FALCO: Detecting Superfluous JavaScript Injection Attacks using Website Fingerprints

Chih-Chun Liu[1], Hsu-Chun Hsiao[1] and Tiffany Hyun-Jin Kim[2]
[1]*National Taiwan University, Taiwan*
[2]*HRL Laboratories, LLC, U.S.A.*

Keywords: JavaScript Injection Detection, Website Behavior Fingerprint, Browser-based DDoS.

Abstract: JavaScript injection attacks enable man-in-the-middle adversaries to not only exploit innocent users to launch browser-based DDoS but also expose them to unwanted advertisements. Despite ongoing efforts to address the critical JavaScript injection attacks, prior solutions have several practical limitations, including the lack of deployment incentives and the difficulty to configure security policies. An interesting observation is that the injected JavaScript oftentimes changes the website's *behavior*, significantly increasing the additional requests to previously unseen domains. Hence, this paper presents the design and implementation of a lightweight system called FALCO to detect JavaScript injection with mismatched *website behavior fingerprints*. We extract a website's behavior fingerprint from its dependency on external domains, which yields compact fingerprint representations with reasonable detection accuracy. Our experiments show that FALCO can detect 96.98% of JavaScript-based attacks in simulation environments. FALCO requires no cooperation with servers and users can easily add an extension on their browsers to use our service without privacy concerns.

1 INTRODUCTION

In a JavaScript injection attack, the *man-in-the-middle* (MitM) attacker (e.g., an Internet Service Provider) injects malicious JavaScript in HTTP connections. Such malicious behaviors could lead genuine users to browse unwanted advertisement websites,[1,2] or launch DDoS attacks to other websites (Marczak et al., 2015). More specifically, JavaScript-based attacks utilize *untrusted intermediaries* between clients and webservers that inject malicious JavaScripts or replace legitimate JavaScripts with malicious ones. By injecting malicious JavaScripts, attackers can launch various attacks that affect clients locally, such as drive-by-downloads, phishing, malicious web advertising, and stealing user information.

Prior work on mitigating JavaScript injection attacks have several limitations, including lack of deployment incentives and difficulties in configuring policies. For example, *Content Security Policy (CSP)* allows a website to restrict script sources by setting the `script-src` directive in HTTP response headers. However, CSP is difficult to correctly configure (Weichselbaum et al., 2016) and can be modified by MitM adversaries on HTTP connections. Moreover, a victim website cannot protect itself from JavaScript-based DDoS with CSP, as the adversary here exploits vulnerable websites (which support neither HTTPS nor CSP) to attack the victim. *Cross-Origin Request Policy (CORP)* (Telikicherla et al., 2014) allows the server to block unwanted cross-origin requests from the browser, and can be applied to prevent JavaScript-based DDoS (Agrawall et al., 2017). However, besides the difficulties in configuring policies, CORP has not been adopted in practice. *Subresource Integrity* (SRI)[3] allows browsers to verify whether the transmitted web resources (e.g., JavaScript) are intact. However, it is not useful when a MitM attack tampers the verification information. *Stickler* (Levy et al., 2016) guarantees the integrity of the website content when the website owner uses a trustless CDN, which stores the website content that was signed by the publisher. However, Stickler requires the client to contact an additional trusted server to bootstrap its service. As prior techniques often require server-client collaboration, therefore further elevating the deployment challenges, Excision (Arshad et al., 2016) presents an

[1]https://www.privateinternetaccess.com/blog/2016/12/comcast-still-uses-mitm-javascript-injection-serve-unwanted-ads-messages

[2]https://zmhenkel.blogspot.tw/2013/03/isp-advertisement-injection-cma.html

[3]https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

Liu, C., Hsiao, H. and Kim, T.
FALCO: Detecting Superfluous JavaScript Injection Attacks using Website Fingerprints.
DOI: 10.5220/0009835101800191
In *Proceedings of the 17th International Joint Conference on e-Business and Telecommunications (ICETE 2020) - SECRYPT*, pages 180-191
ISBN: 978-989-758-446-6

in-browser solution to detect malicious JavaScript inclusion. However, Excision's browser instrumentation increases the browsing time by 12.2%.

To address these limitations, we present FALCO—a historical behavior-based robust system using website *fingerprints* while protecting enduser privacy. In particular, this paper attempts to address attackers that launch *additional network activities*, for example, to force users to visit advertisement pages or continuously send out requests to other websites. We refer to these types of attacks as *superfluous JavaScript injection*, or superfluous JS injection in short. If malicious scripts launch an excessive number of HTTP requests to a victim server or try to inject arbitrary advertisements, additional HTTP requests to external domains result in different dependency relationships between the website and external domains, compared to the condition when the website is free from such JavaScript attacks. To balance performance and accuracy, we generate compact fingerprints that encode the appearance of external domains using Bloom filters. We detect the unusual traffic that is triggered by the MitM attacker in two ways, trading off fingerprint availability and privacy: (1) A browser refers back to its own previous visits for the same website, or (2) the browser contacts an external fingerprint server that collects and manages website fingerprints reported by other nodes in a privacy-aware manner. Querying an external fingerprint server for each website visit raises privacy concern as the external server can learn about the user's browsing history. Besides maintaining the fingerprint database locally, FALCO can address this privacy concern by letting users download a group of fingerprints in advance or adapt existing private information retrieval techniques (Chor et al., 1995) to acquire the website's fingerprint without revealing its browsing history.

We simulated two types of JavaScript-based DDoS attacks to evaluate the detection effectiveness of our system. We also computed website fingerprints of the top 10k websites according to the Majestic top million sites database[4] to investigate the robustness of the proposed fingerprint representations. Our experiments show that FALCO can detect 96.98% of superfluous JS injection attacks in simulation environments.

