TRIDENT: Towards Detecting and Mitigating Web-based Social Engineering Attacks

Zheng Yang[†], Joey Allen[†], Matthew Landen[†], Roberto Perdisci[†‡], Wenke Lee[†]
[†]*Georgia Institute of Technology* [‡]*University of Georgia*

Abstract

As the weakest link in cybersecurity, humans have become the main target of attackers who take advantage of sophisticated web-based social engineering techniques. These attackers leverage low-tier ad networks to inject social engineering components onto web pages to lure users into websites that the attackers control for further exploitation. Most of these exploitations are *Web-based Social Engineering Attacks* (WSEAs), such as reward and lottery scams. Although researchers have proposed systems and tools to detect some WSEAs, these approaches are very tailored to specific scam techniques (i.e., tech support scams, survey scams) only. They were not designed to be effective against a broad set of attack techniques. With the ever-increasing diversity and sophistication of WSEAs that any user can encounter, there is an urgent need for new and more effective in-browser systems that can accurately detect generic WSEAs.

To address this need, we propose TRIDENT, a novel defense system that aims to detect and block generic WSEAs in real-time. TRIDENT stops WSEAs by detecting *Social Engineering Ads* (SE-ads), the entry point of general web social engineering attacks distributed by low-tier ad networks at scale. Our extensive evaluation shows that TRIDENT can detect SE-ads with an accuracy of 92.63% and a false positive rate of 2.57% and is robust against evasion attempts. We also evaluated TRIDENT against the state-of-the-art ad-blocking tools. The results show that TRIDENT outperforms these tools with a 10% increase in accuracy. Additionally, TRIDENT only incurs 2.13% runtime overhead as a median rate, which is small enough to deploy in production.

1 Introduction

Social Engineering (SE) has become an ever more sophisticated and common attack method [1]. Recent surveys report that 84% of hackers leverage *Web-based Social Engineering Attacks* (WSEAs) in the cyber kill chain with a high success rate [2–4]. Moreover, 64% of companies have experienced web-based attacks, and 62% have seen phishing and WSEAs [5]. Attackers also target regular Internet users. The Federal Trade Commission received 2.8 million fraud reports in 2021 in the United States, which led to a $5.8 billion financial loss [6]. The top 3 fraud categories – impostor scams (e.g., tech support scams), online shopping scams, and reward and prize scams (e.g., survey scams) – are commonly seen on the Internet [7–10]. These scams account for $2.3 billion of losses, almost doubling from 2020.

Researchers have studied countermeasures to mitigate the impact of WSEAs. For example, Miramirkhani *et al.* analyzed tech support scams [7]; Kharraz *et al.* built Surveylance [8], which is specifically designed to detect survey scams; and Invernizzi *et al.* developed EVILSEED [11], a crawler that searches the Internet to identify risky websites that install unwanted software. However, these previous works only focus on specific SE attack vectors. Because of the diversity of WSEAs that users can encounter [1], there is an urgent need for new and more effective in-browser defense systems that can accurately detect generic WSEAs.

This paper proposes a new defense system that aims to detect and block generic WSEAs in real-time while the user is browsing the web. The main challenge we face is that *directly* detecting malicious web pages related to WSEAs is extremely difficult due to the large variety of SE tactics attackers can employ and the freedom they have in building malicious content. Therefore, in this work, we investigate how to *indirectly* detect and block WSEAs at their inception before the user interacts with the related scam content.

Recent works have shown that users often reach *Social Engineering Websites* (SE-websites) by interacting with malicious ads [7–9, 12–16]. More specifically, attackers are inclined to leverage low-tier ad networks to inject ads into many different publisher websites at scale and use these ads to lure users to their SE-websites so that various attacks such as lottery scams, reward scams, tech support scams, etc., can be launched. Importantly, these low-tier ad networks often do not inject traditional ads onto the page. Instead, they inject DOM elements into ad-publishing web pages and leverage different social engineering tricks to lure users into clicking

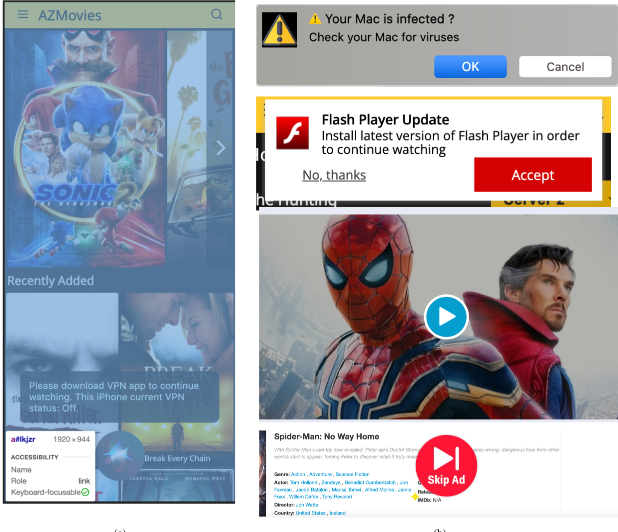


Figure 1: Example SE-ads: (a) An invisible link covering the whole viewport to force users to click; (b) Deceptive elements (fake notification, "Play" and "Skip Ad" buttons) to lure users into interacting with them.

these elements to trigger ad network-driven navigation to a WSEA page. For instance, the ad network may inject a transparent overlay covering the entire publisher page and listen to users' clicks on any portion of the page. We refer to these non-traditional ads that leverage various SE tricks to lure users' clicks as *Social Engineering Ads* (SE-ads).

As mentioned above, SE-ads are non-traditional ads. They are often invisible, malicious ads that, when interacted with, navigate the browser to a landing page containing SE attacks. A previous study [13] reported that attackers often leverage two types of techniques (registering click event listeners and injecting invisible links shown in Fig. 1a) to deploy invisible, malicious ads to steal users' clicks. In addition, SE-ads also appear as misleading in-page components, such as an in-page push notification or fake "Skip Ads" or "Play" buttons, as illustrated in Fig. 1b, to induce users to interact with them. Given these features, we can see that SE-ads are not traditional ads, although we still refer to them as ads because they are injected into a publisher page by ad networks. Therefore, rather than attempting to detect WSEAs directly by analyzing their contents and/or URLs related to the WSEAs, we focus on detecting their leading causes, namely SE-ads.

Although most SE-ads come from ad networks, existing ad-blocking tools are not effective in detecting SE-ads for two major reasons. First, the ads are not generally visible, so ad-blocking tools such as PERCIVAL [17] which block ads through the image rendering pipeline, cannot detect them. Second, the ad networks that distribute these SE-ads are extremely motivated to evade ad blockers [9]. For example, in our evaluation (see Tab. 10 and Tab. 11), we show that neither commercial ad blocker [18] nor the most recent state-of-the-art ML-based ad-blocker [19] is effective against SE-ads.

To address the challenge of detecting SE-ads to mitigate WSEAs, we propose TRIDENT – a novel system that detects SE-ads distributed by low-tier ad networks at scale in real-time and blocks the subsequent web-based social engineering attacks. To this end, TRIDENT develops an in-memory graph representation of a web page and its activities, (e.g. registering event listeners to intercept clicks, manipulating Document Object Model (DOM) to inject deceptive elements shown in Fig. 1), which we call the *Web Action History Graph* (WAHG). During a user's browsing session, TRIDENT uses the WAHG to protect users from potential SE attacks that are launched through SE-ads in real-time. Specifically, during a user's browsing session, TRIDENT vets each navigation event to determine if a sea initiates it.

When TRIDENT detects the navigation is related to a SE-ad, it redirects the user to an interstitial page to warn the user.

To extensively evaluate TRIDENT, we crawled over 100K websites from October 2021 to January 2022 and collected 258,008 unique navigation events initiated by JavaScript (JS), including 1,479 events resulting in SE attacks. In our evaluation, we found that TRIDENT can detect SE-ads with an accuracy of 92.63%, a precision of 90.63%, and a recall of 96.28%, outperforming prior work [19] by more than 10%.

In summary, we make the following main contributions:

- **Blocking generic web-based social engineering attacks.** We propose TRIDENT, a browser extension to Chromium-based browsers that blocks generic web-based SE attacks by detecting SE-ads. TRIDENT achieves an accuracy of 92.63% with a precision of 90.63%, a recall of 96.28%, and an F-1 score of 93.37%. We will release TRIDENT source code at https://github.com/ian7yang/trident.
- **Real-time detection.** TRIDENT extends the Chrome DevTools (CDP) to provide real-time JS activity monitoring with minimal instrumentation. When an SE-ad is detected, TRIDENT determines whether to warn the user with an interstitial alert page before the final social engineering attack landing page is rendered.
- **Comprehensive evaluation.** Our evaluation includes a comparison with previous work and shows that TRIDENT can detect SE-ads more effectively than state-of-the-art ad-blocking tools. Moreover, we show that the detection features used by TRIDENT's classifier are robust against evasion attempts and concept drift. At the same time, we demonstrate that TRIDENT's implementation only incurs a median of 2.13% runtime overhead.

2 A Motivating Example & Challenges

This section presents a real-world example of SE-ads hosted on a high-ranking search result from Google Search and discusses the limitations of prior, generic ad-blocking work.

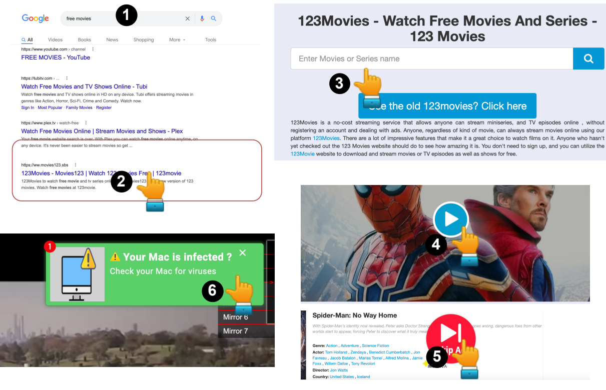


Figure 2: Motivating Example: Alice typed "free movies" in Google Search but ended up landing on SE-websites.

2.1 A Motivating Example

This section introduces a real-world motivating example demonstrating how victims arrive on SE-websites by interacting with the SE-ads. Fig. 2 gives a clear description of how an ad network manipulates users to interact with SE-ads by including JavaScript (JS) code into a content-sharing website, also known as an ad publisher.

Google Search Result Leads to SE Attacks. The attack begins on the popular Google search engine where the victim, Alice, completes a Google search for the phrase, "free movies" at step ❶. Despite Google Search is one of the most highly-respected search engines, it still struggles to filter out websites that include malicious content from the top results of the search. For instance, at the time and location of writing, Google Search returns an illegal movie-sharing website (*ww.movies123.sbs*) in the *top 4 results* for the query "free movies" at step ❷. As a result, Alice is unfortunately supplied with a mixture of benign and malicious search results. As this is one of the top results, many users may click on the link to *ww.movies123.sbs*, which is not considered malicious by VirusTotal [20] or Google Safe Browsing [21].

