On the Integrity of Cross-Origin JavaScripts

Jukka Ruohonen, Joonas Salovaara, Ville Leppänen

On the Integrity of Cross-Origin JavaScripts

Jukka Ruohonen[0000−0001−5147−3084], Joonas Salovaara, and Ville Leppänen
{juanruo, joosal, ville.leppanen}@utu.fi

Department of Future Technologies, University of Turku, Finland

Abstract. The same-origin policy is a fundamental part of the Web. Despite the restrictions imposed by the policy, embedding of third-party JavaScript code is allowed and commonly used. Nothing is guaranteed about the integrity of such code. To tackle this deficiency, solutions such as the subresource integrity standard have been recently introduced. Given this background, this paper presents the first empirical study on the temporal integrity of cross-origin JavaScript code. According to the empirical results based on a ten day polling period of over 35 thousand scripts collected from popular websites, (i) temporal integrity changes are relatively common; (ii) the adoption of the subresource integrity standard is still in its infancy; and (iii) it is possible to statistically predict whether a temporal integrity change is likely to occur. With these results and the accompanying discussion, the paper contributes to the ongoing attempts to better understand security and privacy in the current Web.

Keywords: same-origin, cross-domain, remote inclusion, subresource integrity

1 Introduction

Most current websites load numerous resources from many distinct third-party sources. Among these resources is JavaScript that is executed by clients visiting the websites. There are many viewpoints to this execution of third-party code.

One relates to network protocols [21]. The *hypertext transfer protocol* (HTTP) over the transport layer security protocol (a.k.a. HTTPS) can only authenticate the server to which a client connects. It does not provide any guarantees about the authenticity of the encrypted content transmitted after the authentication. From this perspective, the authenticity (integrity) of web content has become a pressing concern as more and more content is transmitted through *content delivery networks* (CDNs) and cloud services, while at the same time legislations all over the world have seen amendments toward mass surveillance. Another viewpoint relates to privacy [26, 32]. In many respects, the execution of arbitrary third-party code is in the interests of those involved in the tracking of the Web's client-side. A further viewpoint relates to web security and web standards.

The execution of third-party JavaScript occurs in the same context as the execution of the primary code present in a website [4, 24]. To patch this limitation, the so-called *subresource integrity* standard has recently been introduced

for allowing enumeration of cryptographic hashes that clients verify before execution.[1] Although the standard is oddly restricted only to certain web elements [10], it is an important step toward at least some theoretical integrity guarantees. The standard and associated considerations are also adopted as frames for this paper. In particular, the paper's motivation builds on the practical challenges facing the widespread adoption of the standard. As will be elaborated, it is also these practical challenges through which the wider security and privacy viewpoints can be reflected. To these ends, the following contributions are made:

1. The paper presents the first empirical study on the temporal (data) integrity of cross-origin JavaScript code used and executed in the wild.
2. The paper shows that temporal integrity changes are relatively common on one hand and subresource integrity checks very uncommon on the other.
3. The paper demonstrates that a limited set of information can be used to predict whether cross-origin JavaScript code is likely to change temporally.

The remainder of the paper is structured into four straightforward sections. Namely: Section 2 discusses the background, Section 3 introduces the dataset, Section 4 presents the results, and Section 5 concludes with a few remarks.

2 Background

In what follows, the rationale for the empirical study is motivated by briefly discussing the background related to remote cross-origin JavaScript inclusions.

2.1 The Same-Origin Policy

The *same-origin policy* (SOP) is a fundamental part of the Web. It governs the ways elements in a *hypertext markup language* (HTML) document can interact. An origin is defined as a tuple containing a scheme, a host name, and a port [2]. The tuple can be elaborated with the syntax for uniform resource identifiers:

$$\texttt{scheme://}\underbrace{\texttt{[user:password@]host:port}}_{\texttt{authority}}\texttt{/path?query\#fragment}, \quad (1)$$

where `host` is a fully qualified domain name or an Internet protocol address [3]. If and only if the `scheme`, the `host`, and the `port` fields are equal between two *uniform resource locators* (URLs), the two locators have the same origin. When this condition is not satisfied, the antonym term *cross-origin* is often used.

A couple of additional points are warranted about the syntax and its semantics. The first point is about the `scheme`. The inclusion of this protocol field is essential for isolating websites served via plain HTTP from those served via HTTPS [2]. The second point is about the `port` field: when it is missing,

[1] In this paper, the term *standard* includes also recommendations, guidelines, and working drafts that are well-recognized but not necessarily yet officially standardized.

the information is derived from the mandatory `scheme`. Thus, the same-origin condition holds for the two tuples within the following two example URLs:

$$\texttt{http://example.com/index.html} \overset{\text{SOP}}{\equiv} \texttt{http://example.com:port} \quad (2)$$

when either `port` $\equiv 80$ or the two URLs are queried with Internet Explorer, which disregards `port` when deducing about origins [14]. For this particular web browser, the tuples from (2) have the same origin for any `port` $\in [1, 65535]$.