Contributions. We introduce FALCO, a historical behavior-based fingerprinting system that detects superfluous JavaScript injection attacks. FALCO is capable of operating *without* any server-side cooperation, resilient to dynamic factors such as session information, and preserves enduser's privacy. According to our evaluation results, FALCO can detect 96.98% of superfluous JS injection attacks in simulation environments.

[4]https://majestic.com/reports/majestic-million

```
function imgflood() {
   var TARGET = 'victim-website.com'
   var URI = '/index.php?'
   var pic = new Image()
   var rand = Math.floor(Math.random() * 1000)
   pic.src = 'http://'+TARGET+URI+rand+'=val'
}
setInterval(imgflood, 10)
```

Figure 1: JavaScript code in Browser-based DDoS.

2 ATTACKER MODEL

We consider a man-in-the-middle (MitM) adversary, existing between the client and the server, injects a malicious script or replaces a benign script with a malicious one when the user browses the website. The MitM adversary (which is an untrusted intermediary) can leverage this malicious script to perform drive-by-downloads (Cova et al., 2010), phishing, malicious web advertising (Li et al., 2012; Thomas et al., 2015), and stealing user information (Huang et al., 2010).

In our attacker model, the malicious script launches superfluous JavaScript injection to force users to continuously send out requests to other websites to DDoS (Section 2.1) or visit advertisements pages (Section 2.2).

2.1 Browser-based DDoS

In contrast to general Distributed Denial of Service (DDoS), browser-based DDoS attacks leverage innocent browsers as bots to send thousands of HTTP requests to bombard the target server (Pellegrino et al., 2015). The attackers can easily harvest a large number of bots by choosing popular non-HTTPS websites to inject the malicious code. Figure 1 shows an example of a malicious script[5] that continuously trigger cross-origin requests to a victim server.

Figure 2 illustrates the attack steps of browser-based DDoS. When a user A visits a website `http://genuine.server.com` hosted on G, her browser sends an HTTP request to G. G returns a webpage referencing a script hosted on another server A_1. After getting the response from G, the browser subsequently sends a request to A_1 to load the script. A_1's response, however, is intercepted by a MitM attacker and replaced with a malicious one (e.g., the script shown in Figure 1). As a result, A's browser

[5]https://blog.cloudflare.com/an-introduction-to-javascript-based-ddos

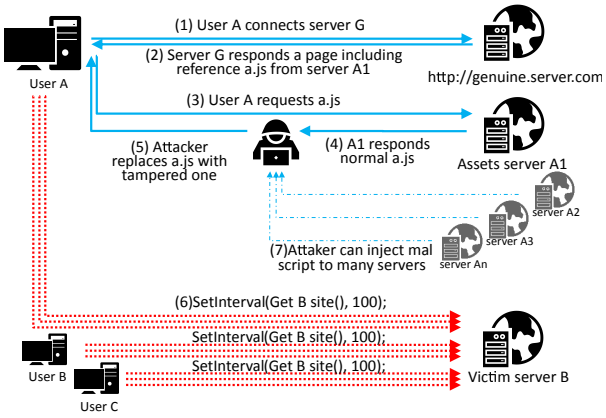

Figure 2: Browser-based DDoS.

loads the tampered script and triggers cross-origin HTTP requests to the victim server B continuously. Every user browsing `http://genuine.server.com` effectively launches a Denial-of-Service (DoS) attack on the server B. If the attacker injects the malicious script to multiple websites $A_1, A_2, A_3, \cdots, A_n$, thousands of innocent users will execute the code, resulting in a browser-based DDoS attack.

Web Browser-based DDoS. In March 2015, a new type of DDoS attack called the Great Cannon was observed (Marczak et al., 2015) targeting two GitHub pages. This attack was caused by an in-path system located at the edge between China's inner network and the Internet, and executed a MitM JavaScript injection for the targeted flow. The attack replaced a Baidu's analytics script with a malicious one that continuously issued requests to the two target pages. By injecting malicious JavaScript to sites loaded by enormous unwitting users, the Great Cannon caused massive HTTP requests per second to target pages and easily overwhelmed them.

Mobile Browser-based DDoS. In September 2015, a customer of Cloudflare was hit by browser-based DDoS attack that leveraged mobile users as DDoS vectors.[6,7] When a user was surfing the web using a mobile device, the browser app injected an iframe that inserted an advertisement in the browser app. Hence, the user was forced to visit the advertisement, and the advertisement forwarded the victim to the attack page. When the malicious JavaScript code in the attack page was executed, it flooded the servers that belong to the victim. This attack event utilized mobile devices as the distributed attack vectors and created distributed flows, causing 275,000 HTTP requests per second during the peak time. Grossmann and Johansen also presented how attackers injected malicious advertisements to launch DDoS (Grossman and Johansen, 2013).

2.2 Advertisements Injection

Several Internet Service Providers (ISPs) were identified to inject advertisements into customer-facing webpages.[8,9,10] These ISPs continuously monitor customer traffic and inject advertisement-incurring JavaScript on webpages that their customers visit *without their consents*. In this case, customers are compelled to receive unexpected information and website owners cannot assure that the content seen by users is unmodified. Such an attack is not only obtrusive, but can cause significant problems with normal Web application functions and network performance. For example, TELCOM—the biggest telecommunication company in Indonesia—was found to secretly inject advertisements to almost every unencrypted webpage that their customers visited; TELKOM sniffed traffic between the client browser and the website owner and injected JavaScript to trigger requests to its advertisement server. Such an injection attack resulted in two undesirable consequences: customers faced unwanted advertisements on the webpages, and such advertisements increased the page loading time, incurring additional usage fees.

In addition to ISPs, some browser extensions are found to be linked to ad-networks by executing JavaScript-based extension.[11] Users installed a highly-rated browser extension to download videos from YouTube. Unfortunately, an additional video appeared on the user's webpage and started playing while loading the page.

