WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention

Weihang Wang[1], Yunhui Zheng[2], Xinyu Xing[3], Yonghwi Kwon[1], Xiangyu Zhang[1],
Patrick Eugster[1,4]
[1]Purdue University, USA [2]IBM T.J. Watson Research Center, USA [3]The Pennsylvania State University, USA
[4]TU Darmstadt, Germany
{wang1315, kwon58, xyzhang, p}@cs.purdue.edu zhengyu@us.ibm.com uxx16@psu.edu

ABSTRACT

Nowadays, a rapidly increasing number of web users are using Ad-blockers to block online advertisements. Ad-blockers are browser-based software that can block most Ads on the websites, speeding up web browsers and saving bandwidth. Despite these benefits to end users, Ad-blockers could be catastrophic for the economic structure underlying the web, especially considering the rise of Ad-blocking as well as the number of technologies and services that rely exclusively on Ads to compensate their cost. In this paper, we introduce WebRanz that utilizes a randomization mechanism to circumvent Ad-blocking. Using WebRanz, content publishers can constantly mutate the internal HTML elements and element attributes of their web pages, without affecting their visual appearances and functionalities. Randomization invalidates the pre-defined patterns that Ad-blockers use to filter out Ads. Though the design of WebRanz is motivated by evading Ad-blockers, WebRanz also benefits the defense against bot scripts. We evaluate the effectiveness of WebRanz and its overhead using 221 randomly sampled top Alexa web pages and 8 representative bot scripts.

CCS Concepts

•Information systems → Display advertising; •Security and privacy → Web application security;

Keywords

Ad-blockers, randomization, web bots

1. INTRODUCTION

Online advertising is the primary source of income for many Internet companies, such as the IT giants Google and Facebook. According to [21], the revenue of U.S. web advertising is as large as $15 billion in Q3 2015. With Ad supports, online services give us instant access to more information than was ever stored in the entirety of the world's libraries just a few decades ago. Ad-supported services also create and maintain systems that allow for instant communication and organization between more than a billion people. Without web advertising, many of the world's most useful technologies may never have occurred.

Ad-blocker is a piece of software that allows a user to roam the web without encountering any Ads. In particular, it utilizes network control and in-page manipulation to help users block most online advertisements. Network control barricades HTTP requests to Ads and thus prevents them from loading. In-page manipulation looks up Ads based on pre-determined patterns and makes them invisible.

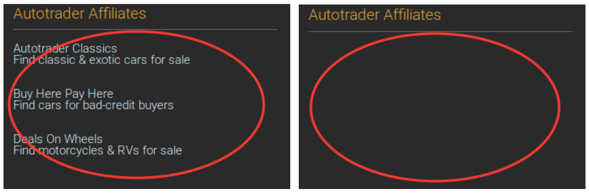
With Ad-blockers, web browsers generally run faster and waste less bandwidth downloading Ads, and the users are no longer distracted by the Ads. Not surprisingly, Ad-blocking is gaining popularity with an astonishing pace. According to a survey by Adobe and an Ad-blocking measurement service PageFair, 16% of the US Internet users run Ad-blocking software in their browsers [27]. Approximately 198 million active users globally used Ad-blocking tools in 2015, up by 41% compared to 2014. Adblock Plus [1], a leading Ad-blocker, claims that their product has been downloaded at an average rate of 2.3 million times per week since 2013 and it is "at a steady clip" [12]. In addition, more iPhone and iPad users start running Ad-blockers thanks to the built-in capacities in the latest iOS 9.

Despite its tangible convenience to the customers, Ad-blocking could devastate the economic structure underlying the web in the long run. This is because many content publishers make their primary legitimate income from Ads, but Ad-blocking is destroying this ecosystem. In 2014, Google made $59.1 billion from advertising, but lost $6.6 billion due to Ad-blocking [18]. During 2015, Ad-blocking has cost Internet companies almost $22 billion [27]. The number will rise to 41.4 billion in 2016. Many games-related websites currently encounter about 50% revenue loss due to Ad-blocking [19]. It is suggested that if everybody used Ad-blockers, Ad-supported Internet services would vanish. *"it's the websites that ad-block users most love that are going out of business first. This is to no one's benefit"* [19].

Furthermore, since Ad-blockers use pre-defined patterns to identify and suppress DOM objects that appear to be Ads, it is often the case that the patterns are so general that part of the regular content is also blocked. As shown in Fig. 1, the text links marked in the red circle on the homepage of www.autotrader.com, as part of the regular content, are inappropriately blocked by Adblock.

To damp the negative effect of Ad-blockers on the web ecosystem, tech companies and service providers introduce

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950352



Autotrader Affiliates

Autotrader Classics
Find classic & exotic cars for sale

Buy Here Pay Here
Find cars for bad-credit buyers

Deals On Wheels
Find motorcycles & RVs for sale

Autotrader Affiliates

Figure 1: Non-ads on www.autotrader.com blocked. Left: no Adblock. Right: with Adblock.

many solutions [8, 10, 15, 24, 11]. For example, one common approach is to integrate to a web page an in-page JavaScript snippet that examines the presentation of Ads and identifies the presence of Ad-blocker. Furthermore, such approaches often demand the users turn off their Ad-blockers or subscribe to a website's paywall. Despite the effectiveness in detecting Ad-blocker's presence, such technologies often substantially degrade the user experience with websites. More importantly, they still fail to punch through Ad-blockers to serve the originally-intended Ads.

In this paper, we develop a technique that allows the content publishers to deliver their Ads without being blocked. *Our goal is to retain a healthy web ecosystem by providing an option for the content publishers to protect their legitimate right.* Our technique, WebRanz, circumvents Ad-blocking using a content and URL randomization mechanism. With WebRanz, the publishers can constantly mutate their web content – including HTML, CSS and JavaScript – so that Ad-blockers cannot find the pre-determined patterns to filter out Ads. More importantly, WebRanz retains the functionalities of the original page and minimizes the visual differences caused by randomization so that the user experience is not affected. WebRanz ensures that multiple accesses of the same website return different content pages that have the same appearances and functionalities.

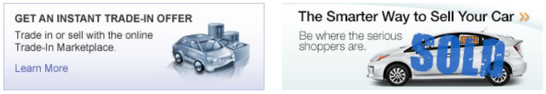
Specifically, since a lot of Ads are loaded by third-party JavaScript on the client side, without going through the original content publisher server, the pages returned by the server need to be instrumented so that the Ad contents generated on the client side can be randomized. WebRanz overwrites native JavaScript APIs so that dynamic DOM objects can be randomized on the fly when they are generated through the overwritten APIs.

Besides, Ad-blockers also cancel network requests to blacklisted URLs by comparing URLs against pre-defined patterns. To bypass, WebRanz randomizes the URLs and resource requested is fetched via a transparent proxy hosted by content publishers.

While WebRanz is developed for circumventing Ad-blocker scourges, last but not the least, our design principle also benefits the defense against many unwanted bot scripts that launch automated attacks on websites as these bots also manipulate DOM objects using pre-determined patterns.

In summary, this paper makes the following contributions.

- We propose a web page randomization technique WebRanz that prevents Ad-blockers from expunging Ads on web pages, and helps defending against web bots. At the same time, WebRanz preserves page appearances and functionalities.
- We address the challenges entailed by URL and web page randomization such as preserving dependencies between DOM objects and CSS selectors, between DOM elements and JavaScript, and handling dynamic generated elements as well as resolving randomized URLs.


GET AN INSTANT TRADE-IN OFFER
Trade in or sell with the online Trade-In Marketplace.
Learn More
The Smarter Way to Sell Your Car »
Be where the serious shoppers are.
SOLD

```
1.  <div id="j_id_1_20n" ...>
2.     <div class="atcui-column atcui-span-8" ...>
3.       <img src="http://www.autotraderstatic.com/.../190954.jpg" alt="Trade In">
4.       ...
5.     </div>
6.     <div class="atcui-column atcui-span-7" ...>
7.       <img src="http://www.autotraderstatic.com/.../153434.jpg" alt="Sell Your Car">
8.       ...
9.     </div>
10.  </div>
```

Figure 2: Static Ads on www.autotrader.com

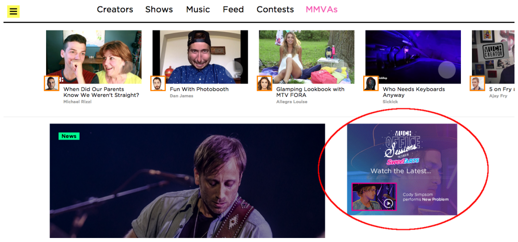


Figure 3: Page www.much.com. The Ad is enclosed by <div id = "div-gpt-ad-300_250-1"> in the red cycle.

- We implement WebRanz and it's publically available at [2]. We evaluate it using 221 randomly sampled top Alexa web pages and 8 real-world bot scripts. We show that WebRanz is effective in circumventing Ad-blockers with negligible overhead. It also defeats all the tested bot scripts.

