All Your Clicks Belong to Me: Investigating Click Interception on the Web

Mingxue Zhang
Chinese University of Hong Kong

Wei Meng
Chinese University of Hong Kong

Sangho Lee
Microsoft Research

Byoungyoung Lee
Seoul National University
Purdue University

Xinyu Xing
Pennsylvania State University

Abstract

Click is the prominent way that users interact with web applications. For example, we click hyperlinks to navigate among different pages on the Web, click form submission buttons to send data to websites, and click player controls to tune video playback. Clicks are also critical in online advertising, which fuels the revenue of billions of websites. Because of the critical role of clicks in the Web ecosystem, attackers aim to intercept genuine user clicks to either send malicious commands to another application on behalf of the user or fabricate realistic ad click traffic. However, existing studies mainly consider one type of click interceptions in the cross-origin settings via iframes, *i.e.*, clickjacking. This does not comprehensively represent various types of click interceptions that can be launched by malicious third-party JavaScript code.

In this paper, we therefore systematically investigate the click interception practices on the Web. We developed a browser-based analysis framework, OBSERVER, to collect and analyze click related behaviors. Using OBSERVER, we identified three different techniques to intercept user clicks on the Alexa top 250K websites, and detected 437 third-party scripts that intercepted user clicks on 613 websites, which in total receive around 43 million visits on a daily basis.

We revealed that some websites collude with third-party scripts to hijack user clicks for monetization. In particular, our analysis demonstrated that more than 36% of the 3,251 unique click interception URLs were related to online advertising, which is the primary monetization approach on the Web. Further, we discovered that users can be exposed to malicious contents such as scamware through click interceptions. Our research demonstrated that click interception has become an emerging threat to web users.

1 Introduction

Clicking an HTML element is the primary way that users interact with web applications. We click hyperlinks to navigate among different documents that are interconnected through the hyperlinks on the Web. We click form submission buttons (*e.g.*, the Facebook like button and the Twitter tweet button) to share data with websites and other people on the Internet. We click custom user interface components (*e.g.*, the video or audio player controls) to command various web applications.

Since clicks are important in modern web applications, attackers have launched UI redressing attacks, namely Clickjacking [26], to hijack user clicks. In particular, malicious websites trick a user into clicking components (*e.g.*, a Facebook like button) different from what the user perceives to click, in order to send commands on behalf of the user to the different application they secretly embed (typically in an iframe tag). To defend against Clickjacking, a rich collection of works has been proposed, which has shown great performance [1, 3, 10, 15, 29, 30].

Clicks are also critical in one pervasive application—online display advertising, which powers billions of websites on the Internet. The publisher websites earn a commission when a user clicks an advertisement they embed from an online advertising network (ad network in short). However, the ad click-through rate is usually very low, *e.g.*, around 2% in business-to-consumer banner ads [18]. To increase revenue that can be made through ad clicks, malicious websites have used bots to automatically and massively send fake click traffic to the ad networks, which is known as ad click fraud [5, 22, 27]. To combat against click frauds, ad networks have developed advanced techniques to determine the authenticity of click traffic [2, 6, 9, 38]. Consequently, traditional bot-based ad click fraud has then become less effective.

Instead of relying on click bots, attackers recently started to *intercept and redirect* clicks or page visits from real users to fabricate realistic ad clicks. First, they infect a victim user's computer with malware to either force or trick a user into submitting an ad click. For example, some "browser redirect viruses" modify a user's default search engine to a malicious one, redirecting the user to an advertiser's page when the user clicks a search result [19]. Second, malicious third-party iframes can automatically redirect users to an ad page. Similarly, a user's current tab may be automatically redirected to unintended destinations when a script opens a new tab upon click. Google recently released a new version of the Chrome browser to automatically prevent these two types of automatic redirects [8]. Nevertheless, Chrome still cannot detect and prevent other possible ways to intercept user clicks, including but not limited to links modified by third-party scripts, third-party contents disguised as first-party contents, and transparent overlays.

A systematic study on click interceptions is necessary to deeply understand this emerging threat to web users. We aim to develop a system to automatically detect such practices on the Web, and investigate what kinds of techniques are exploited and who are involved in. We first design and develop a system to detect various techniques employed by JavaScript to intercept user clicks. Using this system, we then perform a large-scale measurement with the goal of finding out those practitioners that hijack links and deceive user clicks. Finally, we analyze our measurement results, and explore the intents and consequences hidden behind the click interception practices.

However, it is challenging to perform the aforementioned systematic study because of the dynamic and event-driven characteristics of web applications. First, JavaScript code can be dynamically loaded. Statically analyzing the HTML source code is insufficient to cover all scripts that can intercept user clicks. Second, hyperlinks can be dynamically created and modified by any scripts. To pinpoint the scripts truly accountable for the interception, we need to re-engineer a browser to differentiate the actions of different scripts in runtime. Third, JavaScript can dynamically bind a URL to user click on an arbitrary HTML element through event listeners (handlers). Monitoring hyperlink creation and modification is insufficient to catch all the click interception practices. Last but not least, a web page may contain a large number of event handlers that respond to user clicks. To perform a large-scale comprehensive study, we have to efficiently interact with all those event handlers.

To tackle the challenges mentioned above, we design our analysis framework by customizing an open-sourced Web browser. We first mediate all JavaScript accesses to hyperlinks in a web page in the browser's renderer. In this way, we can identify the *initiator* of the URL associated with each hyperlink. Second, we monitor the creation and execution of JavaScript objects so that we can track down the *provenance* of dynamic inline JavaScript code. Third, we monitor all event handlers registered on every HTML element and hook navigation-related JavaScript APIs. With this design, we can develop an automated approach to monitor the event handlers accordingly, and determine if an event handler might be used to hijack user clicks. Last but not least, we derive the navigation URL without really firing the navigation that is initiated by a user click. This allows us to interact with all the click event handlers in an efficient way. It also helps us understand the reason why a particular user click is of the interest of a script.

In this work, we developed OBSERVER, a prototype of the aforementioned analysis framework by customizing and extending the Chromium browser. Using this framework, we performed a large-scale data crawling on the Alexa top 250K websites. We discovered that 437 third-party scripts exhibited the activities of intercepting user clicks on 613 websites. They combined receive 43 million visits on a daily basis. In particular, we observed that some scripts tricked users into clicking their carefully crafted contents, which were usually disguised as first-party contents, or intentionally implemented as barely visible elements covering first-party elements. In addition, we revealed that these third-party scripts intercepted user clicks in order to monetize user clicks, which is a new practice we observe as committing ad click frauds. It is worth noting that we will make our implementation publicly available.

In summary, this paper makes the following contributions.

- We design and develop OBSERVER, a framework for studying click interception practices. This facilitates our capability in automatically detecting a wide range of click interception cases on various websites.
- We perform a large-scale measurement study to explore and understand how attackers manipulate web pages in the wild and thus intercept user clicks.
- We characterize the activities of click interceptions on top Alexa websites and discover the intents and consequences hidden behind the activities of click interception.

2 Related Work

In this section, we introduce existing studies about how attackers intercept user clicks or generate fake clicks, and how to detect and prevent such attempts. We also explain other studies analyzing how JavaScript libraries are included and what their behaviors are.

Clickjacking. Clickjacking, also known as UI redressing, is a popular attack designed to trick a victim into doing some tasks on another website the user has logged in, bypassing the same-origin policy. It is one type of *inter-page* click interception in which a malicious *first-party website* tricks a victim into clicking components in another website loaded in an *iframe*. For example, a malicious website could load a specific page of a target website via an invisible iframe, and place it on top of a crafted object that looks benign and independent to the target page. The malicious website then can trick a victim into unintentionally clicking the target page via the crafted object to activate some operations defined in that page. Framebusting [29–31] is a well-known defense to prevent clickjacking by disallowing untrusted websites to load specified pages via an iframe. However, framebusting is incompatible with third-party mashup or other techniques that demand cross-origin framing [15]. Rather, other studies including ClickIDS [3] and InContext [10] rely on human perception to verify whether a click was intended by a user. Akhawe *et al.* [1], however, identified that such mechanisms are not comprehensive or suffer from an unacceptable usability cost.

Our research complements these studies by investigating new practices of *intra-page* click interception by *third-party scripts*, which intercept a victim's clicks on components (including iframes) within the same page/frame. Further, we demonstrate that the scripts can use hyperlinks, event listeners, and visual deceptions, to intercept user clicks.

Link Hijacking. Link hijacking is an attack to modify the destination of links on websites. Nikiforakis *et al.* [24] investigated ad-based URL shortening services and discovered link hijacking by an embedded third-party iframe on a "waiting page" through automatic tab redirects, which the new Chrome browser can prevent [8]. Our research demonstrates a new form of link hijacking that modifies all first-party hyperlinks before the user even clicks them, and shows our system can automatically detect them.

Visual Deception. Prior works have studied how visual deceptive contents can be used to intercept user clicks. Duman *et al.* [7] studied trick banners (*e.g.*, download buttons) that look similar to first-party contents, and further proposed a defense based on a supervised classifier. Rafique *et al.* [28] discovered overlay ads and invisible banners in free live-streaming services. Note that our research does not focus on a specific category of visual deceptive contents or services. Moreover, OBSERVER is able to distinguish deceptive contents created by *different scripts* because of its provenance tracking capability, allowing us to detect the real culprits.

