ShadowBlock: A Lightweight and Stealthy Adblocking Browser

Shitong Zhu
University of California, Riverside
shitong.zhu@email.ucr.edu

Umar Iqbal
The University of Iowa
umar-iqbal@uiowa.edu

Zhongjie Wang
University of California, Riverside
zwang048@ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Zubair Shafiq
The University of Iowa
zubair-shafiq@uiowa.edu

Weiteng Chen
University of California, Riverside
wchen130@ucr.edu

ABSTRACT

As the popularity of adblocking has soared over the last few years, publishers are increasingly deploying anti-adblocking paywalls that ask users to either disable their adblockers or pay to access content. In this work we propose ShadowBlock, a new Chromium-based adblocking browser that can hide traces of adblocking activities from anti-adblockers as it removes ads from web pages. To bypass anti-adblocking paywalls, ShadowBlock takes advantage of existing filter lists used by adblockers and hides all ad elements stealthily in such a way that anti-adblocking scripts cannot detect any tampering of the ads (e.g., absence of ad elements). Specifically, ShadowBlock introduces lightweight hooks in Chromium to ensure that DOM states queried by anti-adblocking scripts are exactly as if adblocking is not employed. We implement a fully working prototype by modifying Chromium which shows great promise in terms of adblocking effectiveness and anti-adblocking circumvention but also more efficient than the state-of-the-art adblocking browser extensions. Our evaluation on Alexa top-1K websites shows that ShadowBlock successfully blocks 98.3% of all visible ads while only causing minor breakage on less than 0.6% of the websites. Most importantly, ShadowBlock is able to bypass anti-adblocking paywalls on more than 200 websites that deploy visible anti-adblocking paywalls with a 100% success rate. Our performance evaluation further shows that ShadowBlock loads pages as fast as the state-of-the-art adblocking browser extension on average.

CCS CONCEPTS

• **Information systems** → **Browsers**; **Online advertising**; • **Security and privacy** → *Usability in security and privacy*;

KEYWORDS

Adblocking; Anti-adblocking; Browser Modification

ACM Reference Format:
Shitong Zhu, Umar Iqbal, Zhongjie Wang, Zhiyun Qian, Zubair Shafiq, and Weiteng Chen. 2019. ShadowBlock: A Lightweight and Stealthy Adblocking Browser. In *Proceedings of the 2019 World Wide Web Conference (WWW'19), May 13–17, 2019, San Francisco, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3308558.3313558

WWW '19, May 13–17, 2019, San Francisco, CA, USA

ACM ISBN 978-1-4503-6674-8/19/05.
https://doi.org/10.1145/3308558.3313558

1 INTRODUCTION

The deployment of adblocking technology has been steadily increasing over the past few years. PageFair reports that more than 600 million devices globally use adblockers as of December 2016 [1]. Many reasons contribute to the popularity of adblocking. First, lots of websites show flashy and intrusive online ads that negatively impact user experience. Second, the pervasiveness of targeted or personalized ads has incentivized a global ecosystem of online trackers and data brokers, which in turn raises concerns for user privacy. Third, the inclusion of numerous advertising and tracking scripts causes excessive website bloat resulting in slower page loads. The rise of adblocking has jeopardized the ad-powered business model of many online publishers. For example, U.K. publishers lose nearly 3 billion GBP in revenue annually due to adblocking [2].

In response to adblocking, many publishers have deployed JavaScript-based, client-side anti-adblockers to detect and circumvent adblockers. An anti-adblocker typically consists of two components: detection and reaction. For adblocker detection, common practices include checking the absence of ad elements and proactively injecting bait ad elements [33]. Both practices exploit the fact that adblockers make observable changes to the DOM by either blocking relevant requests or hiding DOM elements directly [37]. As a result, these DOM changes can be perceived by the detection part of anti-adblockers through invocation of JavaScript APIs such as `getElementById()`. After adblocker detection, the reaction component can perform different subsequent operations. It can be aggressive paywalls that prevent users from accessing the content or even switching ad sources.

Adblockers have addressed anti-adblockers in one of the following three ways: (i) blocking the JavaScript code of anti-adblockers using filter lists [29], (ii) disrupting anti-adblocker code based on program analysis [39], and (iii) hiding the trace of adblocking to fool anti-adblockers [37]. The first countermeasure is currently adopted by the adblocking community using filter lists such as Anti-Adblock Killer and Adblock Warning Removal [29]. However, the coverage and accuracy of these filter list is lacking. Our manual evaluation on 207 websites using anti-adblockers with visible reactions (i.e. warning message or paywall), only less than 30% of them are correctly identified by Adblock Warning Removal [4] or Anti-Adblock Killer [5]. This is likely due to the manual nature of filter list curation and maintenance which is cumbersome and error-prone. The second countermeasure of rewriting JavaScript to deactivate anti-adblockers is prone to false positives causing site breakage with unacceptable user experience degradation [39]. The

third countermeasure of hiding the trace of adblocking, as implemented in prior work [37], is not stealthy because it injects new JavaScript which is easily detectable by anti-adblockers.

In this paper, we aim to completely hide the traces of adblocking in a stealthy manner by going deep into the browser core. This is analogous to the rootkits in the OS kernel where user applications are unable to detect the presence of a malicious process [36]. Since the browser core is at a lower level (more privileged), it is in theory capable of hiding the states of adblockers from the anti-adblockers while presenting an ad-free view to the user. Specifically, since anti-adblocker is implemented as client-side JavaScript, it can only access web-page states through a number of predefined Web APIs including the ones used to probe the the presence/absence of ad elements. These APIs are standardized by W3C and implemented eventually by web browsers. ShadowBlock hooks any JavaScript API that can potentially distinguish the difference before and after hiding ad elements, and assures no information about the element hiding can be leaked through such API.

We tackle three major challenges in designing and implementing ShadowBlock. First, existing adblockers' model of blocking ad-related URLs (e.g., scripts, iframes, images) does not fit well in our requirement of presenting the same exact DOM view as if no adblocker is employed. For example, if a DOM element is not even retrieved, ShadowBlock would have no way to fake its size, dimension, and other properties. Second, given that the Web APIs suite and the rendering process implemented in modern web browsers are highly complex and intertwined, there may exist unexpected channels that leak information about adblocking deployment. To achieve stealthy adblocking, we need to ensure that no such channel discloses differentiable information about our element hiding action in a conclusive way. Third, modern web browsers make significant efforts in improving their page loading and rendering performance. As we develop ShadowBlock on open-sourced Chromium, we need to minimize the overhead it incurs during the page load process.

Contributions. We summarize our key contributions as follows.