2. MOTIVATION

In this section, we show how Ad-blockers and web bots work in practice. We explain how pattern matching based page element lookup plays an important role in these two. We then overview our approach.

2.1 Ad-Blocking

Web advertising is one of the foundations of Internet economy. Most content publishers earn their revenue by delivering Ads. When a page is loaded, the Ads loaded often go through many layers of delegations and redirections. We call it the *Ad delivery path*. We classify Ads into two categories based on the loading procedure. Those that are literally included by the content page are called the *static Ads* and those dynamically loaded are called the *dynamic Ads*.

Static Ads. Static Ads are usually served by the content publishers and delivered from a central domain. For example, Fig. 2 shows two static Ads on the home page of www.autotrader.com. They are two images with their URLs at lines 3 and 7 in the HTML file. Observe that the URLs are explicitly encoded in the source code. In other words, the Ads are not dynamically loaded by JavaScript.

Dynamic Ads. Dynamic Ads are usually provided by online advertisement vendors or Ad networks, and hosted on servers other than the publisher server. Compared to static Ads, they can be served from multiple domains owned by various Ad networks. They are usually dynamically loaded and may be different in each load. Hence, Ads networks are able to deliver customized Ads to maximize their revenue.

We use www.much.com as an example to illustrate dynamic Ad delivery. It serves the latest music videos and entertainment news. It ranks #6 in the category "Music

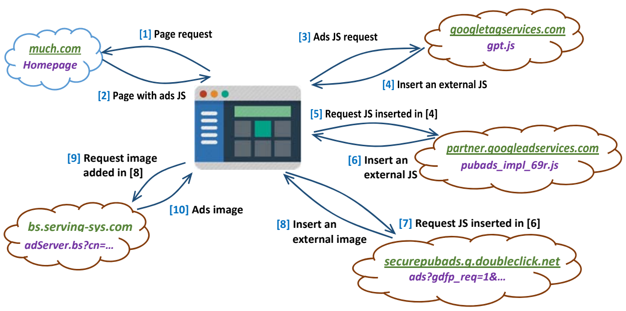

much.com
Homepage
[1] Page request
[2] Page with ads JS
[3] Ads JS request
googletagservices.com
gpt.js
[4] Insert an external JS
[5] Request JS inserted in [4]
partner.googleadservices.com
pubads_impl_69r.js
[6] Insert an external JS
[9] Request image added in [8]
[10] Ads image
bs.serving-sys.com
adServer.bs?cn=...
[8] Insert an external image
[7] Request JS inserted in [6]
securepubads.g.doubleclick.net
ads?gdfp_req=1&...

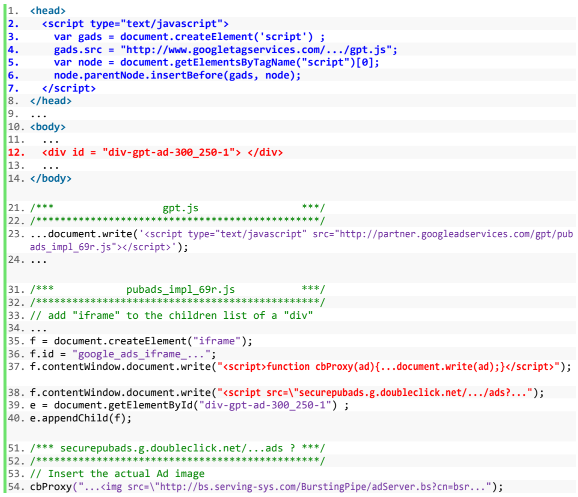
Figure 4: An example of dynamic Ads loading.

Videos” of Alexa top sites [13]. Fig. 3 shows the top portion of the website. The Ad marked in the red cycle is dynamically loaded from Google DoubleClick Ad Exchange service. Fig. 4 shows its loading procedure which consists of 10 steps, each loading and executing some JavaScript, until the Ad is finally displayed. The related source code is shown in Fig. 5. The code is substantially simplified for readability. The HTML after Ad loading is shown in Fig. 6. The script in blue in Fig. 5 is replaced with the script in blue in Fig. 6 after loading. The div in red in the original page is replaced with that in red after loading, containing an iframe with the Ad image. Particularly, the loading procedure contains the following 10 steps (Fig. 4):

- **Steps 1-2**. The browser loads the home page. The HTML returned by the server is shown lines 1-14 in Fig. 5. When the script in lines 3-6 is executed, a new script element is added to the page (line 3 in Fig. 6).
- **Steps 3-4**. The browser parses the new script element and then executes gpt.js from www.googletagservices.com. The script (i.e. line 23 in Fig. 5) inserts another script pubads_impl_69r.js to the page (line 4 in Fig. 6).
- **Steps 5-6**. The browser loads pubads_impl_69r.js from partner.googleadservices.com. This script creates an iframe (lines 35-36) and adds it to the div at line 12 in the original page (lines 39-40). In lines 37-38, the script writes two new script elements to the iframe. At line 37, it adds the definition of a function cbProxy() to the iframe. The function writes an Ad (provided as the parameter) to the page. At line 38, the script adds a new script element whose source is hosted by securepubads.g.doubleclick.net, a Google Ad service provider. The URL is parameterized for targeted advertising. In particular, the server side logic takes the parameters and identify the Ad(s) that should be pushed to a particular client.
- **Steps 7-8**. The script is loaded from securepubads.g.doubleclick.net. The source code is shown in line 54. The script invokes cbProxy(), which is supposed to be defined by other scripts in the earlier steps, to write an Ad image URL to the page. The image is hosted by the MediaMind Advertising Server (serving-sys.com) owned by an advertising company Sizmek. This leads to the iframe in red in Fig. 6.
- **Steps 9-10**. Finally, the browser loads and renders the actual Ad image.

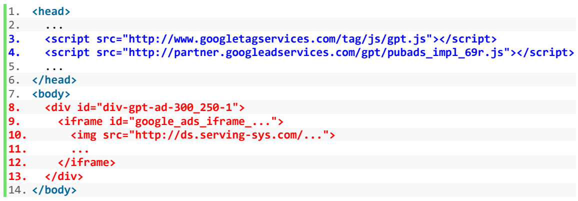
2.1.1 How Ads are Blocked

From the previous discussion, the Ad delivery path may be long and complex. If any of the steps along the path are broken, the Ad is blocked. Most Ad-blockers leverages this characteristics. They recognize steps on Ad delivery paths by pattern matching. They maintain a long list of patterns that can be used to distinguish Ads from normal page content. Take Adblock Plus [1] as an example. Ads are usually blocked by the following two means.



```
1.  <head>
2.    <script type="text/javascript">
3.      var gads = document.createElement('script') ;
4.      gads.src = "http://www.googletagservices.com/.../gpt.js";
5.      var node = document.getElementsByTagName("script")[0];
6.      node.parentNode.insertBefore(gads, node);
7.    </script>
8.  </head>
9.  ...
10. <body>
11.   ...
12.   <div id = "div-gpt-ad-300_250-1"> </div>
13.   ...
14. </body>

21. /***                    gpt.js                  ***/
22. /*********************************************/
23. ...document.write('<script type="text/javascript" src="http://partner.googleadservices.com/gpt/pub
    ads_impl_69r.js"></script>');
24. ...

31. /***              pubads_impl_69r.js            ***/
32. /*********************************************/
33. // add "iframe" to the children list of a "div"
34. ...
35. f = document.createElement("iframe");
36. f.id = "google_ads_iframe_...";
37. f.contentWindow.document.write("<script>function cbProxy(ad){...document.write(ad);}</script>");

38. f.contentWindow.document.write("<script src=\"securepubads.g.doubleclick.net/.../ads?...");
39. e = document.getElementById("div-gpt-ad-300_250-1") ;
40. e.appendChild(f);

51. /*** securepubads.g.doubleclick.net/...ads ? ***/
52. /*********************************************/
53. // Insert the actual Ad image
54. cbProxy("...<img src=\"http://bs.serving-sys.com/BurstingPipe/adServer.bs?cn=bsr...");
```

Figure 5: Dynamic Ads loading on www.much.com.



```
1.  <head>
2.    ...
3.    <script src="http://www.googletagservices.com/tag/js/gpt.js"></script>
4.    <script src="http://partner.googleadservices.com/gpt/pubads_impl_69r.js"></script>
5.    ...
6.  </head>
7.  <body>
8.    <div id="div-gpt-ad-300_250-1">
9.      <iframe id="google_ads_iframe_...">
10.       <img src="http://ds.serving-sys.com/...">
11.       ...
12.     </iframe>
13.   </div>
14. </body>
```

Figure 6: HTML after loading.

Network Control by URL Filtering. Since advertising companies serve Ads on a limited number of servers, it's possible to collect a set of domain names and block requests sent to domains on the URL blacklist. For example, the request to “https://securepubads.g.doubleclick.net/.../ads?...” sent in step [7] in Fig. 4 will be blocked by a URL based rule “/securepubads.”. As a result, the loading procedure is interrupted since the browser cannot obtain the actual Ads enclosed in the response.