Click Fraud and Click Spam. Click fraud and click spam are attempts to raise revenue by submitting fake ad clicks to an ad network. In traditional click fraud, attackers usually operate a botnet to fabricate a large number of ad clicks automatically to an ad network. For example, Pearce *et al.* [27] estimated that the ZeroAccess click-fraud botnet incurred advertising losses on the order of $100,000 per day. In click spam, unethical content publishers or ad injection attackers [32, 37] either trick the users into clicking ads, or use malware to click ads on behalf of the users. Click spams could even lead victim users to malicious ads [16, 37, 39]. Defenses against click fraud and click spam mostly aim to distinguish fake clicks from real clicks by analyzing their patterns [5, 6, 12, 17, 21, 22, 38]. Thus, attackers try to make their click traffic look as benign as possible. For example, some attacks hijack real human clicks through rogue DNS servers and redirect them to ad networks [2]. We discover that the click interception techniques we identify have already been used by attackers for generating realistic click traffic in the wild.

JavaScript Inclusion and Behavior Analysis. Numerous researchers have analyzed the behavior of third-party JavaScript libraries and how they are included. Nikiforakis *et al.* [23] investigated the Alexa Top 10K websites to discover how many remote JavaScript libraries they include and from which library hosting servers they include the scripts. They also assessed the security of those hosting servers to infer whether they could serve malicious JavaScript code. Lauinger *et al.* [14] and Retire.js [25] studied the semantics of JavaScript libraries, by considering whether hosted JavaScript libraries are outdated or have known vulnerabilities. Systems like EvilSeed [11] and Revolver [13] focus on detecting malicious web pages using content or code similarities. Also, ScriptInspector [40] inspects API calls from third-party scripts to study how they interact with critical resources, such as the DOM, local storage and network. It is able to detect suspicious third-party scripts that violate some access policies. These studies, however, rely on the origin of a JavaScript script to determine whether it is a first-party or third-party script. This implies that they cannot properly handle the situation where a website includes JavaScript libraries from their subdomains or other domains, and from other CDNs (§4.2). Furthermore, unlike ScriptInspector, OBSERVER can track the dynamic creation of JavaScript objects and DOM elements such that it can accurately attribute hyperlink modifications and event listener registrations.

3 Overview of OBSERVER

In this section, we present OBSERVER, an analysis framework that is designed to comprehensively log all potential click-interception-related events performed by JavaScript code in a best-effort manner. OBSERVER focuses on three fundamental actions that JavaScript code might rely on to intercept clicks: 1) *modifying* an existing hyperlink in a page; 2) *creating* a new hyperlink in a page; and 3) *registering* an event handler to an HTML element to hook a user click. Whenever OBSERVER identifies any of such actions, it *tags* the corresponding element with the unique identifier of the script that *initiates* the action. Further, OBSERVER logs the reaction (*i.e.*, *navigation*) of a page after it *intentionally* clicks a hyperlink or an element associated with an event handler in the page, to know the URLs to which a click interceptor aims to lead a user.

In the following, we first demonstrate our threat model (§3.1). We then describe how OBSERVER monitors the JavaScript accesses to HTML anchor elements (§3.2), and how it tracks the dynamic creation of HTML anchor elements and HTML script elements (§3.3). Further, we show how OBSERVER hooks several APIs to catch navigation-related JavaScript event listeners (§3.4). Finally, we detail our prototype implementation based on the Chromium browser (§3.5).

3.1 Threat Model

In our threat model, we consider only click interception activities performed by *third-party* scripts as malicious. Although first-party websites might exhibit similar activities to intercept user clicks, we do not consider them as malicious, because they have the full privilege to control their own applications. Nevertheless, OBSERVER can comprehensively collect all data related to click interception.

3.2 Recording Accesses to HTML Anchor Elements

Modifying a hyperlink in a web page is one of the most explicit methods to intercept and navigate a user click into a different URL rather than the original one. OBSERVER aims to record any accesses to all hyperlinks in a web page to detect any such attempts. In HTML, a hyperlink is defined with an anchor element (*i.e.*, an <a> tag), and its `href` attribute specifies the associated destination URL. Thus, by monitoring and recording which script modifies the `href` attribute of an <a> tag, OBSERVER is able to recognize a script's potential click interception.

JavaScript can modify the `href` attribute through DOM APIs in several ways. We use the keyword `a` to represent an HTML Anchor Element object and the keyword `url` to represent a URL string in the following examples. First, a script can directly assign a new value to the attribute as in `a.href = url;`, or in `a.attributes["href"] = url;`. Second, it may also call the `setAttribute()` API as in `a.setAttribute("href", url)` to perform a similar operation. Note that developers may leverage APIs defined in some third-party JavaScript libraries, *e.g.*, jQuery, to change the attribute. OBSERVER can cover all these wrapper libraries because they would still need to call the above APIs defined in the DOM standard, which is implemented by all browsers to ensure cross-browser compatibility.

OBSERVER hooks all these DOM APIs to monitor modifications to the `href` attribute of <a> tags in the DOM. Specifically, it intercepts any call to such an API. Once intercepted, it inspects the current JavaScript call stack to reason about the origin of API invocation. It locates the bottom JavaScript frame in the call stack to find the JavaScript function that initiates the API call.

Script Identification. To attribute the API access to a specific script, we need to obtain the identity of the accessing JavaScript code. OBSERVER assigns a `scriptID` to each script object to uniquely identify it in the JavaScript runtime. In HTML, JavaScript code is usually enclosed between <script> and </script> tags as an inline script, or stored in an external JavaScript file and loaded with <script> tags as an external script. Each <script> tag is compiled into an individual JavaScript object in the JavaScript engine. There are also other types of inline JavaScript code. For example, JavaScript code can be written as the on-event listener attributes of HTML elements. This kind of inline scripts that are not wrapped within a <script> tag are also compiled into separate JavaScript objects, which are identified by the unique scriptIDs.

OBSERVER associates the scriptID of a script with its `sourceURL`, which is the URL the browser uses to load the remote JavaScript code. The `sourceURL` of an inline script, however, is empty. Instead, we use the URL of the embedding frame, *i.e.*, the URL that the browser uses to load the HTML document into the embedding frame, as the `sourceURL` of *static* inline scripts. However, inline scripts can also be created on-the-fly by JavaScript. We will discuss how we attribute a DOM access to a dynamic inline script in §3.3.2. Besides the scriptID, we also record the row number, column number, and name of the function in the accessing script in a *shadow* data store associated with the element. It is worth noting that JavaScript code cannot modify the shadow data store because it is a C++ data structure that is not writable on the JavaScript side.

3.3 Tracking Dynamic Element Creation

Dynamically creating a new hyperlink in a web page is another method to intercept a user click. In short, OBSERVER considers direct and indirect approaches that a script can exploit to achieve this goal: 1) creating a hyperlink and 2) creating a script that creates a hyperlink.

3.3.1 HTML Anchor Elements

JavaScript code can dynamically create any HTML elements, including an anchor element, in a web page. Specifically, JavaScript can insert a new <a> tag into the DOM tree of a web page through APIs such as `document.write("<a>...</a>")` and `document.createElement("a")`. A script can even replace the entire element with a new element by changing the `outerHTML` attribute of it, *e.g.*, `a.outerHTML = '<a href="' + url + '">...</a>'`. These techniques could be exploited by scripts as another way to intercept user clicks instead of modifying existing hyperlinks. Thus, OBSERVER needs to track the dynamic creation of <a> tags in the browser.

OBSERVER attaches a *shadow* `initiator` attribute to each anchor element in the DOM tree to represent the creator of the object. The initiator attribute is the scriptID of the script that creates the corresponding element. OBSERVER assigns a special initiator value—`0`, which represents the owner of a document—to all static elements that are built by the browser parser. The static <a> tags are the *first-party* hyperlinks. OBSERVER intercepts all the element creation APIs in the web browser to find the initiating JavaScript frame in the call stack. The scriptID of the initiating script is used as the initiator of the dynamically created elements (hyperlinks). OBSERVER would also record any accesses to the `href` attribute of the dynamically created anchor elements.

3.3.2 JavaScript

JavaScript code can also be dynamically generated in web applications, just like HTML elements. Specifically, as one class of HTML elements, new <script> elements can be dynamically created by JavaScript using the same APIs for creating elements. OBSERVER aims to assign unique identifies to all of such dynamically created scripts. If an external script file is loaded from a remote host into a dynamically inserted <script> element, getting its identity is not different from getting the sourceURL of one static <script> element. Some strings can also be dynamically parsed as inline JavaScript code if they are defined as inline event handlers or passed in the call of APIs like `window.eval("...")`.

However, it is not straightforward to tell the identity of a dynamically generated inline script because its sourceURL is blank. To overcome this difficulty, OBSERVER hooks the APIs that are used to generate dynamic scripts. It saves the sourceURL of the JavaScript code that calls the script generation API as the sourceURL of the newly generated inline script. To distinguish the dynamically generated script, or the *child* script (either an inline script or an external script), from the generating script, or the *parent* script (the one that generates the script), OBSERVER records the scriptID of the parent script as the `parentScriptID` attribute of the child script. The parentScriptID of all scripts that are initially statically embedded by the document owner is set to `0`. This allows us to construct a script dependency graph in the analysis.

OBSERVER also logs all accesses to any inline on-event handlers of any DOM object as it does with the `href` attribute of <a> elements. It finds the last script that sets an inline on-event handler as its parent script and derives the sourceURL from it. If no such an entry can be found, OBSERVER sets the script that creates the receiver object as its parent script.