(1) We design a well-reasoned solution where we present two different views to anti-adblockers and users. On one hand, ad elements are never directly blocked (so they remain visible to anti-adblockers); on the other hand, these ads are stealthily hidden from users.
(2) We reuse the rules from filter lists used by adblocking browser extensions to element hiding decisions. On top of existing lists that are community-backed and have been widely adopted, we replicate 98.3% of their ad coverage according to manual inspection over Alexa Top 1K websites, with less than 0.6% breakage rate.
(3) We design and implement a fully functional prototype which is open-sourced at the time of publication. We evaluate the effectiveness of ShadowBlock prototype on 207 websites with visible anti-adblockers. We pick real anti-adblockers with different trigger mechanisms and of different complexity. All of them are successfully evaded by our new adblocker design.
(4) Our performance evaluation of ShadowBlock shows that it loads pages comparably fast as Adblock Plus on average, in terms of Page Load Time and SpeedIndex.

2 BACKGROUND AND RELATED WORK

2.1 Adblockers And Filter Lists

Mechanisms. Adblockers rely on manually curated filter lists to identify ads on web pages. EasyList and EasyPrivacy are two most widely used filter lists to block online ads and trackers, with about 71000 and 15000 rules [8], respectively. These lists consist of two types of rules which are basically regular expressions. One of them is HTTP rules that block HTTP requests to URL addresses that are known to serve ads. For example, the first filter rule below blocks all third-party HTTP requests to amazon-adsystem.com, preventing any resource on this domain from being downloaded. The other type is HTML rules, which generally hides HTML elements that are identified as ads. For example, the second filter below hides all HTML elements with ID `promo_container` on wsj.com.

```
||amazon-adsystem.com^$third-party
wsj.com###promo_container
```

It is noteworthy that HTML rules are mostly only introduced to complement HTTP rules while dealing with first-party text ads. This is because these text ads are directly embedded into the HTML itself with no associated additional resource loads, making it inevitable to be included while the browser downloads the web page. Otherwise, HTTP rules are preferred as they prevent ad resources from being loaded in the first place, saving unnecessary network traffic and avoiding execution of ad-related scripts to speed up page loading and rendering.

Limitations. A group of volunteers that maintain filter lists carry out the manual process to add new rules, correct and remove erroneous or redundant rules all based on informal feedback from users [7]. Due to its crowd-sourced nature, this laborious effort faces challenges from both the completeness and soundness. In the context of adblocking, the former results in missed ads and the latter often translates to site breakage or malfunction [23]. At the same time, as adblockers gain their popularity rapidly (11% of global Internet population is blocking ads as of December 2016 [1]), online publishers are also fast adopting countermeasures against adblockers that we summarize below.

2.2 Countermeasures Against Adblockers

Concealing ad signatures. First, online advertisers can bypass rules on filter lists by concealing the signatures these lists use to identify ads. At a high level, this line of countermeasures attempts include first-party advertising and rotation of ad serving domains. First-party advertising exploits the fact that many rules on filter lists are designed to block ads from being loaded from third-party servers. Instead, ads are served from the same domain of the web page hosting them and their nature as ads is concealed as normal content [11]. However, adblockers can easily hide any HTML element on a web page by applying HTML rules that are crafted to target elements based on any combination of their CSS/HTML properties. In other words, any CSS selectors used to create and/or

locate ads elements, can also let adblockers identify and hide these elements in turn.

Domain name rotation is another tactic for obfuscating advertising content. It relies on ad rotation networks that serve ads from frequently-changing, or even automatically generated [38] domain names, which overwhelms volunteers that maintain the filter lists so they are hard to keep pace with the rule updates. This can result in, however, the indifferent blocking of all third-party resources on websites that show such ads [9]. By only whitelisting legitimate scripts that support core functionalities, any possible ads or tracking JavaScript are prevented from running, leaving no chance for loading domain rotating ads. Moreover, AdBlock Plus recently launched its Anti-Circumvention Filter List [3] that specifically counter "circumvention ads", including ads adapting the two countermeasures above.

Deploying anti-adblockers. Second, many publishers choose to deploy anti-adblocker JavaScript code to battle the rise of adblocking. Specifically, such client-side scripts consist of two main components, *trigger* that detects the presence of adblockers by checking whether ads or bait elements are still present, and *reaction* that can display warning messages and/or simply report the results to a remote server [33, 39]. Prior work [29] showed that 686 out of Alexa top-100K websites detect and visibly react to adblockers on their homepages. Even worse, these visible ones only account for less than 10% of all anti-adblockers [39]. Zhu et al. [39] showed that among Alexa top-10k websites, 30.5% are countering adblockers in some form, with most of them being silent reporting. In summary, because of their flexibility and ease of deployment, anti-adblockers are considered the most widely used countermeasure against adblockers adopted by online publishers.

2.3 Countermeasures Against Anti-adblockers

Dedicated anti-adblocking filter lists. As a response, adblockers attempt to circumvent anti-adblockers by blocking their JavaScript code snippets, whitelisting bait scripts/elements, or hiding warning notifications. To this end, adblockers once again rely on manually curated filter lists such as Anti-Adblock Killer [5] and Adblock Warning Removal [4]. These lists either trick anti-adblockers' trigger so they cannot detect adblockers, or mute their reaction component to prevent responses after successful detection.

```
/kill-adblock/js/function.js$script
@@||removeadblock.com/js/show_ads.js$script
ilix.in,urlink.at,priva.us###blockMsg
```

For example, the first HTTP rule above blocks the code snippets containing implementation of an anti-adblocking library `KillAdBlock`, and the second whitelists a bait script file named `show_ads.js` that is used to detect adblockers. The third HTML rule hides the warning message with ID `blockMsg` issued by the associated anti-adblocker. However, our manual evaluation (§4) shows that these lists targeting anti-adblockers are generally ineffective. Only less than 30% of the anti-adblocking warning messages can be removed by the state-of-the-art filter lists. This is again partly because of the crowdsourcing nature of these lists, and also the rising popularity of third-party anti-adblocking services that deploy sophisticated techniques dedicated for detecting/circumventing adblockers [33].

Disrupting anti-adblocker code. Other than the filter lists that have been officially adopted by adblockers, there are also research efforts for detecting and evading anti-adblockers. One solution to measure the anti-adblockers is to perform program analysis techniques that automatically determine if a script functions for anti-adblocking purposes. Such analysis can be static that is based on syntactic and structural features extracted from JavaScript code, and utilizes machine learning approaches to classify the code from ground-truth-labeled training data [29]. It can also be dynamic that captures JavaScript behavior at runtime by collecting and analyzing differential execution trace with the adblocker turned on and off [39]. After successfully pinpointing the critical conditions that are used by anti-adblockers to assert/react against the presence of adblockers, one can choose to rewrite these conditions to prevent the anti-adblockers from functioning. This approach is generally intrusive (patching Javascript can be tricky and cause breakage) and easy to evade. Indeed, the overall success rate of this strategy is shown to be only less than 80%.

