assuming an ATS publisher as the adversary. Finally, we visit each webpage and measure the performance metrics. This experimental setting is designed to minimize performance variance caused by network conditions or routing paths on the Internet to the websites in our dataset.

5.2 Effectiveness of the Overload attack

We evaluate the computational overhead that our attack selectively imposes on users with ad blockers enabled. For Attack I, we inject 1,000 and 150 HTML elements that initiate network requests to URLs with maximum lengths of 12,500 and 10,000 characters under the GS and RS strategies, respectively. For Attacks II and III, we inject 100 and 300 iframes, respectively. We further discuss the effect of using different parameter values in §5.3.

Table 1 summarizes the performance degradation caused by the Overload attack across the three attack scenarios. The second

through the sixth rows report the additional overhead experienced by users when visiting the adversary's page with ad blockers, which the adversary seeks to maximize. The last row shows the overhead introduced for users without ad blockers, compared to visiting the original (unmodified) page. To incentivize users to disable their ad blockers, the adversary aims to minimize this overhead.

Attack I. As shown in the second column, Attack I under the GS strategy incurs additional PLTs exceeding 40 seconds for all five ad blockers when enabled on Chrome. In contrast, when the ad blocker is disabled, our attack adds less than one second of PLT, resulting in minimal impact for users without ad blockers. We emphasize that AdBlock users under this attack can improve their page load speed by up to 76.94× (= 66.94/0.87) simply by disabling their ad blocker, providing a strong incentive to do so. This trend also holds for the CPU time. The attack introduces additional CPU overhead ranging from 50.36 to 99.86 seconds when ad blockers are enabled, while keeping the overhead under 3.02 seconds when they are disabled.

Note that the GS strategy is designed to maximize the computational overhead at the expense of a higher likelihood of detection. Compared to the RS strategy, whose results are shown in the fourth and fifth columns, the GS strategy indeed imposes significantly greater overhead. For instance, against AdBlock, the Overload attack causes 28.49× and 15.90× greater increases in PLT and CPU time under the GS strategy, respectively. Nevertheless, even with the RS strategy, the adversary is still able to impose at least 2.12 seconds of page-loading delays.

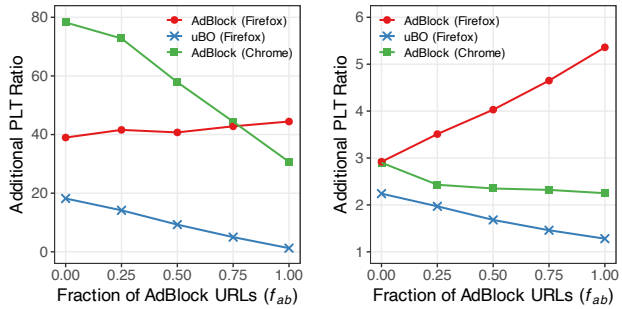
Notably, the attack results are highly consistent across the five target ad blockers on Chrome. This consistency arises because Chrome ad blockers merely supply their filter rules to the browser, which then performs the actual rule matching. As Chrome ad blockers resort to the same rule matching algorithm, our attack imposes similar computational overhead regardless of installed ad blockers.

Attack I also introduces significant page-loading delays against Firefox ad blockers, although the impact varies across ad blockers because each employs its own keyword extraction method. Under the GS strategy, the attack incurs more than 300 seconds of additional PLT against ABU, exceeding our timeout and corresponding to over a 146.34× (=300/2.05) increase compared to the case without ad blockers. In contrast, it introduces only 3.47 seconds of additional delay on AdGuard; however, it is still a 1.69× increase compared to the case without ad blockers.

These results demonstrate the general applicability and transferability of our attack. Although our attack generates URLs using keyword sets tailored to AdBlock and uBO, it remains effective against all target ad blockers—including those on Chrome—that employ different keyword extraction methods.

Attack II. As shown in the Attack II columns, our attack—which repeatedly triggers the execution of cookie-removal JS snippets—successfully introduces additional page load delays ranging from 1.60 to 6.55 seconds when ad blockers are enabled. In contrast, when ad blockers are disabled, the attack incurs only 0.87 and 0.63 seconds of additional delay on Chrome and Firefox, respectively. As a result, enabling ad blockers results in up to a 10.40× (= 6.55/0.63) increase in PLT under this attack.

Attack III. The Attack III columns summarize the results of the attack variant that triggers repeated injection of content scripts. The results demonstrate that the attack successfully achieves the adversary's two objectives. It effectively delays page rendering for users with ad blockers, introducing additional PLT ranging from 1.34 to 7.14 seconds. In contrast, when ad blockers are disabled, the attack incurs only 1.00 and 0.43 seconds of additional PLT for Chrome and Firefox, respectively.



(a) Greedy sampling (GS) (b) Random sampling (RS)

Figure 4: Ratio of additional PLT incurred by Attack I with ad blockers enabled versus disabled while varying the fraction f_{ab} of URLs generated using the AdBlock keyword set.

5.3 Effect of Parameter Selection

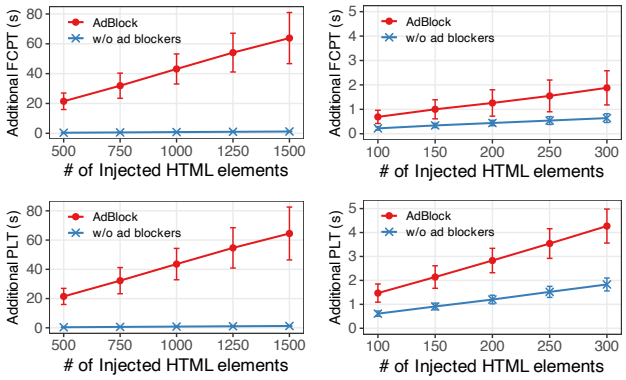
We now evaluate how adversary-controllable parameters influence the trade-off between the adversary's two objectives: maximizing overhead for ad blocker users while minimizing it for users without ad blockers. We omit CPU time results in this section, as they exhibit trends similar to page load time.