3.4 Monitoring JavaScript Event Listeners

Instead of modifying or creating hyperlinks, a script can register an event listener or handler to an HTML element. The event handler is asynchronously executed whenever there is a user click on the element. In particular, a script may open an arbitrary URL in a new browser window/tab, or send an HTTP request in the background, when a user clicks any element it listens for. Therefore, OBSERVER aims to monitor all event listeners registered by JavaScript code in a page to identify whether they will navigate a user to a different URL according to a user click.

OBSERVER first monitors event listener registration by hooking the `addEventListener()` API and monitoring accesses to the on-event listeners, to identify the scripts that are interested in user interactions. It then intercepts any click-related user events (*e.g.*, *click* and *mousedown*) when they are fired in the web browser and detects the event target element in the DOM tree. Since a script may not necessarily initiate a page navigation in its event handler (*e.g.*, an analytic script), OBSERVER filters those scripts by hooking several APIs that can be used for starting a navigation, *e.g.*, `window.open('...')`, `window.location = '...';`, *etc.* OBSERVER detects the bottom frame in the JavaScript call stack and further constructs and logs the navigation URL in these APIs in the shadow data store of the target element.

One challenge we met in our design is that one event handler can be activated multiple times. In the DOM, the events are propagated in three phases: capturing, target, and bubbling. For example, in the capturing phase, an event is propagated from the root node in the DOM tree—the <html> node, then through any intermediate parent nodes, before finally reaching the target node. An event handler registered in the capturing phase at the <html> tag will always be triggered whenever any of its child elements is clicked[1]. To avoid activating such event listeners multiple times, OBSERVER would skip calling an event listener at a node if the `Event.currentTarget` object (*i.e.*, the current node) is different from the `Event.target` object in event propagation. We further set a flag in OBSERVER to abort all page navigations, including those caused by clicking the <a> tags, after the navigation URLs are saved in the logs. This enables us to efficiently interact with all elements in a web page without really visiting the linked URLs.

3.5 Implementation

We implement a prototype of OBSERVER in the Chromium browser (version 64.0.3282.186). We will release our prototype implementation as an open source software. We implement OBSERVER in a full-fledged browser to escape any artificial result that might be caused by using a simpler and uncommon user agent. We add several custom attributes (*e.g.*, initiator, accessLog, scriptID, parentScriptID, sourceURL) to the Node[2] objects to save the monitoring data. All these custom attributes can be read but not written by JavaScript for further analysis. For performance concerns, we implement a lazy update mechanism for setting the above attributes. The values of these attributes are kept in the hidden attribute members of the modified C++ classes. They are updated in the DOM tree only when the attributes are first accessed by JavaScript.

We hook the above DOM APIs by inserting custom monitoring code in the C++ implementation of the V8 binding layer between the V8 JavaScript engine and the DOM implementation in WebKit. The custom monitoring code identifies the JavaScript caller by fetching the scriptID of the bottom frame in the JavaScript call stack. It appends the logs of accesses to the `href` attribute and the inline on-event handlers to the hidden `accessLog` attribute of the corresponding DOM object. The code sets the `initiator` attribute of an anchor element when it is created by either JavaScript code or the browser parser. Furthermore, the sourceURL and parentScrip-

[1] An event handler registered in the bubbling phase at a parent node may not be activated because the event propagation can be stopped by some other event handler registered at its child node.

[2] Node is the base class of HTML elements in WebKit.

tID of all scripts are stored with a <script> object. We further store the scriptID in the sourceURL dictionary at the global Document object.

The prototype of OBSERVER can comprehensively log all click-interception-related events. In the browser, a click-driven navigation can be started by the built-in default event handler of anchor elements (hyperlinks) and the developer-defined event handlers, which we have introduced in §3.2 and §3.4. OBSERVER ensures complete mediation of element accesses and event handler registrations in the C++ implementation of the corresponding DOM APIs (including the built-in default event handler), which cannot be bypassed by any JavaScript code. In other words, the browser must go through the underlying C++ APIs and our monitoring code when JavaScript code accesses any hyperlink or registers an EventListener to any HTML element.

4 Methodology

In order to study the click interception problems in the wild, we perform a large-scale data crawling of the Alexa top 250K websites. We describe our data collection method in §4.1, how we determine the owner and privilege of JavaScript code as well as HTML elements in §4.2, and finally how we detect three classes of click interception in §4.3.

4.1 Data Collection

We use the OBSERVER prototype to collect data for investigating the click interception problem. In particular, we aim to identify all hyperlinks and scripts that react on user clicks, and the destination URLs that the browser would visit after the clicks. We leverage the Selenium WebDriver Python binding to automatically drive OBSERVER and interact with the web page it renders. To this end, we run our analysis framework on a 64 core CPU Linux server and collect data from the Alexa top 250K websites.

We collect data in two phases for each web page: 1) collecting *default data* right after page rendering; and 2) collecting *reaction data* by interacting with a rendered page. In each page navigation, we first asks OBSERVER to wait for a page to be completely rendered by the browser for up to 45 seconds. After that, we insert a script into the page to traverse the DOM tree in pre-order to collect all the data OBSERVER has logged with each element. In addition, we log for each element several display properties (*e.g.*, width, height, position, opacity, *etc.*) to study additional tricks that may be used to intercept user clicks (*e.g.*, some third-party contents overlap with or appear similar to first-party contents). We then save a snapshot of the current DOM tree into an external HTML file as well as a full-page screenshot for further analysis.

Next, we interact with a rendered page to collect data about how the page reacts to our clicks, such as navigation and DOM modification. We disable the navigation flag in OBSERVER to deactivate real navigations that may be caused by event handlers or hyperlinks. We then automatically click all elements in the DOM tree through Selenium to trigger the click event listeners and hyperlink navigations to collect navigation logs. For each navigation triggered by a click, we log the information regarding the navigation URL, the clicked element, and, if exist, the corresponding event listeners and scripts that initiate the navigation. In addition, we traverse the DOM tree again, as we do in the first phase, to identify whether scripts update the DOM elements due to user clicks.

4.2 Third-party Content Detection

In this section, we explain our techniques to distinguish first-party scripts/contents from third-party scripts/contents, which is necessary to detect click interceptions driven by third-party scripts. A naïve technique that merely relies on the exact origin of scripts is not enough because a website frequently loads its own scripts from its subdomains, its different domains, and domains operated by others such as content delivery network (CDN) services. For example, the main page of https://www.google.com/ includes scripts from its subdomain apis.google.com and its CDN domain gstatic.com. If we use only origin information, we may misidentify these scripts as third-party scripts. We aim to solve this problem using *domain substring matching* and *DNS record matching*.

Domain substring matching is a heuristic technique to infer that a remote script is a first-party script if the remote script's domain name is similar to the current page's domain name. It first checks whether the main domain names of a remote script and the current page are the same while excluding domain suffixes. For example, a script loaded from https://apis.google.com/ on https://www.google.co.jp/ is determined as a first-party script because its main domain name excluding the suffix *com* is *google*, which is identical to that of the current page excluding the suffix *co.jp*. Second, it tests whether the proper subdomain name of a remote script consists of the main domain name of the current page without suffixes, to come up with CDN practices that maintain custom subdomain names for individual websites. For example, a script loaded from https://static-global-s-msn-com.akamaized.net/ on https://www.msn.com/ are inferred as a first-party script because the proper subdomain name *static-global-s-msn-com* contains the main domain name *msn*. We do realize that our technique has limitations, which we will discuss in §6.

DNS record matching leverages several DNS records to decide whether two distinct domains are operated by the same organization. Specifically, we inspect the DNS SOA records [36] and the DNS NS records [34] of the two hostnames (domain names). An SOA record includes the email address used to register the domain. Many organizations would use the same email address to register multiple domains. For instance, the SOA email addresses of google.com and gstatic.com are both *dns-admin@google.com*. However, there are also exceptions.

Different organizations may use the same Managed DNS providers [35] to register domains. Accordingly, their SOA same email addresses are identical. For example, both dropbox.com and bitbucket.org use *awsdns-hostmaster@amazon.com* as their SOA email address.

We address this limitation by further examining if the name server (NS) records of a script/URL and the first-party web page have an intersection. Specifically, we use the *domain name* instead of the full hostname of a NS, because one domain may use several NSs from a large pool. If the first-party domain name is found in a common NS, we mark the external script as a first-party script. For instance, both gstatic.com and google.com use NSs *nsX.google.com*, where *X* is a numeric value. Therefore, we determine the two domains belong to the same organization because they have a common NS domain name—*google.com*, and an identical SOA email address. Note that we exclude all common NSs that are operated by any known managed or dynamic DNS providers.

Dynamic Element. Recognizing the sources of dynamic elements is also important to identify cross-party accesses. We classify dynamic elements into two groups based on which parties their initiating scripts belong to. This allows us to distinguish first-party contents from third-party contents.

4.3 Click Interception Detection

Normally, a user may explicitly click a hyperlink to navigate to another web page, or click some components such as images or buttons to interact with the current web page. However, some scripts may deliberately intercept a user's clicks to override the default action that the user may expect. Furthermore, a user could also be fooled by a script into clicking some components she/he would not click. We designate such undesired click manipulation caused by privilege abuse as **click interception** in web applications. As discussed earlier, we do not consider click interceptions exhibited by first-party scripts as malicious.