Hiding adblockers. Besides disrupting the functionalities of anti-adblockers, researchers also have proposed a way to hide the trace of using adblockers, or known as *stealthy adblocking*. In [37], Storey et al. created a shadow copy of the DOM that anti-adblockers operate on before any adblocking actions take place, and then redirects all JavaScript APIs (e.g. `getElementById()`) that can be used to detect the presence of ad elements to the copy instead of the original DOM. However, this so-called rootkit-style stealthy adblocker has inherent drawbacks. First, unless it lives in browser core and with significant engineering efforts, the underlying DOM mirroring and propagation are difficult to be complete in all cases. This is especially problematic in the context of web browsing as any site breakage causes unacceptable user experience degradation. Second, even with a perfect implementation, maintaining a live copy of complicated data structures such as DOM poses a prohibitively high overhead onto the rendering performance of modern web browsers. Given that modern browsers place significant emphasis on performance, heavy operations like such at runtime are generally not acceptable.

3 SHADOWBLOCK

In this section, we first provide an overview of ShadowBlock's architecture. We then discuss ShadowBlock's two building blocks: (1) the identification of ad elements by translating filter list rules to per-element hiding decisions and (2) the concealment of our hiding actions. Finally, we summarize the modifications we make in the relevant modules of Chromium.

3.1 ShadowBlock Overview

Figure 1 illustrates ShadowBlock's architecture. It consists of two sub-systems: one translates rules from filter lists and use them for identifying ad elements in DOM to hide; the other hooks necessary points in Chromium to ensure that the hiding actions are transparent to the trigger/detection component of anti-adblockers. Recall from Section 2.1 that filter lists contain tens of thousands of rules that either block HTTP requests to fetch ad resources or hide HTML ad elements. To prevent exposing adblocking actions to anti-adblockers, we need to hide the changes in DOM or other states (e.g. resource loads) introduced by adblocking because these changes

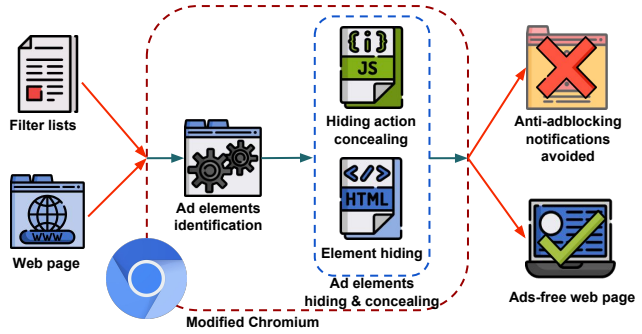


Figure 1: Architectural overview of ShadowBlock

can be detected by anti-adblockers through JavaScript APIs such as `getElementById()`. In order to do so, we first allow all HTTP requests to proceed, then mark any element that results in ads, and subsequently hide the marked elements. It is important that we allow these elements to be retrieved so when the anti-adblocking script queries the state of the element, ShadowBlock will be able to generate valid responses (e.g., dimension of the element).

3.2 Identifying Ad Elements

Next, we explain our approaches for marking different types of ad elements. In general, there are two types of elements: (1) those that are statically embedded in the HTML, and (2) those that are dynamically created by JavaScript. First, some ad elements are statically embedded in the HTML, including ad images, iframes, media files (i.e. video and audio). Such ad elements can be typically identified by matching against the HTML rules on filter lists. Second, ad scripts usually *create* new ad elements that display advertising content. These dynamically created elements should be identified, marked, and hidden.

Ad elements loaded statically. ShadowBlock does not need to do anything special for such elements. Ad filter lists already contain extensive rules that cover them. For example, the rule `||googlesyndication.com/safeframe/` in EasyList blocks the HTML file from https://tpc.googlesyndication.com/safeframe/1-0-23/html/container.html on site cnbc.com, which prevents a container frame used by Google Ads from being loaded. To hide such an ad element, ShadowBlock needs to first identify the iframe element with the blocked URL as its source attribute and then hide it.

Alternatively, Easylist rules may hide ads based on element properties (e.g., element id). We simply reuse these rules to match elements in the page, and mark them accordingly.

Ad elements generated dynamically by ad scripts. There are generally two cases. The easy and the hard. For the easy case, the dynamically generated ads may contain URLs or ids that already show up on Easylist. This allows us to directly mark them as ads using very much the same strategy as mentioned above.

For the hard case, the dynamically generated elements are not on Easylist. This is because it is assumed that ad scripts are blocked upfront and therefore there is no need to block the elements generated by them. In ShadowBlock, in contrast, we need to allow ad scripts to load and execute, which mandates us to track such elements.

To identify elements dynamically created by ad scripts, we need to attribute each element to the script that created it. More formally, we can define this process of attribution as tracking the *data provenance* of HTML elements using taint analysis. Note that there are in general two types of element creation that can be initiated by a script: (1) control-flow-based creation, in which the script directly invokes JavaScript API (e.g. `createElement(tagName)`) and only propagates data from itself into the new element; and (2) data-flow-based creation, in which the script uses data from sources other than itself into the new element (e.g. `createElement(fetchTagNameFromServer())`). Dynamic taint analysis [31] can accurately track the data provenance through taint propagation for both types. Simply put, taint analysis involves *taint source* (where data comes from), *taint sink* (where data ends), and *propagation policies* that define how that tainted data are propagated through the program execution. In our case, all data derived from an identified ad script as "tainted" (i.e., data downloaded through an ad URL, or retrieved/generated by an ad script), then such data can be tracked standard taint analysis propagation policies for JavaScript (e.g., [27]). Finally we will hide any tainted HTML elements (i.e. taint sinks).

Unfortunately, such dynamic taint analysis at runtime usually incurs significant overhead for web browsing [31]. In addition, we argue that the cases where taint analysis will be required are limited. Specifically, even if ad elements take dynamically fetched data, e.g., `createElement(fetchTagNameFromServer())`, they are most likely created by ad scripts. This is sufficient for us to mark the element and hide it (irrespective of what data are actually fetched from the server). The reasoning is that if we consider extension-based adblockers as our baseline, the dynamically created element wouldn't even exist in the first place (as the script as a whole would have been blocked). Based on the above, we devise a simple technique which we call *execution projection* using the call stack information extensively (which are used for various other purposes as well [30, 32]). At a high level, we maintain an "execution stack" that keeps track of what scripts are being executed at any given time point, and mark an element as ad if there is *any* ad script in the stack when the element is being created. For example, consider a simple ad script `ad_loader.js` in Code 1.

```
1 var ad_img = document.createElement("img");
2 ad_img.src = "https://some_ad_publisher.com/ad.jpg";
3 document.body.appendChild(ad_img);
```

Code 1: Example ad script `ad_loader.js`

In this example, at the time of the image element creation, `ad_loader.js` would be at the top of the execution stack because it is being executed.