3 SYSTEM ARCHITECTURE

FALCO detects superfluous JS injection that inflates network activities based on behavior fingerprints, which capture a website's dependency to its external domains.

[6]https://blog.cloudflare.com/mobile-ad-networks-as-ddos-vectors/

[7]https://threatpost.com/javascript-ddos-attack-peaks-at-275000-requests-per-second/114828/

[8]https://www.privateinternetaccess.com/blog/2016/12/comcast-still-uses-mitm-javascript-injection-serve-unwanted-ads-messages

[9]https://zmhenkel.blogspot.tw/2013/03/isp-advertisement-injection-cma.html

[10]https://medium.com/@grumpyuser/telkom-indonesia-secretly-injects-advertisements-a3bf10b447ee

[11]https://staging.hanselman.com/blog/can-you-trust-your-browser-extensions-exploring-an-adinjecting-chrome-extension

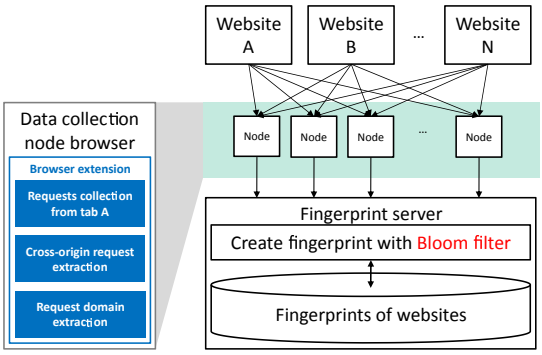

Figure 3: System architecture of data collection phase.

FALCO consists of three phases: data collection (Section 3.1), fingerprint management (Section 3.2), and attack detection (Section 3.3).

Overview. Figure 3 shows an overview of the FALCO system. In the data collection phase, *data collection nodes* collect HTTP requests when loading websites and send the collected data to a *fingerprint server*. In the fingerprint management phase, the fingerprint server extracts, stores, and updates website fingerprints on behalf of clients. Fingerprints are represented compactly using Bloom filters or their variants. In the attack detection phase, the client's *browser extension* downloads fingerprints from the fingerprint servers, and compares the fingerprints with the client's local observation. Once detecting suspicious fingerprint changes, the browser extension will alert the user.

To ensure privacy, a client can run its local version of data collection node and fingerprint server. The client can also retrieve fingerprints via privacy-preserving manners, as described in Section 6.1.

3.1 Data Collection

During the data collection phase, we assume that the data collection nodes and websites are not under attack.

A data collection node collects raw HTTP requests when loading each website of interest, and strips off privacy-sensitive information. Only the domains and their occurrences are sent to the fingerprint server, for the following reasons.

Privacy Considerations. Since an HTTP-request URL can leaks clients' sensitive information such as personal accounts, session ID, or cookies, we consider privacy as a fundamental design parameter as follows: the data collection nodes extract the domains from HTTP-request URLs before they are delivered to the fingerprint server. Consequently, the fingerprint server receives the connection data without any private information and uses it to create the fingerprint.

For example, National Taiwan University's website launches 61 requests, only two of which are cross-origin requests. These two cross-origin requests contain the user's location or environment information in the parameters.[12] To mitigate this privacy concern, we focus only on the URL domain of the cross-origin requests, truncating the request and obtaining the domain (e.g., "*stats.g.doubleclick.net*") for the fingerprint.

Fingerprint Robustness. Ideally, a website's behavior fingerprint should be robust against factors such as location and time. We conducted small-scale experiments to observe the impact of location and time. Specifically, we investigated the HTTP requests for launching *http://www.mit.edu* from the US, Japan, and Taiwan, and Figure 4 summarizes the triggered time for each request. This figure can be considered as an approximate shape for all the fingerprints from different locations. Even though the time duration for each HTTP request is not always the same across different locations, we can still infer that the additional HTTP requests preserve the order.

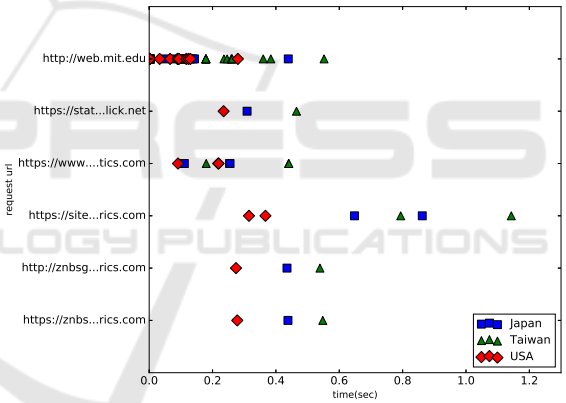

Figure 4: Fingerprints of mit.edu from USA, Japan, and Taiwan.

In addition to the location, we speculated that the website browsing time may affect the fingerprint. Indeed, we observed that the fingerprints collected at different times were not always the same, due to network conditions (e.g., speed, server status, processing speed, etc.) at different times. To confirm this conjecture, we accessed *http://www.ntu.edu.tw* once every hour within a day with the same client environment in Taiwan, and the connection results are shown in Figure 5.

Based on the small-scale experiments, we decided to take into account the frequency of requests but not

[12]For instance, one was sent to Google Analytics (*stats.g.doubleclick.net*) and contained parameters of screen resolution, browser language, etc.

the precise timestamps or sequence of requests for generating a fingerprint. In addition, to reduce the variance introduced by location-dependent requests, we recommend deploying data collection nodes at different locations across the world.

3.2 Fingerprint Management

The fingerprint server is responsible for extracting, storing, and updating fingerprints. For each website, the fingerprint server receives a list of accessed domains and their frequencies from the data collection nodes. The fingerprint server then extracts and encodes important information into a Bloom filter as a fingerprint. We will explain why using Bloom-filter-like data structures shortly. The fingerprint server stores fingerprints of the popular websites collected from different locations, updates fingerprints when necessary, and provides them when users query.