Attack I. Recall that our first attack has three parameters: (1) the fraction f_{ab} of adversarial URLs generated from the AdBlock keyword set, (2) the number of injected HTML elements, and (3) the maximum URL length.

Figure 4 shows how the effectiveness of Attack I against three target ad blockers, each employing a distinct keyword extraction method, varies with the fraction f_{ab} of URLs crafted from the AdBlock versus uBO keyword sets. We measure attack effectiveness as the ratio of additional PLT with ad blockers enabled to that with ad blockers disabled. We fix the number of injected elements and the maximum URL length at 100 and 10,000 for this experiment, respectively.

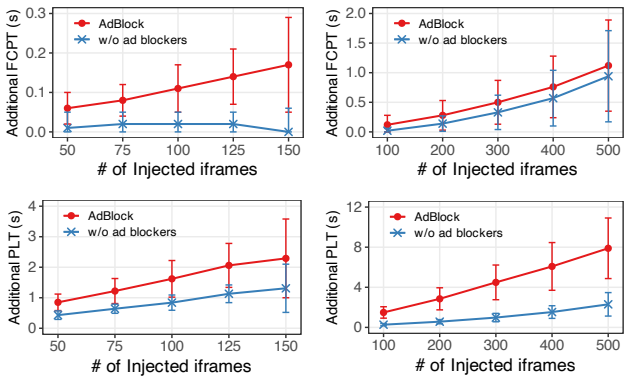
As f_{ab} increases (i.e., as more adversarial URLs are generated using the AdBlock keyword set), the attack becomes more effective against AdBlock on Firefox because the generated URLs become increasingly tailored to this ad blocker. In contrast, as f_{ab} decreases—meaning that URLs are generated more frequently from the uBO keyword set—the attack becomes more effective against uBO on Firefox and AdBlock on Chrome.

These results indicate that the most effective keyword set depends on the target ad blocker. Nevertheless, Attack I remains broadly effective across different ad blockers, incurring PLT increases ranging from 1.25× to 78.24× relative to the case without an ad blocker, regardless of the value of f_{ab}. We further note that an adversary can tune f_{ab} to target specific ad blockers—for example, by decreasing f_{ab} to more effectively target users who installed uBO on Firefox. For the remaining experiments, we generate half



(a) Greedy sampling (GS) (b) Random sampling (RS)

Figure 5: Additional FCPT and PLT measured as the number of injected adversarial HTML elements increases in Attack I.



(a) Attack II. (b) Attack III.

Figure 6: Additional FCPT and PLT introduced by the cookie-setting (Attack II) and iframe-inserting (Attack III) attacks as the number of injected iframes varies.

of the adversarial URLs using the AdBlock keyword set and the other half using the uBO keyword set (i.e., $f_{ab} = 0.5$).

Figure 5 illustrates how performance degradation changes with different numbers of inserted HTML elements. We focus this evaluation on AdBlock installed on Chrome, as it represents the most widely deployed configuration, with 61M+ users. In this experiment, we fixed the maximum URL length to 10,000. Intuitively, adding more network-initiating HTML elements raises the overhead for ad blockers. However, this also introduces additional overhead for browsers without ad blockers, as they must still process the injected elements.

The results in the figure align with this intuition. As more elements are injected into the webpage, PLT increases linearly, regardless of whether ad blockers are present. However, the overhead with ad blockers disabled grows much more slowly, while it increases rapidly when ad blockers are enabled. For example, when we inject 1,500 and 300 HTML elements using the GS and RS strategies, our attack incurs 64.50 and 4.27 seconds of additional PLTs for users with AdBlock, respectively. In contrast, it only imposes 1.26 and 1.83 seconds of overhead when ad blockers are disabled.

According to a report by Google [12], one second is considered the maximum page load delay before users are likely to abandon the page. Based on this, we identify parameter values that keep the page load delay under one second for users without ad blockers. The figure shows that injecting 1,000 and 150 HTML elements keeps the page load delay under one second for the GS and RS strategies, respectively. Hence, we adopt these values for all subsequent experiments, including those presented in Table 1.

We note that the adversary can inject a much larger number of HTML elements under the GS strategy, which contributes to improving attack efficacy. We attribute this to using only stylesheets when injecting HTML elements. As shown in Figure 3, stylesheets complete page loading faster than any other element type. As a result, the adversary has more room to inject additional HTML elements before exceeding the one-second threshold for users without ad blockers.

Figure 5 also shows that under the GS strategy, the FCPT closely tracks the PLT, whereas under the RS strategy, FCPT is substantially shorter than PLT. This difference arises because the two strategies inject adversarial elements at different locations within the page.

Recall that under the GS strategy, adversarial elements are injected into the <head> tag of the webpage. As a result, the ad blocker becomes overloaded while processing head elements, which blocks the rendering of the page body and leads to a large FCPT. Once the head is processed, the page finishes loading quickly because the body contains no adversarial elements, resulting in a small gap between FCPT and PLT.

In contrast, under the RS strategy, adversarial elements are injected as siblings of randomly selected elements within the page body. Consequently, the browser begins rendering the page normally, yielding a small FCPT, but progressively encounters adversarial elements during rendering. This causes cumulative delays in loading the remainder of the page, leading to a larger gap between FCPT and PLT. We refer readers to Appendix A for an evaluation of the impact of varying URL lengths in Attack I.