The SOP is used for many functions explicitly or implicitly related to privilege separation [4, 12]. While these functions cover numerous web elements, the most important function is to restrict the execution of JavaScript by a web browser (refer to [37] for slightly outdated but still useful, extensive technical discussion). In essence, two same-origin documents have full access to each other's web resources. They can make HTTP requests to each other via JavaScript, they can manipulate each other's *document object model* (DOM) that acts as an interface between JavaScript and HTML, and they can even share information about cookies. Thus, without the SOP, a JavaScript running in one tab of a user's browser could do practically anything with the content in another tab.

Despite the SOP restrictions, *cross-origin embedding* is often allowed [14]. In particular, cross-origin requests are allowed for `<script>` tags equipped with the `src` attribute. When a client's browser encounters such a tag, it will issue a HTTP GET request to retrieve the content, which is executed immediately in the current origin's context. Although additional (security) information can be added to the GET requests via the `query` and `fragment` fields [17], the basic security issue thus is that the JavaScript response has full privileges within the requesting web page [11, 24]. In addition, the *cross-origin resource sharing* (CORS) standard [35] can be used to relax the SOP policy by whitelisting (in HTTP header responses) those hosts from which cross-origin resources can be loaded dynamically with JavaScript. Either way, the fundamental security risk remains identical: if the host behind a script's source is compromised, arbitrary code can be executed in the context of all websites having included the script. The risk is not only theoretical; a recent data breach allegedly involved a single line of misplaced JavaScript and a compromised third-party [33]. One way to analytically approach the risk is to consider the integrity of remote JavaScripts.

2.2 Integrity of Cross-Origin JavaScripts

There are numerous distinct aspects to the integrity of websites. For instance, at the DOM and HTML levels, integrity "ensures that the contents of an interaction cannot be modified without the knowledge of the interacting components" [11]. Another example would be the concept of web session integrity, which "ensures that an attacker can never force the browser into introducing unintended messages in sessions established with trusted websites, or into leaking the authentication credentials (cookies and passwords) associated to these sessions" [5]. When remote JavaScripts are included in a website, a prerequisite

for this notion of web session integrity is the integrity of the remote JavaScripts themselves. For this purpose, the traditional notion of data integrity is suitable.

Thus, let $h(\cdot)$ denote a cryptographic hash function and $\mathcal{H}$ a message digest outputted by the function for a given input. (For Internet measurement research, the first secure hash algorithm, or SHA-1, is sufficient and used also in the empirical part of this paper.) Let $\mathcal{J}$ further denote a unique cross-origin JavaScript that was included by a web page $\mathcal{P}$ with a `<script>` tag whose `src` attribute pointed to a unique uniform resource locator, $\mathcal{U}$. Consider then that this URL,

$$\mathcal{U} \in \mathcal{P} \quad \text{but} \quad \mathcal{U} \overset{\text{SOP}}{\not\equiv} \mathcal{P}, \tag{3}$$

was downloaded at time index t through

$$f(\mathcal{U}, t) = \begin{cases} \mathcal{J}_t & \text{if HTTP GET of } \mathcal{U} \text{ was succesful at time } t \text{ and} \\ \varnothing & \text{otherwise (including all network errors, etc.).} \end{cases} \tag{4}$$

By further defining $t = 1, 2, \ldots, T$ and

$$h(f(\mathcal{U}, t)) = \mathcal{H}_t \text{ if } f(\mathcal{U}, t) \not\equiv \varnothing \text{ and } \varnothing \text{ otherwise,} \tag{5}$$

the integrity of the remote $\mathcal{J}$ from the viewpoint of $\mathcal{P}$ can be evaluated by comparing the output from $h(\cdot)$ for two inputs $f(\mathcal{U}, t) \not\equiv \varnothing$ and $f(\mathcal{U}, t+k) \not\equiv \varnothing$ with $k \neq 0$ and $0 < t + k \leq T$. To a reasonable degree for empirical research purposes, the integrity has been intact if and only if $\mathcal{H}_t \equiv \mathcal{H}_{t+k}$ (see [16] for a formal exposition of the same temporal idea). In theory, an integrity violation might be defined to occur also when $h(f(\mathcal{U}, t)) \equiv \mathcal{H}_t$ but $h(f(\mathcal{U}, t+k)) \equiv \varnothing$. As the Internet is not entirely reliable in terms of content transmission via either HTTP or HTTPS, the weaker definition is used in the empirical analysis.