Fingerprint Construction. A straightforward approach is using the list of accessed domains and their frequencies as a website's fingerprint. When visiting a website, the client will compare its domain access patterns to the website's fingerprint. If the client observes a new domain that is not on the list, or accesses a domain more frequently than it was documented in the fingerprint, the client reports detection of spurious requests, which indicates a potential superfluous JS injection attack. However, this straightforward approach is inefficient (in terms of storage cost on the fingerprint server and the client, as well as the computation cost on the client side) and leaks unnecessary information to whoever queries the fingerprint server.

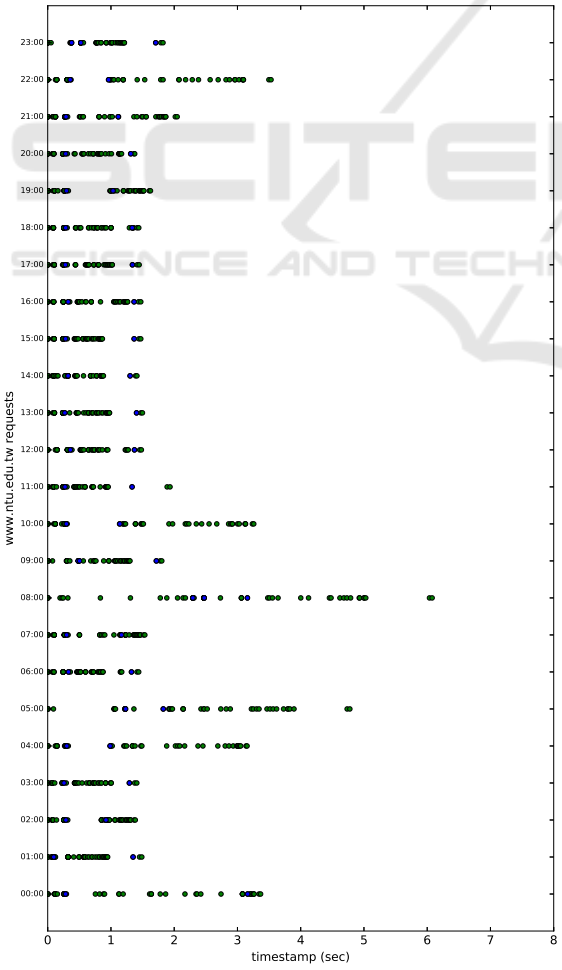

Figure 5: Fingerprints of ntu.edu.tw.

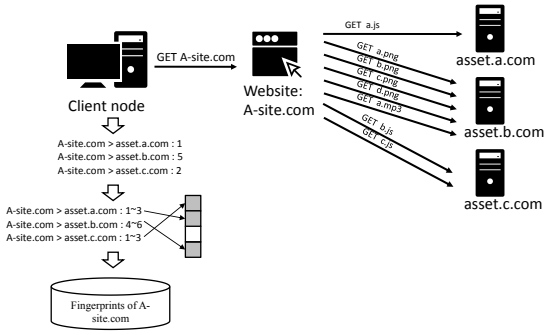

Figure 6: Obtain fingerprint from requests using Bloom filter.

To improve efficiency and privacy, FALCO uses a Bloom filter or a counting Bloom filter to represent a website's fingerprint. At a high level, a Bloom filter is a compact data structure that supports efficient membership queries. Thus, given a Bloom filter that contains the set of accessed domains, one can easily determine whether a new domain is visited by querying the Bloom filter. After a brief introduction of Bloom filters, we describe three fingerprint construction methods that utilize Bloom filters and discuss their applicability in Section 3.2.1.

Bloom Filters & Counting Bloom Filters. A Bloom filter is a data structure that uses a bit array for set membership queries. This space-efficient structure offers operations, such as adding elements to the set and querying if the element exists in the set. Bloom filters might claim an element is one of the set when it does not belong to the set, but never reports an existing element to be absent from the set.

An empty Bloom filter is a bit array of n bits, representing a set of n elements with initial value 0 as Figure 7(a) shows. When an element x from the set $S = \{x_1, x_2, \cdots, x_n\}$ is inserted, the Bloom filter uses k independent hash functions to calculate the position $h_1(x), h_2(x), \cdots, h_k(x)$ in the array and set these bits to 1 as shown in Figure 7(b). To determine if y exists in this set S, we only need to check the k hash functions with y. If any value in $h_1(y), h_2(y), \cdots, h_k(y)$ is 0, then y does not belong to the set S (Figure 7(c)), and

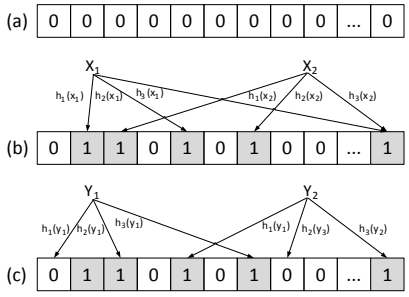

Figure 7: Bloom filter operation.

when there are more 1s in the array, the false positive is higher.

An enhanced version called a counting Bloom filter uses an array of n counters to replace the n bits in a Bloom filter. The counters show the number of inserted elements of the hash values that are indexed at each position. To confirm that an element is in the set, we can check if the values in the hash-indexed positions of the element are all above zeros.

In summary, Bloom filters provide a fast membership queries operation with space-efficient feature. Although the length of the array and the number of hash functions could directly affect the false positive rate, we can reduce it by extending the array of the Bloom filter. Because of the benefits of space efficiency and querying speed, we use the Bloom filter to create a fingerprint of the cross-origin domains for each website in our system architecture.