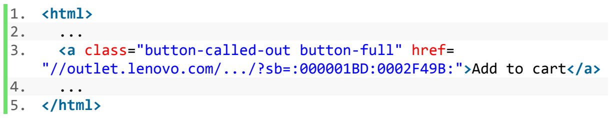
In-page Manipulation by Selector-based Filtering. Elements not blocked by network control can be successfully loaded from the remote servers. However, they can still be blocked before they are rendered by the browser. This is done by identifying Ad elements inside the browser using selector patterns and setting these elements to invisible. For example, in Fig. 5, two Ad related DOM elements are set to invisible based on the following two selectors. Consequently, the Ad image is not rendered.

(1) Selector “##*div*[*id*ˆ= “*div-gpt-ad-*”]” matches the div element with *id* that starts with “*div-gpt-ad-*”. As such, the div (line 8 in Fig. 6) is set to invisible.

(2) Selector “##*iframe*[*id*ˆ= “*google_ads_iframe*”]” selects the iframe with id begins with *google_ads_iframe*. Therefore, the iframe at line 9 in Fig. 6) is hidden.

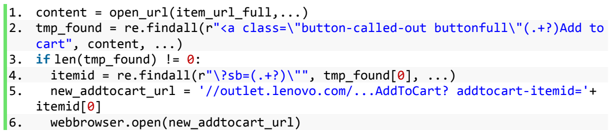
2.2 Content-sensitive Web bots

Web bots are programs that simulate user browsing behavior. They read the HTML code, analyze the content and interact with the web app just like humans. Web bots are commonly used for various purposes such as searching, scraping and impersonation. According to a recent study on



```
<html>
  ...
  <a class="button-called-out button-full" href=
"//outlet.lenovo.com/.../?sb=:000001BD:0002F49B:">Add to cart</a>
  ...
</html>
```

Figure 7: The source of the "Add to cart" button. The page has totally 109 <a> elements.



```
1.  content = open_url(item_url_full,...)
2.  tmp_found = re.findall(r"<a class=\"button-called-out buttonfull\"(.+?)Add to
    cart", content, ...)
3.  if len(tmp_found) != 0:
4.    itemid = re.findall(r"\?sb=(.+?)\"", tmp_found[0], ...)
5.    new_addtocart_url = '//outlet.lenovo.com/...AddToCart? addtocart-itemid='+
    itemid[0]
6.    webbrowser.open(new_addtocart_url)
```

Figure 8: A web bot [9] for the Lenovo Outlet.

20,000 popular websites by Incapsula Inc., out of 15 billion visits observed, 56% of all traffic is generated by bots [22].

Bots can be roughly classified into two groups based on the targets. Some do not focus on particular items but grab all contents. Bots targeting search engines are examples of this type. The bots in the other group focus on specific elements. They parse the HTML and locate the targets using predefined patterns. Once found, they either simulate human behaviors (e.g., clicking buttons) or extract valuable data. Data theft by web scraping and human impersonators are typical examples. Since being able to locate the targets is important, we dub them *content-sensitive web bots*.

Content-sensitive web bots such as scrappers have caused substantial loss. According to ScrapeSentry, 39% of booking traffic on travel industry websites is generated by scraping bots [28]. As a result, airlines have increased booking fees to balance the cost [17]. A scraper also demonstrated its capabilities in causing damages through the Twitter earnings leak incident [20, 30]. In April 2015, Twitter planed to release its Q1 earning report after the stock market was closed. However, the report link was mistakenly posted online before the schedule. Although it was deleted immediately, it stayed online for 45s. A financial scraper owned by Selerity discovered the report in such a short time and tweeted Twitter's disappointing result when the market was still open. As a result, shares of Twitter fell as much as 22%.

Content-sensitive web bots are also widely used on Ecommerce websites. Take a bot [9] targeting at the Lenovo outlet store as an example. The Lenovo outlet store offers substantially discounted computers but the quantity is limited. It is usually hard to get one since many people keep refreshing the inventory and grab a deal as soon as it becomes available. While it is tedious for a human to repeat this procedure, a bot was programmed to monitor the store and add deals to the shopping cart automatically.

Fig. 7 shows the HTML code of the "Add to cart" button. It is an <a> element representing a link. There are totally 109 <a> elements on this page. Fig. 8 shows a snippet of the web bot. The script loads a product page at line 1. It hence tries to locate the "Add to cart" link. Since there are many <a> elements, the script has to distinguish the target from the others. It does so by comparing the style class name and the element id of a candidate link with some patterns. In this case, at line 2, the script uses two style class names "button-called-out" and "button-full" as the signature. If such an <a> is found, at line 4, it further extracts the id from the content after "sb=" in the link. In this example, the itemid is ":000001BD:0002F49B:". Then it constructs the actual add-to-cart link at line 5 and invokes the browser at line 6 to add the item to the shopping cart.

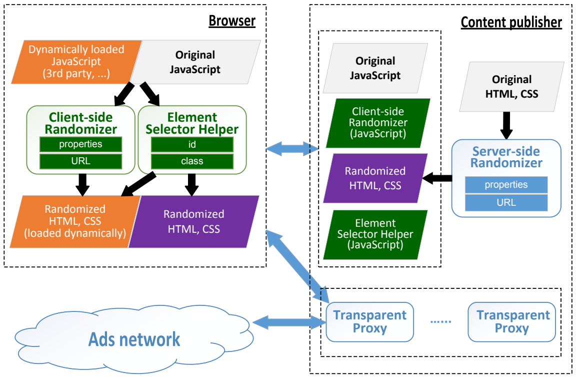


Figure 9: Overview of the web page randomization.

As observed above, a critical precondition for content-sensitive web bots is that they identify important DOM objects by pattern matching, which is similar to Ad-blockers.

2.3 Our Solution: Web Page Randomization

WebRanz is a technique that randomizes both URLs and content in web pages so that Ad-blockers and content based web bots can no longer use pattern matching to identify and manipulate DOM elements.

As mentioned before, web pages may be changed dynamically on the client side. Therefore the randomization is performed not only on the server side by the content publisher, but also on the client side through JavaScript instrumentation added to the page by the content publisher. In particular, upon receiving a client side request, the publisher prepares the page as usual. Before delivering it to the client, WebRanz randomizes the page by randomizing the element ids, style sheets, URLs, and so on.

The randomization is designed in such a way that it does not change the visual effects and functionalities of the page but rather the internal representations of the page. As such, the changes are transparent to the end user. WebRanz also inserts JavaScript code to the page to randomize the page content dynamically generated on the client side.

Fig. 9 shows the overview of the web page randomization procedure. WebRanz is deployed on the server side (publishers). The input is an HTML file and its corresponding external CSS style-sheets. The HTML file can be a static HTML page or the output of a dynamic server side script before being sent out. The existing external JavaScript and other resource files (e.g., images) are untouched. They will be sent to the clients upon requests.

We randomize the attributes (e.g. element id or style class name) and URLs in HTML and external CSS style-sheets. As such, the selector-based and URL based filters used in Ad-blockers and web bots can no longer identify the page elements. The output is a randomized version of the input HTML page and CSS style-sheets. In the HTML, we also append utilities that redirect the DOM selectors in the existing JavaScript to the corresponding randomized HTML elements and handle DOM elements dynamically inserted in the browser. The former is to make sure the JavaScript in the original page can work properly with the randomized version, especially when the script accesses DOM elements. The latter is to handle DOM elements we do not see during the server side randomization.

For the www.much.com example, the div and iframe in red in Fig. 6 are given random ids such that the aforementioned

Table 1: Adblock Plus EasyList Filters

URL Blocking Rules		Element	Exception
Domain only	Resource only	Hiding Rules	Rules
5,054	13,811	27,114	3,973

Adblock Plus patterns fail to locate and suppress the Ad image. For the Lenovo web bot example, the class name at line 3 of Fig. 7 "`button-called-out button-full`" is replaced with a random string that changes each time the page is loaded. As such, the web bot in Fig. 8 cannot identify the link and fail to put the product in cart.

Besides, to resolve the randomized URLs, we leverage transparent proxies deployed on the publisher side. In particular, all URLs (either statically embedded or dynamically loaded) randomized by server-side and client-side randomizer point to publisher's transparent proxies. Note that the proxy URL can be arbitrary and different in each load. Or, we may just use the publisher's top-level domain (e.g. cnn.com) to bypass the URL filter. Once requested, the transparent proxies recover the real URLs, fetch the resource requested and send them to the client.

3. WEB PAGE RANDOMIZATION

As third party contents (including ads) are usually loaded dynamically and could be different in each load, a publisher may not be able to determine which elements should be randomized or which paths should be changed in advance. In order to achieve practical web page randomization, WebRanz needs to handle a few technical challenges:

- Randomization causes inconsistencies between DOM objects and their style specification, between HTML elements and JavaScript.
- Client side randomization should be supported by instrumenting the page with the randomization logic (in JavaScript) executed on the client side.
- The server side needs to be extended to resolve the randomized URLs that change constantly.