After attributing elements to ad scripts, which are identified using filter lists, we can mark all ad elements and hide them accordingly. We illustrate this projection/marking process in Figure 2. In more complex cases, there can be many scripts in the executing stack, because code in one script can invoke functions in other scripts, and so on. More importantly, ad scripts can invoke non-ad libraries (e.g. jQuery) to create ad elements so the top script in stack at the time of element creation is not necessarily ad script. To

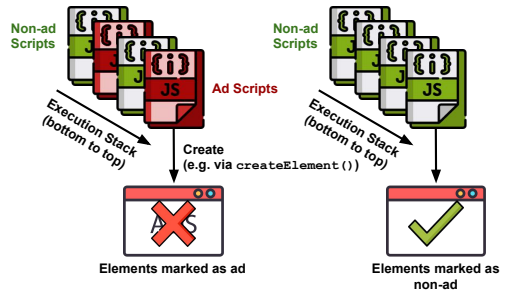


Figure 2: Execution projection for marking script-created ad elements

tackle this challenge, we need to scan the entire execution stack. If there is any ad script in the execution/invocation chain, ShadowBlock should mark the element being created as ad. This is because any script (ad-related or not) invoked by a known ad script should never have been executed in the first place, given that adblocking extensions simply block the whole ad script altogether.

Note that our approximate solution does not handle a special case when an element is created via a JavaScript API that gets overridden (hooked) by an ad scripts . Since client-side JavaScript is allowed to override (i.e. injecting code containing a callback to its own) arbitrary JavaScript API functions (e.g. `createElement()`) with its own version, we would see ad script in the stack when an element is being created via the API overridden by an ad script. In this case, we may mistakenly mark a non-ad element as ad when the overriding ad script does not propagate any data into the newly created element. To tackle this issue, at the time of element creation, we would need to further check whether the code injected by the overriding ad script alters the element. If it alters the element then we mark the element as ad, and non-ad otherwise.

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  report_statistic_to_server(new_elem);
  return new_elem;
};
```

Code 2: API overriding with the element intact

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  ad_elem = change_to_ad_elem(new_elem);
  return ad_elem;
};
```

Code 3: API overriding with the element altered

3.3 Stealthily Hiding Ad Elements

After identifying ad elements, ShadowBlock needs to stealthily hide them so as to not leaving its trace to anti-adblockers. Next, we discuss how we realize such stealthiness through CSS property access API hooking.

Choice of hiding mechanism. We first need to decide how to hide ad elements within Chromium. Given the complexity of modern web browsers and API standards, we can hide an HTML element in several different ways. To better understand different hiding mechanisms, we illustrate Blink's rendering process in Figure 3 [16]. Blink's rendering path, from parsing an HTML file to the pixel display on user's screen, can be summarized in the following phases.

(1) **Parse** flat HTML and CSS in plain-text to DOM and CSS Object Model (CSSOM) in tree structure.
(2) **Combine** DOM and CSSOM to Render Tree, which captures all the visible DOM content and all the CSSOM style information for each node.
(3) **Paint** the rendered pixels to user's display according to Render Tree.

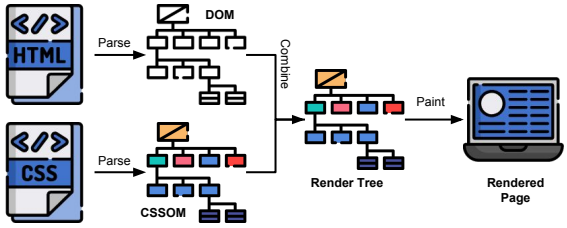


Figure 3: Rendering Path for Blink

In theory, each of these 3 phases contain APIs/modules that we can leverage for hiding an HTML element (or in other words, preventing it from being rendered). We outline different possible strategies to hide HTML elements below:

(1) **DOM/CSS layer**: (i) remove the element from DOM; (ii) set the element's style to `display:none`, `visibility:hidden` or `opacity:0`
(2) **Render Tree layer**: remove the `LayoutObject` (i.e. a styled node) from Render Tree
(3) **Paint layer**: prevent the region in pixels associated with the element from being painted

Note that generally, the higher-layer (i.e. closer to DOM/CSS layer) we tweak around in the hierarchy, more engineering effort is required for ensuring its stealthiness against anti-adblockers, while narrower gap we have to bridge for translating the identified ad HTML elements to the data structure corresponding to that layer (e.g. CSS properties for DOM/CSS layer, `PaintBlock` for Paint layer etc.). In contrary, the lower-layer (i.e. closer to Paint layer) our modifications reside, fewer unexpected channels there are that can potentially leak the hiding activities, but at the same time more engineering efforts are required for translating the identified ad elements to that layer's data structure.

After investigating different possibilities, we find the CSS property `visibility:hidden` achieves the most suitable trade-off for our objective. It persists in phase (1), functions in phase (2) of the rendering path, and eventually affects both Render Tree and painted/rendered page in phase (2) and (3). On one hand, `visibility` is a property associated with every HTML element so we can easily identify after marking its effective element as ad, without the need of further tracing. On the other hand, unlike `display:none`, it by design preserves the space taken up by the hidden element so it causes no side-effect to the layout of the document, minimizing the impacted points that need to be hooked for covering the hiding action. Figure 4 shows the visual difference between their respective effects. Compared to `opacity:0` that also preserves the occupied

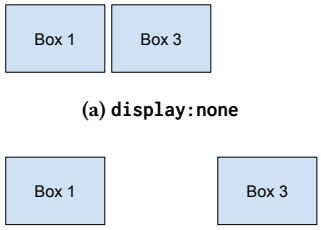


Figure 4: Comparison of toggling different CSS properties (Box 2 is hidden)

space, `visibility` is a categorical CSS property instead of numerical as `opacity` so its implementation in Chromium is considerably less complex, simplifying our hooking logic as well.

Hooking for concealing hiding actions. After hiding the target ad element by setting its `visibility` CSS property value [21] to `hidden`, we next need to cover any traces that result from this change of value and can be detected by anti-adblockers. Since anti-adblocking scripts are client-side JavaScript code, which can only access the change of states happened in the page through JavaScript Web APIs [22], we search through the source code of Chromium, analyze relevant modules and locate the following three categories of APIs that are impacted by the `visibility` CSS property:

- **CSS/Style-related**: changing a CSS property value immediately affects its own return value to JavaScript APIs via `getComputedStyle()`. We hook this value to `visible` to fool anti-adblockers. Fortunately, for our case, setting `visibility:hidden` preserves the space the element occupies, so it does not collapse the page layout nor affect other relevant CSS properties such as `offsetHeight/offsetWidth`.
- **Event-related**: flipping the `visibility` property prevents an element from receiving some DOM events, such as `onfocus`. Since anti-adblockers can leverage these events as a side-channel to infer the real visibility of an element, we hook relevant modules in Chromium so hidden elements can receive these events just like visible elements.
- **Hit-testing-related**: another effect of setting `visibility:hidden` is Blink treats elements with it as inapplicable to Hit Testing, which checks if an element is clickable by users in their viewpoints. This removes the hidden element from return values to APIs like `elementFromPoint()`, which can potentially be used by anti-adblockers to differentiate hidden elements from visible ones. We hook relevant modules in Blink to cover it.