The use of cryptographic hashes has been adopted also for recent web standards, recommendations, and guidelines. Namely, the subresource integrity standard allows to specify one or more hashes for `<script>` and `<link>` tags with an additional `integrity` attribute [36]. When a client's browser retrieves the content referenced with these tags and a correctly specified `integrity` is present, it refuses to process the content in case the predefined hashes do not match the hash of the content retrieved. Furthermore, the *content security policy* (CSP) dialect can be used to instruct browsers to enforce the subresource integrity constraint for all external (non-inline) scripts [15]. In this case a browser refuses to execute a JavaScript either in case a valid `integrity` attribute is missing or the integrity check fails. Additional information can be passed to browsers by using the `crossorigin` attribute from the CORS standard to tell browsers that credentials (including cookies and certificates) are not transmitted through the tags equipped with the subresource integrity checks. By using the symbol $\mathcal{H}$ to again denote a message digest, $g(\cdot)$ a base64-encoding function, and referring to the earlier example in (2), the following excerpt can be used to illustrate the syntax of a subresource integrity check enforced for a cross-origin JavaScript:

```
<script src="https://example.com/javascript.js"                    (6)
 integrity="sha256-g(H)" crossorigin="anonymous"></script>
```

Thus, the basic idea is simple but not bulletproof. The standard mentions three potential weaknesses [36]. The first weakness is cryptographic: potential hash collisions undermine the foundations of all integrity checks done with a particular algorithm. The second weakness relates to transmissions: a malicious proxy can obviously strip the attributes in case plain HTTP is used or the context is otherwise insecure (see [9] for a survey of these man-in-the-middle scenarios). The third weakness originates from information leakages: it may be possible to deduce about the critical parts of a website protected by integrity checks. By repeatedly loading resources for which integrity checks are enforced, it may be possible to gain information about whether the content protected is static or dynamic. Particularly in case CORS is not used in conjunction with the integrity checks, these information leakages may allow an attacker to eventually guess authentication details [36], for instance. In addition to these three explicitly mentioned weaknesses, so-called browser cache poisoning may potentially circumvent the integrity checks [18]. As always, there may be also other already known or yet unknown weaknesses affecting the subresource integrity standard.

2.3 Practical Integrity Challenges

There are many practical challenges for widespread integrity checking of cross-origin scripts. Arguably, the cardinal challenge has never been the lack of technical solutions and standards, but rather the adoption of these solutions and standards among clients, servers, software producers, web developers, and numerous other actors involved. In terms of the standardized solutions, a major practical challenge relates particularly to web development practices and the manual work entailed in the implementation and enforcement of the solutions [29]. The pocket-sized analytical framework from the previous section can be used to exemplify three practical scenarios on how integrity may vary for cross-origin scripts.

First, two distinct websites $\mathcal{P} \not\equiv \mathcal{P}'$ may include a script with the same unique $\mathcal{U}$ pointing to the same unique $\mathcal{J}$ with the same unique $\mathcal{H}$. Second, it is possible that two unique URLs from two distinct websites point to the same unique JavaScript content, possibly at different times t and $t + k$. In this case

$$h(f(\mathcal{U}, t)) \equiv h(f(\mathcal{U}', t + k)) \equiv \mathcal{H}_t \tag{7}$$

holds for $\mathcal{U} \in \mathcal{P}$, $\mathcal{U}' \in \mathcal{P}'$, $\mathcal{U} \not\equiv \mathcal{U}'$, and $\mathcal{P} \not\equiv \mathcal{P}'$. These cases occur because web developers may copy JavaScripts from different Internet sources to their own websites. Furthermore, it is relatively common that a same small script is used in various parts of a website, such that (7) holds for $\mathcal{U} \in \mathcal{P}$, $\mathcal{U}' \in \mathcal{P}$, and $\mathcal{U} \not\equiv \mathcal{U}'$.

In other words, certain JavaScripts are included by many websites, scripts from one website may be plagiarized to other sites, and some JavaScripts are used in multiple parts of a single website. All three scenarios have security implications. Besides outright duplicates, the common occurrence of approximately highly similar JavaScript code (a.k.a. code clones) [7] is problematic because a cloned script may contain vulnerabilities. Cloned scripts are also unlikely to be rigorously maintained already due to the lack of strict references to the original sources for which vulnerabilities may be fixed by the original authors. The

problem is not only theoretical: recent Internet measurement studies indicate that many websites include vulnerable and outdated JavaScript libraries [22]. Regardless whether a vulnerable script is cloned or original, the potential attack vector also increases in case the script is used in multiple parts of a website.