At first glance, this website appears innocuous while also providing a diverse selection of popular, well-known movies. However, under the hood, *ww.movies123.sbs* includes scripts obtained from low-tier ad networks with one goal: to trick visitors into clicking on the SE-ads these scripts inserted so they can make money from their malicious activity. Looking at Fig. 2, several mouse event listeners, registered on `#document`, intercept Alice's click on the search box in step ❸. In fact, any click on the page triggers the listeners, which dynamically determine what page to open for Alice. Due to these click interceptions, Alice is obligated to interact with SE-ads when searching for a movie to watch. Before Alice can type the movie name, the SE-ad opens up a new tab, asking Alice to install "Rainbow Blocker", a known AdWare [22]. When Alice arrives at the spider-man movie, she clicks on the play button in step ❹ and "Skip Ads" in step ❺. Unfortunately, the SE-ads are attempting to trick Alice into downloading browser extensions, which claim to be necessary to watch the movie. However, after further manual analysis of their code, we found that these extensions were trackers and AdWare, which track users and harm their digital privacy. After seven clicks, Alice could watch the movie after closing all the opened tabs. While Alice is watching, an in-page notification pops up to warn Alice that her Mac is infected. Alice becomes nervous and clicks on the banner to download software to clean her Mac in step ❻. This software was confirmed to be an AdWare by VirusTotal [20].

```
<!--ad slot on nytimes.com-->
<div id="dfp-ad-top" class="place-ad placed-ad"
  data-google-query-id="CNrG4fK85PcCFWcrwQodUhgBGQ">
<!--the iframe injected by the ad script below-->
<iframe src="https://...safeframe.googlesyndication..."/>
</div>

// an inline script to configure the ad size
var adConfig = function() {...};
// a remote script: doubleclick/pubads_impl_*.js
var adFrame = createAdIframe(adConfig);
appendAdFrame('#dfp-ad-top', adFrame);
```

(a) Ad scripts from Google Ads

```
var func = {init: function(event) {
      return setTimeout(function() {
      windowOpenerNull(), removeTransparentLayer()},
      500), sendClickMetrics()},
  createTransparentLayer: function(){...},
  removeTrasparentLayer: function(){...},}
// register click event listener
document.addEventListener(
  isChrome ? 'mousedown': 'click',
  handler(e){removeTransparentLayer();func.init(e);})
```

(b) In-lined ad scripts from adSterra

Figure 3: Script snippets from Google Ads which follows ad standard and AdSterra which inject SE-ads.

Low-tier Ad Networks Are Popular, but Hijack Clicks. Ad publishers are inclined to cooperate with low-tier ad networks, which pay more than high-profile advertising platforms [23]. For example, AdSterra pays up to USD $25 for a click [24], which is 10x more than what Google ads pay. Therefore, these low-tier ad networks are strongly motivated to elicit clicks to collect more money [25]. These low-tier ad networks may use SE tricks to harvest as many clicks as possible. As described by the reverse-engineered ad scripts in Fig. 3b in the Appendix, they inject in-line scripts to insert a transparent DOM layer and register a mouse event listener. The visitor is then forced to trigger the event listener, which opens a new window and loads ads. This approach is highly different from what the high-profile ad networks do and does not follow the general standards [26–29]. In contrast, looking at the pseudo code from Google ads in Fig. 3a, the ad publisher prepares a container for the ad script to inject an iframe that can isolate the ad's contents such that it cannot directly access the first party's contents. Unfortunately, these ads will likely collect fewer clicks than low-tier networks.

Therefore, as ad publishers, these content-sharing websites

prefer low-tier ad networks even though these ad networks may use SE tricks to get more clicks. Thus, the low-tier ad networks can transfer a fraction of their high revenue from advertisers to those ad publishers. The advertisers are satisfied by having more ads exposed to users, which results in a higher conversion rate. This business model undoubtedly is intriguing to attackers and provides them with opportunities to spread malicious content (e.g. unwanted software, WSEAs).

2.2 Challenges

Next, we will discuss in detail the limitations of prior approaches [12, 13, 17, 19, 30–32] on generic ad blocking, and the major challenges in detecting the malicious behavior demonstrated in the motivating example.

Limitations of filter-list-based ad-blockers. Traditional ad-blockers leverage human-created blacklists or whitelists to determine what network requests can pass. Every request URL that matches a pattern in the list is blocked, regardless of the request being benign or malicious. While these ad-blockers may block some malicious content, they are not robust because once malicious ad networks change the domain or URL parameters to serve their malicious content, they can evade these blocking tools [9, 10]. They also hurt the earnings of ad publishers. Therefore, some content providers refuse to display their content if an ad-blocker is detected. To avoid the pitfalls of URL-based detection systems, TRIDENT does not factor URL parameters into the feature extraction process except by using the URL to determine whether a resource is from the first party or a third party.

Limitations of ML-based ad-blockers. ADGRAPH is the most recent, state-of-the-art, ML-based ad-blocker, which achieves 95.33% accuracy and can identify many mistakes in the filter lists as mentioned above. However, as mentioned in WEBGRAPH [33], ADGRAPH is not robust. Adversaries can evade ADGRAPH by tweaking the domains and URL parameters. Unfortunately, even though WEBGRAPH is a more robust version of ADGRAPH, it focuses on network information flows, which SE-ads do not necessarily rely on. For example, when a SE-ad script loads, it does not need to issue network requests to track users or fetch ad resources, resulting in no network information flows. This allows the attack to hide from WEBGRAPH. To this end, analyzing the behaviors of the script becomes more critical. TRIDENT builds a graph-based representation of a website, which we call the *Web Action History Graph* (WAHG), to learn what scripts do on the web page, which provides more insightful information to determine whether a script is related to SE-ads.

Detecting invisible SE-ads. Traditional ads come with attractive images and words to draw users' attention. This inspired Din *et al.* to build PERCIVAL [17], which is a deep learning model inside the image rendering pipeline of the Chromium browser that blocks rendering ad images. PERCIVAL achieves

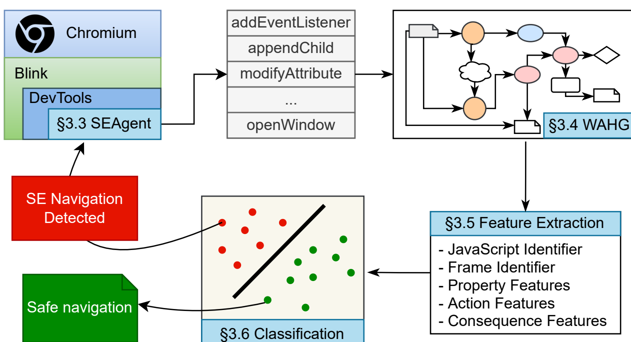


Figure 4: The design of TRIDENT. TRIDENT instruments Chromium to collect features related to JS activities. The features are fed into a classifier when navigation takes place.

96.76% accuracy and can effectively block ad images to make users less likely to interact with the ads. Unfortunately, the DOM elements that intercept users' clicks, such as fake play buttons and invisible overlays [12, 13], do not need to go through the image rendering pipeline. These SE-ads evade PERCIVAL by nature. Additionally, adversaries can circumvent PERCIVAL using the attacks proposed in [34]. To address this challenge, TRIDENT translates invisible formats into features. And these features are good indicators for SE-ads as shown in the evaluation of feature importance in §3.5.

3 TRIDENT

3.1 Overview

In this section, we introduce TRIDENT, a novel real-time detection system for identifying *Social Engineering Ads* (SE-ads) and blocking navigation to potential *Social Engineering Websites* (SE-websites). At a high-level, TRIDENT takes advantage of two intuitions: (1) SE-ads use tricks (e.g., click-jacking and social engineering) to lure users into interacting with strategically placed DOM elements and triggering unwanted browser navigation; and (2) SE-ads often navigate the user to malicious websites that host social engineering attacks (e.g., tech support scams, malicious downloads, etc.). Therefore, to detect SE-ads and block the subsequent events, TRIDENT monitors the user's browsing session and vets each navigation to determine if it may be related to an SE-ad. More specifically, during this vetting process, TRIDENT extracts features related to how this navigation was initiated and passes these features to its classification module. Finally, if TRIDENT determines this navigation is SE-ad related, TRIDENT presents an interstitial page to warn the user.

While prior approaches [7, 8, 11] focus on specific SE attack vectors, TRIDENT takes a more generic approach that relies on the causality of how users end up in SE-websites. Namely, TRIDENT detects WSEAs by detecting the anomalous techniques, which intercept users' clicks by any means, routinely used by SE-ads, which often lead to

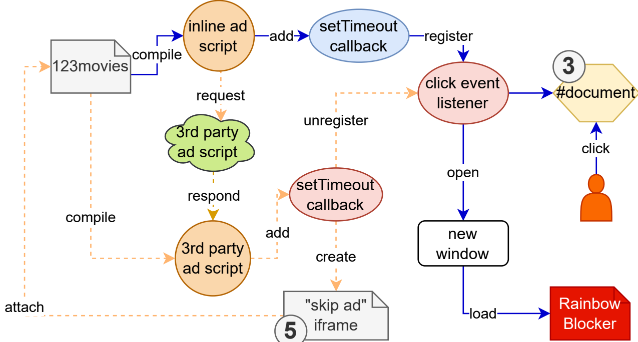


Figure 5: WAHG based on step 3 in the motivating example.

websites that host SE attacks. TRIDENT achieves this by leveraging the design illustrated in Fig. 4. First, TRIDENT instrument Chromium by extending the Chrome DevTools Protocol framework (CDP) [35] with a new agent, Social-Engineering agent (`SEAgent`). While a user is visiting a website, the `SEAgent` collects JS actions (e.g., event listener registrations, DOM modifications) and sends them to a background daemon. The background daemon builds an in-memory graph representation of the web page and its activities, which we call *Web Action History Graph* (WAHG). While TRIDENT builds and updates the WAHG, it also extracts *property features*, *action features*, and *consequence features* about the page's JS code from the graph. These features describe how these scripts are included, what contexts the scripts are running in, and what the scripts do on a web page. These features are passed to TRIDENT's classification module, which classifies the navigation as related to SE-ads or benign.

In the remainder of this section, we first give an example of the WAHG of the motivating example in §3.2, and then explain how TRIDENT instruments Chromium in §3.3. Next, we discuss how to construct WAHG while the user is browsing a website in §3.4, and the feature extraction along with it in §3.5. Finally, we introduce the classifier in §3.6.