In the following subsections, we first discuss what elements should be randomized. We then discuss how they are randomized. The discussion is divided into two parts: randomization performed on the (publisher) server side and that on the client side by the JavaScript instrumentation.

3.1 What to Randomize

One of the most important design goals of WebRanz is to retain the appearances and functionalities of web pages while breaking the patterns used by Ad-blockers and web bots. Hence, not all HTML elements or element attributes are subject to randomization. For example, changing the type of a DOM object (e.g., a radio box to a check box) or changing the style may cause undesirable visual differences. To identify randomization candidates, we perform the following two studies: First, we analyze EasyList, the blacklist used by Adblock Plus, to select important attributes. Second, we visit the home pages of Alexa Top 500 websites using different blacklist settings and evaluate the effectivenesses of URL-based and element hiding filters.

3.1.1 Interpreting Patterns in Blacklisting Rules

Adblock Plus works based on a set of rules. We classify these rules to three categories, as shown in Table 1:

- The *URL blocking rules* specify URL filters for network control. Any requests to blacklisted URLs are canceled. They can be further divided into two subgroups: *domain only* patterns (e.g., http://*.much.com) and *resource* patterns with both domains and resource paths (e.g., http://*.much.com/banner.jpg).
- The *exception rules* disable existing rules on particular domains. Filters are not applied on domains that match the whitelist.
- The *element hiding rules* specify selector based filters. The HTML elements on the page that match the rules are prevented from rendering.

Table 2: Element Hiding Filters

	Selectors	id	class	id and class	id or class
General Rule	16,037	7,269	8,538	23	15,784 (98.42%)
Site Specific	11,077	3,979	4,331	112	8,198 (74.01%)
Total	27,114	11,248	12,869	135	23,982 (88.45%)

Table 3: Elements Blocked on Alexa Top 500 Sites

	Full EasyList		Domain	Resource	Element Hiding
	URL	Hiding	Only	Only	Rules Only
Min	0	0	0	0	0
Max	171	18	91	171	19
Average	5.3	0.8	3.8	4.2	0.9

Observe that the URL blocking and element hiding rules are dominant, and most URL blocking rules are those that contain both the domain and the resource path.

Each element hiding rule is a selector defined by one or more attributes. To determine the attributes that are important for randomization, we further study the 15,701 element hiding rules in the Adblock Plus's list. The results are presented in Table 2. We observe that attributes id and class are most commonly used: id is a unique identifier of an HTML element in the document; class provides a class name for one or more HTML elements.

Ad-blockers hide elements using style rules. A style rule consists of a selector and a declaration block. The selector points to the HTML element(s) where the style declared is applied to. The declaration block contains visual effect specifications such as size and layout. The style rules can be a piece of in-line script in HTML or an external stylesheet CSS file. Examples of element hiding rules are shown as follows. ##.googleads-container matches elements with class googleads-container. A composite hiding rule ###resultspanel > #topads is to select the element that has id topads and is enclosed in an element with id resultpanel.

3.1.2 Evaluating Filters on Popular Websites

In the second study, we collect the number of elements removed by the URL blocking rules and the element hiding rules and evaluate their effectiveness on real world websites.

We first collect the data using the original full EasyList. As the URL blocking rules and element hiding rules are not orthogonal, we repeat the experiment using different subset rules in EasyList: (1) URL blocking rules that only contain domains (*domain only*), (2) URL blocking rules that have both domain and resource paths (*resource only*) and (3) Element hiding rules. The results are shown in Table 3. On the one hand, the URL based rules block more than 6 times elements than the element hiding rules on popular web pages. On the other hand, the fact that more than half rules are element hiding rules shows that they are important as URL blocking rules cannot handle all cases without any overkills.

As suggested by above two studies, id and class are critical attributes used by selectors in the element hiding rules. Besides, bypassing the URL blocking rules is crucial. Therefore, WebRanz aims to randomize id, class and URLs. That would allow us to counter 85.8% of all rules, which represent the dominating majority applied in practice.

3.2 Server Side Randomization

WebRanz is deployed on the publisher server. Part of the randomization is performed directly on the server side. In this subsection, we discuss the server side randomization. It mainly consists of randomizing the id and class attributes, fixing styling rules and randomizing URLs.

3.2.1 Randomizing Element Id and Class

Before a page retrieved/generated by the publisher server is returned to the client, WebRanz parses the page and then traverses the DOM tree. During traversal, it replaces each id or class name encountered with some random value. It also maintains a one-to-one mapping between the original id/class name and its randomized counter-part.

This mapping is the key to preserving the semantics and functionalities and will be used in later steps. Note that it is possible that multiple HTML elements have the same id. In this case, WebRanz also projects them to the same random value. We also want to point out that the server does not need to keep the mapping in any permanent storage as it is never reused. In other words, each page returned to some client is randomized independently.

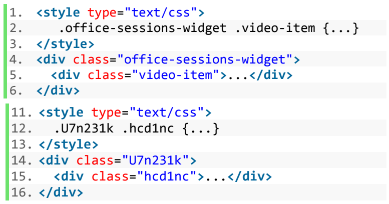
3.2.2 Fixing Static HTML Style Rules

Style rules determine the visual effects of a class of DOM elements. Styles can be specified in the following ways:

- An inline attribute is defined along with the DOM element. E.g., <div style="border: 10pt"> specifies that the div element has a border of 10 points.
- An internal style is defined in the header. E.g., <style type="text/css">.sidebar{width:100%;}</style> sets the width of element(s) with class of "sidebar".
- An external rule is specified in a CSS file and embedded in the HTML. E.g. <link rel = "stylesheet" href = "1.css"> includes style rules defined in 1.css.
- A style can be dynamically defined by a property setter in JavaScript. E.g. getElementById("x1").style= {border:1pt} sets the border of the element with id of "x1". This is often used to define styles on the client side.

The inlined style is not affected by randomization. In contrast, the internal and external rules have to be updated since the id/class names in those style rules need to be made consistent with the randomized id/class names.

Example. In Fig. 10, the HTML code contains a div (lines 4-6) whose style is specified in lines 1-3. Observe that the strings "U7n231k" and "hcd1nc" are the randomized values for "office-sessions-widget" and "video-item" respectively. Since the rule name and the class name of the HTML element are updated consistently, the visual representation is preserved.



```
1.  <style type="text/css">
2.     .office-sessions-widget .video-item {...}
3.  </style>
4.  <div class="office-sessions-widget">
5.    <div class="video-item">...</div>
6.  </div>
```

```
11. <style type="text/css">
12.   .U7n231k .hcd1nc {...}
13. </style>
14. <div class="U7n231k">
15.   <div class="hcd1nc">...</div>
16. </div>
```

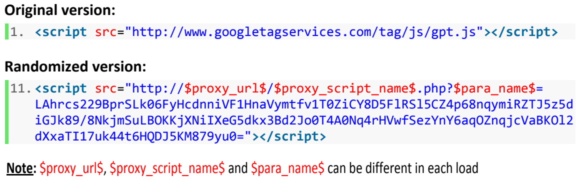
Figure 10: An Example of Class Randomization.

WebRanz only fixes static style rules on the server side. For dynamic styles that are defined by property setters or inserted by in-page JavaScript, WebRanz relies on the JavaScript instrumentation to ensure consistency (on the client side). More details can be found in Section 3.3.

3.2.3 Randomizing Static URLs

As mentioned earlier, Ad-blockers work by blocking URLs or hiding elements. Randomizing id/class prevents element lookup by selectors. WebRanz also performs URL randomization to evade URL blocking.

WebRanz traverses the DOM tree of the page and identifies all URLs in the page. For those that can be matched by URL blocking rules, WebRanz randomizes the whole URL including domain, resource path and all parameters. In particular, it takes the whole URL and randomize the string using public-key cryptography. This is to make sure that the ad-blocker on the client side cannot recover the original URL for the randomized version, as the private key will not be sent to the client.



Original version:

```
1.  <script src="http://www.googletagservices.com/tag/js/gpt.js"></script>
```

Randomized version:

```
11. <script src="http://$proxy_url$/$proxy_script_name$.php?$para_name$=
    LAhrcs229BprSLk06FyHcdnniVF1HnaVymtfv1T0ZiCY8D5FlRSl5CZ4p68nqymiRZTJ5z5d
    iGJk89/8NkjmSuLBOKKjXNiIXeG5dkx3Bd2Jo0T4A0Nq4rHVwfSezYnY6aqOZnqjcVaBKOl2
    dXxaTI17uk44t6HQDJ5KM879yu0="></script>
```

Note: $proxy_url$, $proxy_script_name$ and $para_name$ can be different in each load

Figure 11: An Example of URL Randomization.