Based on how a user click could be manipulated, we categorize click interception into three classes—interception by hyperlinks, interception by event handlers, and interception by visual deception. In particular, a script can intercept user click by 1) using an existing hyperlink or creating a new hyperlink; 2) registering a click event handler with an element; and 3) manipulating the UI to deceive a user into clicking elements controlled by the script.

In the following, we explain the methods to detect the three classes of click interception. Specifically, we leverage the *navigation URL* and the *navigation APIs*[3] (§3.4), and the display properties of the element (§4.1).

4.3.1 Interception by Hyperlinks

In general, a script can intercept user clicks with hyperlinks in two ways: modifying one existing (first-party) hyperlink, and adding one hyperlink to a huge element.

Modifying Existing Hyperlinks. A *third-party* script can intercept a user's click through a *first-party* hyperlink by overwriting the *href* attribute. A *third-party* script might also employ a similar approach to intercept a user's click on *another third-party* hyperlink. Therefore, we search in the *href* attribute log of an anchor element the last script that modifies its value. If a (different[4]) third-party script is found, the script is marked as one click interception script. We use the technique in §4.2 to determine if the script and the anchor element belong to the same organization. A third-party script might also intercept a user's click through attaching an event listener to a first-party hyperlink, which we discuss in the following section. Note that although a *first-party* script may modify a *third-party* hyperlink, we think this is legitimate because the first party as the owner of the web page is entitled to include or remove any third-party contents.

Creating Huge Hyperlinks. A script can trick users into clicking its hyperlink by enclosing a huge clickable element. In particular, it can enclose a significant part of its web page within one <a> tag such that a click on any of the enclosed contents would result in a page navigation that is controlled by it. Therefore, we also check the size of an anchor element relative to the browser window[5]. Specifically, we use 75% as the threshold to detect the suspicious huge hyperlinks that can be used to intercept user clicks. According to our knowledge, most (but not all) links on the web are relatively small compared to the browser window. Therefore, we think 75% is a reasonably large threshold to help quickly identify the suspicious ones. Further, we exclude any hyperlinks pointing to a first party navigation URL, because the first party has the right to use huge hyperlinks in its own pages.

4.3.2 Interception by Event Handlers

The event handlers are the second technique that a script can use to intercept user clicks. However, a script listening for user click may not necessarily navigate the user to another URL. For instance, an analytic script may observe user clicks to determine and log only user engagement within the current page. We leverage the navigation-related APIs to solve this problem.

To start a new navigation, a developer needs to either call the `window.open()` API or change the `location` of the current frame. The two JavaScript DOM APIs are implemented by the C++ methods `LocalDOMWindow::open()` and `Location::SetLocation()` in WebKit, respectively. For each element, we

[3] The default event handler of <a> tags is also considered as one API.

[4] We use the term a *different script* to represent a script of a different organization in the rest of the paper.

[5] We used 1024px x 768px as the browser window size in our experiments.

examine if the two C++ methods are (indirectly) called upon a click on the element. We then extract the navigation URLs from the associated logs.

Third-party Interception Scripts using Event Handlers. We determine a *third-party* script as a click interception script if it (indirectly) calls either one of the above two C++ methods in its click event listener that is added to a *first-party* element. We name such a click event listener as a *navigation event listener*. Similarly, if such a navigation event handler is added to a *third-party* element created by the script of a different organization, the third-party script implementing the event handler is also determined as a click interception script.

Intercepting Huge Elements with Event Handlers. We use the same 75% relative size threshold to detect suspicious huge elements that are registered with a *third-party navigation event handler* and can be used to intercept user clicks. We also filter the elements that are associated with a first-party navigation URL.

4.3.3 Interception by Visual Deception

Third party scripts can also intercept a user's clicks through visual implementation tricks to deceive a user. In particular, the third-party contents are designed in some way such that a user is likely to click. We do not consider first-party contents with similar characteristics malicious because the first-party websites have the complete freedom to design their contents.

This last click interception category could be controversial in our opinion, as some third-party developers may argue that they *do not intend to deceive the end users*. Nevertheless, we still classify such practices as click interception (but not necessarily malicious) because the users *can be deceived through the visual tricks*.

We have identified two possible visual deceptions—mimicry, and transparent overlay. We detect these visual deceptive tricks for each *group of third-party elements*, which are the largest sub DOM tree that consists of only elements of the same third-party script (organization).

Mimicry. Some third-party script would deliberately decorate its elements such that they are almost visually indistinguishable from first-party contents. A user might consequently click these mimic elements. However, the imitating elements are usually not exact copies of some first-party elements. As a result, we cannot use pixel-wise comparison to detect such mimic elements.

We utilize the structural information as well as the display properties of a third-party element group to detect mimicry. Specifically, we compute the relative size of media contents, *e.g.*, images, videos, and iframes, in *a group of third-party elements*, as well as the size of the largest container of them. We then compute the same metrics for any *group of first-party elements* whose root node is a sibling (neighbor) to that of the third-party element group. Next, we calculate a similarity score between the two groups of elements using: 1) the CSS class names of the two root nodes, which are primarily used to describe the representations of HTML elements; 2) the numbers of each kind of media tags, which indicate how media contents are implemented; and 3) the relative sizes of media contents in two groups and the sizes of the largest container nodes, which represent the visual layout of an element group.

We set a threshold learned from our training phase to keep only third-party element groups that are very similar to some first-party element groups. Note that we compute the similarity scores using the display property data before we click the elements to find the elements whose default representation is likely to fool a user. We do acknowledge that there are other features (*e.g.*, the DOM tree structure, color histogram) that may better determine the similarity. However, we find the ones that we select work well in our manual test over a small set of samples. We plan to leverage more sophisticated techniques (*e.g.*, image classification [7]) in our future work.

Transparent Overlay. A third-party script can inject contents that partially overlap with or completely cover first-party contents. In the case that some first-party contents are completely covered, the user might not notice their existence and treat the covering third-party contents as first-party ones. Further, a script can make some of its elements barely visible by setting a small value to their *opacity* style property. Subsequently, a user's click could be delivered to these "hidden" elements when the user is intending to click some other elements beneath them. We detect transparent overlay third-party contents in the following two steps.

First, for each group of third-party elements, we compute the *minimum* portion of a *first-party* element that it overlaps with. Specifically, we scroll the browser window virtually to compute all the possible overlapped regions with each *first-party* element. If the covered portion of a first-party element is always greater than a pre-defined threshold (*e.g.*, 25%), we label this group of third-party elements as overlay elements. Since some third-party scripts may implement components allowing a user to cancel out the overlay elements, we further exclude those that no longer significantly overlap with any first-party element after our automatic clicks, which must include a click on one of such cancel-out buttons if there are any. However, this method may not work well in some cases. For example, the covering elements could first be hidden by a click on a cross button, and later be revealed by another click on another button. We consider it as a limitation and plan to leverage knowledge in computer vision to develop a better automated testing method in our future work.

Next, we detect third-party transparent overlay element groups by comparing the *opacity* value collected in the display properties with a small threshold (*e.g.*, 0.1). A zero opacity value indicates complete transparency. We do not consider elements whose style is *visibility: hidden* or *display: none* because user clicks are not passed to these invisible elements. In addition, we keep only the transparent third-party element groups that are big enough to be easily clickable, *i.e.*, the container size is greater than 1% of the browser window size.

5 Click Interception in the Wild

In this section, we first present our analysis on data collected in our web crawl (§5.1), then characterize click interception by demonstrating *how* different techniques (§5.2) are employed by *which* scripts (§5.3) to intercept user clicks, and finally explain *why* they do it and its consequences (§5.4).

5.1 Dataset

We crawled data from the main pages of Alexa top 250K websites in May 2018. Excluding those that timed out or crashed in our data collection process, we were able to gather valid data of 228,614 (91.45%) websites. We identified *third-party* navigation URLs (the first URL the browser would visit upon a user click) collected in a web page using the method described in §4.2. We obtained 2,065,977 unique third-party navigation URLs, which corresponded to 427,659 unique domains. On average, a web page contains 9.04 third-party navigation URLs, pointing to 1.87 domains.

We visited each of the 2M navigation URLs and recorded both the intermediate redirect URLs and the landing URL. We could not visit 39 URLs in our experiment because of various errors (*e.g.*, HTTP 404 status code, too many redirects, *etc.*). We managed to obtain 1,982,613 unique landing URLs.

We collected 413,075 intermediate redirect URLs (excluding the navigation URLs and the landing URLs) in this process. Specifically, we observed no redirection for 1,263,754 (61.17%) navigation URLs. We encountered at most 29 intermediate hops before we reached a final landing URL.

We detected 2,001,081 distinct third-party scripts that were loaded from 1,170,582 different domains. On each page, there are on average 8.75 third-party scripts.

5.2 Click Interception Techniques

In this section, we demonstrate how the different techniques that we identify in §4.3 are employed for click interception.

5.2.1 Interception by Hyperlinks

We identify three possible ways that a *third-party* script can intercept user clicks through hyperlinks (§4.3.1). In total, we observe that 4,178 hyperlinks on 221 websites were intercepted, which can lead a user to 2,695 distinct *third-party* URLs. We present in Table 1 the breakdown of the 4,178 links and the total number of daily visits to the affected websites[6].