Attacks II and III. In Attacks II and III, the adversary adjusts the number of injected iframes to balance the two objectives. Figure 6 shows the impact of varying the number of injected iframes for these attack variants. As in Attack I, increasing the number of injected HTML elements leads to longer PLTs for both users with and without ad blockers. The results also show that FCPT is much shorter than PLT under these attacks. Similar to Attack I under the RS strategy, this gap arises because Attacks II and III insert iframes as siblings of randomly selected elements within the webpage body. As a result, the browser initially renders the page normally but slows down when it encounters and processes the injected iframes at random locations in the body. Since the overhead without ad blockers remains below one second when the adversary injects 100 and 300 iframes for Attacks II and III, respectively, we use these values in all experiments.

5.4 Analysis in Real-World Network

We have so far evaluated the Overload attack by statically serving real-world webpages in a local network environment to avoid the impact of network fluctuations. We now extend our evaluation to

Table 2: Performance degradation of Chrome ad blockers incurred by our attack measured in a real-world network environment. Values are reported in seconds.

	Attack I (GS)		Attack I (RS)		Attack II		Attack III	
	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU
AdBlock	66.30 (± 20.51)	117.14 (± 44.02)	2.42 (± 2.78)	5.89 (± 2.26)	1.14 (± 2.98)	8.40 (± 5.51)	4.15 (± 3.05)	12.73 (± 6.20)
uBO Lite	43.61 (± 13.14)	46.26 (± 14.48)	2.11 (± 3.21)	3.22 (± 1.66)	1.13 (± 2.31)	5.80 (± 6.20)	2.23 (± 2.36)	4.83 (± 2.65)
ABP	66.52 (± 20.45)	119.42 (± 44.00)	2.37 (± 2.78)	5.61 (± 2.28)	1.73 (± 4.72)	6.78 (± 5.43)	4.05 (± 2.51)	12.72 (± 6.44)
AdGuard	41.75 (± 13.38)	48.14 (± 16.80)	1.82 (± 1.99)	4.45 (± 1.61)	2.90 (± 3.05)	11.68 (± 4.49)	3.90 (± 2.17)	10.46 (± 4.80)
ABU	42.77 (± 13.70)	46.87 (± 15.38)	1.99 (± 2.18)	4.70 (± 1.83)	0.75 (± 2.71)	7.76 (± 4.50)	5.60 (± 3.35)	18.17 (± 8.07)
w/o ad blockers	1.94 (± 1.55)	4.23 (± 2.01)	1.55 (± 2.89)	2.23 (± 1.50)	0.23 (± 1.81)	2.13 (± 2.06)	1.51 (± 2.26)	2.98 (± 2.03)

examine whether our attack is also effective in the presence of dynamically generated network requests. To this end, we repeat the same experiment from §5.2 while simulating a real-world adversary using a proxy setup. Specifically, we employ the mitmproxy library [19] to redirect only the initial request for the main page to our server, which serves an attack-instrumented version of the page. All subsequent requests—both first-party and third-party—are sent to their original remote servers.

Table 2 summarizes the results measured against Chrome ad blockers. Despite increased variance from network fluctuations, all three variants of the Overload attack successfully imposed substantial, selective page-loading delays on users with ad blockers. For example, webpages manipulated by Attack I under the GS strategy load up to 34.29× slower. For Attacks II and III, enabling ad blockers slows page loading by up to 12.61× and 3.71×, respectively.

However, we observed that the PLT gap between enabling and disabling ad blockers was slightly smaller in this setting. This occurs because disabling ad blockers allows the browser to initiate dynamically generated ATS-related traffic, thus slowing down the page loading. Consequently, the difference in loading time between enabling and disabling ad blockers decreases, independent of our attack. These findings demonstrate that the Overload attack constitutes a practical DoS threat, though its impact may vary depending on network conditions and website characteristics. We also provide results for Firefox ad blockers in Appendix B.

5.5 Breakage Analysis

We finally evaluate whether our attack negatively affects users' browsing experience when ad blockers are disabled. Since the attack manipulates a target page by injecting adversarial HTML elements, it could potentially disrupt the original functionality of the page (e.g., sign-in functionality, search functionality, and apparent visual breakages). To assess this, we manually inspected 100 randomly selected manipulated pages for each attack. For Attack II, where only 84 target webpages were available, we inspected all of them. In total, we evaluated 384 webpages spanning Attack I under both the GS and RS strategies, Attack II, and Attack III.

In our manual analysis, only three of the 384 evaluated pages—one from Attack I under the RS strategy and one each from Attacks II and III—exhibited functionality disruptions. This corresponds to a breakage rate of only 0.78% (= 3/384). Moreover, since the adversary controls the manipulated page, they could simply relaunch the attack to avoid such cases. We therefore believe that our attack incurs only minimal functionality disruption on target webpages. In Appendix C, we provide a detailed discussion of an example visual breakage.

6 Mitigation

We propose three mitigation strategies, each targeting a specific variant of the Overload attack. We implement these mitigations in AdBlock and uBO, two of the most widely employed ad blockers. Because both ad blockers share the same implementation across Firefox and Chrome (except for Attack I), we evaluate their effectiveness only on Firefox. For Attack I, since Chrome ad blockers rely on Chrome's native rule matching algorithm, we integrate the corresponding mitigation directly into Chrome and additionally evaluate its effectiveness in Appendix D.