Fig. 11 shows an example of URL randomization. Line 1 is an external script. Line 11 is the corresponding randomized version. The randomized URL is sent to $proxy_url$, a URL pointing to the transparent proxy, as a request parameter. The original URL is replaced with this randomized version. Please note that the proxy URL $proxy_url$ and $proxy_script_name$ can be arbitrary valid URL pointing to a publisher's proxy. We can make it a moving target or simply use the publisher's top-level domain to host a pool of proxy scripts. Therefore, *blocking the URL to transparent proxies is impractical*. Otherwise, *all content* hosted by this publisher will be blocked.

3.2.4 Transparent Proxy

The original URLs are reshaped to evade URL blocking rules by randomization. Since domain names are also changed, to make sure the client can correctly load the original resource, we use a group of transparent proxies deployed on the publisher side to fetch the resource for the client.

In particular, when the transparent proxy receives a request, it decodes the parameter and recovers the original URL. It then sends a request with original data from the client (including cookies) to the original destination. Once the response arrives, the proxy forwards it to the client. Note that if the requested URL points to a local resource, the proxy directly loads and returns it to the client in order to minimize the performance overhead.

In addition, besides URL, the src of an image or script may be a *data URI* that is the file content itself [3]. Data URIs are also randomized as they can be used to deliver Ads. When the proxy receives such data URI request, it directly returns the decoded data with its corresponding header. For example, if a data URI request is an image type, the proxy decodes the image data in the request and adds an image header to the response.

3.3 Preserving Client-side Functionalities and Handling Dynamically Loaded Elements

Both the DOM interface and third-party libraries (e.g. *jQuery*) provide many ways to access HTML elements. After randomization, the original attribute values are replaced with the random ones. However, the DOM element selectors in existing JavaScript still use the original values such that they cannot locate the correct elements anymore.

In order to provide consistent accesses to randomized DOM elements, one way is to scan the JavaScript in the page and update all the selector values, similar to fixing the static style rules. However, since third-party JavaScript files are loaded dynamically on the fly, it is infeasible to update all these files during the server side randomization. Instead, we choose to override the relevant JavaScript APIs and map the original attribute values to their corresponding randomized versions at runtime whenever the attributes are accessed.

3.3.1 Overriding Element Selectors

The Document Object Model (DOM) defines a programming interface to access and manipulate HTML elements and their properties. Specifically, the reference to an HTML element can be acquired using an id selector (getElementById) or a class selector (getElementsByClassName).

```
1. var byId = document.__proto__.getElementById;
2. document.__proto__.getElementById = function(id) {
3.   if (idMap[id]) {
4.     return byId.call(document, idMap[id]);
5.   }
6.   else {return byId.call(document, id);}
7. };
```

Figure 12: Override document.getElementById().

Therefore, WebRanz overrides these two methods. It projects the original attribute value to the corresponding randomized value as follows: if a value has been randomized, the original selector value is replaced by the corresponding randomized value. Otherwise, the same value is used. The overridden version of document.getElementById() is shown in Fig. 12. At line 1, the original document.getElementById() is saved in variable byID. Line 3 checks if a mapping exists. If so, it calls byID with the corresponding randomized value. Note that idMap[] projects an original value to its randomized version. The overridden functions and the mappings are inserted to the page and sent over to the client. As such, the element access redirection happens at runtime on the client side.

WebRanz uses a predefined set of obfuscations for preventing the mappings from being automatically reverse engineered. For example, idMap[] is encrypted differently each time the page is loaded and the overridden element lookup functions have the corresponding decryption methods. Even though in theory the attackers (to our approach) can still inject some exhaustive scanning JavaScript (to the page) to repetitively call the overridden API functions to reverse engineer the mappings, the cost is so high that the user experience of the page would be substantially degraded. More discussion can be found in Section 5.

3.3.2 Overriding Third-party JavaScript Library APIs

Although third-party JavaScript libraries can provide various element access interfaces using different kinds of selectors, many of them eventually need to invoke some primitive DOM interface function. Take jQuery, one of the most popular third-party JS libraries, as an example.

jQuery provides efficient ways to access HTML elements through web API functions. For example, $(.className) selects all elements with class className. Its underlying implementation is based on the primitive API getElementsByClassName(). Since WebRanz has overridden the primitive element lookup functions, the corresponding jQuery selectors can locate the correct HTML elements as well.

```
1. hasClass: function(a) {
2.   if (classMap[a])
3.     a = classMap[a];
4.   // the actual HTML element look-up
5. }
```

Figure 13: Override jQuery .hasClass().

Besides the two primitive element look-up functions, WebRanz also overrides a set of higher-level DOM manipulation APIs, such as .addClass(), .attr() and .hasClass(). Fig. 13 shows the handling of .hasClass() provided by *jQuery*. In lines 1-2, classMap maps a class name to its randomized version. The actual look-up is done using the value after mapping. Obfuscation is also used to protect classMap.

3.3.3 Randomizing Dynamically Generated Elements

Elements Dynamically generated are common on the client side. The script manipulating them may be from the publisher or third parties such as Ad-networks. The new elements and the modified element attributes need to be randomized as well otherwise Ad-blockers may block them.

```
1. var idSetter = idProperty(element.__proto__.__proto__).set;
2. var idGetter = idProperty(element.__proto__.__proto__).get;
3. Object.defineProperty(element.__proto__.__proto__, 'id', {
4.   set: function(arg) {
5.     randId = randIdGenerator(...);
6.     idSetter.call(this, randId);
7.   },
8.   get: function() {
9.     return reverseIdMap[idGetter.call(this)];
10.  }
11.});
```

Figure 14: Override the Setter and Getter of id.

Overriding Property Setters and Getters. Existing JavaScript may modify or read attribute values, such as the values of id and class. For example, element.id = id1 sets id1 as the id of an HTML element. To make them consistent, we also need to provide bi-directional mappings between the original values and the corresponding randomized versions. Fig. 14 shows an example of such wrappers. The references to the original getter and setter are saved in lines 1-2. In lines 4-7, the 'id' setter is overridden. When set, a random id is generated at line 5 and set by the original setter at line 6. The getter is shown in lines 8-10. Function reverseIdMap() projects a randomized value to its original one. At line 9, the getter function returns the original id value.

Overriding DOM Elements Creating Function. WebRanz also intercepts a set of DOM object creation functions that require id and/or class as part of the parameters, such as element.setAttribute(class, class1) sets attribute class of a newly created element to class1. It then generates random id and class. The instrumentation is similar to that of setters and getters and hence elided.

Randomizing URL Appended. WebRanz intercepts APIs that append elements to the web page and rewrites enclosed URLs using the same method discussed in Sec. 3.2.3.

4. EVALUATION

In this section, we describe our implementation and evaluate the effectiveness and efficiency of WebRanz in preventing ad-blocking and web bots. Experiments are done on a machine with Intel i7 2.8GHz CPU, 16GB, and OS X Yosemite.

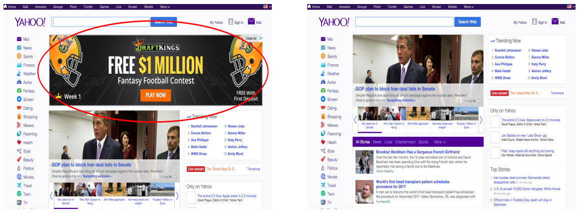


Figure 15: Top Banner Ad on www.yahoo.com is blocked. Left: no Adblock. Right: with Adblock. The distance score of the two screenshots is 4.75.

4.1 Implementation

We implemented WebRanz using Node.js [23]. In particular, we augment the *htmlparser2* [4] with the ability to parse static HTML pages and randomize the attribute values of its DOM elements. We extend a CSS parser [5] to process CSS and overwrite the corresponding selectors. The URL resource path randomization is implemented with initiating AJAX requests on the client. We also implement a php program on the server side to intercept the AJAX requests.

4.2 Ad-blocking Evasion

To study the effects of WebRanz on preventing Ad-blocking, we run WebRanz on real world web pages. We randomly collected the home pages of 1426 websites from Alexa top 10,000 list. In particular, we gather HTML, JavaScript and CSS files of each home page. We first remove duplicated websites hosted under different country domains (53). For example, google.de is removed because google.com is in the dataset. We further exclude those home pages that do not contain ads, the urls are obsolete, and those that incur connection reset error (1152). Finally, 221 unique web pages are obtained. We host these web pages on our own web server where we run WebRanz and perform randomizations.

Ad-blocker deletes Ads and the screen area used to place ads is taken by surrounding content. Therefore, Ad-blocker changes visual appearances of web page that contains Ads. Fig.15 shows the homepage of www.yahoo.com without and with Ad-blocker. On the left snapshot (without Ad-blocker), the top area marked by the red circle is a banner Ad. On the right snapshot (with Ad-blocker), the Ad space is removed and all contents underneath the Ad is moving upward. To study the effectiveness of WebRanz, we compare the visual appearances of a web page in different settings. In particular, we developed a program using Selenium [25]. It automatically installs Adblock Plus 1.9 on a Firefox browser and drives the browser to visit the home pages hosted on our server with and without WebRanz enabled. For each visit, we took a snapshot and stored the image. To get a baseline for visual comparison, we also captured the snapshots of home pages with both Adblock and WebRanz disabled. In this way, we have snapshots taken in three different settings. For each web page, we measure the visual differences of its snapshots. More specifically, we compute the the distance between snapshots using an image comparison algorithm in the Lucene Image Retrieval (LIRE) library [6].