Hyperlink Modifications. Surprisingly, the *href* attribute of 4,027 first-party <a> tags on 100 websites were directly tampered by a third-party script. For instance, the ad URL shortening script https://cdn.adf.ly/js/link-converter.js modified the *href* attribute of one anchor element to http://ay.gy/2155800/... on the website http://magazinweb.net/. Similarly, the third-party script https://cpm4link.com/js/full-page-script.js modified hyperlinks on the website https://www.lnmta.com/ to https://cpm4link.com/full/?api=.... They are obviously privilege abuses. In addition, we find that 31 third-party hyperlinks on 2 websites were modified by a different third-party script. For example, the script https://s7.addthis.com/js/300/addthis_widget.js modified 11 third-party hyperlinks on the website https://www.crazy-net.com/ to https://plus.google.com/110631064773293614230; the script http://media1.admicro.vn/core/log_cafef.js modified 20 third-party hyperlinks on the website http://cafef.vn/ to http://lg1.logging.admicro.vn/nd?nid=.... This indicates that those third-party scripts indiscriminately modify anchor elements to intercept user clicks.

Table 1: Categorization of Click Interception Techniques

Technique	#Cases	#Websites	%Cases	#Visits/day
Hyperlinks	4,178	221	89.52	12,686,591
Modifying 1st-party links	4,027	100	86.29	2,496,620
Modifying 3rd-party links	31	2	0.66	638,247
Inserting huge 3rd-party links	120	119	2.57	9,551,724
Event Handlers	203	172	4.35	5,455,821
On 1st-party nodes	189	161	4.05	4,636,145
On 3rd-party nodes	14	12	0.30	819,676
On huge 3rd-party nodes	0	0	0	0
Visual Deceptions	286	231	6.13	25,269,314
Mimicry	140	87	3.00	16,604,258
Transparent Overlay	146	144	3.13	8,665,056

Huge Hyperlinks. We observe 120 huge *third-party* <a> tags on 119 websites. These anchor elements enclose contents whose size is at least 75% of the browser window size. As a result, a visitor has a very high chance to click such an anchor element. For example, on the website http://torrents73.ru/, the third-party script http://gynax.com/js/MjgxMw==.js created a large anchor, which encloses a huge background image. Users would be directed to another page https://wheel.grand-casino48.com/ upon a click. We also identify that 135 websites used 148 huge *first-party* <a> tags, which we currently consider as legitimate as we discussed in §3.1.

5.2.2 Interception by Event Handlers

We analyze how event handlers are exploited to intercept user clicks. Overall, we find 203 elements across 172 websites were attached with *navigation event handlers*, which would drive a user to a *third-party* URL upon click.

We observe that 189 *first-party* elements of 161 websites were added at least one *third-party* navigation event handler. For example, the third-party script https://smashseek.com/rq/4949 intercepted user clicks on the website https://www1.mydownloadtube.com by adding a navigation event listener to the <html> element. The user's browser would open a new URL (the specific URL changes upon each user click) when

[6] We get the statistics using the SimilarWeb API.

a user clicks any element on this page[7]. Another example is detected on the page http://azasianow.com/, where the third-party script http://fullspeeddownload.com/rq/4297 registered an event handler on the <body> element. We also consider such practices as a type of privilege abuse, as they force a user to visit a URL when the user interacts only with first-party contents. What is worse, even an experienced user with some technical background cannot easily find out that the navigation is actually controlled by a third-party script rather than the website she/he directly visits.

Interestingly, we find on 12 websites that 14 *third-party* elements were attached with navigation event handlers by a third-party script of a different organization. For example, the website https://www.mlbstream.io/ included the third-party script https://amadagasca.com/rgCQwi5INUm04AxMu/5457, which registered an event handler on an <img> element. The user would be directed to https://jackettrain.com/imp/5457/?scontext_r=... upon clicking on that image and finally land at a random website. One possible reason is that the attaching scripts were loaded after the other third-party scripts had inserted those elements, so that they mistakenly attached event handlers to the other third-party elements.

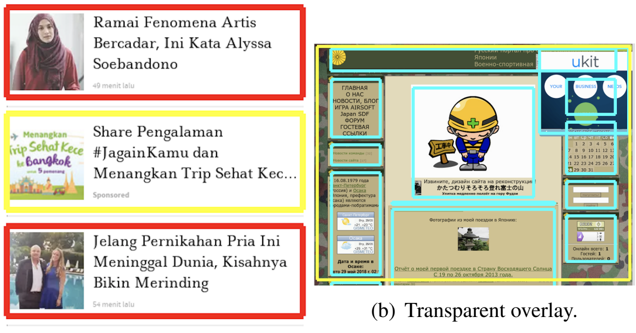
We do not find any third-party script intercepting user clicks by registering navigation event handlers with huge third-party elements. On the other hand, we discover 2 websites added navigation event handlers to their own huge elements. In particular, the websites http://www.force-download.net/ and http://www.force-download.es/ both registered a navigation event handler to the <html> node to intercept user clicks, just as the above-mentioned third-party scripts. Nevertheless, we do not consider them as malicious.

5.2.3 Interception by Visual Deception

We analyze how the two visual deception techniques, *mimicry* and *transparent overlay* (§4.3.3) are used in the wild.

Mimicry. We discover 140 *mimic* third-party element groups on 87 websites. These *third-party* contents are carefully designed to resemble nearby *first-party* contents. Hence, unwary users are very likely to be fooled and consequently click them.

Figure 1(a) shows an example of such a *mimicry* trick that we detect on the website https://www.bintang.com. The contents enclosed within the yellow rectangle were inserted by the third-party script https://securepubads.g.doubleclick.net/gpt/pubads_impl_207.js, whereas those in the red rectangles were the organic first-party contents. Without scrutiny, they just look like each other. The only visual hint for discriminating them is the text *Sponsored*, which was displayed in a very small font size just as the first-party sub captions in the red rectangles. Even though a user may notice this small text, she/he may still decide to click the third-party elements as they appear to be provided directly by the first-party website which



Ramai Fenomena Artis Bercadar, Ini Kata Alyssa Soebandono

49 menit lalu

Share Pengalaman #JagainKamu dan Menangkan Trip Sehat Kec...

Sponsored

Jelang Pernikahan Pria Ini Meninggal Dunia, Kisahnya Bikin Merinding

54 menit lalu

(b) Transparent overlay.

(a) Mimicry.

Figure 1: Examples of visual deceptive third-party contents.

she/he trusts. However, such trust would be abused in this case because those contents were generated solely by a third-party script the user does not know. In particular, the navigation URL was under the full control of this unknown third-party script and could take the user to any (potentially unsafe) page. We will discuss more about the security implication in §5.4.

Transparent Overlay. We detect 146 *transparent overlay* third-party element groups on 144 websites. Specially, they covered a significant portion (at least 25%) of first-party elements regardless of mouse scroll. We could not cancel them out by automatically clicking elements in those websites. Further, they were either completely transparent or translucent with a very low *opacity* style value. What is worse, many of them contained NO user-perceivable content (*e.g.*, texts or images), hence being *transparent*. As a result, they were almost—if not absolutely—invisible and thus difficult to be noticed.

Figure 1(b) demonstrates an example of such a visual trick that we identify on the website http://jgsdf.ucoz.com. The yellow rectangle includes the *third-party* contents that overlapped with the underlying *first-party* contents, which are enclosed by the cyan rectangles. The script that created these third-party contents is http://pl14318198.puserving.com/a2/49/14/a2491467a19ffc3f9fe0dbe66e54bae0.js. Although the overlay third-party contents were not visible in this case, they constantly covered about 50% of the first-party contents in the cyan rectangles no matter how a user scrolled this page. As a result, this script could intercept any click on the covered first-party elements, because the click would be first passed to the overlay third-party elements. When a user clicked within the area of yellow rectangle, an ad link was opened in a new window.

Although third-party scripts can deceive a user with different tricks, the effectiveness can vary dramatically depending on their implementation and the end user's technical background. In general, we think they are less effective compared with the other two direct techniques we have discussed above. In particular, whether the *mimic* contents are deceptive is really subjective. We leave it for our future work to examine

[7] This is not true for elements with other click event listeners that stop the event propagation.

how effective the visual deceptions are on real users.

5.2.4 Evasion of Detection

We also detect a few cases that third-party scripts *selectively* intercepted user clicks. In particular, they would limit the rate at which they intercept the clicks to avoid a user's suspicion. For instance, some scripts would activate the page navigation code in their event handlers only when a user *first* visits a page. This can be easily implemented by dropping a cookie in a user's browser. They might clear this flag after some time (*e.g.*, a day) to reactivate the click interception code. However, we do not have enough data to learn the timeouts they use. We discuss next such a detection evasion example.

The script https://pndelfast.com/riYfAyTH5nYD/4869—included by the website https://torrentcounter.to/—selectively intercepted the user clicks on the background of the website. We observed the interception only when we visited the page with a clean cookie, which suggests the script used a cookie to log click interception status. Interestingly, we find the script was obfuscated to prevent a normal user from analyzing it. We deobfuscate the script (Listing 1), and search for the keyword *cookie*. As expected, we find several functions that are used to control the rate of click interception. Lines 8, 13, and 16 define the functions `"setCookie"`, `"removeCookie"`, and `"getCookie"`, respectively. Line 6 defines the `"timeout"` variable that we suspect to control the interception timeout or interval. It sets the cookie in Line 28, if the return value of the function `init` defined in Line 20 is not true. The cookie is deleted in Line 33. This script also defines several variables, *e.g.*, `"certain_click"`, `"every_x_click"`, `"delay_before_start_clicks"`, `"click_num"`, `"interval_between_ads_clicks"`, which we believe to be used to control click interception. As is limited by the space, we do not discuss in more details how the script works. It would be an interesting research topic to investigate how these scripts cloak their malicious activities to avoid detection.