6.1 Early Blocking based on zlib Entropy

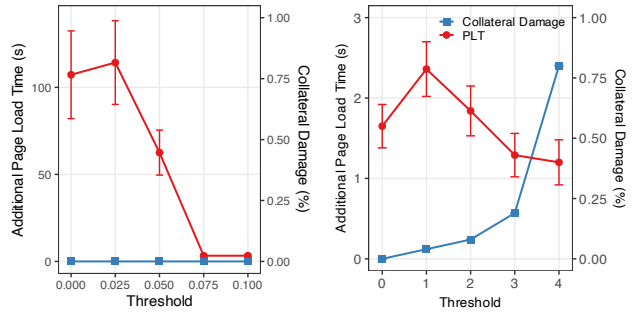
Method. To mitigate Attack I, we implement a novel blocking mechanism that prevents filter rule lookups for URLs likely to impose excessive overhead. Specifically, before the ad blocker queries the filter rule hash table with keywords extracted from a network request URL, we first assess whether the URL is likely to trigger an excessive number of filter rule evaluations. If so, the ad blocker blocks the request early, before actual rule evaluation occurs.

For this classification, we propose using the *zlib* entropy of the URL [41]. zlib entropy is computed as the ratio of compressed size to the original text length, a measure used to estimate text redundancy [17]. It provides an effective signal for detecting URLs crafted to maximize rule matches by repeatedly using a limited set of keywords. Our intuition is that such attack URLs will yield lower zlib entropy values.

Metrics. Because the proposed mitigation is designed to additionally block network requests before rule evaluation, it cannot result in side effects of permitting traffic that ad blockers would normally block. Therefore, its objective is to minimize two metrics: (1) additional PLT caused by enabling ad blockers and (2) collateral damage caused by our mitigation. We define collateral damage as the percentage of network requests originally classified by ad blockers as non-ATS resources but blocked after applying our mitigation. Such collateral damage may disrupt the page's functionality.

Result. To implement our mitigation strategy, we modified AdBlock's source code to block URLs with zlib entropy values below a given threshold. Using this modified AdBlock, we measured the two metrics on the same target pages from §5.3. Figure 7 presents the results of applying early blocking to AdBlock. When the threshold is set to zero, the browser applies no mitigation and behaves identically to the original ad blockers. For uBO, we refer readers to Appendix D, which reports similar findings.

As shown in the figure, under the GS strategy, our attack introduces an additional 107.25 seconds of PLT when AdBlock operates without mitigation. However, when the browser blocks URLs with



(a) Greedy sampling (GS). (b) Random sampling (RS).

Figure 7: Additional PLT and collateral damage after applying early blocking-based mitigation to AdBlock on Firefox.

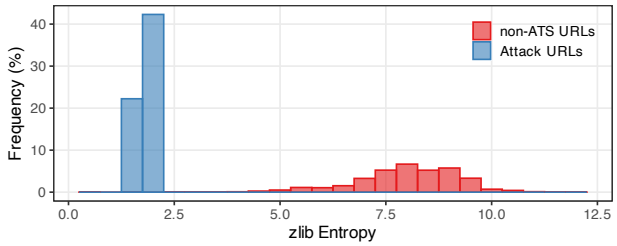


Figure 8: zlib entropy distribution of attack URLs crafted under the RS strategy and non-ATS URLs from target pages.

zlib entropy values below 0.075, the additional PLT is reduced to 3.29 seconds. While this still introduces about three seconds of delay due to the extra step of computing zlib entropy for all requests, we emphasize that this corresponds to a 32.60× reduction in overhead. Since the zlib entropies of URLs crafted under the GS strategy are 0.057 and 0.049 when constructed from the AdBlock and uBO keyword sets, respectively, our mitigation successfully blocked them. Moreover, it caused *zero* collateral damage: no non-ATS resources were blocked, as their zlib entropies were at least 1.54.

For the RS strategy, our attack initially caused 1.65 seconds of page-loading latency when the threshold was set to zero. As the threshold increased, the overhead decreased, but at the cost of incorrectly blocking some non-ATS resources. Nevertheless, our mitigation was able to lower the overhead to 1.20 seconds when the threshold was set to 4, with collateral damage limited to only 0.80%. This demonstrates that the mitigation strategy introduces only minimal side effects.

To understand how early blocking selectively blocks attack URLs, we compared the zlib entropy distributions of URLs crafted under the RS strategy with those of non-ATS URLs collected from target webpages. Figure 8 shows a clear separation between attack URLs and benign requests, highlighting the effectiveness of zlib entropy in distinguishing adversarial patterns from legitimate traffic. Since attack-generated URLs repeatedly use keywords from a limited set (i.e., 105 keywords), they exhibit notably low zlib entropy values. These results show that one can carefully tune the threshold by referencing such distributions to effectively mitigate our first attack.

6.2 Lazy Removing of Adversarial Cookies

Method. For Attack II, we modified ad blockers to postpone the repeated removal of adversarial cookies until page loading completes, thereby minimizing negative impact on users' browsing experience. This mitigation preserves the original functionality of webpages without interfering with benign cookies while still blocking adversarial cookies.

Result. We measured the PLTs of the sites from §5.3 after applying this mitigation. Without mitigation, enabling AdBlock and uBO on Firefox incurred additional PLTs of 3.67 and 3.85 seconds, respectively. With our mitigation in place, the additional delay was reduced to 1.18 ± 0.19 and 1.04 ± 0.32 seconds, respectively.

6.3 Removing Blank Iframes

Method. For Attack III, we modified ad blockers to remove blank iframes once their number exceeds a given threshold, thus preventing them from repeatedly injecting content scripts. However, this mitigation risks legitimate webpage functionality, since many sites also use blank iframes. To assess real-world usage, we collected the index pages of 8,441 websites from Tranco's Top-10K websites that were accessible without errors. We observed that 90% of these sites use no blank iframes, and 99% use fewer than six. The maximum observed was 24. Based on this distribution, we set the threshold to 24 in our experiment to avoid disrupting normal webpage functionality.