3.2.1 Fingerprint Extraction Methods

We propose three fingerprint construction methods as shown in Figure 8. They provide different levels of efficiency and accuracy depending on the information encoded in Bloom filters.

Method I: Store the Visited Domains in Bloom Filters. To generate a fingerprint of a website, Method I inserts the domains of the website's cross-origin HTTP requests to the Bloom filter, *without* considering the timestamp and the frequency of each domain.

A superfluous JS injection attack is detected if the client browser observes a new domain that does not appear in the fingerprint (i.e., the Bloom filter returns false). Figure 8 (a) shows an example using Method I.

Method II - Store the Pairs of Visited Domain and its Frequency in Bloom Filters. As shown in Figure 8 (b), compared with Method I, Method II additionally considers the frequency of each domain in fingerprint generation. For each cross-origin HTTP request, Method II inserts the pair of its domain and frequency to the Bloom filter. The frequency is quantized into coarse-grained ranges to reduce false detection. For example, if a website launched requests to domain `a.com` 1 time, to domain `b.com` 5 times and to `c.com` 2 times, Method II with a range of three will insert three elements: `a.com`:[1-3], `b.com`:[4-6], `c.com`:[1-3].

To detect whether a visited website is under a superfluous JS injection attack, a client browser accumulates and computes the frequency of each cross-origin domain. The client browser then check whether every pair of `domain:range` is in the Bloom filter.

Method III - Store the Pairs of Visited Domain and its Frequency in Counting Bloom Filters. Similar to Method II, Method III also considers the frequency of each domain in fingerprint generation. Instead of using a Bloom filter to encode the pair of domain and frequency range, Method III uses a *counting Bloom filter*, in which each element is naturally associated with a count. Method III uses the counter in the counting Bloom filter to record the frequency of each domain.

To detect whether a visited website is under a superfluous JS injection attack, a client browser accumulates and computes the frequency of each cross-origin domain, as in Method II. The client browser then checks whether the frequency of each domain is smaller than or equal to what the counting Bloom filter returns. An attack is detected if any of the domain has a frequency higher than the return value.

3.2.2 Website Classification and Method Selection

We observe that each fingerprint construction method presented in Section 3.2.1 is suitable for different types of websites. To improve detection accuracy, we propose heuristics for website classification and method selection.

Website Classification. We divide websites into four types according to their dynamics in the visited domains and the number of requests. We assume a fingerprint server can determine the type of a website after seeing several rounds of data provided by the data collection nodes. As the classification should be relatively stable, popular websites (e.g., the top one million) can also be classified and labelled in advance.

The first category consists of websites that load a fixed number of requests and visit identical domains every time. These websites often contain no media or advertisement. An example of such websites is Wikipedia.

The second category consists of websites that issue requests to different domains over time, such as websites embedding real-time bidding advertisements. In this case, the JavaScript first queries an advertisement network, and then the advertisement net-

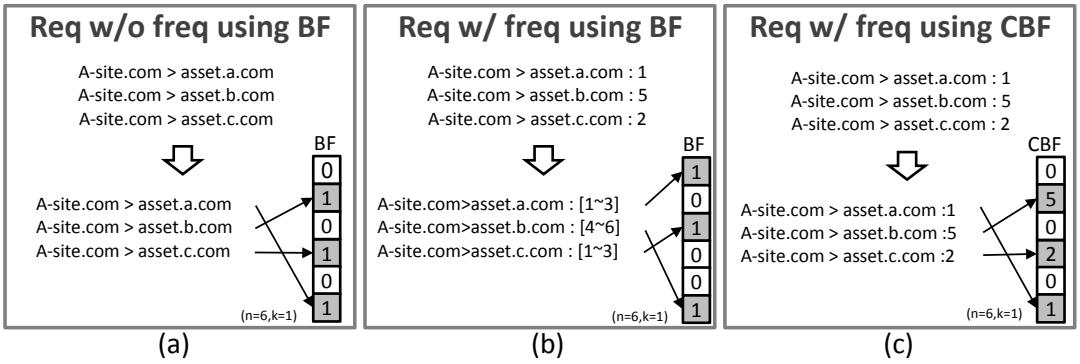

Figure 8: Examples of three fingerprint construction methods.

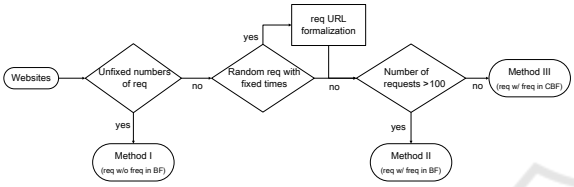

Figure 9: A decision diagram for selecting a fingerprint construction method based on the website type.

work asks the browser to contact different advertisers. The Yahoo website belongs to this type. To reduce false positive during detection, for this type of websites, the data collection nodes should access the websites for multiple times, ensuring that the website's fingerprint captures most of the normal web activities.

The third category consists of websites that issue requests to a fixed set of domains but the number of requests may vary over time. For example, a website embedding autoplay videos (e.g., Disney) will instruct the browser to send periodic requests in order to play the videos. For this type of websites, we select Method I to collect the fingerprint.

The last category consists of websites that issue a fixed number of requests, but the destinations of the requests may change over time. An example is a website that relies on a CDN service: A website publisher hosts the website assets on CDN servers; when a user wants to visit the website, he or she will be directed to one of the CDN servers based on location, load, price, etc. Similar to the second category, for this type of websites, the data collection nodes should access the websites for multiple times. Because the number of requests stay the same, we use Method II or III to create fingerprints.

Selection of Fingerprint Construction Method. Figure 9 shows the decision diagram for selecting a fingerprint construction method based on the website type. We first check whether the website falls in the first category. If the website does not have a fixed number of requests, we use Method I to create the fingerprint. Then we check if the website belongs to the second category or not. If yes, the requests will be formalized before adding to the fingerprint. Additionally, if the number of requests is more than 100, we choose Method II for space saving. The rest will be handled by Method III.