As discussed in Section 3, web page randomization needs to be carried out on both the web server and the client browser. Thus, we quantify the efficiency of our WebRanz on both ends. In particular, we measure the latency of mutating DOM elements on the web server as well as the latency introduced by the interception of Ad rendering.

4.2.1 *Effectiveness*

Table 4 demonstrates 50 randomly selected web pages (from the aforementioned 221 web pages with Ads) and the distance measures between their snapshots. Fig. 16 shows the average distance measures between snapshots across the 221 web pages. As is shown in the figure, we observed an average visual deviation of 3.828 on a web page before and after Ad-blocker is enabled. Note that 3.828 suggests substantial visual differences. Fig. 15 presents two images (with and without Adblock Plus) with the distance of 4.75.

Table 4: Results for 50 randomly selected web pages.

Web Page	Orig vs. Orig + Adblock	Orig vs. Rand + Adblock	Page Load Time(s)	Random-ization Time(s)	Overhead
www.autotrader.com	1.22	0.6	7.58	0.72	9.50%
www.capital.gr	1.15	0	8.85	0.45	5.08%
www.celebritynetworth.com	5.37	0.58	12.1	0.252	2.08%
www.christianpost.com	0.60	0	4.41	0.215	4.88%
www.classiccars.com	21.34	0.49	5.49	0.346	6.30%
www.dostor.org	1.33	0	3.91	0.222	5.68%
www.espnfc.us	2.19	0	6.63	0.765	11.54%
www.goodreads.com	0.73	0	5.31	0.433	8.15%
www.groupon.com	0.44	0	4.15	0.269	6.48%
www.haber7.com	6.97	0.61	7.27	0.306	4.21%
www.head-fi.org	4.67	0	7.34	0.589	8.02%
www.hqrevshare.com	2.00	0	4.69	0.166	3.54%
www.huffingtonpost.co.uk	2.74	0.57	11.99	0.99	8.26%
www.lotterypost.com	4.88	0	4.36	0.176	4.04%
www.marktplaats.nl	3.36	0	3.65	0.337	9.23%
www.mery.jp	0.72	0	5.18	0.445	8.59%
www.meteofrance.com	6.70	0	6.39	0.401	6.28%
www.microcenter.com	4.71	0	5.80	0.280	4.83%
www.mobile01.com	1.82	0	3.08	0.128	4.16%
www.much.com	2.53	0	7.36	0.481	6.53%
www.myanimelist.net	2.67	0	8.93	0.314	3.52%
www.niuche.com	0.59	0	8.9	0.168	1.89%
www.nk.pl	5.05	0	7.18	0.332	4.62%
www.nytimes.com	0.65	0	6.95	0.495	7.12%
www.olx.co.id	6.52	0	3.66	0.351	9.59%
www.phys.org	2.86	0	12.31	0.19	1.54%
www.popmaster.ir	3.57	0	8.48	0.243	2.87%
www.posta.com.tr	6.15	0	18.28	0.221	1.21%
www.prothom-alo.com	2.99	0	8.53	0.674	7.90%
www.qz.com	3.41	0	4.91	0.246	5.01%
www.reclameaqui.com.br	5.48	0	3.71	0.423	11.40%
www.sabq.org	7.35	0	2.5	0.198	7.92%
www.setn.com	7.32	0.51	9.64	0.196	2.03%
www.shufoo.net	1.64	0	5.57	0.214	3.83%
www.siteprice.org	6.25	0	4.85	0.195	4.03%
www.smartinf.ru	15.88	0	2.35	0.082	3.49%
www.softonic.com	3.66	0	10.14	0.424	4.18%
www.southcn.com	1.53	0	6.65	0.21	3.16%
www.stcatharinesstandard.ca	8.18	0	21.23	0.453	2.13%
www.subscene.com	1.16	0	2.03	0.109	5.37%
www.theepochtimes.com	4.75	0	13.33	0.378	2.84%
www.u-car.com.tw	4.66	0.51	6.78	0.238	3.51%
www.torrenthound.com	1.21	0	3.9	0.082	2.10%
www.trend.az	0.97	0	5.04	0.177	3.51%
www.uol.com.br	1.66	0	11.32	0.41	3.62%
www.vietq.vn	2.13	0	7.36	0.199	2.70%
www.weeb.tv	6.27	0	3.97	0.089	2.24%
www.wmaraci.com	1.36	0	2.23	0.157	7.04%
www.wral.com	4.73	0.58	12.26	0.525	4.28%
www.yahoo.com	4.75	0	4.2	0.348	8.29%

Observe that after a web page is randomized, Table 4 and Fig. 16 illustrate a significant drop of the visual deviation. In Table 4, we noted many distance scores between snapshots drop to zero after the web pages are randomized. This indicates that WebRanz can effectively prevent Ad-blocking, display Ads to end users and preserve the visual layout of the original web pages. For those non-zero distance scores after randomization, we manually examine the corresponding web pages. We found their visual differences result from CSS and JavaScript. In particular, we noted that some web pages

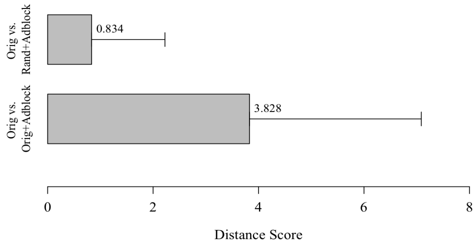
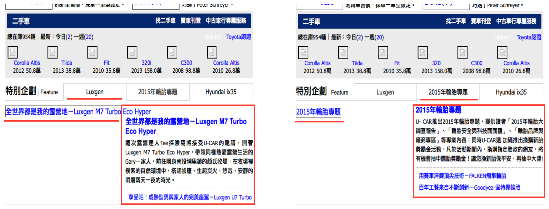


Figure 16: The average distance between snapshots. The average distance between original page and the page with Adblock is 3.828. The average distance between the original page and the randomized page with Adblock is 0.834. The standard deviations are 3.258 and 1.391 respectively.



全世界都是我的露營地－Luxgen M7 Turbo Eco Hyper

全世界都是我的露營地－Luxgen M7 Turbo Eco Hyper

這次露營達人Tee探險窩將接受U-CAR的邀請，開著Luxgen M7 Turbo Eco Hyper，帶領同樣熱愛露營生活的Gary一家人，前往隱身南投埔里鎮的顏氏牧場，在牧場裡樸素的自然環境中，搭起帳篷、生起炭火，悠哉、安靜的消磨兩天一夜的時光。

享受吧！成熟型男與家人的完美座駕－Luxgen U7 Turbo

2015年輪胎專題

2015年輪胎專題

U- CAR推出2015年輪胎專題，提供讀者「2015年輪胎大調查報告」、「輪胎安全與科技面面觀」、「輪胎品牌與廠商專區」等專業內容；同時U-CAR還 加碼推出換購新胎獎勵金活動，凡於活動期間內，換購指定胎款的網友，將有機會抽中購胎獎勵金！讓您換新胎保平安，再抽中大獎!

用賽車淬鍊頂尖技術－FALKEN飛隼輪胎

百年工藝來自不斷創新—Goodyear固特異輪胎

Figure 17: Distance score is affected by sliding panels on www.u-car.com.tw. Left: no Adblock. Right: after randomization with Adblock. The distance score of the two snapshots is 0.516.

employ publicly available JavaScript and CSS library (e.g., YUI [7]) to develop animation such as sliding panels whose dynamics introduce the slight differences between snapshots. As a result, the non-zero distance scores do not suggest the failure of WebRanz. Fig. 17 shows a typical example for the differences between an original page (without Adblock) and the randomized page (with Adblock). Observe that all Ads are unblocked in both images. The differences are caused by that the JavaScript in the page decided to display different Ad contents in the two loads. We suspect that a faithful replay technique that can get rid of non-determinism should be able to generate a distance score close to 0.

4.2.2 Efficiency

Fig. 19 illustrates the average latency of rendering a web page before and after web page randomization. For each page, we load it 10 times and take the average. As shown in the figure, a web browser takes longer time to render a web page when it is randomized because (1) page randomization introduces additional JavaScript and the browser needs to take extra time to parse and render the code; (2) randomization needs to intercept Ad rendering on the fly. As the code size of additional JavaScript is small and the interception is lightweight, the overhead of rendering a randomized web page is negligible (1.38%). Besides, although loading dynamic contents introduces deviations in page loading time, it affects the overhead in a much smaller scale.

Table 4 shows the randomization latency introduced on the server side. On average, this randomization latency is 235 milliseconds. As this latency influences the response to page requests, we compare this latency with the page response time. Table 4 (column overhead) shows the overhead of page response. On average, server side randomization introduces negligible overhead to web page response (4.76%).