Result. We measured the PLTs of the sites from §5.3. Without mitigation, AdBlock and uBO on Firefox incurred additional PLTs of 8.06 and 1.43 seconds, respectively. With the mitigation in place, these delays were reduced to 1.42 ± 1.22 and 0.49 ± 0.22 seconds.

7 Related Work

There have been numerous attacks against ad blockers [11, 15, 26, 35, 40, 46, 49, 50, 53, 60, 61, 63, 66, 67], which have evolved in various ways to either bypass or detect them.

To evade ad blockers, Wang *et al.* [63] proposed a randomization mechanism that mutates HTML elements in webpages. Lin *et al.* [40] studied how advertisers serve ATS content from rapidly changing domains to evade filter lists. For instance, advertisers employ CNAME cloaking [26] or domain generation algorithms [11, 50] to frequently rotate their domains. Ukani *et al.* [60] reported that advertisers load ad content into blank iframes to bypass ad blockers. Nithyanand *et al.* [49] observed that advertisers inject bait ad containers and compare their rendered appearance against expected values, thereby identifying whether ad blockers are active.

To mitigate such attacks, recent studies have proposed machine learning (ML)-based ad blockers [8, 27, 36, 37, 47, 54, 64, 65]. In particular, the latest ML-based ad blockers [36, 39, 54, 55] employ features carefully engineered from the graph representation of a webpage, which are difficult for adversaries to manipulate.

In response, recent studies have suggested attacks against these ML-based ad blockers. Tramèr *et al.* [59] demonstrated adversarial attacks against perceptual ad blocking techniques, such as Percival [27]. Zhu *et al.* [67] presented A^4, the first attack framework that performs optimization-based adversarial attacks tailored to each network request. Shin *et al.* [53] further showed that a single

HTML manipulation can universally bypass ML-based ad blockers across different requests.

However, none of these works have examined whether ad blockers themselves might introduce new security threats to their users. In this paper, we presented a selective DoS attack that exploits vulnerabilities within ad blockers.

8 Conclusion

In this paper, we introduced a novel attack vector, the Overload attack, which exploits the internal optimization mechanisms of ad blockers to selectively impose excessive page-loading delays and CPU usage on users with ad blockers. Our evaluation shows that the attack is able to increase page-loading time by up to 146.34×, highlighting its effectiveness.

We also propose straightforward mitigation strategies that effectively address the presented threats and can be readily integrated into existing ad blockers. Furthermore, we responsibly disclosed our findings to Chrome and all target ad blocker developers, who acknowledged the reported vulnerabilities and committed to releasing patches. We hope that this study raises awareness within our security community about the importance of properly defending against denial-of-service threats in ad blockers.

Ethical Considerations

We responsibly disclosed all our findings to the evaluated ad blocker vendors and Chrome developers. Following standard disclosure practices, we provided detailed reports describing the vulnerabilities, their potential impact, and proof-of-concept demonstrations. By engaging directly with the affected parties, we ensured that our research contributes to improving the security and reliability of widely deployed ad blockers without placing users at risk.

Additionally, our primary evaluation was conducted in a controlled local environment (§5.2–§5.3), and each site was visited no more than five times under real-world network conditions in §5.4 to avoid disrupting normal service operations.

Open Science

We publicly release our attack and evaluation frameworks at https://github.com/WSP-LAB/Overload and https://doi.org/10.5281/zenodo.20771078, respectively. The attack framework includes proof-of-concept scripts for each attack variant, along with two demonstration videos illustrating Attack I under the greedy and random sampling strategies, respectively. The evaluation framework provides scripts for reproducing the evaluation results reported in Table 1. In addition, the Tranco Top-10K website list is publicly available at https://tranco-list.eu, and the EasyList filter list can be accessed at https://easylist.to.

Acknowledgments

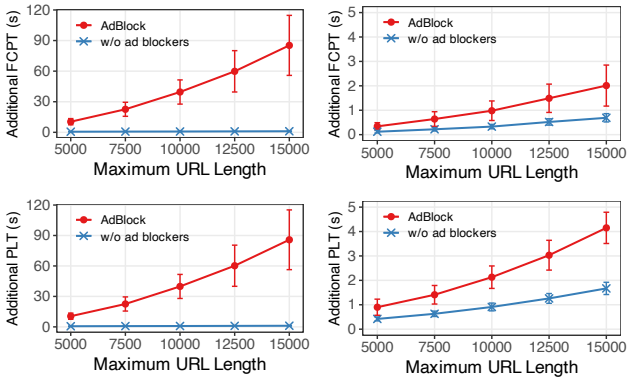
We thank the anonymous reviewers for their constructive feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2026-25470691). The authors used ChatGPT for minor grammatical corrections and manuscript refinements. All such edits were thoroughly reviewed and verified by the authors.