3.2 Web Action History Graph

The *Web Action History Graph* (WAHG) is a graph-based representation of a web page. Nodes in the graph represent web objects (e.g., window, resource, DOM node, etc.) and edges represent causal relationships between objects. For example, when a script inserts a new DOM element into the DOM tree, an edge from the script to the element will be connected into the WAHG. We formally define all graph objects and relationships in Tab. 1.

To demonstrate the WAHG's capability to represent SE-ads, we provide an example WAHG of the suspicious publishing page, "www.movies123.sbs", that Alice encountered in the motivating example (❸) in Fig. 5. For clarity, the example only contains the portions of the WAHG related to two SE-ad attacks on the page. The first SE-ad is launched by an inline script on "ww.movies123.sbs" and is represented by the set

Object Type	Attributes
Frame	security_origin, url, is_page
Window	url
Resource	url, type
Script	url, is_isolated, frame_owner
Function	url, is_eval_or_new_function, location
DOM Node	tag_name, is_inserted_by_js
HTML Parser	frame_owner

(a) Graph Objects. The unique ID for each object is omitted.

Relationship	Example
Attached	Frame → Frame
Compiled by	Script → Frame
Created	Script/Function → Frame
Add event listener	Script/Function → Function
Listen to events	Function → DOM Node
Add callback function	Script/Function → Function
Navigated	Frame → Frame
Opened	Frame → Window
Load	Window → Frame
Respond	Parser/Script/Function → Resource
Response	Resource → Parser/Script/Function

(b) Relationship between objects.

Table 1: WAHG objects, relationships, and key attributes.

of nodes connected by the solid blue edges. The inline script initiates the deployment of the SE-ad by scheduling a delayed callback to be executed using `setTimeout`. When this callback is executed, it adds a new mouse event listener onto the `#document` element which consequently covers the whole viewport. When Alice clicks on the input box to search for a movie, the click is effectively hijacked. The mouse event listener on `#document` is fired and redirects Alice to the malicious website called "Rainbox Blocker". The second SE-ad attack is shown by the dashed yellow path, which is initiated by the same inline script, but with a different deployment technique. More specifically, the inline script injects a third-party ad script that also uses `setTimeout` to create an iframe and insert it onto the page. If Alice clicks on the "Skip Ad" button, which is rendered in the iframe, it would cause Alice to download a malicious Chrome Extension. This example demonstrates the fine-grained details related to a web page that is embedded into the WAHG.

3.3 Social-Engineering Agent

The Social-Engineering Agent (`SEAgent`) module resides within the browser to emit event logs for constructing the WAHG. To minimize our footprint in the browser, we implemented the `SEAgent` on top of the Chrome DevTool's Protocol (CDP) [35] which can be easily updated and maintained.

CDP is a debugging tool to assist web developers with

UI development. In addtion to debugging a website, CDP can also be used to analyze the website for security and privacy purposes. More specifically, CDP implements several "domains" where each domain has a set of APIs and events related to a particular aspect of a web application (e.g., `DOM`, `Network`, or `DOMDebug`). Internally, each domain relies on a backend "Inspector Agent" that encapsulates the necessary instrumentation to support the domain. For example, the `DOM` domain provides events for DOM modifications, and the `DOMDebug` domain exposes an API to collect current event listeners. Unfortunately, existing CDP domains could not support real-time information collection for some cases. For example, the `DOMDebugger.getEventListeners` API collects current event listeners on the DOM at the query time. We would have to call this API frequently to capture every registered and removed listener, which is cumbersome and risky because a malicious listener may be removed when we call this API. Moreover, the `Debugger` domain does not implement event hooks for JS executing stack, which is essential for JS action attribution, which will be discussed in §3.4.

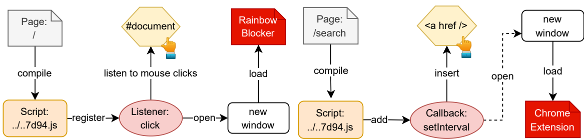
To meet the real-time requirement, we implement the `SEAgent` with less than 800 lines of C++ code. `SEAgent` is a plug-n-play component to the existing CDP, which means it is easy to update and maintain with browser updates. `SEAgent` implements four types of hooks to collect JS actions for constructing the WAHG in real-time. Whenever a hooked API is called, it emits an event immediately. For example, the instrumentation in event listener registration collects the `event_target`, `event`, and the listener `function` whenever a script or function calls `addEventListner` to meet the real-time requirement for feature collection. The details of the hooks are listed in Tab. 2.

3.4 WAHG Construction

In this section, we discuss in detail how TRIDENT uses the event logs collected by the `SEAgent` to construct the WAHG in real-time progressively.

TRIDENT parses every event and translates the results into nodes and edges. There are two important attribution steps that TRIDENT performs: JS attribution, which associates DOM events to a responsible JS file, and navigation initiator attribution, which determines which script requests the navigation such that TRIDENT only needs to inspect paths to this script node on the WAHG instead of inspecting all the script nodes. We discuss in detail how both tasks are completed in the remainder of this section.

JavaScript Attribution. TRIDENT needs to attribute all DOM events to the accountable script. To do so, for each interaction and event, we attribute the event to the current executing JS function. For instance, when the script "../..7d94.js" inserts an event listener onto the page, we connect the script to the listener in Fig. 6a. This approach addresses most cases for finding the responsible JS file, However, the two global



(a) Navigation Initiator as a JavaScript Function

(b) Navigation Initiator as an Anchor tag

Figure 6: TRIDENT finds the responsible JavaScript function that initiates the navigation. The functions in pink elliptical are accountable for SE-ads.

JS functions, `eval` and `Function`, pose challenges when we try to attribute events to the correct functions or scripts. For example, when an external script loads, it invokes `eval` to evaluate a JS code snippet. This process requires compiling the snippet and generating a new script object, but this snippet will not have a valid URL. In these cases, we assign events caused by the snippet to its caller's URL. We use the same approach for the `Function` API as it works similarly to `eval`.

Navigation Initiator. There are two types of navigation initiators: a script or a user's action (e.g., clicking a link, typing in the address bar). Finding the initiator of a navigation event helps us reduce the analysis space by only having to analyze scripts and events related to the navigation, which leads the users to the websites under the attacker's control. Fig. 6a presents a JS function initiator, the click listener, which opens a new page. By analyzing the WAHG, TRIDENT can locate the responsible script that may lead to a SE-websites. Obviously, not all navigation events are initiated by JS code directly. In Fig. 6b, an anchor tag is inserted by a timer callback function. When the user clicks on the link, it opens a new window. Based solely on the information on this path, TRIDENT cannot determine what code is responsible for the navigation. To handle these cases, TRIDENT learns what `href` attribute is assigned to or updated for all the anchor nodes. It connects the JS function that modified or updated the anchor node to the new window by matching their URLs.

3.5 Feature Extraction

In this section, we discuss the features used to learn the characteristics of malicious and benign scripts. Next, we describe how TRIDENT extracts features in real-time.

3.5.1 Feature Descriptions

TRIDENT's features are divided into three groups – *property*, *action*, and *consequence* features – as shown in Tab. 3. The first group introduces the script's properties; the second group describes the script's behaviors; and the last group contains redirect information. We leverage our domain experts' intuitions on web development and experiences from previous studies [13, 14, 36, 37] to choose these features to describe what happens before and after navigation.

Hooks	Description	Locations
DOM	Record DOM activities including DOM manipulation, etc. Attribute the operation to a JS function.	Node creation, insertion, and removal
		Node attributes modification
Page	Record frame activities including iframe creation and deletion, frame navigation, and opening new tabs. Attribute the operation to a JS function.	iframe attach and detach
		Frame navigation / Opening new windows
Network	Record network activities including what resources are being requested and who are responsible for these requests	Network requests
		Network responses
Script	Record JavaScript activities including what scripts are compiled and executed, what user callbacks are added, and what event listeners are registered.	Script compilation, execution
		Function invocation
		Add user callbacks / event listeners

Table 2: Instrument hooks to construct WAHG.

Property Features
execution context (first party or third party frame)
script type (inline, remote file, `eval`, or `function`)
owner (first party or third party)
requestor (HTML parser or another script)
requestor's properties
Actions Features
register event listeners (`event_type`, `event_target`)
add timer callbacks (`setTimeout`, `setInterval`)
insert DOM nodes (`node_type`)
open new windows (`url`, `target`)
initiate navigation (`url`, iframe, origin, client_redirect, browser_initiated)
modify DOM node attributes (attributes)
send network requests (`resoure_type`, `url`)
Consequence Features
# of redirect hops
# of unique domains
redirect type (JS-driven, response-header-driven)

Table 3: Feature groups used by TRIDENT.

Property Features. *Property* features target the properties of a script, including how the script is included in a web page, who owns the script, and the context it is running in. TRIDENT determines the property features when a script is compiled and executed. If the script is inserted into the web page by another script, TRIDENT adds the requestor's properties too. First-party scripts are usually included by the website operator, which implies they can be trusted, whereas third-party scripts (e.g., ad scripts from ad networks) are unverified and should not be trusted. Legitimate ad scripts follow the FTC rules [27] to inject ads, for example, by isolating their ad contents inside an iframe as shown in Fig. 3a. In contrast, SE-ad scripts are strongly motivated to elicit user's click by any means. Therefore, TRIDENT uses this feature group to learn whether a suspicious action can be trusted.

Action Features. *Action* features represent the behaviors exhibited by a script on the web page. These actions are primarily related to click hijacking, including registering event listeners, adding large hyperlinks, and injecting visually deceptive elements. Each action becomes an edge in the WAHG. TRIDENT then extracts these features from both the node's and the edge's properties. For instance, the *register event listeners* feature considers the `event_type` of the edge and `event_target` of the target node. More specifically, a JS function registers an event listener that listens to mouse events on a specific DOM element. This DOM element is the `event_target`. TRIDENT checks whether this DOM element is a JS inserted DOM Node or a built-in large element (e.g. `#document`, `body`). For actions involved in network requests such as `open new windows`, `attach iframe`, `initiate same-tab navigation`, and `send network requests`, TRIDENT examines the URL to determine where the resources are from. This feature group helps TRIDENT learn to separate malicious activities from benign ones. For example, appending a transparent hyperlink covering the whole viewport is more suspicious than adding a visible iframe to load content.

Consequence Features. *Consequence* features describe what happens after the navigation. We extract the URLs in the redirect chain and collect the number of unique domains. TRIDENT also checks whether the redirect is initiated by JS or an HTTP response header. We consider the redirect chain between the first page and the eventual landing page because the window directly opened by clicking an ad usually is not the eventual landing page [9, 38]. Usually, ad networks need to determine what ad to present by collecting the user's cookies before deciding where to send the user. Unlike clicking on ads, clicking on a link to an article usually directly opens the article without any redirects because the website knows where the user is heading. Therefore, redirects between the opening action and the final landing action are good indicators of ads. This is useful for TRIDENT to determine whether a newly opened tab is for ads. Moreover, analyzing these consequence features is mandatory because popular websites may also deploy techniques to intercept the users' clicks for benign purposes [36] and merely relying on the features of

the current page can cause high false positive [13].