3.3 Attack Detection

Given the website type and its fingerprint of a website, the FALCO browser extension on the client side can detect whether the website is under a superfluous JS injection when the user browses the website.

The client-side detection is performed through the following procedure.

1. *Install* - The user installs the FALCO browser extension.
2. *Fetch* - When the user accesses a website using the browser, the detector (browser extension) will first fetch the website's fingerprint from the fingerprint server. The user can also fetch the fingerprint of the website in advance.
3. *Detect* - All the cross-origin requests initiated by the website will be checked by the detector before they are sent out.
4. *Block* - If a superfluous request is detected (i.e., it does not exist in the fingerprint or the number of requests to the same domain exceeds the record in the fingerprint), the request will be blocked. The user can decide whether to add the request to a blacklist.

4 EVALUATION

To generate a fingerprint of a website, we load the website for 20 seconds[13] and collect its cross-origin requests. We assume that the website is not under attack during data collection. Data collection nodes and a fingerprint server are hosted using AWS, and the data is collected from USA, Japan, and Taiwan.

4.1 Fingerprint Generation

We investigate the robustness of the fingerprint generation methods presented in Section 3.2.1 across different locations. In the first part of the experiment, we collect 3,000 websites fingerprints from the US using AWS WorkSpace and set up a client browser in Taiwan. In the second part of the experiment, we collect 3,000 websites fingerprints and set up a client browser in Taiwan to test the fingerprints from the same location. The results are shown in Table 1.

Table 1: True positive detection rate using fingerprint.

	TP in different location	TP in same location
Req w/o freq in BF	90.58%	96.68%
Req w/ freq in BF	85.28%	87.48%
Req w/ freq in CBF	82.42%	88.24%

A true positive represents a consistent detection in different locations. The parameters of the Bloom filters are $n = 500$ and $k = 3$, and the parameters of the counting Bloom filters are $n = 1000$, $k = 3$. The results show that more than 82.42% websites have the same fingerprints in different locations, and more than 87.48% websites have the same fingerprints in the same location. The true positive rate increases as the size of the Bloom filter increases. As the size of Bloom filter increases by four times, the true positive rate approaches 100%.

According to our classification in Section 3.2.2, we divide the websites into four categories, and create fingerprints using different methods, as shown in the procedure of Figure 9.

As shown in Table 2, by performing the classification, the true positive rate of the two-location experiment increases to 93.08%, where 6.97%, 42.97%, and 50.06% of the websites use method I, II, and III, respectively, to create the fingerprints. Similarly, the true positive rate of the same-location experiment increases to 98.36%, where 12.5%, 39.4%, and 48.1% of the websites are assigned to method I, II, and III, respectively.

[13]We performed a small-scale measurement of the top 100 websites and observed that 58% of them completed loading in 10 seconds and 87% in 20 seconds. Thus, we use 20 seconds as a threshold.

Table 2: True positive detection rate using fingerprint with websites classification.

	Different locations	Same location
Method I (BF w/o times)	6.97%	12.5%
Method II (BF w/ times)	42.97%	39.4%
Method III (CBF w/o times)	50.06%	48.1%
Avg TP	93.08%	98.36%

Table 3: False positive rate under attack.

	FP when under attack
Req w/o freq in BF	3.02%
Req w/ freq in BF	0.57%
Req w/ freq in CBF	0.07%

4.2 Attack Detection

We simulate two types of attacks and evaluate the detection rate using FALCO.

4.2.1 Attack Simulation I - Ad Injection

We first simulate the ad-injection attack introduced in Section 2.2. When a website is under ad-injection, it will send out extra cross-origin requests to ad domains. Therefore, in this attack simulation, we insert arbitrary cross-origin HTTP requests into 3,000 request lists from different websites, and detect the extra traffic by fingerprints. The result of this experiment is shown in Table 3. A false positive means that the request is not in the fingerprint but is detected. The false positive for FALCO is less than 3.02%. The results also confirm that the larger the fingerprint size is, the more accurate the result is, as we can see that the size of Bloom filter and the numbers of hash functions impact the false positive of fingerprints.

4.2.2 Attack Simulation II - Browser-based DDoS

The second attack we simulated is browser-based DDoS attack. As shown in Figure 10, we use mitmproxy[14] to MitM attack our browser in the simulation environment. When the client browses website A-site.com (http://www.ntu.edu.tw) and gets the example.js from A-site.com, the mitmproxy replaces the example.js to mal.js. This mal.js attempts to send a request to the victim server continuously to perform an HTTP flooding attack.

Figure 11 shows the collected fingerprints from multiple countries. The purple dots represent the

[14]https://mitmproxy.org/

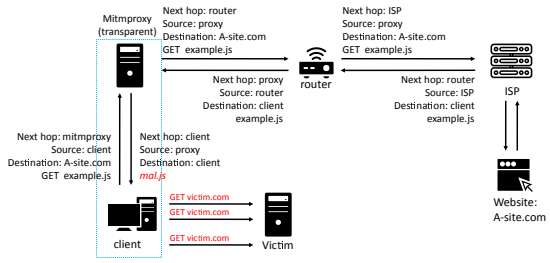

Figure 10: Attack environment simulation.