References

[1] 2026. ABP anti-circumvention filter list. https://easylist-downloads.adblockplus.org/abp-filters-anti-cv.txt.
[2] 2026. AdBlock. https://getadblock.com/.
[3] 2026. EasyList. https://easylist.to/easylist/easylist.txt.
[4] 2026. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.
[5] 2026. uAssets. https://github.com/uBlockOrigin/uAssets.
[6] 2026. uBlock Origin. https://ublockorigin.com.
[7] 2026. uBlock Origin Lite. https://github.com/uBlockOrigin/uBOL-home.
[8] AdblockPlus. 2018. Sentinel is Online. https://blog.adblockplus.org/blog/sentinel-is-online.
[9] Mir Masood Ali, Binoy Chitale, Mohammad Ghasemisharif, Chris Kanich, Nick Nikiforakis, and Jason Polakis. 2023. Navigating Murky Waters: Automated Browser Feature Testing for Uncovering Tracking Vectors. In *Proceedings of the Network and Distributed System Security Symposium*.
[10] Valentin Allaire. 2017. FuckAdBlock. https://github.com/sitexw/FuckAdBlock.
[11] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *Proceedings of the Internet Measurement Conference*. 230–244.
[12] Daniel An. 2018. New Industry Benchmarks for Mobile Page Speed. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/.
[13] The Chromium Authors. 2012. Chromium repository - CookieMonster. https://chromium.googlesource.com/chromium/src/+/main/net/cookies/cookie_monster.cc.
[14] Adam Barth. 2011. HTTP State Management Mechanism. https://www.ietf.org/rfc/rfc6265.txt.
[15] Muhammad Ahmad Bashir, Sajjad Arshad, Engin Kirda, William Robertson, and Christo Wilson. 2018. How Tracking Companies Circumvented Ad Blockers Using WebSockets. In *Proceedings of the Internet Measurement Conference*. 471–477.
[16] Aaron Cahn, Scott Alfeld, Paul Barford, and S. Muthukrishnan. 2016. An Empirical Study of Web Cookies. In *Proceedings of the International Conference on World Wide Web*.
[17] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2021. Extracting Training Data from Large Language Models. In *Proceedings of the USENIX Security Symposium*. 2633–2650.
[18] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. 2021. Cookie Swap Party: Abusing First-Party Cookies for Web Tracking. In *Proceedings of the ACM Web Conference*. 2117–2129.
[19] Aldo Cortesi, Maximilian Hils, and Thomas Kriechbaumer. 2026. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/.
[20] Savino Dambra, Iskander Sanchez-Rola, Leyla Bilge, and Davide Balzarotti. 2022. When Sally Met Trackers: Web Tracking From the Users Perspective. In *Proceedings of the USENIX Security Symposium*. 2189–2206.
[21] Google Developers. 2012. Content scripts. https://developer.chrome.com/docs/extensions/develop/concepts/content-scripts.
[22] Google Developers. 2021. Manifest V2 support timeline. https://developer.chrome.com/docs/extensions/develop/migrate/mv2-deprecation-timeline.
[23] Google Developers. 2023. Replace blocking web request listeners. https://developer.chrome.com/docs/extensions/develop/migrate/blocking-web-requests.
[24] Google Developers. 2024. Migrate to Manifest V3. https://developer.chrome.com/docs/extensions/develop/migrate.
[25] Google Developers. 2025. chrome.declarativeNetRequest. https://developer.chrome.com/docs/extensions/reference/api/declarativeNetRequest.
[26] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. 2021. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. In *Proceedings on Privacy Enhancing Technologies*. 394–412.
[27] Zainul Abi Din, Panagiotis Tigas, Samuel T. King, and Benjamin Livshits. 2020. Percival: Making in-Browser Perceptual Ad Blocking Practical with Deep Learning. In *Proceedings of the USENIX Annual Technical Conference*. 387–400.
[28] eyeo GmbH. 2025. Adblock Plus filters explained. https://adblockplus.org/filter-cheatsheet.
[29] eyeo GmbH. 2025. Adblock Plus: The world's #1 free ad blocker. https://adblockplus.org.
[30] eyeo GmbH. 2025. How to write filters. https://help.adblockplus.org/hc/en-us/articles/360062733293-How-to-write-filters.
[31] Mozilla Foundation. 2026. PerformanceNavigationTiming. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceNavigationTiming.
[32] Mozilla Foundation. 2026. PerformancePaintTiming. https://developer.mozilla.org/en-US/docs/Web/API/PerformancePaintTiming.
[33] Xuehui Hu and Nishanth Sastry. 2019. Characterising Third Party Cookie Usage in the EU after GDPR. In *Proceedings of the Web Science Conference*. 137–141.
[34] Fortune Business Insights. 2026. Online Advertising Market Size, Share & Growth Report, 2034. https://www.fortunebusinessinsights.com/online-advertising-