To conclude, TRIDENT's primary goal is to detect navigation made by clicking benign ads, links, or SE-ads. Simply put, benign ads follow FTC rules which create iframes that do not intercept users' clicks; anchor links usually do not need to redirect the users multiple times; and SE-ads steal users' clicks by any means and redirect the users to SE-websites.

3.5.2 Real-time Feature Extraction

Unlike prior approaches [9, 37] that collect features offline, TRIDENT extract features while the user is browsing so that TRIDENT can timely detect and block SE-related navigation. This process is asynchronous to the browser rendering process, so collecting features for each event will not impact the user experience. For instance, when a script registers an event listener, TRIDENT creates or finds the script node, creates or finds the function node (the event listener), and creates or finds the event target node (e.g., a DOM node). Then, TRIDENT updates the WAHG by connecting them. Meanwhile, TRIDENT updates the action features of this script for adding a listener.

When a navigation event is received, TRIDENT only needs to update the WAHG one last time to insert the target frame node and connect the frame to the script or function node which initiated the navigation. The initiator then becomes the entry point for backtracking on the WAHG. Taking the example in Fig. 5, when the user clicks the `#document`, it triggers the event listener to open a new window. At this point, TRIDENT has already learned that the in-line script added a `setTimeout` which registered the event listener. As the features have already been stored in memory for the in-line script, TRIDENT only needs to update the features by adding that the script also opens a new window. Therefore, TRIDENT does not need to make expensive queries to traverse the WAHG for feature collection at the last point. Then, TRIDENT translate these features into a feature vector that captures the actions done by this script under its owner frame's context and pass down to the classifier.

3.6 Blocking SE-ads related Navigation

The final portion of TRIDENT is its classification module. When a navigation event is about to occur, the extracted features discussed in §3.5 are passed to the classification module, which will classify the navigation as SE-ad-related or benign. If the navigation is determined to be SE-ad-related, TRIDENT will block the navigation to prevent the user from being directed to the SE attack. Internally, TRIDENT uses a random forest [39] classifier for classification. We configure the random forest as an ensemble of 100 decision trees with each decision tree using $\sqrt{N}$ features, a default value that works well for TRIDENT, where N is the total number of features.

When visiting a website, the `SEAgent` continuously sends events to the post-processing daemon, which builds the WAHG, extracts the features, and runs the classifier.. When navigation is scheduled, the features, except for the consequence ones, are sent to the classifier. When the navigation is about to commit, the daemon receives the updated consequence features and reruns the classifier before the landing page commits. When the classifier classifies a navigation request as malicious, the `SEAgent` inserts an interstitial warning page to make the user aware of the dangers ahead. Note that we use one single model rather than two, trained with and without consequences features because the performance difference is minimum as shown in §4.4.

4 Evaluation

This section discusses the extensive experimental evaluations we completed for TRIDENT and compares TRIDENT with the state-of-the-art tools. Our evaluations address the following research questions:

RQ1: How accurately can TRIDENT detect navigation initiated by SE-ads?

RQ2: Are the features used by TRIDENT understandable and robust?

RQ3: How well does TRIDENT perform compared with the state-of-the-art tools?

RQ4: What is the runtime performance and resource consumption overhead for `SEAgent`?

4.1 Experiment Setup

This section discusses the websites used in our evaluation and how we simulated user actions to trigger SE-ads and navigation to SE-websites for data collection.

Data Source. Our data collection process relied on *publicwww.com* (P.W.) [40], a popular source code search engine, to collect scripts that may deploy SE-ads. We obtained over 100,000 ad publisher websites by searching JS code snippets on P.W. by following the approaches used in the study [9]. These JS code snippets were obtained by analyzing websites, which were open-sourced in that study, and websites we encountered by searching for free content-sharing websites, which prefer to include low-tier ad networks as suggested by prior research [12].

Crawler Design. Unlike prior works [19, 33, 41] that only crawl the Internet by loading the home page, this work requires a crawler to interact with as many SE-ads as possible. To achieve this, we built the crawler on top of Puppeteer [42] to simulate users' interactions with web pages, and developed a clicking strategy conducive to triggering navigation. First, we collect anchor elements that point to a different origin and place them in an anchor node pool. Additionally, we collect elements with mouse listeners in a mouse event pool. Because large elements have a higher chance of being clicked, we sort the DOM nodes in descending order of the element's bound-

ing box size to prioritize the elements that are most likely to capture a real user's clicks. Then, our crawler clicks the elements in these pools one by one. If a click triggers navigation, the crawler takes a screenshot of the navigated page.

We deployed the crawlers in 20 docker containers simulating users' interactions with websites from October 2021 to January 2022 to collect training data and in October 2022 to collect data for examining TRIDENT's robustness. We use these two datasets to evaluate TRIDENT's accuracy, investigate TRIDENT's false positives and false negatives, and compare with the state-of-the-art tool. We will discuss these experiments in detail in the following sections.

4.2 Ground Truth & Dataset Cleaning

This section first introduces the techniques we used to collect ground truth for the datasets and then discusses our approaches to cleaning and balancing the datasets.

Labeling. Prior works [19, 41] rely on EasyList and EasyPrivacy [30] as ground truth to label ads-related URLs. Unfortunately, these lists focus on generic ads. Using these lists as the ground truth would make TRIDENT target generic ads, rather than SE-ads, which is not our goal. To identify the ground truth in our datasets, we developed a semi-automated approach as the following to identify whether navigation lands on a malicious (SE) website.

- **L1: Landing page screenshots clustering.** During crawling, when a new tab is open, or cross-origin navigation occurs, the crawler will take a screenshot of it. Following the methodology in the study [9], we use DBScan on the perceptual hashes [43] of those screenshots to cluster them. Then, we review each cluster to visually identify whether a landing page is a SE-website. If it is, we label this navigation malicious.

- **L2: Categorical BlockList, Google Safe Browsing, and VirusTotal.** We choose three additional services for identifying whether a website is malicious or not, a categorical BlockList [31] on Github, which is popular in the community and is updated frequently, Google Safe Browsing (GSB) [21], and VirusTotal (VT) [20]. We consider a URL malicious if it falls in the buckets of Malware, Scam, Abuse, Phishing, and Fraud in the BlockList, is determined unsafe by GSB, or is flagged out by at least one of the engines in VirusTotal. Then, we feed all landing page URLs and the URLs in the redirect chain to these three services to label them automatically. If a page's URL is labeled malicious, we mark this navigation event as malicious.

Then, we label a navigation event malicious when either of the two components says it is malicious. Although these two labeling techniques may mislabel some examples due to the imperfection of the chosen block lists and image clustering algorithm, they make the labeling process much more efficient for a large dataset. Therefore, we use semi-auto-labeled ground truth to train TRIDENT.

Ground Truth. In total, we obtained 258,008 navigation events initiated by JS code. Using those two labeling techniques, we identified 1,479 navigation events resulting in SE attacks. Note that we obtained more than a million JS files, but most did not demonstrate behaviors related to our features, so we excluded them from the ground truth.

Next, we show the statistics of the training (ground truth) dataset in Tab. 4. The ground truth covers more than ten low-tier ad networks (e.g., AdSterra, PopCash, etc.) and major top-tier ad networks (e.g., Google, Facebook, etc.). We did not identify the brands of some ad networks. Therefore, if their domain(s) have a pattern, we group them, such as ___*cdn.com* and *cdn.*___*.xyz* where the blanks are random strings. Otherwise, they are labeled "Unknown". Some low-tier ad networks (e.g., "PopAds") are known to distribute SE-ads [9, 37]. However, we did not find positive samples from the training dataset for these ad networks. After investigation, we found that most of the navigation went to adult or benign websites (e.g., yahoo.com) due to cloaking. These adult websites did not show SE attacks at the time of data collection.

Then, we list the types of SE attacks discovered by the labeling techniques in this ground truth dataset in Tab. 5. We categorize those SE attacks based on the screenshots of the landing pages obtained by our crawlers. We have six categories of SE attacks in total. And we give examples in Fig. 7.

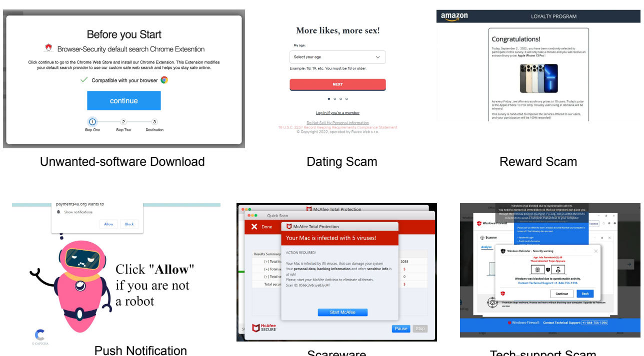


Figure 7: SE-websites examples in the training dataset.

Datasets Cleaning. We found that the training dataset was heavily imbalanced after labeling. There were two problems in the dataset: (1) the data was heavily imbalanced between classes, and (2) the data was imbalanced within the negative class (e.g., more scripts for rendering first-party content than scripts for injecting third-party ads). This is expected because benign scripts are ubiquitous. Training TRIDENT directly on this imbalanced dataset would undoubtedly produce a poor model. There are generally two strategies to overcome the imbalanced dataset problem: (1) over-sample the minor (malicious/positive) class or (2) under-sample the majority (be-

Ad Network	# of navigation events	# of SE-websites landed	% of SE attacks
Unknown	119,391	438	0.37%
AdSterra	1,247	350	28.07%
PopCash	1,085	267	24.61%
___cdn.com	559	141	25.22%
lkqd / Nexstar	236	105	44.49%
RevenueHits	276	41	14.86%
cdn.___.xyz	77	36	46.75%
whos.amung.us	29	29	100.00%
ZarPop (Persian specific)	25	16	64.00%
AdMaven	324	13	4.01%
OnClasrv	20	12	60.00%
uTarget (Russian specific)	32	11	34.38%
realsrv.com	650	10	1.54%
Propeller	4	4	100.00%
AdExtrem	21	3	14.29%
AdFly	61	3	4.91%
AddThis	5,552	0	0.00%
Google Ads	66,677	0	0.00%
AdGebra	311	0	0.00%
AdPartner	93	0	0.00%
Amazon Ads	24	0	0.00%
Facebook Ads	13,983	0	0.00%
Infolinks	15,162	0	0.00%
Mgid	6,290	0	0.00%
PopAds	1,087	0	0.00%
Rekmob	248	0	0.00%
ShareThis	17,973	0	0.00%
TeckAd	22	0	0.00%
Twitter	6,549	0	0.00%
Total	258,008	1,479	0.05%

Table 4: Statistics of the ground truth dataset by ad network.