Summary. We confirm that various click interception techniques have been used in the wild. Third-party scripts intentionally intercepted user clicks using event listeners, and manipulate user clicks through visual deceptions. They also leveraged huge anchor elements to deliberately intercept user clicks. Further, many *third-party* scripts even modified *first-party* hyperlinks to intercept user clicks.

5.3 Click Interception Scripts

In this section, we characterize click interception based on the third-party scripts that intercept user clicks. Further, we investigate how they were embedded to intercept user clicks.

```
1  var _0x3e0d = ["...", "certain_click", "every_x_click"
       , "delay_before_start_clicks", "click_num", "
       interval_between_ads_clicks", "has_adblock", "...
       "];
2  var build = function() {
3    var target = {
4      "data" : {
5        "key" : "cookie",
6        "value" : "timeout"
7      },
8      "setCookie" : function(value, name, path, headers)
         {
9        var cookie = name + "=" + path;
10       headers["cookie"] = cookie;
11     },
12     "removeCookie" : function() {
13       return "dev";
14     },
15     "getCookie" : function(match, href) {
16       var v = match(new RegExp("(?:^|; )" + href["
       replace"](/([.$?*|{}()[]\/+^])/g, "$1") + "
       =([^;]*)"));
17       return v ? decodeURIComponent(v[1]) : undefined;
18     }
19   };
20   var init = function() {
21     var test = new RegExp("\\w+ *\\(\\) *{\\w+
       *['|\"].+['|\"];? *}");
22     return test["test"](target["removeCookie"]["
       toString"]());
23   };
24   target["updateCookie"] = init;
25   var array = "";
26   var _0x418128 = target["updateCookie"]();
27   if (!_0x418128) {
28     target["setCookie"](["*"], "counter", 1);
29   } else {
30     if (_0x418128) {
31       array = target["getCookie"](null, "counter");
32     } else {
33       target["removeCookie"]();
34     }
35   }
36 };
```

Listing 1: A simplified click interception script from https://pndelfast.com.

5.3.1 Third-party Scripts Characterization

Our results in §5.2 demonstrate that third-party scripts leverage all the three techniques to intercept user clicks. We present the statistics of these scripts—the *unique* number of script URLs, origins, and domains in Table 2.

Huge Hyperlinks. We detect 86 unique *third-party* scripts that injected huge <a> tags into their embedding pages. We show the top 5 origins of such scripts in Table 3. The noticeable scripts are those loaded from http://gynax.com. They were found to create one huge <a> element on each of 47 websites they were included. Each <a> tag was enclosed within a <noindex> element, which further contained a full-page image. All the hyperlinks would finally reach https://wheel.28grand-casino.com/, which is an online gambling game website.

Hyperlink Modifications. We detect 57 unique *third-party* scripts that directly intercepted user clicks by modifying *first-party* hyperlinks. We show the top 10 origins of such scripts in Table 4. The top script https://cdn.adf.ly/js/link-converter.js

Table 2: Statistics of unique click interception scripts.

Technique	#URLs	#Origins	#Domains
Hyperlinks	145	76	63
Modifying 1st-party links	57	41	35
Modifying 3rd-party links	2	2	2
Inserting huge 3rd-party links	86	33	26
Event Handlers	106	72	58
On 1st-party nodes	103	69	55
On 3rd-party nodes	7	7	7
On huge 3rd-party nodes	0	0	0
Visual Deceptions	197	173	95
Mimicry	78	60	54
Transparent Overlay	119	114	42

Table 3: Top 3rd-party script origins injecting huge anchors.

Script	#Websites	#Elements
http://gynax.com	47	47
https://securepubads.g.doubleclick.net	7	7
https://yastatic.net	7	7
http://bgrndi.com	6	6
http://js883.guangzizai.com	5	5

was found on 18 websites. *Adf.ly* is a short URL service that helps websites monetize their links. As its name suggests, this script converts *every first-party* hyperlinks to a *third-party* hyperlink. If a user clicks any converted hyperlink, the user would be taken to an intermediary page of *adf.ly* hosted on http://clearload.bid/. This page displayed an advertisement as shown in Figure 2. The user can click the *SKIP AD* button on the right top corner to continue to visit the original *first-party* hyperlink. Many other top scripts in Table 4, *e.g.*, https://linkshrink.net/fp.js, https://api.getsurl.com/js/get_auto.js and https://adshort.co/js/full-page-script.js, worked in a very similar way. This is definitely very distracting to users. However, as we will demonstrate next in §5.4, the first-party websites explicitly included these click interception scripts to monetize their websites.

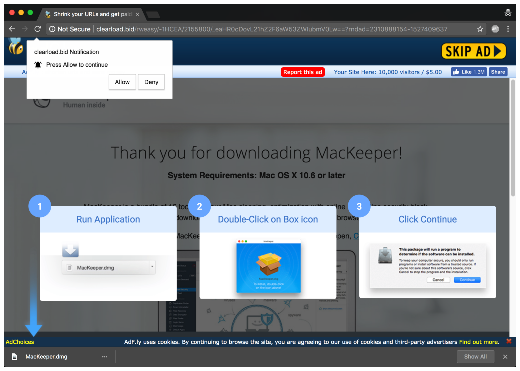
Event Handlers and Visual Deceptions. We find 103 unique third-party scripts which listened for clicks on first-party elements to intercept user clicks. We also discover 78 and 119 unique *third-party* scripts that injected mimic and transparent overlay contents, respectively, into the embedding websites. We discuss next that how these click interception third-party scripts were included in those "victim" websites.

5.3.2 Click Interception Script Inclusion

While we discover that third-party scripts deliberately intercepted clicks via several tricks, it is not clear if they were intentionally included by the first-party websites. To this end, we analyze the script dependency data to figure out the inclusion relationship between third-party scripts and first-party websites. In particular, we aim to determine if a click interception third-party script was *directly included by the website itself*, or *indirectly included by another third-party script*.

Table 4: Top 3rd-party script origins modifying first-party links.

Script	#Websites	#Elements
https://cdn.adf.ly	18	583
https://cdn.shopify.com	11	245
https://static.v2.paysites.czechcash.com	9	640
https://www.sc.pages02.net	7	82
https://linkshrink.net	7	190
https://api.getsurl.com	5	384
https://static-js.sixshop.co.kr	4	59
http://cdn.adf.ly	2	190
http://shinkme.com	2	38
https://adshort.co	2	28



clearload.bid Notification

Press Allow to continue

Allow Deny

Thank you for downloading MacKeeper!

System Requirements: Mac OS X 10.6 or later

To install, double-click on the icon above!

This package will run a program to determine if the software can be installed.

To keep your computer secure, you should only run programs or install software from a trusted source. If you're not sure about this software's source, click Cancel to stop the program and the installation.

AdF.ly uses cookies. By continuing to browse the site, you are agreeing to our use of cookies and third-party advertisers Find out more.

Figure 2: A drive-by download page visited via click interception.

We categorize how a remote third-party script can be included into three classes. First, a third-party script is *statically included by the first-party website*, if the corresponding <script> tag is statically defined in the original web page HTML source. Next, a third-party script is *dynamically included by the first-party website*, if it is loaded through a <script> tag that is dynamically created by a first-party script, including those first-party scripts hosted on a different domain. Finally, a third-party script is *dynamically included by another third-party script*, if it is loaded through a <script> tag that is dynamically created by another third-party script. We summarize the results in Table 5.

Static Inclusion. We find that the majority of these third-party scripts, *i.e.*, 280 unique scripts (64.07%) out of 437 third-party click interception scripts, were statically included by 397 websites. This indicates that these websites deliberately included the click interception scripts, even though they may not intercept user clicks by themselves. In particular, the short URL monetization script https://cdn.adf.ly/js/link-converter.js was found to be statically included by those 18 websites. The script https://wchat.freshchat.com/js/widget.js was statically included by 17 websites. These websites *explicitly allowed such scripts to intercept their users' clicks in exchange for payments*.

Dynamic Inclusion. We discover that 103 unique third-party scripts (23.57%) were dynamically included by first-party

Table 5: How third-party click interception scripts are included.

Inclusion Type	#Websites	#Scripts
Statically included by 1st-party website	397	280
Dynamically included by 1st-party website	112	103
Included by another 3rd-party script	104	63

Table 6: Advertising click interception navigation URLs.

Technique	#URLs	#Ad URLs	%Ad URLs
Hyperlinks	2,695	1,088	40.37
Event Handlers	186	21	11.29
Visual Deceptions	380	74	19.47

websites. For instance, the scripts script=http://gynax.com/j/w.php and http://bgrndi.com/js/NTQw.js were dynamically included by 5 and 4 first-party websites, respectively. In other words, these websites used JavaScript to dynamically create <script> tags to include those scripts. Such websites would be responsible for the privilege abuses by those click interception scripts even if they do not intercept user clicks. They either *did not scrutinize the scripts before including them*, or *deliberately allowed them to intercept user clicks*.