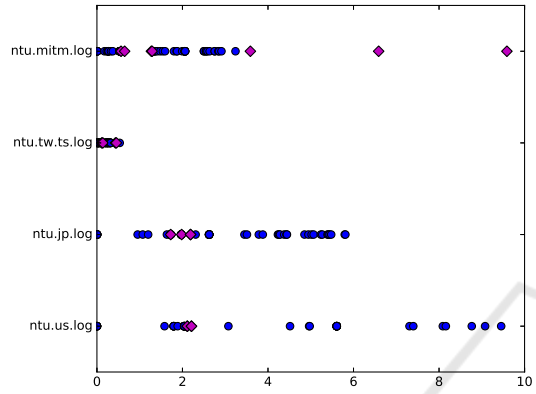

Figure 11: Fingerprint of MitM attack simulation.

cross-origin requests, and the blue spots represent the same-origin requests. The first log on top shows the fingerprint when the browser is under attack, the second one is the fingerprint from Taiwan, the third one is from Japan and the last is from USA. As our results confirm, FALCO can detect all of the simulated attacks.

4.3 Performance Evaluation

In this section, we compare the latency between a browser without the extension and a browser which installs the extension we implemented. The comparison results are shown in Figure 12. We tested top 10 websites from *majestic.com*. Each bar displays the loading time with dark color, and the finish loading time with light color. Pink bars show the latency of visiting websites using the browser without the extension, and blue bars show the latency of visiting websites using the browser with the extension. The average latency of loading one website is 0.29 seconds. The additional latency introduced by the browser extension may result from the number of cross-origin requests, the performance of the web server, or the client network environments.

5 RELATED WORK

In this section, we describe existing defense techniques and their limitations.

Same-Origin Policy (SOP). The same-origin policy[15] is a web security policy enforced by modern browsers. In principle, the same-origin policy restricts scripts in one page from accessing another page unless they belong to the same origin. SOP can prevent scripts from accessing Document Object Model (DOM), cookies, and local storage data belonging to another web origin, but it cannot block cross-origin requests from HTML tags such as ⟨img⟩, ⟨script⟩ and ⟨iframe⟩, because the requests triggered by HTML tags inherit the origin of the main page. Since browsers can still be deluded with sending malicious requests, the same-origin policy cannot prevent or detect superfluous JS injection.

Cross Origin Request Policy (CORP). CORP (Telikicherla et al., 2014; Agrawall et al., 2017) is a browser security policy against browser based DDoS attacks. It enables the server to control cross-origin requests from the browser and block illegal requests by the server. For example, with CORP response header, site alice.com can block unauthorized requests from site bob.com. As a browser receives the response with CORP header, it records the policy in the browser cache when encountered for the first time. When further requests are triggered, the browser will enforce the CORP rule on the request. As mentioned above, executing CORP mechanism involves both the server and browser. For example, when only the server-side supports CORP, the browser can still send illegal cross-origin requests to the server. Moreover, besides the difficulties in configuring policies, CORP has not been adopted in practice.

Content Integrity. SRI[3] guarantees the integrity of resource by checking its hash value. More specifically, users compute the hash value of a resource and declare it alone with the resource path. Upon fetching the resource, a browser can compute the hash and verify the integrity of resource. Although SRI can ensure that the JavaScript is not tampered by an attacker, it is not flexible for either a dynamic webpage or updating the JavaScript. Furthermore, if attackers act as man-in-the-middle, it is easy to replace both the hash value and the resource at the same time.

Stickler (Levy et al., 2016) provides a service that supports a website publisher to place its static resource, such as JavaScript, CSS, images, or other media files, around the world via CDN network. By using CDN, website owners can efficiently deliver

[15] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

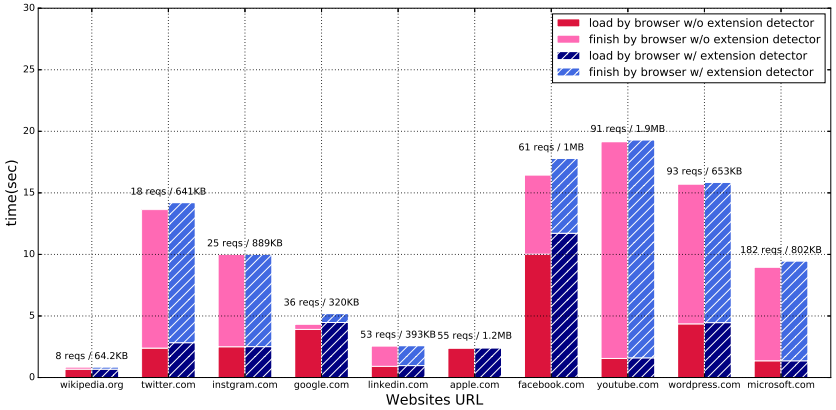

Figure 12: Average latency of browsing websites by fingerprint extension.

their content and reduce bandwidth costs, under the premise that they trust CDN unconditionally. Stickler provides a system that guarantees that the website asset would not be tampered by the trustless CDN service, by allowing a website publisher to digitally-sign all their assets on CDN. Users who want to access the website will first fetch the bootloader from website's server, in which case the bootloader downloads the signed assets from CDN and verified them. According to the digital signature and the operation of the bootloader, Stickler can ensure the integrity of the website content from CDN.

Client-side Protection. Prior techniques often require server-client collaboration, therefore further elevating the deployment challenges. Excision (Arshad et al., 2016) presents an in-browser solution to detect malicious JavaScript inclusion. However, Excision's browser instrumentation increases the browsing time by 12.2%.

Comparison We compare FALCO with existing defense mechanisms introduced in Section 5 with respect to the deployment requirement, policy setting, detection indicator, and threat model. The results are summarized in Table 4.

For the deployment requirement, FALCO and Excision only need to modify the client side and does not require server collaboration, whereas the others require participation of both the client and server. For the policy setting, mechanisms such as CORP are only effective if the web administrators can correctly configure access policies to block unwanted traffic; FALCO does not require any complex setting. FALCO is more lightweight than Excision, because FALCO focuses on superfluous JS injection, which incurs observable inflation in network traffic, whereas Excision aims to detect generic content injection.