SE Attacks	# of SE attacks	# labeled by L1	# labeled by L2
Unwanted-software Download	857	817	539
Dating Scam	222	204	48
Reward / Lottery Scam	177	156	92
Push Notification	148	148	25
Scareware	51	29	42
Tech-support Scam	24	20	13

Table 5: SE attack types in the ground truth dataset. The unwanted-software download includes binary files and browser extensions. We identify SE attacks based on the union of L1 and L2.

Class Label	New-Tab Nav.	Same-Tab Nav.
Malicious	1,358	121
Benign	5,726	250,803

Table 6: Navigation events made by scripts in the training dataset. The data within the benign class is imbalanced in terms of navigation pattern.

nign/negative) class. To address our problems, we decided to under-sample the negative class as recommended by the state-of-the-art techniques [44, 45] to reduce the false-negative rate as our goal is to detect SE-ads as accurately as possible.

Additionally, we removed "silent" scripts that do not invoke any DOM APIs of our interest and under-sampled the same number of positive class from the negative class, which addressed the first problem. To address the second problem, we analyzed the distribution of the features and found that benign scripts tended to navigate the users in the same tab. In contrast, the malicious scripts preferred to open new windows, as shown in Tab. 6. Random sampling from the benign class would yield a large portion of same-tab navigation entries, making a performant classifier. However, this classifier would not generalize to websites that open windows in new tabs, which are data points near the classification border. Therefore, we need to choose more samples near this border, in this case, more entries in the new-tab navigation from the benign class. After analyzing the distribution of benign navigation events, we chose 50% from the NT entries and 50% from the ST ones. We will explain why we choose this ratio in §4.3.1.

4.3 TRIDENT Performance

To answer **RQ1**, we evaluated our model using 10-fold cross-validation on the training dataset and reported the average accuracy. Next, we discuss the disagreement between TRIDENT and the ground truth data.

New-tab Nav.	Same-tab Nav.	Accuracy	Precision	Recall	F-1 Score
100%	0%	87.76%	86.69%	89.31%	87.98%
90%	10%	88.30%	86.09%	91.68%	88.80%
50%	**50%**	**92.63%**	**90.63%**	**96.28%**	**93.37%**
0%	100%	99.76%	99.78%	99.43%	99.60%
Random Sampling		99.36%	99.14%	99.59%	98.17%
No Sampling		97.69%	89.71%	76.39%	82.52%

Table 7: Model accuracy with different approaches of under-sampling the majority (negative) class.

4.3.1 Accuracy

First, we trained the model with the raw imbalanced dataset (no sampling), which reported good accuracy, but bad precision and recall, as shown in Tab. 7. Next, to improve the performance, we used five approaches to balance the dataset. Tab. 7 presents the results. Notably, the more Same-tab Navigation (STN) entries we sample, the better the model performs. However, it lacks generality. When we trained the model with (New-tab Navigation) NTN benign samples (all benign data points near the borderline), the accuracy dropped to 87.76%. Although the model has the lowest accuracy, this situation (each navigation opens a new tab) is implausible. As shown in Tab. 6, 97.77% of the navigation events happened in the same tab for the benign class. Therefore, to be conservative and include a good number of data points near the borderline from the benign class, we decided to use 50% from the NTN entries and 50% from the STN entries for the benign samples to balance the dataset.

With this balanced training dataset, TRIDENT detects SE-ads related navigation with 92.63% accuracy, 90.63% precision, 96.28% recall, and 93.37% F-1 score.

4.3.2 TRIDENT on Tranco Top 1K

TRIDENT is designed to detect and block navigation initiated by SE-ads. In other words, TRIDENT is not supposed to block normal ads and harm the user experience. Therefore, we tested TRIDENT on popular websites from the Tranco Top 1k list. In

addition to the auto-crawling process, we also collected data from 10 popular news, e-commerce websites, and social platforms (*nytimes, washingtonpost, cnn, forbes, bestbuy, newegg, ebay, twitter, facebook, reddit*) by explicitly interacting with online ads and links.

In total, we obtained 109,744 script-frame combinations. However, we only found 78 navigation events initiated by JS. When labeling these 78 events, we only found one website (*yts.mx*, ranked at 981/1,000 as of writing) served SE-ads and lead us to a SE-website. Next, we fed this labeled data into TRIDENT's model and achieved 100% accuracy, which means TRIDENT allowed navigation made by interacting with normal ads, and links and blocked the navigation initiated by the SE-ads served on *yts.mx*.

4.3.3 False Positives & False Negatives

We now analyze TRIDENT's FP and FN cases. We reused the dataset collected for the Tranco top 1k list and crawled 1,000 websites from our ad publisher website lists. In total, we collected 14,045 navigation events made by JS code. These navigation events involved 3,611 unique scripts and 5,823 unique web pages crawled from the 2,000 websites. We first tested this dataset on the original model using our semi-auto labeling techniques described in §4.2 and obtained a false positive rate of 5.86% (827 FPs). After manual investigation, we discovered 477 mislabeled entries which were primarily from adult websites, and corrected the labels. Finally, we obtained 763 positive samples and 13,283 negative samples. These samples yielded a false positive rate of 2.57% and a false negative rate of 0.13% on the original model. We present the statistics of the positive class for the testing dataset in Tab. 8. TRIDENT detected all navigation to SE-websites initiated by at least seven low-tier ad networks. We also found that TRIDENT detected "PopAds" and "PopMyAds". These two ad networks were known to distribute SE-ads [9, 37], however, they did not show any SE-websites when we were collecting the ground truth. Fortunately, TRIDENT detected them when they took our crawlers to SE-websites.

Ad Network	# of navigation events	# of SE attacks	# of SE attacks detected
AdSterra	1,519	511	560
Others	8,793	151	297
PopCash	916	76	76
realsrv	129	16	33
___cdn.com	156	4	30
PopAds	596	2	2
PopMyAds	349	2	4
AdMaven	62	1	6

Table 8: Statistics of positive class for the testing dataset to show TRIDENT can detect all SE attacks. The negative class is omitted for brevity.

False Positives. TRIDENT achieved a 2.57% false positive rate after correcting the labels. After looking at these false positives, we identified three types. We now first discuss each type and then propose mitigation approaches.

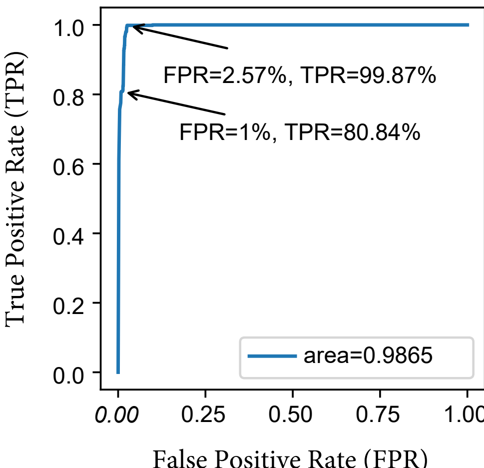


Figure 8: Model performance by different false positive rates.

In the first type, the ad script injects DOM elements into the websites for benign purposes. For example, AddThis [46] accounts for 26% of the false positives. This ad network primarily injects clickable DOM elements into web pages. By clicking those elements, the user can share the website with others by posting a message on Twitter, Google+, or Facebook. This type of false positive does not have a pattern. Some of them are close to the decision boundary and some are not.

In the second type, the ad script injects SE-ads, but interacting with the SE-ads takes the user to normal advertiser websites directly. For example,*absoluteroute.com* accounts for 8.8% of the false positives. The ad script from this domain injected invisible overlays, but took our crawlers to normal advertiser websites without redirection, e.g, *worldofwarships.com* based on the crawling records. These data entries are very close to the classifier's decision boundary, with an average probability of 66.79% being positive.

In the third type, the ad scripts inject SE-ads. And interacting with the SE-ads leads to adult websites. In the auto-labeling and manual labeling review processes, we did not find those adult websites launching social engineering attacks immediately. However, those adult websites usually track users and can launch sophisticated attacks. This type of false positive is far from the decision boundary because the ad scripts do inject SE-ads and redirect users multiple times and TRIDENT determines they are SE-ads related navigation. The only difference is that the landing page does not show social engineering attacks immediately. Note that we found some mislabeled adult websites during the manual labeling review, but these false positives were not.

To allow for easy comparison with TRIDENT, in Fig. 8 we show the ROC curve and highlight the TPR at an FPR of 1% and at an FPR of 2.57%. For a practical deployment, we believe that setting the detection threshold to achieve a TPR

of 99.87% at an FPR of 2.57% offers a better trade-off. The main reason is that the baseline for computing the false positives consists only of navigation events that are initiated by JS. As discussed in 4.3.2, navigation events initiated by JS are relatively rare; we only found 78 navigation events initiated by JS out of thousands of instances. Therefore, TRIDENT's classifier is rarely invoked, and only a small fraction of those rare events result in a potential false positive (i.e., an erroneously blocked navigation). Furthermore, to further improve TRIDENT by reducing the FPR, we can use a whitelist-based approach to avoid incorrectly blocking trusted ad networks, e.g., AddThis, to reduce the first type of FP. This whitelist is configurable, allowing the user to decide what to include.

False Negatives. TRIDENT only found one false negative, which converts to 0.13% false negative rate. The adult website *hentaibedta.net* embedded malicious links in its first-party content. Specifically, it included ad images that pointed to an external website (*ouo.io/QqJgfz*). During the investigation, this external website eventually landed the user on a malicious browser extension downloading page and two reward scam pages. The SE-ads on the adult site were injected by the first-party script and behaved as if they were the first-party content. Although TRIDENT failed to detect the script, we argue that this type of ad script is considered out of scope as TRIDENT focuses on ad networks that distribute malicious ad scripts at scale. If we changed its property to third-party, TRIDENT can detect the navigation initiated by this script. We will discuss TRIDENT's limitations in §5.

4.4 Feature Importance and Robustness

To answer **RQ2**, we assessed TRIDENT's classifier by analyzing its feature importance to confirm that the features were understandable and reflected domain experts' intuition. Beyond explaining feature importance, we analyzed our model's robustness against concept-drift [47] and evading techniques.

4.4.1 Feature Importance

We selected TRIDENT's features based on our domain knowledge, expert intuition, and previous studies [13, 14] to obtain meaningful and understandable features. To this end, we evaluated the feature group importance, guided by the Leave-One-Group-Out approach proposed by Au *et al.* [48]. We reported the results in Fig. 9a using ROC curves. The *property* feature group has the lowest AUC score, whereas the *action* feature group has the highest score. This result is understandable as the properties of a script do not indicate its maliciousness, and what a script does reflects its objective the most. To better understand what matters most in the action feature group, we also present a breakdown in Fig. 9b, which depicts that the navigation features are more important than others. The rest of the features contribute almost equally.