Indirect Inclusion. On the other hand, we discover that only 63 third-party click interception scripts (14.42%) were indirectly included by other third-party scripts. One such a top script is https://tags.bkrtx.com/js/bk-coretag.js, which was included by other third-party scripts on 6 websites. For example, it was included by the script https://s.accesstrade.net/js/atd/bluekai/atd_bluekai.js?id=... on the website https://haken-mikata.com. The latter script was also indirectly included by another script https://s.accesstrade.net/js/atd/satd.js?pt=824F2E4C4077D97ECC014C7A3DE07136725853, which was statically included by the first-party website. In such cases, we cannot blame the first-party websites for indulging those suspicious scripts. Click interception caused by these scripts could be prevented if the websites configure a proper Content Security Policy (CSP) [33] that disallows the browser to load scripts from unknown sources. However, in practice it is difficult and even infeasible to use CSP because many websites need to allow dynamic inclusion of advertising scripts that may be loaded from arbitrary sources due to ad syndication. Therefore, a finer-grained security policy that limits the privilege of included scripts would be more desirable in preventing such privilege abuses.

Summary. We discover that 437 third-party scripts attempted to intercept user clicks on a total of 613 websites. Several top third-party scripts deliberately intercepted user clicks on all their embedding websites. Surprisingly, many of them were included directly by the first-party websites, to monetize the hyperlinks, or more accurately, the user clicks, of those websites.

5.4 Click Interception Reasons and Consequences

We have demonstrated that some third-party scripts intercepted user clicks through various tricks. In this section, we seek to understand the motivations and consequences of such undesired activities.

5.4.1 Monetization

As we have demonstrated in §5.3.1, many *third-party* scripts offer monetization services by converting *first-party* hyperlinks into *third-party* ad links. They force a user to view an advertisement before navigating to the original destination page when the user clicks any hijacked link. As a result, both the third-party click interception script and the first-party website can earn some commission from those participating advertisers. Similarly, we find many other cases where a click was intercepted by a third-party script to visit an advertiser's landing page.

Identifying Advertising URLs. To understand if monetization via advertising is really a common reason for click interception, we compare the navigation URLs in the click interception cases with all the other navigation URLs in our dataset. Specifically, we leveraged the Ghostery extension to determine if one navigation URL is *advertising-related* by testing if it matches the URL pattern of any known advertising company. A navigation URL is marked as an advertising URL, if a positive match is found for any of its intermediate redirect URLs (if any) and the landing URL. We also manually labeled the URLs generated by those short URL monetization scripts as ad URLs because they are not known to the extension.

Surprisingly, we find that 1,183 (36.39%) out of the 3,251 unique click interception navigation URLs are advertising URLs (Table 6), which is a *18.7 times higher* rate than that of normal third-party navigation URLs[8]. In total, only 40,278 (1.95%) out of the 2,065,977 third-party navigation URLs are identified as advertising URLs.

Potential Click Fraud. These click interception websites and scripts have a "good" reason to trick users into clicking those advertising URLs. In online display advertising, the publishers and the ad networks are paid by an advertiser when a user clicks the advertiser's ad under the pay-per-click billing mode. Although they can also earn some commission for an ad impression in the pay-per-view billing mode, the money is much less than what they can get paid when the ad is clicked. However, the ad click-through rate is usually very low—around 2% (in a business-to-consumer banner ad case [18]). To boost ad revenue, the straightforward and effective approach is to leverage real user clicks, as modern ad networks can accurately detect bot-based click frauds [2, 6, 9, 38]. On the other hand, the third-party scripts also have the incen-

[8] We exclude all first-party navigation URLs in our analysis.

tive to cheat advertisers for higher income because many of them are also ad networks. This well explains why the short URL monetization scripts, which also operate as ad networks, have been helping websites intercept user clicks.

In our research, we observe that third-party scripts have leveraged various click interception techniques to monetize user clicks. Further, our results demonstrate that click interception has become an emerging way for generating realistic click traffic to commit ad click fraud.

5.4.2 Distributing Malicious Content

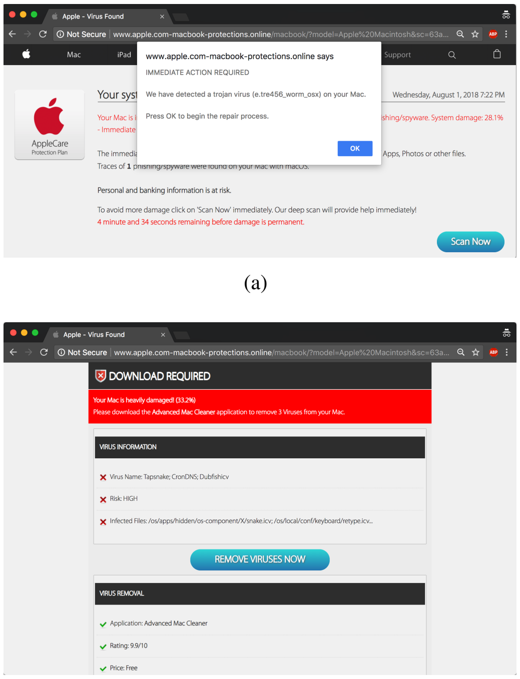
Besides monetization, we find that click interception can lead a user to visit malicious contents. In particular, we were directed to some fake anti-virus (AV) software and drive-by download pages when we manually examined some of the click interception URLs.

For instance, we were forced to visit an ad click URL by the script https://pndelfast.com/riYfAyTH5nYD/4869 on the website https://torrentcounter.to/. Since the navigation URL is an ad click URL, the landing URL is random each time we visit. Nonetheless, one landing URL we visited is a fake AV website, as shown in Figure 3(a). This website showed some fake warnings about virus infection with alarm to fool the user into clicking the *Scan Now* button. After that, it displayed some scanning animation and finally generated a fake scan report to trick the user into installing the fake AV software, as shown in Figure 3(b). The Google search results of the domain *1bcde.com* also suggest it is a malicious redirect website.

We also find that the script http://cdn.adf.ly/js/link-converter.js converted one link of the website http://magazinweb.net/ into http://ay.gy/2155800/..., which is an advertising link. It once took our browser to a drive-by download page, as shown in Figure 2. When we visited the page, our browser automatically started downloading the MacKeeper installer, which is considered as scamware [20]. The page even shows detailed instruction to trick the user into installing this scamware.

These are just two of many malicious examples we have encountered in our manual investigation. We think that there were much more malicious cases that we have yet to discover. Unfortunately, manually verifying all the 2 million URLs in our dataset is infeasible. We plan to leverage automated URL scanning techniques to automatically detect the malicious URLs associated with click interception in the future.

Summary. We identify that many third-party scripts intercept user clicks to monetize user clicks. In particular, they intercept real user clicks to fabricate ad clicks as a new form of committing ad click fraud. Further, the landing URLs that they trick the users into visiting can be malicious.


Apple - Virus Found
Not Secure
www.apple.com-macbook-protections.online/macbook/?model=Apple%20Macintosh&sc=63a...
ABP
Mac
iPad
Support
AppleCare
Protection Plan

www.apple.com-macbook-protections.online says

IMMEDIATE ACTION REQUIRED

We have detected a trojan virus (e.tre456_worm_osx) on your Mac.

Press OK to begin the repair process.

Your syst

Wednesday, August 1, 2018 7:22 PM

Your Mac is i ishing/spyware. System damage: 28.1%
- Immediate

The immedia Apps, Photos or other files.
Traces of **1** phishing/spyware were found on your Mac with macOS.

Personal and banking information is at risk.

To avoid more damage click on 'Scan Now' immediately. Our deep scan will provide help immediately!
4 minute and 34 seconds remaining before damage is permanent.

(a)

Apple - Virus Found
Not Secure
www.apple.com-macbook-protections.online/macbook/?model=Apple%20Macintosh&sc=63a...
ABP

DOWNLOAD REQUIRED

Your Mac is heavily damaged! (33.2%)
Please download the **Advanced Mac Cleaner** application to remove 3 Viruses from your Mac.

VIRUS INFORMATION

- ✗ Virus Name: Tapsnake; CronDNS; Dubfishicv
- ✗ Risk: HIGH
- ✗ Infected Files: /os/apps/hidden/os-component/X/snake.icv; /os/local/conf/keyboard/retype.icv...

VIRUS REMOVAL

- ✓ Application: Advanced Mac Cleaner
- ✓ Rating: 9.9/10
- ✓ Price: Free

(b)

Figure 3: A fake AV website visited because of click interception.

6 Discussion and Future Work

We discuss the limitations of our work, the possible mitigation of the click interception threat, and our future work.

Third-party Script Detection. Our methodology for distinguishing first-party scripts from third-party scripts is not 100% accurate. First, the domain substring matching can be problematic if an adversary can create victim-specific subdomains. For example, a third-party can intentionally generate a subdomain xyz.third-party.org by adding a new entry in its name server. Our technique would mislabel this subdomain as a first-party URL if it is included by xyz.com. Second, an organization may use distinct email addresses for its subsidiaries. For instance, the SOA email address of https://www.instagram.com/ is *awsdns-hostmaster@amazon.com*, whereas that of https://www.facebook.net/ is *dns@facebook.com*. We classify scripts loaded directly from Facebook on Instagram as third-party scripts even though Instagram is owned by Facebook. Although our approach to determining the relationship between two hosts is not complete, it is good enough for achieving our goal and provides better results compared with a similar approach using only whois records [4].