Since all JavaScript APIs, to our best knowledge, directly or indirectly rely on the modules above to determine an element's visibility, we ensure the completeness of our hooking against potential information leakage to client-side anti-adblockers. It is worth noting that to avoid affecting existing `visibility:hidden` CSS property value, in Chromium we create a new `visibility:fake-visible` value that completely mimics what `visibility:hidden` behaves, except for the points that are intentionally hooked. Moreover, since `visibility` is a CSS property that inherits from parent node to child nodes, we can make the identified ad elements invisible to user's display even though we only identify the top-most element as ad according to filter lists.

3.4 Chromium Modification

Next, we describe our modifications to Chromium for implementing ShadowBlock. We start with a brief introduction of Chromium's architecture, then move to the instrumentation we use for identifying ad elements, and lastly discuss the modules we leverage for hiding identified ad elements and hook for eliminating the traces resulted from the hiding action. We implement the prototype of ShadowBlock with 1307 LOC (1265 LOC addition and 42 LOC deletion) in C++ on top of Chromium [1]. Note that we re-use `SubresourceFilter` [18] and `libadblockplus` [10] for parsing filter lists with production-level robustness.

Chromium Architecture. Chromium's rendering engine is called Blink and its JavaScript engine is called V8. Blink [6] is responsible for fulfilling the rendering path shown in Figure 3, in which its core module renders all HTML elements and handles their `visibility` CSS properties we use for hiding ad elements. V8 [19] handles the compilation and execution of all JavaScript code, including the ad scripts ShadowBlock needs to identify and anti-adblocker scripts that intend to detect our hiding action over ad elements. Blink has a bindings module to handle interactions between rendering and JavaScript execution. Rendering related script tasks are passed through bindings module. For example, JavaScript API calls such as `getComputedStyle()` are handled through the bindings module.

Instrumentation for identifying ad elements. As discussed in Section 3.2, there are three types of ad elements ShadowBlock needs to identify.

First, for ad elements generated by ad scripts, we rely on execution projection. In Chromium, we leverage `blink::SourceLocation::Capture` to capture the full `v8::v8_inspector::V8StackTrace` that includes the entire JavaScript call stack[2] at any given time point. It serves as the underlying stack tracing mechanism for V8's debugger/inspector, and has therefore been optimized with low overhead [20].

Second, we instrument the constructor of `blink::Element` class in Blink, which captures the earliest point of creation event for all HTML elements. Because of V8's single-threading nature, we can safely associate the current stack trace to the element creation event, and scan the stack to match the scripts in it against filter lists. If it is a match, we then mark the element as ad. Additionally, we instrument the `DispatchWillSendRequest` event in both `blink::FrameFetchContext` and `blink::WorkerFetchContext` to intercept the point when an ad script loads another script, and mark the loaded script as ad script. By adding such loaded ad scripts to a set, we match the stack trace against them as well at element creation, ensuring we can mark all ad elements.

Third, for elements loaded with resources that match rules in the filter lists, we intercept the `AttributeChanged` event in `blink::`

[1] We open source our implementation at https://github.com/seclab-ucr/ShadowBlock to allow reproducibility as well as help future extensions by the research community.
[2] We admit that there are cases where asynchronous tasks are not correctly captured by the default V8 call stack trace. However, we argue this incompleteness only translates to very limited number of missing ads according to our manual evaluation in Section 4.2.

`Element` and match the URL against filter lists, if it is a match we mark this element as ad. For element hiding rules, we adopt `libadblockplus` [10], a C++ wrapper library around the core functionality of Adblock Plus to parse filter lists and generate the CSS selectors for matching ad elements for a particular domain. Then, we mark the ad elements that match the generated CSS selectors by calling `ContainerNode::QuerySelectorAll()`.

Since many web pages are dynamic due to JavaScript execution over time, we also need to monitor attribute changes of each element. For this purpose, we instrument the `AttributeChanged` event and match any element with newly changed attributes against CSS selectors from HTML rules. We mark an element as ad if it is a match, or un-mark the element if this element has been marked but it is not matched this time. Note that in order for minimizing the number of matches needed to perform, we conduct the first batch match (via `QuerySelectorAll()`) after the `load` event of DOM is fired, and then match elements upon their attribute changes. This design choice leaves a short period of time (few milliseconds) between page navigation and `load` DOM event in which ads are displayed. We make this trade off to reduce the overhead incurred by `QuerySelectorAll()`. In comparison, adblocking extensions such as Adblock Plus inject CSS rules when `document.readyState` turns `interactive` [26], which happens before the `load` event. However, it is important to note that most ads in current web ecosystem are loaded in an asynchronous manner and are unlikely to appear before the `load` event in first place.

Stealthy modifications for hiding ad elements. As mentioned earlier, we leverage `visibility` CSS property to hide identified ad elements by creating a new `fake-visible` enumerate and visually hide elements with this enumerate, as if it behaves as `hidden`. In the meantime, we hook relevant modules in both Blink and its bindings with V8 to ensure the stealthiness of our hiding action. More specifically, for eliminating traces accessible by CSS/Style-related APIs, we hook `CSSComputedStyleDeclaration::GetPropertyCSSValue` in Blink and force return `visible` to queries about hidden elements. For event-related APIs, we hook `Element::IsFocusableStyle()` and other conditions that determine if an element can receive events. Lastly, we hook `ComputedStyle::VisibleToHitTesting()` so ad elements are still regarded "visible" from the viewpoint perspective of Blink. In principle, our hooking guarantees that the identified ad elements are invisible to user's display as pixels on screen but appear as visible to APIs accessible to client-side JavaScript.

4 EVALUATION

We evaluate ShadowBlock in terms of its (1) stealthiness against anti-adblockers, (2) ad coverage, and (3) performance as as compared to adblocking extensions.

4.1 Stealthiness Analysis

Takeaway: ShadowBlock has 100% success rate against anti-adblockers whereas state-of-the-art anti-adblocking filter lists have only 29% success rate.

Experimental Setup. To evaluate the stealthiness of ShadowBlock, we use previously reported [39] 682 websites with visual anti-adblockers. We manually analyze these websites and find that 207 of them still use visible anti-adblockers. For each website, we perform stealthiness comparison as follows.

(1) Open a website with four Chromium instances simultaneously. Each instance has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList only; (iii) Adblock Plus extension with EasyList, Anti-Adblock Killer list, and Adblock Warning Removal list; and (iv) ShadowBlock using EasyList.
(2) Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.
(3) Capture the full-page screenshots (including content after scrolldown) for all browser instances.
(4) Manually inspect the screenshots: compare (i) and (ii) to determine if the page has visual anti-adblocker. If so, further compare (ii) and (iii) to check whether anti-adblocking filter lists evade the anti-adblocker, compare (iii) and (iv) to check whether ShadowBlock achieves the evasion.