Finally, we also investigate the relation between server side randomization and the size of web page. As shown in Fig. 18, the randomization latency is dependent upon the size of web page because WebRanz needs to mutate the attributes of DOM elements. A larger web page may contain more DOM elements and thus WebRanz needs to take more time to parse content.

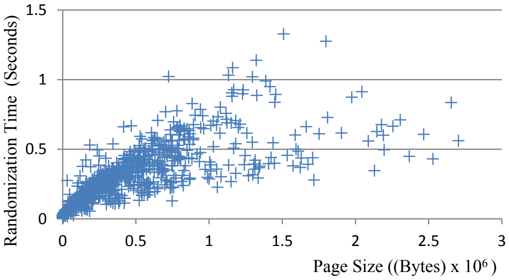


Figure 18: Randomization latency vs. page size.

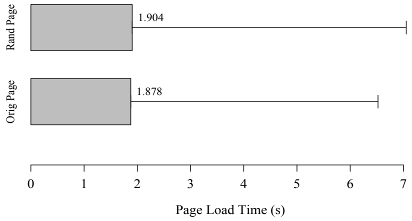


Figure 19: Page load latency before (avg:1.878, std:4.645) and after (avg:1.904, std:5.149) randomization.

4.3 Web Bot Prevention

Next we investigate how WebRanz benefits the defense against web bots. From open-source projects, we selected 8 representative unwanted bots targeted at popular sites. We summarize these bots in Table 5. Column Web Libraries indicates the utilities that a bot relies on. Column Type represents the way that a bot locates a DOM element. Column LOC describes the code size.

We run the bots on both the original and the randomized versions of the target pages. Table 6 summarizes our findings. Succeed? shows whether a script can accomplish its task. To understand why a script fails on the randomized version, we identify the shortest execution path to complete a task. We then count the number of critical element look-ups (i.e. that cannot be circumvented) and the number of successful look-ups before failing along the path. We present the numbers in Look-ups: Total means the total number of look-ups; Passed means the number of look-ups working correctly. Finally, Reason shows why the pattern fails.

As shown in the result, all content-sensitive bots are broken on the pages after randomization. Almost all failed at a very early stage except one that finished 50% of the look-ups. The results show that id and class are important selectors. Randomizing their values is an effective countermeasure against web bots. Note that some look-ups succeeded because the selectors used are very general (e.g. “url:*”).

5. DISCUSSIONS

Our experiments show the effectiveness of WebRanz. We argue that it helps maintaining the health of web ecosystem. However, it is merely one step towards our ultimate goal of constituting a fair and sustainable Ad delivery mechanism and preventing web bots. It has the following limitations.

Table 5: Characteristics of Content-sensitive Bots

Bot	Target	Web Libraries	Type	LOC
[9]	Lenovo Outlet	urllib, webbrowser	regex matching	348
[36]	Google Search	Selenium	element selector	517
[36]	Google Search	BeautifulSoup	element selector	3066
[31]	Yelp	BeautifulSoup	element selector	266
[14]	Amazon	—	regex matching	250
[29]	Twitter	BeautifulSoup, Selenium	element selector	277
[26]	NewEgg	BeautifulSoup	element selector	215
[16]	Groupon	BeautifulSoup	element selector	106

Table 6: Results on the original and randomized page

Target	Succeed?		Look-ups		Reason
	Orig	Rand	Total	Passed	
Lenovo Outlet	√	×	8	4	class
Google Search	√	×	2	0	class
Google Search	√	×	2	0	id
Yelp	√	×	27	0	class
Amazon	√	×	10	1	class
Twitter	√	×	18	5	class
NewEgg	√	×	9	1	class
Groupon	√	×	6	0	class

First, WebRanz is not intended to distinguish legitimate Ads from the unsolicited ones. For example, Adware may leverage WebRanz to circumvent Ad-blockers. We argue this is an orthogonal challenge beyond the scope of the paper.

Second, to preserve web page appearances and functionalities, WebRanz has to send the mappings (from the original attributes to their randomized versions) and JavaScript code that performs the runtime de-randomization to the client side. In other words, they are accessible by the Ad-blockers and web bots, and theoretically can be reverse engineered and circumvented. However, we argue that reverse engineering WebRanz is impractical because the mappings and the de-randomization logic are also randomized and encrypted differently each time a page is loaded. As such, reverse engineering can hardly be automated. Even if reverse engineering may occasionally succeed, the information becomes useless when the page is loaded another time.

Third, Ad-blockers and web bots may inject in-page script to probe the mappings on the fly or simply reuse the instrumented APIs. For example, to apply an element hiding rule, Adblock can inject in-page script to invoke the instrumented APIs which translate the ids and classes in the rules to the randomized versions. However, since Adblock does not know which rules may apply for a given page, it has to test each rule, which is prohibitively expensive (due to the sheer volume of the rules). The resulting user experience degradation would easily force the end user to uninstall Adblock.

Fourth, WebRanz cannot block web bots if they only use general rules, even though this does not happen in the real world according to our experience.

6. RELATED WORK

To the best of our knowledge, the line of work most closely related to ours is anti-adblocker solutions [15, 24, 11]. They exploit the power of in-page JavaScript to identify ad-blocker installers and then demand them turn off their ad-blockers or subscribe to paywalls. Unlike existing anti-adblockers, which aim to discover and lock down ad-blocker installers, WebRanz helps websites deliver the originally-intended ads by preventing ad-blockers from expunging them.

Another relevant work is PolyRef, a polymorphic defense against automated attacks on the web [35]. PolyRef utilizes polymorphism to dynamically reshape sensitive page content and thus impede certain class of automated attacks. Different from WebRanz that randomizes DOM attributes and overwrites native JavaScript APIs for the purpose of handling dependency between JavaScript and HTML, PolyRef achieves web content randomization by directly reshaping DOM attributes and at the same time updating corresponding JavaScript because they assume there is no dependency between third-party JavaScript and obfuscated DOM attributes. Ads on web pages are tied to third-party JavaScript. As a result, PolyRef cannot be general and applied to the context of ad-blocking. In addition, PolyRef only reshape sensitive web content typically resistant to JavaScript dynamics. Thus, it could ensure partially obfuscated web page functional even though neglecting to deal with JavaScript dynamics. Ads are typically rendered by dynamic JavaScript and failure to deal with JavaScript dynamics could make obfuscated web pages dysfunctional. Consequently, PolyRef is unlikely to be effective against ad-blocking.

Last but not least, our work is also relevant to the research in moving target defenses in web applications [34, 32, 33]. To defend Cross-Site Scripting attacks, Portner et al. [32] mutate JavaScript lexical rules uniquely for different users. Taguinod et al. [33] propose to change the underlying language implementation of the web application with the goal of preventing certain categories of vulnerabilities from being effectively exploited. However, these two solutions [32, 33] need system modifications which are not practical for circumventing Ad-blocking. NOMAD [34] targets at preventing web bots from imitating a real user's actions of submitting an HTML form. It randomizes HTML form element parameters in each session but does not handle HTML loaded dynamically. Therefore, this technique does not apply to bypassing Ad-blocking.

7. CONCLUSION

In this paper, we present WebRanz, a technical approach that allows websites to fend off Ad-blockers and serve their originally-intended Ads. Our work is motivated by Ad-blockers that threaten advertising-dependent Internet services and technologies. The main idea of WebRanz is to constantly reshape web pages through a randomization mechanism, so that Ad-blockers never identify the pre-determined patterns that they use to barricade Ads. Our randomization mechanism preserves the layout of web pages as well as the dependency between JavaScript and HTML with negligible overhead. As a result, it has no impact on user experience and could potentially help website owners regain their revenue lost to ad-blockers. Unwanted bot scraping share common characteristics with Ad-blocking. Our results also show that WebRanz helps defending against such bots.

8. ACKNOWLEDGMENTS

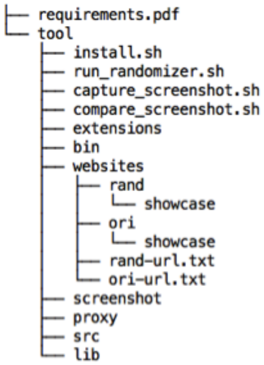
We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 14096 68, 1320444, 1320306, and 1117065, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. P. Eugster was partly supported by European Research Council under grant FP7-617805. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

9. ARTIFACT DESCRIPTION

WebRanz is a tool for randomizing Web pages for better advertisement delivery and web bot prevention. WebRanz instruments the DOM APIs of in-page scripts and randomizes the signatures of web elements. The instrumentations are performed on both server side and client side. At the same time, WebRanz preserves the appearances and functionalities of web pages.

9.1 Directory Structure

We provide a downloadable link [2] which includes a zip file webranz.zip. The structure of the zip file is as follows:



```
├── requirements.pdf
└── tool
    ├── install.sh
    ├── run_randomizer.sh
    ├── capture_screenshot.sh
    ├── compare_screenshot.sh
    ├── extensions
    ├── bin
    ├── websites
    │   ├── rand
    │   │   └── showcase
    │   ├── ori
    │   │   └── showcase
    │   ├── rand-url.txt
    │   └── ori-url.txt
    ├── screenshot
    ├── proxy
    ├── src
    └── lib
```

Figure 20: WebRanz source code directory.