6 DISCUSSION

6.1 Privacy Concern

In FALCO, the client-side's detection relies on the fingerprint provided by the fingerprint server. Therefore, the fingerprint server must know which websites the users attempt to browse. We briefly describe possible approaches to prevent the fingerprint server from explicitly recording the browsing history. The first approach is prediction; users can *regularly* download a group of fingerprints in advance, which they may visit in the future. The second approach is by increasing the anonymity set; users can download the fingerprint of the website which they want to browse with other random fingerprints at the same time. The third approach is by maintaining a local fingerprint server with a group of users; the users in the group fetch the fingerprint through this server, then the centralized fingerprint server will know that this group attempts to view some websites but never learns about the specific user. The last approach is adapting existing private information retrieval techniques (Chor et al., 1995) to acquire the website's fingerprint without revealing its browsing history.

6.2 Limitations and Challenges

There are several limitations of FALCO. To detect potential abnormal traffic sent from the client browser, users have to install a browser extension in their browsers and the extension will generate and compare fingerprints while browsing, which may affect the browsing performance. Also, we assume that a website's fingerprint is computed when the website is not under attack, which may not always hold in practice.

Table 4: Comparison of FALCO with related work.

	Deployment	Policy Setting	Detection Basis	Threat Model
FALCO	browser only	N/A	external dependency	superfluous JS injection
SOP	browser & web server	N/A	requests header	cross-origin request
CORP	browser & web server	CORP rule	request header	JavaScript-based DDoS
SRI	browser & web server	N/A	hash value of script	tampered JavaScript
Stickler	browser & web server	N/A	digital signature	malicious CDN
Excision	browser only	N/A	inclusion sequences	3rd-party content inclusions

Some challenges remain as well. First, further investigation is needed to determine the effect of location, browsing time, network environments, browser versions, etc. Second, as web applications today are highly personalized, it would be interesting to explore whether the fingerprints should be personalized and how to efficiently and effectively create personalized fingerprints. Third, it is worth investigating the incentive model of adopting defense mechanisms against superfluous JS injection attacks, as users may not have enough incentive to install an extension just for avoiding their browsers from being leveraged to attack others. Instead, ISPs may have strong motivation to defend against such attacks, but there are technical challenges to identify unwanted requests trigger by each website from the ISPs' perspectives.

7 CONCLUSION AND FUTURE WORK

This paper presents a robust detection system called FALCO against superfluous JavaScript injection attacks using website fingerprints without server-side cooperation. To analyze normal website behavior, we extracted the fingerprints of the top 10,000 websites, and our evaluation results confirm that FALCO can detect 96.68% superfluous JS injection attacks in our simulation environment.

As for the future work, we are studying how to improve FALCO as follows. (1) To expand the deployment of our system, we have to improve the transparency of our system, such as providing our service under a third-party certification authority to improve users' willingness to use FALCO. (2) On the client-side, a personal fingerprint from the user's browsing history is better than a fingerprint from others. We plan to develop a local fingerprint system in which a fingerprint can be extracted for each connection from the user's browser and detect attacks based on his/her own local observations. (3) Many dynamic factors may impact the accuracy of fingerprints such as the location, browsing time, network environment and the browser version. Websites continuously send out requests (e.g., website analytic services, websites with session-replay scripts).[16] We plan to focus on investigating such dynamic factors to improve the accuracy. (4) We plan to enhance the website classification approach by websites' characteristics.

ACKNOWLEDGEMENTS

This research was supported in part by the Ministry of Science and Technology of Taiwan (MOST 109-2636-E-002-021).

REFERENCES

Agrawall, A., Chaitanya, K., Agrawal, A. K., and Choppella, V. (2017). Mitigating browser-based ddos attacks using corp. In *Proceedings of the 10th Innovations in Software Engineering Conference*, pages 137–146. ACM.

Arshad, S., Kharraz, A., and Robertson, W. (2016). Include me out: In-browser detection of malicious third-party content inclusions. In *International Conference on Financial Cryptography and Data Security*, pages 441–459. Springer.

Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. (1995). Private information retrieval. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*, pages 41–50.

Cova, M., Kruegel, C., and Vigna, G. (2010). Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, pages 281–290. ACM.

Grossman, J. and Johansen, M. (2013). Million browser botnet. Black Hat USA.

Huang, L.-S., Weinberg, Z., Evans, C., and Jackson, C. (2010). Protecting browsers from cross-origin css attacks. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 619–629. ACM.

Levy, A., Corrigan-Gibbs, H., and Boneh, D. (2016). Stickler: Defending against malicious content distribution networks in an unmodified browser. *IEEE Security & Privacy*, 14(2):22–28.

[16] https://freedom-to-tinker.com/2017/11/15/no-boundaries-exfiltration-of-personal-data-by-session-replay-scripts/

Li, Z., Zhang, K., Xie, Y., Yu, F., and Wang, X. (2012). Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 674–686. ACM.

Marczak, B., Weaver, N., Dalek, J., Ensafi, R., Fifield, D., McKune, S., Rey, A., Scott-Railton, J., Deibert, R., and Paxson, V. (2015). China's great cannon. *Citizen Lab*, 10.

Pellegrino, G., Rossow, C., Ryba, F. J., Schmidt, T. C., and Wählisch, M. (2015). Cashing out the great cannon? on browser-based ddos attacks and economics. In *WOOT*.

Telikicherla, K. C., Choppella, V., and Bezawada, B. (2014). Corp: a browser policy to mitigate web infiltration attacks. In *International Conference on Information Systems Security*, pages 277–297. Springer.

Thomas, K., Bursztein, E., Grier, C., Ho, G., Jagpal, N., Kapravelos, A., McCoy, D., Nappa, A., Paxson, V., Pearce, P., et al. (2015). Ad injection at scale: Assessing deceptive advertisement modifications. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 151–167. IEEE.

Weichselbaum, L., Spagnuolo, M., Lekies, S., and Janc, A. (2016). Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1376–1387.