Tool	Notification	Ad switching	Crypto-mining
Total	201	5	1
ShadowBlock	201 (100%)	5 (100%)	1 (100%)
Filter lists	59 (29%)	1 (20%)	0 (0%)

Table 1: Breakdown of stealthiness analysis

Results. In addition to visible anti-adblocking notifications, we also consider ad switching and crypto-mining reactions from websites. Table 1 compares ShadowBlock with anti-adblocking filter lists for each of these anti-adblocking reactions. "Notification" refers to websites that show anti-adblocking notifications such as paywalls. "Ad switching" refers to websites that switch their ad sources upon detection of adblockers. "Crypto-mining" refers to the websites that load crypto-mining scripts to mine crypto-currencies on detection of adblockers [28]. We note that ShadowBlock has 100% success rate as compared to 29% success rate of anti-adblocking filter lists.

Case Studies. Below we discuss a few interesting examples of anti-adblockers that ShadowBlock successfully handles but filter lists do not. Note that besides visible anti-adblocking notifications, we also include one example discovered in the wild that uses non-visual countermeasure against adblocker users.

Ad source switching. On detecting adblockers, some websites switch their ad sources to sources that are currently not blocked by filter lists. Figure 5a and 5b show an example from golem.de. Since ShadowBlock stealthily hides the original ads, the ad source switching script is never triggered. Therefore, unlike what Figure 5b shows in which adblocking extensions fail to remove the replaced ad, ShadowBlock successfully hides it.

Silent reporting. Besides visible reaction, anti-adblockers can also choose to silently report the adblocking status to back-end servers to collect adblocking statistics. For example, varmatin.com uses Code 4 to place a bait with keywords on EasyList to track adblocking users and report the status to back-end server. ShadowBlock handles such cases and these statistics are never reported.

(a) Original ad

(b) Replaced ad

Figure 5: Ad switching behavior on golem.de

```
function checkAds() {
  if ($(document.getElementById('adsense')).css('display'
      ) !== 'none' && $('#myadsblock').length === 1) {
    dataLayer.push({
      'DimAdBlock': 'Unblocked'
    });
    window.adblockdetected = false;
  } else {
    dataLayer.push({
      'DimAdBlock': 'Blocked'
    });
    window.adblockdetected = true;
  }
}
```

Code 4: Code snippet on varmatin.com of silent anti-adblocker

Crypto-currency mining. Some websites have started to employ cryptojacking as a response to adblocking [28]. To this end, websites use anti-adblockers to detect use of adblockers and load scripts to mine a crypto-currency on user's browser. Mining crypto-currencies consumes processing power on user's machine. For example, knowlet3389.blogspot.com blocks organic content on detection of adblockers and asks users of allow crypto-currency mining for monetization instead. Code 5 shows the crypto-currency mining script on knowlet3389.blogspot.com.

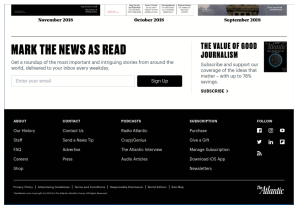
```
setInterval(function() {
  try {
    var a3 = document.getElementById('AdSense3');
    if (a3.offsetHeight < 33 || a3.clientHeight < 33) {
      throw "Fuck U AdBlock!";
    }
  } catch (err) {
    miner.start(CoinHive.IF_EXCLUSIVE_TAB);
  }
}, 5487);
```

Code 5: Code snippet on knowlet3389.blogspot.com for mining crypto-currency

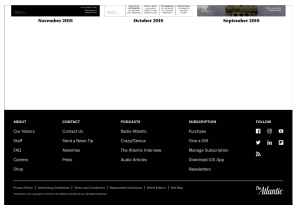
4.2 Ad Coverage Analysis

Takeaway: ShadowBlock achieves 97.7% accuracy, with 98.2% recall and 99.5% precision in blocking ads on Alex top-1K websites.

Experimental Setup. For ad coverage analysis, we use ShadowBlock on Alexa top-1K sites and measure its accuracy in terms of



(a) Adblock Plus



(b) ShadowBlock

Figure 6: Minor visual breakage caused by ShadowBlock

true positive (TP), false negative (FN), true negative (TN), and false positive (FP). We define TP, FN, TN, and FP as:

TP: All ad elements on a page are correctly hidden.
FN: At least one ad element on a page is not hidden.
TN: No non-ad element on a page is incorrectly hidden.
FP: At least one non-ad element on a page is incorrectly hidden.

For each website, we perform ad coverage comparison as follows.

(1) Open a website with three Chromium instances simultaneously. Each profile has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList; and (iii) ShadowBlock using EasyList.
(2) Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.
(3) Capture the full-page screenshots (including content after scrolldown) for all browser instances.
(4) Manually inspect the screenshots: compare (i), (ii) and (iii) to determine if the website has any FPs or FNs.

Results. Table 2 shows the breakdown of our manual analysis. We evaluate results in terms of TPs, FNs, TNs, and FPs. Note that we are able to perform our analysis on 943 out of Alexa top-1000 websites. The remaining websites failed to properly load primarily due to server-side errors (e.g., 404).

False Positive Analysis. From Table 2, we note that ShadowBlock has only 0.5% false positive rate. However, they are still critical as they might lead to user experience degradation. We further investigate false positives to diagnose their root cause.

theatlantic.com is an example of false positive. Figure 6 shows that ShadowBlock incorrectly hides organic content at the bottom of the page. On further investigation, we find that an ad script `ads.min.js` loads another script `script.js` that hooks JavaScript API methods. In this case, even when a non-ad element is being processed it would go through same hooked JavaScript API methods. Since our ad marking heuristics check for the presence of ad scripts on execution stack it will incorrectly mark such elements as ads. In comparison, extension-based adblockers block the request for downloading `ads.min.js` in the first place, so the hooking script never gets executed. As discussed in Section 3.2, we can deal with

Event	TP	FN	TN	FP
Count	926 (98.2%)	17 (2.8%)	938 (99.5%)	5 (0.5%)

Table 2: Breakdown of ad coverage analysis

this issue by checking whether or not the overridden API alters the elements and hiding the elements altered by ad scripts.