Although the *property* feature group scores 0.03% lower than the best score by using all the three feature groups, based on the FN discussion in §4.3.3, it is helpful when a data point is near the decision boundary. Also, the scores of training with and without consequence features only have a 0.67% difference. Therefore, we can use one single model before and after navigation as mentioned in §3.6.

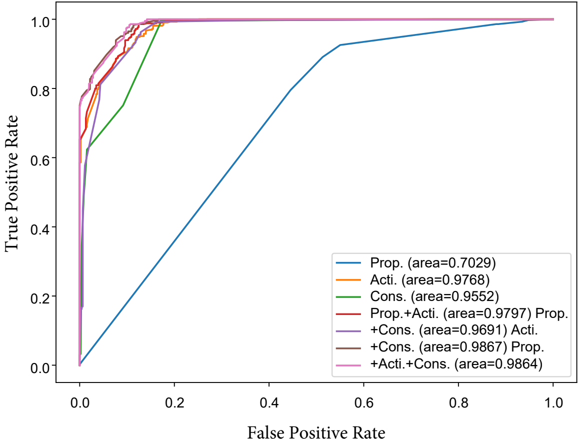
4.4.2 Robustness

In this section, we evaluated how well TRIDENT performs against concept-drift [47] by testing the model using the testing dataset. Next, we tested the robustness of TRIDENT's classifier by altering feature values to simulate evading TRIDENT.

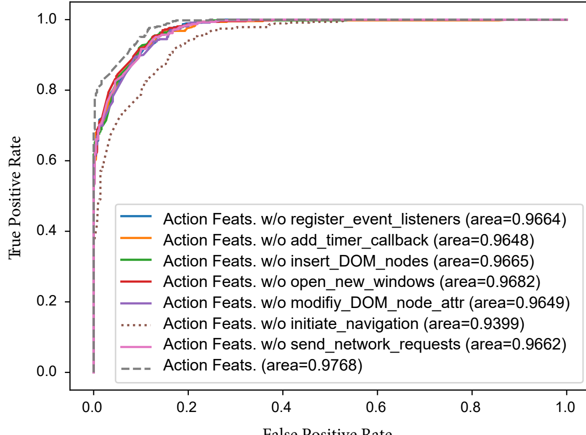
Concept Drift. Machine learning models are known to lose their effectiveness over time due to the underlying changes in the data distribution used to train the model. We build TRIDENT to slow down the degradation process by focusing on the behaviors of the scripts that inject SE-ads. To this end, we evaluated TRIDENT's accuracy over time by testing it on a dataset crawled in October 2022, almost one year after the initial model was trained. We obtained a similar result for the dataset used for FP and FN analysis. TRIDENT achieves an accuracy of 97.37 % with a precision of 98.25% and a recall of 97.37%. These results indicate that we do not need to frequently retrain TRIDENT because the fundamental techniques used by those SE-ads do not change often. However, we recommend updating TRIDENT and retraining the model every several months for the potentially new JS APIs introduced and employed by ad networks.

Evasion Simulation. We have discussed one FN that evaded TRIDENT in §4.3.3. Given the limitation of gathering more evading samples, we simulated evasive SE-ads by altering feature values. We generated four guidelines based on our domain expert intuition of feasible evading techniques: (1) include the malicious script as the first-party script; (2) put the script as an inline script; (3) directly bring the user to SE-websites without redirects; and (4) behave as benign scripts while stealing clicks. Based on these four guidelines, we reported the evasion rates in Tab. 9 by techniques.

First, we changed the *property* feature groups to make the scripts first-party and/or inline. This alternation yields a maximum of 5.11% evasion rate. Next, we let the attacks directly bring the users to the SE-websites. This change alone leads to a 3.62% evasion rate. When combining the techniques used for the *property* features, the evasion rate went up to 9.17%. Finally, we tested altering the *action* features, which is the most challenging part since we need to keep the attacks valid. We took a conservative approach, keeping the features related to DOM manipulations, including event listener registrations, DOM node modifications, etc. We only updated the remaining features in this feature group and reported the result in the lower part in Tab. 9. We did not report the combination of these behaviors since the evasion rate did not increase signifi-



(a) Action feature group has the most impact on the classifier.



(b) The navigation information weighs more than others.

Figure 9: Feature importance illustrated by the performance of models trained on different combinations of features.

Approaches	**Evasion Rate**
First-party script (Fst.Pty.)	2.13%
Inline script (Inl.)	5.11%
No redirects (NoRdr.)	3.62%
NoRdr. + Fst.Pty.	2.56%
NoRdr. + Inl. + Fst.Pty.	9.17%
Do not request external resources	1.49%
Do not add callbacks	1.49%
Do not attach iframes	1.92%
Do not modify node attributes	1.70%

Table 9: Evasion rates by altering key feature values.

cantly. The highest evasion rate was 1.92% by not attaching iframes on the page.

In summary, we found that the attackers can evade TRIDENT at a high rate only if they could include their malicious scripts as first-party by colluding with the website owner or compromising the web servers. However, this is unlikely because the attackers can have better choices of compromising visitors when they can access the web servers.

4.5 Comparison to State-of-The-Art Systems

To answer **RQ3**, we compared TRIDENT with two state-of-the-art tools: Brave Shields, the adblocking module for Brave Browser [18] from industry and ADGRAPH [19] from academia. We first show Brave Shields is insufficient using a filter-list-based approach and then show ADGRAPH is not suitable for SE-ads.

4.5.1 Traditional Blacklist-Based Ad-Blockers

Adblock Plus is the most popular blacklist-based ad-blocker. It leverages manually maintained blacklists to deny or whitelists to allow ad or tracker traffic. Brave Browser has integrated a variety of filter lists, which are a superset of Adblock Plus's, so we set up its ad-blocking component [49] locally to see how well TRIDENT performs against traditional ad-blockers. Brave Shields takes in a script URL and a frame URL and returns a binary decision. We feed Brave Shields our script URLs along with their corresponding running frame's URLs and analyze the disagreements between Brave Shields and the ground truth. As described in dataset cleaning section, we obtained 1,479 positive samples for the training dataset, of which Brave Shields missed 14.74%. To make a fair comparison, we tested Brave Shields on our two batches of datasets. First, we performed a 70/30 training/testing split of our training dataset, following the data balancing method we used previously, and trained a model to test the testing split. The second dataset was the testing dataset we collected in March 2022. To evaluate how well TRIDENT performs against Brave Shields, we only need to focus on the false negative rate, the rate of evading the detection. Tab. 10 reports that TRIDENT outperforms Brave Shields almost by 7 times.

4.5.2 Machine Learning Based Ad-Blockers

We focus on two related prior works on ad-blocking: ADGRAPH [19], the first ML-based ad-blocking tool that is based on the contents of ads and trackers, and WEBGRAPH [33], the first ML-based ad-blocking tool that is based on the action of ads and trackers. In the following, we discuss why ADGRAPH and WEBGRAPH cannot solve the problem TRIDENT is trying to solve.

First, we replicated ADGRAPH by crawling *Alexa Top 10k* using the open-sourced ADGRAPH binary, labeled the data using the latest filter lists as of writing, and built the same classifier as described in the paper. We then created the testing dataset by letting ADGRAPH crawl random P.W. 1k websites from our website seed list. The accuracy on these sites

Dataset	FNR by Brave Shields	FNR by TRIDENT
First batch	15.14%	2.13%
Second batch	12%	1.49%

Table 10: FNR of detecting SE-ads by Brave Shields and TRIDENT. The first batch is 30% split from the training dataset. The second batch is from the testing dataset.

Model	Accuracy	Precision	Recall	F-1 Score
ADGRAPH for Generic Ads	83.25%	80.12%	81.65%	80.88%
ADGRAPH for SE-ads	81.51%	71.34%	75.33%	73.28%
TRIDENT	95.07%	96.11%	95.49%	95.79%

Table 11: TRIDENT outperforms both ADGRAPH models for detecting SE-ads. The "Generic Ads" is the original ADGRAPH model and tested on the SE-ads dataset, whereas the "SE-ads" is trained and tested on the SE-ads datasets.

dropped to 83.25%. This shows that ADGRAPH for generic ads does not work well for SE-ads.

Next, we sampled 1,000 websites from the training dataset and 1,000 websites from the testing dataset, respectively. We refer to the two datasets as *P.W. 1k Trn.* and *P.W. 1k Tst.* for simplicity. For each batch of P.W. 1k, 500 sites were from websites known to publish SE-ads and 500 were from benign websites. Then, we let ADGRAPH crawl these 2,000 websites and labeled the datasets using our ground truth. Finally, we trained ADGRAPH and TRIDENT on the same training dataset and tested them on the same testing dataset. As shown in the lower part in Tab. 11, TRIDENT outperforms ADGRAPH by over 10%. ADGRAPH trained by *P.W. 1k Trn.* performs even worse than the generic model. However, this is not an apple-to-apple comparison. The ADGRAPH for Generic and ADGRAPH for SE-ads are two different models as they are trained on different datasets which are labeled differently. The former targets generic ads while the latter targets SE-ads. Moreover, while replicating ADGRAPH, we found URLs with protocol `data:` will be considered as NON-AD in the labeling process of ADGRAPH. This implies resources using base64 encoded URL would likely escape ADGRAPH's detection because ADGRAPH can extract nothing from such URLs. This gives the adversaries opportunities to import external scripts using `"data:text/javascript,ZG9Tb21ldGhpbmcoKQ=="` which means `doSomething()` to evade ADGRAPH.

WEBGRAPH improves the robustness of ADGRAPH by removing the content features and adding information flows of network, storage, and shared. Because WEBGRAPH is not open-sourced as of writing, we are not able to evaluate it with our datasets. However, we argue WEBGRAPH is not designed to capture how a script manipulates the DOM to lure users to social engineering websites. Hence, its performance on our datasets should be equivalent to ADGRAPH's.

4.6 Runtime Overhead

To answer **RQ4**, we evaluated the runtime performance of `SEAgent`, the major component that may induce overhead, including running time and memory and CPU usage.

Runtime Overhead. To quantify the impact on the user experience, we measured the page load time to evaluate the runtime overhead for the Tranco top 1k websites [50]. To measure this, we leveraged Chromium's `TRACE_EVENT` instrumentation infrastructure for profiling [51]. We added a new trace category named `blink.seagent` and put `TRACE_EVENT0` marco at the beginning of each instrumentation hook. Then, we enable `blink.user_timing` to measure the page load time, which is defined as the time spent between the navigation request start and the load event end [52]. For each website, we loaded the page into the browser 10 times and selected the median page-load overhead.