market-110252.
[35] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the Internet Measurement Conference*. 171–183.
[36] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the IEEE Symposium on Security and Privacy*. 895–904.
[37] Andrew J. Kaizer and Minaxi Gupta. 2015. Towards Automatic Identification of JavaScript-oriented Machine-Based Tracking. In *Proceedings of the ACM International Workshop on Security and Privacy Analytics*. 33–40.
[38] Amit Klein and Benny Pinkas. 2019. DNS Cache-Based User Tracking. In *Proceedings of the Network and Distributed System Security Symposium*.
[39] Kiho Lee, Chaejin Lim, Beomjin Jin, Taeyoung Kim, and Hyoungshick Kim. 2024. AdFlush: A Real-World Deployable Machine Learning Solution for Effective Advertisement and Web Tracker Prevention. In *Proceedings of the ACM Web Conference*. 1902–1913.
[40] Su-Chin Lin, Kai-Hsiang Chou, Yen Chen, Hsu-Chun Hsiao, Darion Cassel, Lujo Bauer, and Limin Jia. 2022. Investigating Advertisers' Domain-changing Behaviors and Their Impacts on Ad-blocker Filter Lists. In *Proceedings of the ACM Web Conference*. 576–587.
[41] Jean loup Gailly and Mark Adler. 2026. zlib. https://zlib.net/.
[42] AdAvoid Ltd. 2026. AdBlocker Ultimate - Block Ads, Popups & Trackers. https://adblockultimate.net.
[43] Adguard Software Ltd. 2026. AdGuard - The world's most advanced ad blocker! Get the best ad-free experience. https://adguard.com.
[44] John V. Monaco. 2022. Device Fingerprinting with Peripheral Timestamps. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1018–1033.
[45] Veelasha Moonsamy, Moutaz Alazab, and Lynn Batten. 2012. Towards an Understanding of the Impact of Advertising on Data Leaks. *International Journal of Security and Networks* 7, 3 (2012), 181–193.
[46] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. 2017. Detecting Anti Ad-blockers in the Wild. In *Proceedings on Privacy Enhancing Technologies*.
[47] Shaoor Munir, Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2023. CookieGraph: Understanding and Detecting First-Party Tracking Cookies. In *Proceedings of the ACM Conference on Computer and Communications Security*. 3490–3504.
[48] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*. 541–555.
[49] Rishab Nithyanand, Sheharbano Khattak, Mobin Javed, Narseo Vallina-Rodriguez, Marjan Falahrastegar, Julia E. Powles, Emiliano De Cristofaro, Hamed Haddadi, and Steven J. Murdoch. 2016. Adblocking and Counter-Blocking: A Slice of the Arms Race. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet*.
[50] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. 2016. A Comprehensive Measurement Study of Domain Generating Malware. In *Proceedings of the USENIX Security Symposium*. 263–278.
[51] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened against Manipulation. In *Proceedings of the Network and Distributed System Security Symposium*.
[52] The Chromium Project. 2026. CookieMonster. https://www.chromium.org/developers/design-documents/network-stack/cookiemonster/.
[53] Dongwon Shin, Suyoung Lee, Sanghyun Hong, and Sooel Son. 2024. You Only Perturb Once: Bypassing (Robust) Ad-Blockers Using Universal Adversarial Perturbations. In *Proceedings of the Annual Computer Security Applications Conference*. 190–206.
[54] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing advertising and tracking information flows for robust blocking. In *Proceedings of the USENIX Security Symposium*. 2875–2892.
[55] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proceedings of the ACM Web Conference*.
[56] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Tales of favicons and caches: Persistent tracking in modern browsers. In *Proceedings of the Network and Distributed System Security Symposium*.
[57] Simon Stewart. 2026. Selenium. https://www.selenium.dev/.
[58] Backlinko Team. 2024. Ad Blocker Usage and Demographic Statistics. https://backlinko.com/ad-blockers-users.
[59] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. 2019. AdVersarial: Perceptual Ad Blocking meets Adversarial Machine Learning. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2005–2021.
[60] Alisha Ukani, Hamed Haddadi, Alex C. Snoeren, and Peter Snyder. 2025. Local Frames: Exploiting Inherited Origins to Bypass Content Blockers. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[61] Robert J. Walls, Eric D. Kilmer, Nathaniel Lageman, and Patrick D. McDaniel. 2015. Measuring the Impact and Perception of Acceptable Advertisements. In *Proceedings of the Internet Measurement Conference*. 107–120.
[62] Binghui Wang, Tianchen Zhou, Song Li, Yinzhi Cao, and Neil Gong. 2022. GraphTrack: A Graph-based Cross-Device Tracking Framework. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*. 82–96.
[63] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention. In *Proceedings of the International Symposium on Foundations of Software Engineering*. 205–216.
[64] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. 2016. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *Proceedings of the European Symposium on Research in Computer Security*. 238–258.
[65] Zhiju Yang, Weiping Pei, Monchu Chen, and Chuan Yue. 2022. WtaGraph: Web Tracking and Advertising Detection using Graph Neural Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1540–1557.
[66] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Proceedings of the Network and Distributed System Security Symposium*.
[67] Shitong Zhu, Zhongjie Wang, Xun Chen, Shasha Li, Keyu Man, Umar Iqbal, Zhiyun Qian, Kevin S. Chan, Srikanth V. Krishnamurthy, Zubair Shafiq, Yu Hao, Guoren Li, Zheng Zhang, and Xiaochen Zou. 2021. Eluding ML-based Adblockers With Actionable Adversarial Examples. In *Proceedings of the Annual Computer Security Applications Conference*. 541–553.
[68] Sebastian Zimmeck, Jie S. Li, Hyungtae Kim, Steven M. Bellovin, and Tony Jebara. 2017. A Privacy Analysis of Cross-device Tracking. In *Proceedings of the USENIX Security Symposium*. 1391–1408.

A Effect of URL Lengths on Attack I



(a) Greedy sampling (GS) **(b) Random sampling (RS)**

Figure 9: Additional FCPT and PLT resulting from initiating network requests with varying URL lengths in Attack I.

Figure 9 shows how increasing URL length impacts the performance degradation caused by Attack I. As the figure illustrates, longer URLs result in greater overhead. This is because longer URLs contain more keywords, which trigger a larger number of rule-matching operations, leading to longer PLTs. Since the maximum URL lengths of 12,500 and 10,000 incur a page-loading delay of less than one second for users without ad blockers for the GS and RS strategies, respectively, we used these values in our attack.

B Analysis of Firefox Ad Blockers in Real-World Network

Table 3 summarizes the attack results measured against Firefox ad blockers under dynamically generated, real-world network conditions. Despite increased variance due to network fluctuations, all

Table 3: Performance degradation of Firefox ad blockers incurred by our attack measured in a real-world network environment. Values are reported in seconds.