Measurement Scope. We visited only the main pages of Alexa top 250K websites, so we could miss scripts that are loaded only in their sub pages. However, our goal is to have a preliminary understanding of the click interception problem. We do not intend to and are not able to cover all pages and scripts that can be found on these websites. In the future, we will consider sub pages of these websites to investigate the differences between the main pages and the sub pages.

Artificial Interaction with Web Pages. OBSERVER applies an artificial way to interact with websites, *i.e.*, using a script to click all the elements on a page, in order to automate the analysis. This could be different from the normal behavior of a real human being. Nevertheless, our goal is to collect as much click-related data as possible in each page visit. It would be an interesting research topic to study if developers would write code to distinguish authentic clicks from automatically generated ones[9].

Generating Security Warnings. Click interception can direct a user to an unknown URL by modifying first-party hyperlinks or hijacking user clicks on first-party elements. It exploits the fact that the user cannot determine the provenance of the URL that he or she is about to visit (unintentionally). To protect a user from visiting potentially attacker-controlled URLs, a possible defense is to provide the user the provenance information regarding each hyperlink and click. In particular, the browser can display a message alongside each hyperlink about its provenance, *e.g.*, if the associated URL is provided by the first-party website or a third party. The additional message needs to be unforgeable and tamper-proof from JavaScript code, such that the adversary cannot manipulate such security-related data. One potential implementation is to utilize the browser UI that is usually not accessible to JavaScript. For example, we can display the message in the status bar when the user hovers the mouse over a link. Similarly, to defend against event-listener interception, we can display an unforgeable warning message if the user hovers over an element that is potentially intercepted by a third-party script. However, this may cause a lot of false positives as an event handler may not necessarily initiate a navigation upon user click. Therefore, it might be better to show such warning when the user actually performs the click, as [10] does. According to our experiment, OBSERVER introduces negligible performance overhead on navigation. It is thus suitable to be extended as a real-time detection tool for the end users. We plan to extend OBSERVER by incorporating these defenses, and conduct a user study to evaluate their effectiveness.

Ensuring Link and Click Integrity. The above defenses require a user to make security decisions, which might not be very effective in practice. Alternatively, we can let the browser automatically enforce integrity policies for hyperlinks and click event handlers. For example, an integrity policy can specify that all first-party hyperlinks shall not be modifiable by third-party JavaScript code. One may further specify that third-party scripts are not allowed to control frame navigations, although listening for user click is still permitted. Enforcing all such policies would effectively prevent click-interception by hyperlinks and event handlers. However, it might also break the functionalities of some third-party components. To give the user and the website administrator better control, the polices can specify the permissions for each script, matched by an absolute URL, a domain name, a wild card, or a secret token, mimicking the Content Security Policy [33]. We plan to develop and evaluate such an integrity protection mechanism as our future work.

7 Conclusion

We have investigated the click interception problem on the Web with a custom analysis framework developed based on the Chromium browser. We collected data from the Alexa top 250K websites and identified several techniques that can be employed to intercept user clicks. We detected that 437 third-party scripts intercepted user clicks using hyperlinks, event handlers and visual deceptions on 613 websites. We further revealed that many third-party scripts intercept user clicks for monetization via committing ad click fraud. In addition, we demonstrated that click interception can lead victim users to malicious contents. Our research sheds light on an emerging client side threat, and highlights the need to restrict the privilege of third-party JavaScript code.

8 Acknowledgments

The authors thank the anonymous reviewers and our shepherd, Franziska Roesner, for their helpful suggestions and feedback to improve the paper. This material is based on research supported by CUHK under grant 4055081. The views, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily represent the views of CUHK.

References

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking Revisited: A Perceptual View of UI Security. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, 2014.

[2] Sumayah Alrwais, Christopher Dunn, Minaxi Gupta, Alexandre Gerber, Oliver Spatscheck, and Eric Osterweil. Dissecting Ghost Clicks: A Tale of Ad Fraud Via Misdirected Human Clicks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2012.

[9] The clicks in our experiment were generated through Selenium and are different from those generated using JavaScript, which can be easily detected.

[3] Marco Balduzzi, Manuel Egele, Engin Kirda, Davide Balzarotti, and Christopher Kruegel. A Solution for the Automated Detection of Clickjacking Attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Beijing, China, April 2010.

[4] Frank Cangialosi, Taejoong Chung, David Choffnes, Dave Levin, Bruce M. Maggs, Alan Mislove, and Christo Wilson. Measurement and Analysis of Private Key Sharing in the HTTPS Ecosystem. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[5] Vacha Dave, Saikat Guha, and Yin Zhang. Measuring and Fingerprinting Click-Spam in Ad Networks. In *Proceedings of the 2012 ACM SIGCOMM*, Helsinki, Finland, August 2012.

[6] Vacha Dave, Saikat Guha, and Yin Zhang. Viceroi: Catching Click-spam in Search Ad Networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, October 2013.

[7] Sevtap Duman, Kaan Onarlioglu, Ali Osman Ulusoy, William Robertson, and Engin Kirda. TrueClick: Automatically Distinguishing Trick Banners from Genuine Download Links. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[8] Google. Expanding user protections on the web. https://blog.chromium.org/2017/11/expanding-user-protections-on-web.html.

[9] Google. Google Ad Traffic Quality. https://www.google.com/ads/adtrafficquality/.

[10] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, August 2012.

[11] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. EvilSeed: A Guided Approach to Finding Malicious Web Pages. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[12] Ari Juels, Sid Stamm, and Markus Jakobsson. Combating Click Fraud via Premium Clicks. In *Proceedings of the 16th USENIX Security Symposium (Security)*, Boston, MA, August 2007.

[13] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the 22nd USENIX Security Symposium (Security)*, Washington, DC, August 2013.

[14] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February–March 2017.

[15] Sebastian Lekies, Mario Heiderich, Dennis Appelt, Thorsten Holz, and Martin Johns. On the Fragility and Limitations of Current Browser-Provided Clickjacking Protection Schemes. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[16] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[17] Bin Liu, Suman Nath, Ramesh Govindan, and Jie Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Seattle, WA, March 2014.

[18] Ritu Lohtia, Naveen Donthu, and Edmund K Hershberger. The Impact of Content and Design Elements on Banner Advertising Click-through Rates. *Journal of Advertising Research*, 43(4):410–418, 2003.

[19] Malwaretips. How to remove Web Browser Redirect Virus (Windows Help Guide). https://malwaretips.com/blogs/remove-browser-redirect-virus/.

[20] Mike Matthews. What MacKeeper is and why you should remove it from your Mac, 2018. https://www.imore.com/removing-mackeeper-your-mac.

[21] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *Proceedings of the 16th International Conference on World Wide Web (WWW)*, 2007.

[22] Brad Miller, Paul Pearce, Chris Grier, Christian Kreibich, and Vern Paxson. What's Clicking What? Techniques and Innovations of Today's Clickbots. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2011.

[23] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, October 2012.

[24] Nick Nikiforakis, Federico Maggi, Gianluca Stringhini, M Zubair Rafique, Wouter Joosen, Christopher Kruegel, Frank Piessens, Giovanni Vigna, and Stefano Zanero. Stranger Danger: Exploring the Ecosystem of Ad-based URL Shortening Services. In *Proceedings of the 21st International World Wide Web Conference (WWW)*, Seoul, Korea, April 2011.

[25] Erlend Oftedal. Retire.js: What your require you must also retire. https://retirejs.github.io/retire.js/.

[26] OWASP. Clickjacking. https://www.owasp.org/index.php/Clickjacking.

[27] Paul Pearce, Vacha Dave, Chris Grier, Kirill Levchenko, Saikat Guha, Damon McCoy, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Characterizing Large-Scale Click Fraud in ZeroAccess. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.

[28] M. Zubair Rafique, Tom Van Goethem, Wouter Joosen, Christophe Huygens, and Nick Nikiforakis. It's Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[29] David Ross and Tobias Gondrom. HTTP Header Field X-Frame-Options. Technical report, 2013.

[30] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting Frame Busting: a Study of Clickjacking Vulnerabilities at Popular Sites. In *Proceedings of the IEEE Web 2.0 Security and Privacy (W2SP)*, 2010.

[31] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, Raleigh, NC, April 2010.

[32] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon McCoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and Moheeb Abu Rajab. Ad Injection at Scale: Assessing Deceptive Advertisement Modifications. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[33] W3C. Content Security Policy Level 3. https://www.w3.org/TR/CSP3/.

[34] Wikipedia. List of DNS record types. https://en.wikipedia.org/wiki/List_of_DNS_record_types#NS.

[35] Wikipedia. List of managed DNS providers. https://en.wikipedia.org/wiki/List_of_managed_DNS_providers.

[36] Wikipedia. SOA record. https://en.wikipedia.org/wiki/SOA_record.

[37] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *Proceedings of the 24th International World Wide Web Conference (WWW)*, Florence, Italy, May 2015.

[38] Haitao Xu, Daiping Liu, Aaron Koehl, Haining Wang, and Angelos Stavrou. Click Fraud Detection on the Advertiser Side. In *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)*, Wroclaw, Poland, September 2014.

[39] Apostolis Zarras, Alexandros Kapravelos, Gianluca Stringhini, Thorsten Holz, Christopher Kruegel, and Giovanni Vigna. The Dark Alleys of Madison Avenue: Understanding Malicious Advertisements. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC)*, 2014.

[40] Yuchen Zhou and David Evans. Understanding and Monitoring Embedded Web Scripts. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.