The distributions of the runtime overhead are shown in Fig. 10. The median runtime overhead is 2.13% which results in a 0.02-second increase in the page load time, which are comparable to previous works [41, 53, 54]. Looking at outliers, we found the websites have more DOM modifications were more impacted by the `SEAgent`. For instance, *kickstarter.com* took the longest to load with 14.34% (0.33 seconds) overhead. After checking this website, we found that JS inserted more than 35,000 DOM nodes and modified their attributes, and then removed half of them before the page was fully loaded. These outliers are rare given that the overhead for the 95% of the Tranco 1k list is less than 5.7%.

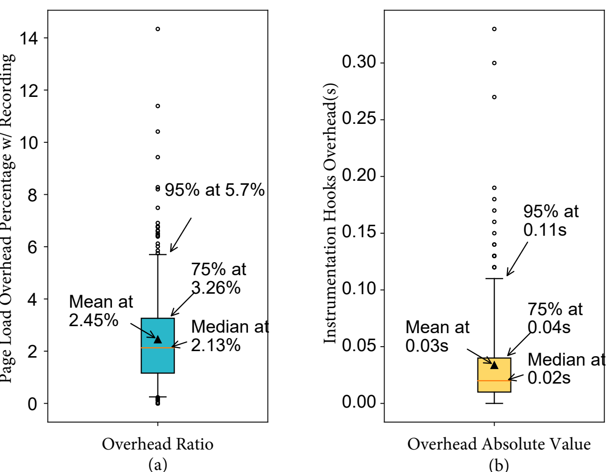
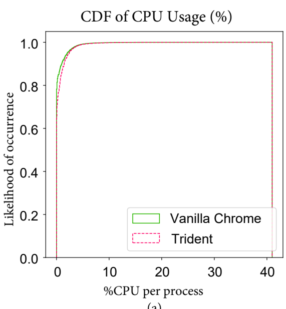


Figure 10: The runtime overhead induced on the page load by TRIDENT for the Tranco 1k. (a) presents the runtime overhead increase for the page load. (b) provides the absolute time induced by TRIDENT.

Resource Overhead. To evaluate TRIDENT's resource usage overhead, we measured the CPU and memory usage for the websites listed in the Tranco top 1k [50]. It is challenging to separately measure the precise resource consumption of

TRIDENT's components because this would require sophisticated code instrumentation to calculate how much memory is allocated and how many CPU cycles are consumed. Therefore, we leverage an alternative approach that allows us to estimate the resource usage overhead. We use the `ps` [55] command to continuously record the CPU and memory usage of the browser processes (with 100ms granularity) while visiting the home page of every website in the Tranco top 1k list ten times (i.e., 10k page loads in total), using both vanilla Chromium and TRIDENT. Every time a page is visited, we wait for the page to be fully loaded and then wait another 10 seconds before visiting the next page.



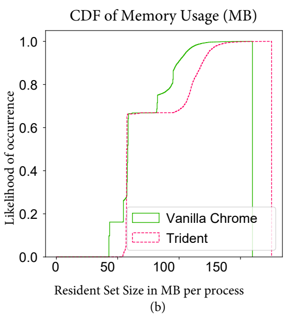


Figure 11: The resource usage induced on the page load by TRIDENT for the Tranco 1k. (a) presents the distribution of CPU usage. (b) provides distribution of the memory consumed by TRIDENT. TRIDENT induced negligible CPU and memory overhead when visiting the Tranco 1k.

To compare the resource usage of vanilla Chromium and TRIDENT, we summarize the results as Cumulative Distribution Function (CDF) graphs in Fig. 11. As seen from Fig. 11, TRIDENT induces negligible CPU overhead and limited memory usage overhead, which is mainly driven by TRIDENT's need to perform data serialization and buffer browser data objects that are then recorded to the TRIDENT's trace files.

Summary. With 2.13% overhead on page load time, negligible CPU overhead, and small memory overhead compared to the memory available on modern devices, we believe TRIDENT could be deployed in real-world environments to work as a real-time classification system.

5 Discussions

5.1 Limitations

This section discusses the limitations of TRIDENT in three ways. One is the runtime environment of TRIDENT which may allow adversaries to learn the existence of TRIDENT and refuse to display malicious content. Another is that the diversity of the training dataset may be limited when we crawl the Internet. The last is when the malicious ad scripts are included as a first party (un)intentionally.

Runtime Environment. We envision TRIDENT being deployed as a browser extension with Chrome DevTools Protocol turned on as a prototype, which exposes TRIDENT's existence. Adversaries may detect TRIDENT and then cloak themselves or refuse to display content until the users turn off TRIDENT. To address this limitation, we could embed TRIDENT directly into the browser to make it invisible to those adversaries. We leave this for future work.

Data Collection and Labeling. Unlike previous works [17, 19, 33, 38, 56] which target at generic ads and trackers, TRIDENT targets at SE-ads, which are not as ubiquitous as those ads and trackers. We rely on *publicwww.com* to collect websites that inject SE-ads. To this end, the diversity of the training dataset is limited to a small number of ad networks we have identified by reverse-engineering their ad scripts and searching on the Internet. While TRIDENT performs well based on this dataset , its accuracy may drop when it encounters unseen ad network scripts. However, TRIDENT can periodically retrain its classifier on improved ground truth as the users provide feedback.

First-party Ad Scripts. In section §4.3.3, we found one FN example: one malicious ad script was (un)intentionally included by the website operator. TRIDENT failed to detect the navigation initiated by the script. The results in Tab. 9 also show that the attackers may have a higher chance to evade TRIDENT by injecting ad scripts as first-party scripts. However, we argue that colluding with the first party to launch SE-ads for SE attacks at scale is implausible. Therefore, we consider first-party SE-ads out of scope. Some website operators may unintentionally include malicious scripts on their websites. To this end, we recommend users look for the suspicious behaviors described in §3.5.

5.2 Ethical Considerations

In line with previous studies [9, 10, 53] that need to crawl the Internet, our crawling experiments simulated user's clicks on ad publishers, which may lead to advertisers' landing pages. Because our primary goal is to analyze the behavior of interacting with SE-ads, we argue it would not be possible without clicking on the websites to trigger SE-ads and navigate to SE-websites. Moreover, our crawlers do not target any specific ads or ad campaigns. They randomly choose ten clickable elements and ten links. These clicks resulted in 5,726 opened windows that loaded benign content. Assuming that all of these windows eventually reached the landing pages of advertisers, we found our crawler made two clicks on the ads for each advertiser on average. Considering the average CPC (cost per click) being USD $0.75 [25], the cost to each advertiser would be USD $1.5 on average. This result shows that our crawling experiment ensured minimal financial losses for legitimate advertisers while generating results that help prevent people from falling into WSEAs.

6 Related Work

Web-based social engineering attacks. While previous works [7–10, 13, 14, 36, 37] have studied web-based SE attacks through malicious advertising, they either focus on detecting specific web SE attack vectors or lack a defensive method towards their findings. For example, Vadrevu and Perdisci [57] used visual clustering and heuristics to identify SE-attack campaigns at the landing page level, which was done "offline". And this work [57] does not focus on detecting SE-ads, which is TRIDENT's focus. Sanchez-Rola *et al.* [36] found that even popular websites intercepted users' clicks and brought them to harmful websites; nonetheless, this study does not provide a solution to mitigate the consequences of those clicks. Zhang *et al.* [13] built OBSERVER to study three click intercepting techniques. However, turning OBSERVER into an accurate detection system is challenging because benign websites may also intercept users' clicks for benign purposes [36]; therefore, analyzing the events triggered by the clicks is mandatory, which OBSERVER does not focus on while TRIDENT does. Koide *et al.* [37] developed STRAYSHEEP to identify SE-websites effectively by using a crawler to interact with the websites. Unfortunately, STRAYSHEEP is also an offline tool and is not designed for online detection. TRIDENT takes a generic approach, considering both what a script is doing on a webpage and what consequences it causes, to detect SE attacks by detecting their leading cause, which is SE-ads that often employ SE techniques to hijack clicks.

Clickjacking. Clickjacking is known as UI redressing attack that uses multiple transparent or opaque layers to trick a user into clicking on third-party content such that to bypass the same-origin policy [58]. Framebusting [59] is a good defense against clickjacking. However, it degrades the user experience on websites that requires cross-origin iframes, and the inconsistencies of implementation are concerning [60]. Previous works [61, 62] rely on the users to verify what they have clicked, which are not comprehensive and have usability concerns [63]. Unlike traditional clickjacking attacks which inject iframes, SE-ads have employed new techniques to steal clicks. Our work takes a new approach by analyzing what JS scripts do on a web page to complement prior studies.

Ad blocking. Generic ad blockers are efficient at blocking generic ads. However, they suffer from incompleteness and are easy to evade [9, 10]. Advanced ad blockers [17, 19, 33, 41] employ ML techniques to complement the generic ad blockers. Unfortunately, they are not trained to detect SE-ads and block the subsequent events triggered by interacting with those SE-ads. TRIDENT will be the second defense to protect users from falling into SE tricks by complementing them.

Browser Provenance Graph. JSGRAPH [54] instruments Chromium to build a graph for forensic analysis offline. MNEMOSYNE [53] builds a graph by leveraging the existing APIs in CDP. As discussed in §3.3, the current domains in CDP can not meet the expectation of recording the JS actions in real-time. PAGEGRAPH [41], as the successor of ADGRAPH [19], instruments the browser and expose its API through CDP. However, this implementation only sends a completed page graph when the web page emits `unload` event. In contrast, our real-time feature extraction requirement needs to access the graph whenever the graph is updated. While PAGEGRAPH maintains an in-memory graph representation as TRIDENT does, it still requires significant engineering work to use it directly. Therefore, we decided to extend the existing CDP with minimal instrumentation for TRIDENT.

7 Conclusion

In this work, we present TRIDENT, a novel online system for detecting and blocking social engineering ads. We show that TRIDENT can effectively detect SE-ads and block the consequent navigation to social engineering websites with an accuracy of 92.63%, which outperforms the state-of-the-art generic adblocking tools by more than 10%. Finally, TRIDENT 's runtime performance is extremely low and only has a 2.13% median increase on the page load time on websites in the Tranco 1k list.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd for their helpful and informative feedback. This material was supported in part by National Science Foundation (NSF) under grants No. CNS-2126641; the Office of Naval Research (ONR) under grants N00014-17-1-2179, N00014-17-1-2895, N00014-15-1-2162, and N00014-18-1-2662; and Cisco Systems under an unrestricted gift. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflects the views of NSF or ONR.

References

[1] Fatima Salahdine and Naima Kaabouch. "Social engineering attacks: A survey". In: *Future Internet* 11.4 (2019), p. 89.

[2] *Fully 84 Percent of Hackers Leverage Social Engineering in Cyber Attacks*. https://www.esecurityplanet.com/threats/fully-hackers-leverage-social-engineering-in-cyber-attacks/. 2017.