	Attack I (GS)		Attack I (RS)		Attack II		Attack III	
	PLT	CPU	PLT	CPU	PLT	CPU	PLT	CPU
AdBlock	98.91 (± 52.75)	115.09 (± 58.67)	16.64 (± 35.79)	21.65 (± 41.98)	3.36 (± 3.48)	8.51 (± 5.43)	7.85 (± 6.67)	15.41 (± 8.70)
uBO	18.13 (± 10.47)	28.15 (± 14.66)	4.91 (± 6.35)	7.19 (± 9.63)	2.13 (± 2.30)	5.79 (± 5.68)	0.61 (± 1.91)	2.71 (± 1.51)
ABP	102.02 (± 50.41)	118.51 (± 55.47)	18.59 (± 37.51)	23.70 (± 43.39)	2.02 (± 1.62)	8.09 (± 5.49)	7.81 (± 7.27)	15.81 (± 9.34)
AdGuard	19.82 (± 23.51)	24.10 (± 24.38)	7.15 (± 6.18)	15.58 (± 7.33)	3.68 (± 3.29)	11.86 (± 5.66)	3.65 (± 3.64)	12.21 (± 4.84)
ABU	> 300	> 300	25.11 (± 14.36)	29.44 (± 16.19)	0.80 (± 2.12)	4.02 (± 3.09)	0.87 (± 2.11)	2.45 (± 1.50)
w/o ad blockers	3.58 (± 4.15)	7.35 (± 4.02)	1.44 (± 3.02)	2.93 (± 2.30)	0.66 (± 2.10)	1.28 (± 1.32)	0.45 (± 1.59)	1.02 (± 0.98)

three variants of the Overload attack successfully imposed substantial, selective page-loading delays on users with ad blockers. For example, webpages manipulated by Attack I under the GS strategy loaded over 83.80× slower. For Attacks II and III, enabling ad blockers increased PLTs by up to 5.58× and 17.44×, respectively.

C Example of Visual Breakage

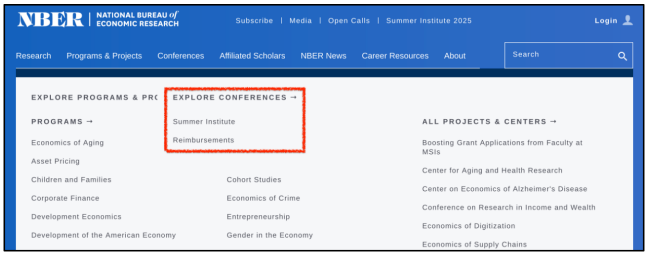
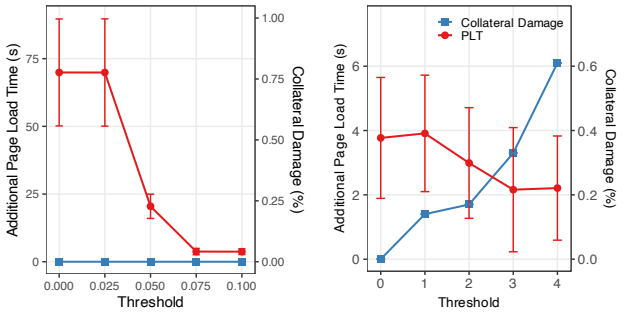


Figure 10: Visual disruption on the webpage (`nber.org`) caused by the Overload attack.

Figure 10 illustrates an example of visual breakage caused by Attack III in a real-world hosting scenario. As highlighted, the dropdown menu layout became misaligned, overlapping with other menus. This occurs because the attack statically injects adversarial HTML elements as siblings of random elements in the page. When the page subsequently manipulates its layout dynamically, these injected elements may trigger visual disruption, as shown in the figure.

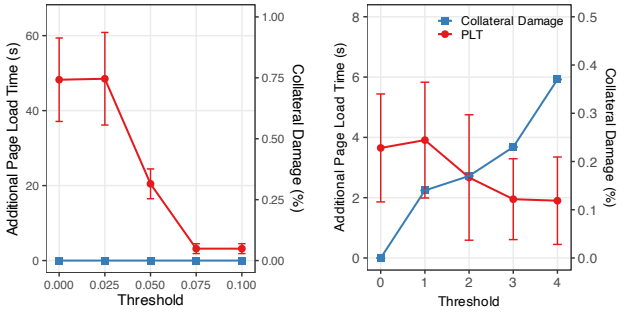
D Effect of Early Blocking on Other Ad Blockers

Figures 11–13 show PLT and collateral damage measured after applying the early-blocking mitigation from §6.1 to AdBlock on Chrome, uBO Lite on Chrome, and uBO on Firefox, respectively. These results demonstrate that our mitigation is broadly effective across different ad blockers, including those integrated with Chrome. Notably, the results in Figures 11 and 12 are highly similar, as the actual blocking of network requests in Chrome ad blockers occurs within the Chrome browser itself, which we modified to implement the mitigation.



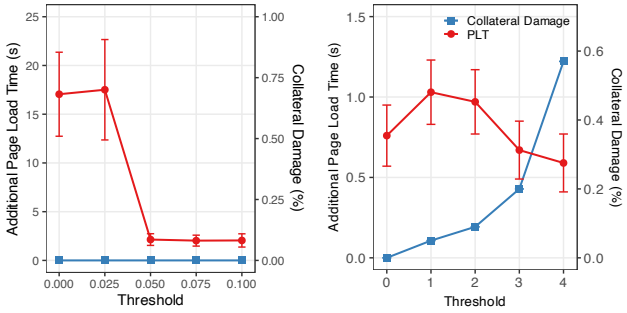
(a) Greedy sampling (GS). **(b) Random sampling (RS).**

Figure 11: Additional PLT and collateral damage after applying early blocking-based mitigation to AdBlock on Chrome.



(a) Greedy sampling (GS). **(b) Random sampling (RS).**

Figure 12: Additional PLT and collateral damage after applying early blocking-based mitigation to uBO Lite on Chrome.



(a) Greedy sampling (GS). **(b) Random sampling (RS).**

Figure 13: Additional PLT and collateral damage after applying early blocking-based mitigation to uBO on Firefox.