- requirements.pdf: Required software with installation links (Note that we also provide a VM [2] that contains all required libraries and programs).
- tool/install.sh: Install node modules
- tool/websites/[ori|rand]/showcase/: The original and randomized version of the websites
- tool/websites/[ori|rand]-url.txt: URLs of the original websites and the randomized websites
- tool/screenshot/: Screen shots of the websites. Each web page has three different screen shots. Details could be found in Section 9.2.3.
- tool/run_randomizer.sh: Randomize sample websites in tool/websites/ori/showcase/
- tool/[capture|compare]_screenshot.sh: Capture and compare screen shots
- tool/extensions: Firefox Add-on installation file
- tool/[lib|src|proxy|bin]/: WebRanz source code and executables

9.2 Run WebRanz

9.2.1 Installation

Run the installation script in directory tool/ to install node.js modules:

```
$ bash install.sh
```

We also provide a VM (Ubuntu) and installed all dependencies. Its home directory is identical to the one listed in Section 9.1. The login and password are **webranz** and **webranz**, respectively.

9.2.2 Run Randomizer

Invoke the randomizer by running the script:

```
$ bash run_randomizer.sh
```

The script randomizes all the websites under website/ori /showcase/ and generates the output in website/rand/showcase/.

An Example. Fig. 21 shows the screen shots of www.yahoo.com before and after randomization: A banner ad on the top center is enclosed in a <div> element with id of "ad-north-base", which matches an Adblock Plus filter and thus blocked. After randomization, the id value is encrypted to a random value "3457323009" so that it will not be blocked anymore. Among 221 web pages we evaluated, we randomly selected 50 web pages and host on our server [2]. The URLs of the original and the randomized versions are in tool/websites/[ori|rand]-url.txt. The pre-generated screen shots for the web pages in Table 4 are also provided in the VM.

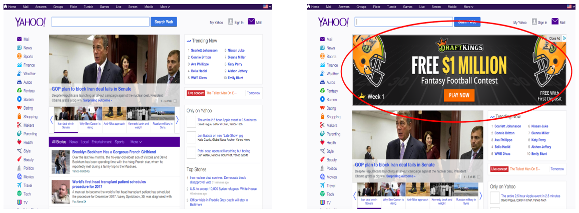


Figure 21: Top Banner Ad on www.yahoo.com is hidden due to a match on an Adblock Plus filter. Left: original web page. Right: randomized web page. Both with Adblock Plus enabled.

9.2.3 Capture Web Page Screen shots

To evaluate the effectiveness, for each web site, we compare the visual differences among screen shots taken in 3 various settings: (1) ***site-1.png***: the original page without Adblock Plus; (2) ***site-2.png***: the original page with Adblock Plus enabled; (3) ***site-3.png***: randomized page with Adblock Plus enabled. We prepared a script ($ bash capture_screenshot.sh) built atop Selenium to capture above screen shots automatically. It drives the browser, visits URLs listed in tool/websites/ori-url.txt and tool/websites/rand-url.txt, and takes screen shots. Screen shots will be generated and stored in tool/screenshot/.

9.2.4 Compare Image Difference

The screen shots of the 50 websites can be found in tool/screenshot. For each site, there are 3 screen shots as described in Section 9.2.3. The expected result is that ***site-1.png*** should be same as or similar to ***site-3.png***, while ***site-2.png*** is different from them as ads are blocked by Adblock Plus. Invoke the script that compares image differences as follows.

$ bash compare_screenshot.sh, and the difference score is printed in the console.

10. REFERENCES

[1] Adblock Plus. https://adblockplus.org/.

[2] WebRanz Project Page. https://www.cs.purdue.edu/homes/wang1315/webranz/.

[3] RFC 2397 - The "data" URL scheme. http://tools.ietf.org/html/rfc2397.

[4] Htmlparser2. https://www.npmjs.com/package/htmlparser2.

[5] CSS Parser. https://github.com/reworkcss/css.

[6] Lucene Image Retrieval. https://github.com/dermotte/lire.

[7] YUI. http://yuilibrary.com/.

[8] A former Googler has declared war on ad blockers with a new startup that tackles them in an unorthodox way. http://www.businessinsider.com/former-google-exec-launches-sourcepoint-with-10-million-series-a-funding-2015-6.

[9] A simple python crawler for Lenovo outlet website. https://github.com/agwlm/lenovo_crawler.

[10] Ad Blockers and the Nuisance at the Heart of the Modern Web. http://www.nytimes.com/2015/08/20/technology/personaltech/ad-blockers-and-the-nuisance-at-the-heart-of-the-modern-web.html.

[11] Adblock Blocker. https://wordpress.org/plugins/addblockblocker/.

[12] Adblock Plus Talks Content-Blocking And The Tricky Shift To Mobile. http://techcrunch.com/2015/07/22/adblock-plus-talks-content-blocking-and-the-tricky-shift-to-mobile/. Jul. 2015.

[13] Alexa: The top ranked sites in Music Videos. http://www.alexa.com/topsites/category/Top/Arts/Music/Music_Videos. Aug. 2015.

[14] Amazon reviews downloader and parser. https://github.com/aesuli/Amazon-downloader.

[15] Anti Adblock Script. http://antiblock.org/.

[16] Crawling Groupon to get all Information about all deals in America. https://github.com/mihirkelkar/crawl_groupon.

[17] Data Theft Watch: Web Scraping Attacks Almost Double. http://www.infosecurity-magazine.com/news/data-theft-watch-web-scraping/. Jun, 2015.

[18] Google losing billions in adblocking devil's deal. http://blog.pagefair.com/2015/google-losing-billions-adblock-devils-deal/. Jun. 2015.

[19] Growth of Ad Blocking Adds to Publishers' Worries. http://blogs.wsj.com/cmo/2015/04/09/growth-of-ad-blocking-adds-to-publishers-worries/. Apr. 2015.

[20] How one tweet wiped $8bn off Twitter's value. http://www.bbc.com/news/technology-32511932. Apr, 2015.

[21] IAB Internet Advertising Revenue Report, Q3 2015. http://www.iab.com/news/q3adrevenue/. Dec. 2015.

[22] Incapsula Inc. 2014 Bot Traffic Report: Just the Droids You were Looking for. https://www.incapsula.com/blog/bot-traffic-report-2014.html. Dec. 2014.

[23] Node.js. http://nodejs.org.

[24] Remove Adblock. http://removeadblock.com/.

[25] Selenium - Web Browser Automation. http://www.seleniumhq.org.

[26] Storage Analysis - GB/$ for different sizes and media. http://forre.st/storage.

[27] The 2015 Ad Blocking Report. http://blog.pagefair.com/2015/ad-blocking-report/. Aug. 2015.

[28] The Scraping Threat Report 2015. https://www.scrapesentry.com/wp-content/uploads/2015/06/2015_The_Scraping_Threat_Report.pdf. Jun, 2015.

[29] tScrape. https://github.com/tranberg/tScrape.

[30] Twitter leak demonstrates power of scraper bots. http://www.usatoday.com/story/tech/2015/04/28/twitter-selerity-leak-tweets-earnings/26528903/. Apr, 2015.

[31] YelpCrawl: Exhaustive Yelp! Scraper. https://github.com/codelucas/yelpcrawl.

[32] Joe Portner, Joel Kerr, and Bill Chu. Moving target defense against cross-site scripting attacks (position paper). In *Foundations and Practice of Security - 7th International Symposium, FPS 2014, Montreal, QC, Canada, November 3-5, 2014. Revised Selected Papers*, pages 85–91, 2014.

[33] Marthony Taguinod, Adam DoupÃľ, Ziming Zhao, and Gail-Joon Ahn. Toward a Moving Target Defense for Web Applications. In *Proceedings of 16th IEEE International Conference on Information Reuse and Integration (IRI)*. IEEE, 2015.

[34] Shardul Vikram, Chao Yang, and Guofei Gu. NOMAD: towards non-intrusive moving-target defense against web bots. In *IEEE Conference on Communications and Network Security, CNS 2013, National Harbor, MD, USA, October 14-16, 2013*, pages 55–63, 2013.

[35] Xinran Wang, Tadayoshi Kohno, and Bob Blakley. Polymorphism as a defense for automated attack of websites. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, volume 8479 of *Lecture Notes in Computer Science*, pages 513–530. Springer International Publishing, 2014.

[36] Xinyu Xing, Wei Meng, Dan Doozan, Nick Feamster, Wenke Lee, and Alex C. Snoeren. Exposing Inconsistent Web Search Results with Bobble. In *Proceedings of the 15th International Conference on Passive and Active Measurement - Volume 8362*, PAM 2014, pages 131–140, New York, NY, USA, 2014. Springer-Verlag New York, Inc.