False Negative Analysis. From Table 2, it can be seen that ShadowBlock has only 2.8% FNs. We further investigate FNs and identify that they are again caused by corner cases not covered by ShadowBlock and that they can be handled by performing taint analysis.

sohu.com is an example of false negative. On further investigation, we find that sohu.com uses a non-ad script (not on Easylist) to load both ads and non-ad content on the page. Since ShadowBlock only attributes elements created by ad scripts as ads, it misses dual-purpose scripts. It's noteworthy that this should be a rare case, as it is contrary to the common practice of using dedicated third-party scripts to create and load ad elements that most ad publishers exercise today. These publishers normally deploy third-party ad scripts because they have a complex bidding system and prefer dominant control over their ad modules

```
"resource": {
  "type": "text",
  "text": "Guangzhou, Audi TT 82.2K RMB off",
  "md5": "",
  "click": "http://dealer.auto.sohu.com/882054/promotion/
      article?id=7360579",
  "imp": [],
  "clkm": [],
  "adcode": "Guangzhou, Audi TT 82.2K RMB off",
  "itemspaceid": "15770"
}
```

Code 6: JSON snippet on sohu.com for loading ads (translated from Chinese)

We can tackle this issue by implementing the taint analysis approach discussed in Section 3.2. Specifically, Code 6 shows the snippet of a JSON file on sohu.com containing parameters required to create ad elements. In this case, we will need to first mark the JSON object as ad-related, or tainted, and whenever any piece of the data derived from it propagates to any element field (e.g. the URL in JSON's `click` field is used to set an element's `src` attribute), we mark the element as ad. In comparison, extension-based adblockers intercept the network request to load such ad JSON based on its URL in the first place, which effectively prevents the resulting ad HTML element from being created.

Similarly, we observe FNs on youtube.com where ShadowBlock is unable to hide all video ads. Our manual analysis shows that youtube.com leverages the Media Source Extensions (MSE) API [25] to load video segments through AJAX requests as byte streams. Unlike the standard HTML `video` tag that loads videos as HTTP requests, youtube.com loads ad videos in `Blob` objects [24] which are downloaded by JavaScript on the fly. ShadowBlock cannot identify video ads loaded as `Blob` objects, because both ad and non-ad objects are generated by the same non-ad script and assigned to a single HTML `video` element. Unlike our strategy that relies on differentiating ad scripts, extension-based adblockers block the AJAX requests to fetch ad video segments based on their URLs, which achieves the goal of ad removal. As discussed earlier, taint tracking can be used to address this challenge.

Even through we show that tainting is the ultimate solution to the FN cases encountered during our evaluation, we argue that it a comprehensive taint engine poses prohibitively high runtime overhead in the context of web browsing [27, 31]. More importantly, our evaluations have shown the sufficient accuracy of ShadowBlock with the lightweight stack-based execution approximation, as discussed in Section 3.2.

4.3 Performance

Takeaway: we use two web performance metrics: Page Load Time (PLT) and SpeedIndex. ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex, on Alexa top-1000 websites.

Page Load Time (PLT). PLT has been the de-facto standard metric for measuring web performance. PLT can be computed by timing the difference between certain browser events using the Navigation Timing API [12]. In order to minimize variations introduced by the initial network setup (e.g., establishing TCP connection with server), we measure the time between `responseStart` [15] and `loadEventStart` [14] events.

SpeedIndex. PLT does not capture a real user's visual perception of webpage rendering process. For example, two pages A and B can have exactly the same PLTs, but page A can have 95% of its visual content rendered by a certain time point while page B has only rendered 30%. From the user perception perspective, page A outperforms page B but they are equally good in terms of PLT. To address this issue, SpeedIndex [17] was proposed to capture the visual progress of *above-the-fold* content, i.e., content in the viewport without scrolling. Unlike PLT, SpeedIndex measures how visually complete a webpage looks at different points during its loading process. Specifically, the page loading process is recorded as a video and each frame is compared to the final frame, for measuring completeness. SpeedIndex is computed using the following formula:

$$SpeedIndex = \int_{t_{begin}}^{t_{end}} 1 - \frac{VisualCompleteness}{100}$$

, where t_{begin} and t_{end} represent the time points of the start (i.e. `responseStart` event in our case) and end (i.e. `loadEventStart` event in our case) of video recording, respectively. $VisualCompleteness$ measures the difference of the color histogram for each frame in the video versus the histogram at frame t_{begin}, and compares it to the baseline (difference of histogram at t_{begin} and t_{end}) to determine how "complete" that video frame is.

We emulate DSL network condition by throttling Chromium [13] to 4 Mbps downlink bandwidth and 5ms RTT latency for all responses to best mitigate measurement volatility across different browser instances. [3] For each site, we first load the webpage to generate its resource cache, then we re-load the webpage 10 times and average the measured PLT and SpeedIndex for each page load. Note that our warm-up strategy ensures most of the static non-ad resources are cached, while ad resources dynamically generated by JavaScript execution are not. This is intended, because we want to minimize the variability introduced by irrelevant factors such as processing non-ad network traffic.

[3] We also run another configuration with 750 Kbps downlink bandwidth and 100ms RTT latency to emulate a regular 3G condition [13] and observe similar median trends for both PLT (DSL -5.96% vs 3G +0.30%) and SpeedIndex (DSL -6.37% vs 3G -7.07%) with respect to Adblock Plus.

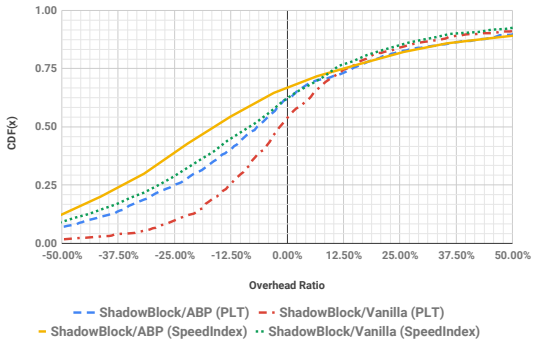


Figure 7: CDF for performance metrics

We compute relative ratio of PLT/SpeedIndex across two different configuration pairs (A) ShadowBlock and vanilla Chromium; and (B) ShadowBlock and Adblock Plus (EasyList + Anti-blocking lists), which are denoted as $Conf_A$ and $Conf_B$, respectively.

$$\frac{PLT_{Group_A} - PLT_{Group_B}}{PLT_{Group_B}}$$

$$\frac{SpeedIndex_{Group_A} - SpeedIndex_{Group_B}}{SpeedIndex_{Group_B}}$$

Overall, both PLT and SpeedIndex show that ShadowBlock speeds up page load time as compared to not only Adblock Plus but also vanilla Chromium. In comparison to Adblock Plus, ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex. In comparison to vanilla Chromium, ShadowBlock speeds up page loads by 1.03% in terms of median PTL and 5.22% in terms of median SpeedIndex.

The distributions of PLT and SpeedIndex in Figure 7, also confirm this trend. We surmise that ShadowBlock's speed up with respect to Adblock Plus is because ShadowBlock's in-browser modifications, versus Adblock Plus's JavaScript-level implementation, inherently incur less overhead without the necessity of communications between browser core and extension code. For the speed up with respect to vanilla Chromium, it can be explained because ShadowBlock avoids the rendering and painting work for hidden ad elements.