[3] *The Social Engineering Infographic - Security Through Education*. https://www.social-engineer.org/social-engineering/social-engineering-infographic/.

[4] Gianpiero Costantino et al. "CANDY: A social engineering attack to leak information from infotainment system". In: *2018 IEEE 87th Vehicular Technology Conference (VTC Spring)*. IEEE. 2018, pp. 1–5.

[5] *15 Alarming Cyber Security Facts and Stats | Cybint*. https://www.cybintsolutions.com/cyber-security-facts-stats/. 2020.

[6] *New Data Shows FTC Received 2.8 Million Fraud Reports from Consumers in 2021 | Federal Trade Commission*. https://www.ftc.gov/news-events/news/press-releases/2022/02/new-data-shows-ftc-received-28-million-fraud-reports-consumers-2021-0. 2022.

[7] Najmeh Miramirkhani, Oleksii Starov, and Nick Nikiforakis. "Dial One for Scam: A Large-Scale Analysis of Technical Support Scams". In: 2017. DOI: 10.14722/ndss.2017.23163.

[8] Amin Kharraz, William Robertson, and Engin Kirda. "Surveylance: Automatically Detecting Online Survey Scams". In: *Proceedings - IEEE Symposium on Security and Privacy*. Vol. 2018-May. Institute of Electrical and Electronics Engineers Inc., July 2018, pp. 70–86. ISBN: 9781538643525. DOI: 10.1109/SP.2018.00044.

[9] Phani Vadrevu and Roberto Perdisci. "What you see is not what you get: Discovering and tracking social engineering attack campaigns". In: *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC* (2019), pp. 308–321. DOI: 10.1145/3355369.3355600.

[10] Karthika Subramani et al. "When Push Comes to Ads: Measuring the Rise of (Malicious) Push Advertising". In: *{IMC} '20: {ACM} Internet Measurement Conference, Virtual Event, USA, October 27-29, 2020*. 2020, pp. 724–737. ISBN: 9781450381383. URL: https://doi.org/10.1145/3419394.3423631.

[11] Luca Invernizzi et al. "EvilSeed: A guided approach to finding malicious web pages". In: *Proceedings - IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers Inc., 2012, pp. 428–442. ISBN: 9780769546810. DOI: 10.1109/SP.2012.33.

[12] M. Zubair Rafique et al. "It's Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services". In: Internet Society, May 2017. DOI: 10.14722/ndss.2016.23030.

[13] Mingxue Zhang et al. "All your clicks belong to me: Investigating click interception on the web". In: *Proceedings of the 28th USENIX Security Symposium*. 2019.

[14] Ting Yu et al. "Knowing Your Enemy: Understanding and DetectingMalicious Web Advertising". In: p. 1070. ISBN: 9781450316514.

[15] USENIX Association., ACM SIGMOBILE., and ACM Digital Library. "Towards Measuring and Mitigating Social Engineering Software Download Attacks". In: USENIX Association, 2005, p. 48. ISBN: 9781931971324.

[16] Apostolis Zarras et al. "The dark alleys of madison avenue: Understanding malicious advertisements". In: *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*. Association for Computing Machinery, Nov. 2014, pp. 373–379. ISBN: 9781450332132. DOI: 10.1145/2663716.2663719.

[17] Zain Ul Abi Din et al. "PERCIVAL: Making in-browser perceptual ad blocking practical with deep learning". In: *Proceedings of the 2020 USENIX Annual Technical Conference*. 2020. ISBN: 9781939133144.

[18] *Blocking goals and policy - Brave Browser Wiki*. [Online; accessed 20-January-2022]. 2021. URL: https://github.com/brave/brave-browser/wiki/Blocking-goals-and-policy.

[19] Umar Iqbal et al. "AdGraph: A graph-based approach to ad and tracker blocking". In: *Proceedings - IEEE Symposium on Security and Privacy* 2020-May (2020), pp. 763–776. ISSN: 10816011. DOI: 10.1109/SP40000.2020.00005.

[20] *VirusTotal*. [Online; accessed 20-January-2022]. 2022. URL: https://virustotal.com.

[21] *Google Safe Browsing | Google Developers*. https://developers.google.com/safe-browsing/. 2022.

[22] *Rainbow Blocker Adware - Easy removal steps (updated)*. https://www.pcrisk.com/removal-guides/23298-rainbow-blocker-adware. 2022.

[23] *How Much Money Do Websites Make From Advertising?* https://adsterra.com/blog/how-much-money-websites-make-from-ads/. 2020.

[24] *Best CPM Rates for Publishers and Webmasters*. https://adsterra.com/blog/geos-with-high-cpm-rates-for-publishers/.

[25] *Google Display Ads CPM, CPC, & CTR Benchmarks in Q1 2018*. https://blog.adstage.io/google-display-ads-cpm-cpc-ctr-benchmarks-in-q1-2018. 2018.

[26] *Better Ads Standards - Google Ad Manager*. https://admanager.google.com/home/resources/feature-brief-better-ads-standards/. 2018.

[27] *Advertising and Marketing on the Internet: Rules of the Road | Federal Trade Commission*. https://www.ftc.gov/business-guidance/resources/advertising-marketing-internet-rules-road. 2022.

[28] *What are IAB Standard Ads? Why are They Important?* https://www.adpushup.com/blog/what-are-iab-standard-ads-why-are-they-important/. 2021.

[29] *IAB New Ad Portfolio: Advertising Creative Guidelines*. https://www.iab.com/guidelines/iab-new-ad-portfolio/. 2022.

[30] *EasyList*. [Online; accessed 20-January-2022]. 2022. URL: https://easylist.to/.

[31] *blocklistproject/Lists: Primary Block Lists*. [Online; accessed 05-June-2022]. 2022. URL: https://github.com/blocklistproject/Lists.

[32] *uBlock Origin - Free, open-source ad content blocker.* https://ublockorigin.com/. 2022.

[33] Sandra Siby et al. *WEBGRAPH: Capturing Advertising and Tracking Information Flows for Robust Blocking*. Tech. rep.

[34] Florian Tramèr et al. “AdVersarial: Perceptual Ad Blocking meets Adversarial Machine Learning”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 2005–2021. ISBN: 9781450367479.

[35] *Chrome DevTools Protocol*. [Online; accessed 20-January-2022]. 2022. URL: https://chromedevtools.github.io/devtools-protocol/.

[36] Iskander Sanchez-Rola et al. “Dirty Clicks: A Study of the Usability and Security Implications of Click-Related Behaviors on the Web”. In: *Proceedings of The Web Conference 2020*. WWW ’20. Taipei, Taiwan: Association for Computing Machinery, 2020, 395–406. ISBN: 9781450370233.

[37] Takashi Koide, Daiki Chiba, and Mitsuaki Akiyama. “To Get Lost is to Learn the Way: Automatically Collecting Multi-Step Social Engineering Attacks on the Web”. In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS ’20. Taipei, Taiwan: Association for Computing Machinery, 2020, 394–408. ISBN: 9781450367509.

[38] Umar Iqbal et al. *KHALEESI: Breaker of Advertising and Tracking Request Chains*. Tech. rep.

[39] Leo Breiman. “Random forests”. In: *Machine learning* 45.1 (2001), pp. 5–32.

[40] *Search Engine for Source Code - PublicWWW.com*. https://publicwww.com/. 2022.

[41] Quan Chen et al. “Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures”. In: *IEEE Symposium on Security and Privacy (S&P)* (2021).

[42] *Puppeteer*. [Online; accessed 20-January-2022]. 2022. URL: https://pptr.dev/.

[43] *Kind of Like That - The Hacker Factor Blog*. https://www.hackerfactor.com/blog/index.php?/archives/529-Kind-of-Like-That.html.

[44] Miroslav Kubat, Stan Matwin, et al. “Addressing the curse of imbalanced training sets: one-sided selection”. In: *Icml*. Vol. 97. 1. Citeseer. 1997, p. 179.

[45] Hui Han, Wen-Yuan Wang, and Bing-Huan Mao. *LNCS 3644 - Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning*. Tech. rep. 2005, pp. 878–887.

[46] *Get more likes, shares and follows with smart website tools - AddThis*. https://www.addthis.com/.

[47] Alexey Tsymbal. “The problem of concept drift: definitions and related work”. In: *Computer Science Department, Trinity College Dublin* 106.2 (2004), p. 58.

[48] Quay Au et al. “Grouped feature importance and combined features effect plot”. In: *arXiv preprint arXiv:2104.11688* (2021).

[49] *brave/adblock-rust*. [Online; accessed 20-January-2022]. 2021. URL: https://github.com/brave/adblock-rust.

[50] Victor Le Pochat et al. “Tranco: A research-oriented top sites ranking hardened against manipulation”. In: *arXiv preprint arXiv:1806.01156* (2018).

[51] *The Trace Event Profiling Tool (about:tracing)*. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/. 2022.

[52] *Page load time - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. https://developer.mozilla.org/en-US/docs/Glossary/Page_load_time. 2022.

[53] Joey Allen et al. “Mnemosyne: An Effective and Efficient Postmortem Watering Hole Attack Investigation System”. In: *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, Oct. 2020, pp. 787–802. ISBN: 9781450370899. DOI: 10.1145/3372297.3423355.

[54] Bo Li et al. “JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions”. In: *Network and Distributed Systems Security (NDSS) Symposium* February (2018). DOI: 10.14722/ndss.2018.23319.

[55] *ps(1) - Linux manual page*. 2022. URL: https://man7.org/linux/man-pages/man1/ps.1.html.

[56] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. “Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1143–1161.

[57] Jochem van de Laarschot and Rolf van Wegberg. “Risky business? Investigating the security practices of vendors on an online anonymous market using ground-truth data”. In: *Proceedings of the 30th USENIX Security Symposium*. 2021.

[58] *Clickjacking | OWASP Foundation*. https://owasp.org/www-community/attacks/Clickjacking.

[59] *X-Frame-Options - HTTP | MDN*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Frame-Options. 2022.

[60] Stefano Calzavara et al. “A Tale of Two Headers: A Formal Analysis of Inconsistent Click-Jacking Protection on the Web”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 683–697. ISBN: 978-1-939133-17-5.

[61] Lin Shung Huang et al. “Clickjacking: Attacks and defenses”. In: *Proceedings of the 21st USENIX Security Symposium*. 2012.

[62] Marco Balduzzi et al. “A solution for the automated detection of clickjacking attacks”. In: *Proceedings of the 5th International Symposium on Information, Computer and Communications Security, ASIACCS 2010*. 2010. DOI: 10.1145/1755688.1755706.

[63] Devdatta Akhawe et al. “Clickjacking revisited a perceptual view of UI security”. In: *8th USENIX Workshop on Offensive Technologies, WOOT 2014*. 2014.