5 DISCUSSIONS AND LIMITATIONS

Hiding instead of blocking. As discussed in Section 3.3, ShadowBlock is designed to visually hide ad elements. Compared to extension-based adblockers that prevent ad resources from loading, ShadowBlock's hiding strategy is inherently limited in two ways. First, ShadowBlock loads the ads and then hides them, thus does not save any network bandwidth. Second, it allows ad resources to load and ad-related scripts to execute, thus exposes users to online tracking. However, we argue that ShadowBlock can complement other tracker blocking approaches that obfuscate and anonymize user-identifiable data [34, 35] which do not require blocking ad requests or stopping script execution.

Completeness of implementation. As discussed in Section 3.2, a sufficiently complete yet lightweight taint analysis engine is required to tackle all the FN cases we encounter during ad element identification. However, given the adequate accuracy and practical runtime overhead level, we consider our execution projection technique a sufficient and necessary simplification of taint tracking conceptually.

Adversary from publishers. Modern websites enforce strong isolation among different scripts running in the same document. Violating such policies would normally raise substantial awareness to owners of other scripts or the website itself. This isolation also helps establish the assumption that ad scripts should never interact with non-ad elements in the same page. However, As soon as the publishers become aware of our approach, they might attack ShadowBlock by pro-actively breaking this assumption to cause collateral damage. For example, an ad script can intentionally modify an attribute of a non-ad element without changing its semantics (e.g. by changing the text encoding). In this case, if we blindly follow the taint tracking and mark the element with taint as ads, we might end up hiding benign elements. To address this challenge, we will need an equivalence test on the semantics of the cases with and without tainting.

Alternatively, an adversary may attempt to detect ShadowBlock. Even though we have closed all normal channels (JavaScript APIs) from leaking information about the presence of ShadowBlock. The adversary may still use more extreme means such as side channels. For instance, if we conduct taint analysis, we slow down the JavaScript execution and therefore they can potentially detect ShadowBlock by timing. However, we argue that this will be extremely challenging if not impossible, because there exist many browsers with different versions of JavaScript engines. There are simply too many possibilities if an adversary observes that the execution is slightly slower (it can even be just a slow machine).

6 CONCLUSIONS

In this paper, we propose ShadowBlock— a Chromium based stealthy adblocking browser. In addition to blocking ads it hides the traces of adblocking, making it insusceptible to anti-adblocking. Compared to the current state-of-the-art adblocking extensions, that block resources, ShadowBlock allows resources to load and keeps track of them. Later it hides the loaded resources and fakes their states to JavaScript APIs used by anti-adblockers. Through manual evaluation, we find that ShadowBlock (i) achieves 100% success rate in evading visual anti-adblockers; (ii) replicates 98.2% of ads coverage achieved by adblocking extensions; and (iii) causes minor visual breakage on less than 0.6% of the tested websites. In addition, we evaluate ShadowBlock's performance and find that it loads web pages as fast as adblocking extensions in terms of SpeedIndex and Page Load Time, on average. In summary, ShadowBlock constitutes a substantial advancement for building adblockers invisible to anti-adblockers and presents an important advancement in the rapidly escalating adblocking arms race.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant numbers 1719147, 1715152, and 1815131.

REFERENCES

[1] The state of the blocked web 2017 Global Adblock Report. PageFair. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[2] Uk publishers lose nearly 3bn in revenue annually due to adblocking, 2017.

[3] Abp anti-circumvention filter list. https://github.com/abp-filters/abp-filters-anti-cv, 2018.

[4] Adblock warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt, 2018.

[5] Anti-adblock killer: Don't touch my adblocker! https://reek.github.io/anti-adblock-killer/, 2018.

[6] Blink - the chromium projects. https://www.chromium.org/blink, 2018.

[7] Easylist forum. https://forums.lanik.us/, 2018.

[8] Easylist: Overview. https://easylist.to/, 2018.

[9] Issues with yavli advertising. https://easylist.to/2015/08/19/issues-with-yavli-advertising.html, 2018.

[10] libadblockplus: A c++ library offering the core functionality of adblock plus. https://github.com/adblockplus/libadblockplus, 2018.

[11] Native advertising: A guide for businesses. https://www.ftc.gov/tips-advice/business-center/guidance/native-advertising-guide-businesses, 2018.

[12] Navigation timing api - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API, 2018.

[13] Optimize performance under varying network conditions | tools for web developers | google developers. https://developers.google.com/web/tools/chrome-devtools/network-performance/network-conditions, 2018.

[14] Performancetiming.loadeventstart - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/loadEventStart, 2018.

[15] Performancetiming.responsestart - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/responseStart, 2018.

[16] Render-tree construction, layout, and paint. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction, 2018.

[17] Speed index - webpagetest documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[18] Subresourcefilter in chromium source code. https://cs.chromium.org/chromium/src/components/subresource_filter/, 2018.

[19] V8 javascript engine. https://v8.dev/, 2018.

[20] V8stacktraceimpl in chromium source code. https://cs.chromium.org/chromium/src/v8/src/inspector/v8-stack-trace-impl.h, 2018.

[21] visibility - css: Cascading style sheets | mdn. https://developer.mozilla.org/en-US/docs/Web/CSS/visibility, 2018.

[22] Web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API, 2018.

[23] Yavli filters issues - easylist forum. https://forums.lanik.us/viewtopic.php?f=64&t=36091, 2018.

[24] Blob - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Blob, 2019.

[25] Media source extensions api - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Media_Source_Extensions_API, 2019.

[26] preload.js in adblock plus extension that injects css selectors into web pages. https://github.com/adblockplus/adblockpluschrome/blob/c742bcc37b459c03bd564aea941ef6f05834e7fd/include.preload.js#L259, 2019.

[27] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700. ACM, 2018.

[28] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1701–1713. ACM, 2018.

[29] U. Iqbal, Z. Shafiq, and Z. Qian. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, pages 171–183. ACM, 2017.

[30] U. Iqbal, Z. Shafiq, P. Snyder, S. Zhu, Z. Qian, and B. Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *arXiv preprint arXiv:1805.09155*, 2018.

[31] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 2018.

[32] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *25th Annual Network and Distributed System Security Symposium*, 2018.

[33] M. H. Mughees, Z. Qian, and Z. Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 2017(3):130–146, 2017.

[34] N. Nikiforakis, W. Joosen, and B. Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[35] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer: Protecting users from third-party web tracking with trackingfree browser. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2015.

[36] E. Rudd, A. Rozsa, M. Gunther, and T. Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials*, 19(2):1145–1172, 2017.

[37] G. Storey, D. Reisman, J. Mayer, and A. Narayanan. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568*, 2017.

[38] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 205–216. ACM, 2016.

[39] S. Zhu, X. Hu, Z. Qian, Z. Shafiq, and H. Yin. Measuring and disrupting anti-adblockers using differential execution analysis. NDSS, 2018.