The inclusion of certain cross-origin scripts by numerous websites raises the attack surface to an entirely different level. By compromising a popular CDN used to distribute JavaScript code, arbitrary code may be injected to thousands (or even millions) of websites, and this code may be executed by millions (or even billions) of clients. Even though this scenario has not fortunately realized, the theoretical possibility cannot be ruled out. On the other hand, the scenario has already realized on the side of privacy: "arbitrary code" is executed by billions of web clients due to the inclusion of cross-origin scripts by millions of websites. By assumption, it is also the third-party tracking infrastructures and web advertisements served via these infrastructures that make it difficult for web developers to enforce temporal integrity checks. In fact, it has been argued that web developers no longer even know who they are trusting with their remote JavaScript inclusions [21, 31]. Given this motivation, the remainder of this paper focuses on the question of how common temporal integrity changes are in reality.

3 Data

In what follows, the dataset is elaborated by discussing the sampling and polling routines used to assemble the dataset and the measurement framework for it.

3.1 Sampling

By following common research approaches for retrieving JavaScript code [20, 26], the initial collection of JavaScripts was done by sampling ten thousand unique second-level domain names from a ranking list made available by Cisco [8]. It is worth remarking that Cisco's lists have been used also previously [25] as an alternative to Alexa's lists, which are no longer available free of charge. More importantly, each domain in the list was transformed to a second-level domain name. This transformation is justified because the ranks are based on the volume of *domain name system* (DNS) traffic passing through Cisco's (OpenDNS) servers. For this reason, the list contains separate entries for example for `microsoft.com` and its subdomains such as `data.microsoft.com` and `ipv6.microsoft.com`.

Five further remarks are required about the sampling. To begin with, (a) each domain was queried with the `http` scheme. Thus, Microsoft's main domain was queried by passing `http://microsoft.com` to a browser, for instance. That said, (b) it is important to emphasize that redirections were followed for all queries. In terms of the running example, the URL requested was actually redirected to a location `https://www.microsoft.com/fi-fi/`. These redirections involved both the DNS and the HTTP protocol; typically, HTTP redirections upgraded the `http` scheme requested to HTTPS connections, while either HTTP or DNS redirections occurred to the subdomains (such as the `www`-prefixed ones) of the

requested second-level domains. The accounting of these redirections is essential for deducing about cross-origin scripts. In addition: when dealing with mostly dynamic content in the contemporary Web, a JavaScript-capable browser is required particularly for executing inline JavaScripts, which may interfere with the execution of external scripts [26]. Therefore, (c) a custom WebKit/Qt-powered headless browser was used with JavaScript enabled for all queries. By again following common practices [30, 31], (d) a 30 second timeout was used for all queries to ensure that the majority of scripts were successfully executed. Finally, (e) all domains were queried two times in order to account temporary failures.

3.2 Polling

The domains sampled were used to construct a pool for temporal integrity polling. To construct the pool, all scripts were collected from each domain sampled, but only cross-origin scripts were qualified to the polling pool. All cross-origin comparisons were done based on the visited URLs (and not the requested ones), which were used also for transforming relative URLs to absolute ones. Thus, an initial redirection to a HTTPS connection or a subdomain did not qualify an entry to the pool. During the construction of the pool, each cross-origin $\mathcal{U}$ in `<script src="`$\mathcal{U}$`">` was downloaded via a GET request. After observing that many prior test requests failed when the `query` and related fields were stripped, all downloads were made with the exact same URLs used in the websites sampled. Although no extensive attempts were made to verify that a given URL actually pointed to a valid JavaScript, `Content-Type` and related HTTP header fields were recorded for the initial downloads. In addition, each download was passed through a program for checking the *multipurpose Internet mail extension* (MIME) type. Finally, the downloads that returned non-empty buffers with a HTTP status code 200 were used to construct the polling pool, $[\mathcal{U}_1, \ldots, \mathcal{U}_{35417}]$. The URLs within the pool were then polled with HTTP GET requests consecutively for $T = 10$ days starting from March 23, 2018.

The empirical focus is on the following representation of the polling pool:

$$[h(f(\mathcal{U}, t)), h(f(\mathcal{U}, t+1)), \ldots, h(f(\mathcal{U}, t+T-1))]_i, \quad i \in [1, 35417], \tag{8}$$

where $\mathcal{U}$ satisfies (3) with respects to a sampled $\mathcal{P}$. To account for temporary transmission errors, domain name resolution failures, and other related networking shortages, the cases marked with a symbol $\varnothing$ in (4) were first removed from each vector. After this removal, each vector was transformed to a set, such that only unique hashes are observed for each script. (None of the transformations resulted a $\emptyset$, which would indicate that all polls would have failed.) This simple operationalization provides a straightforward way to observe the temporal integrity of cross-origin scripts: if $H_{\mathcal{U}}$ denotes a set of unique hashes, the temporal integrity of a given script residing at $\mathcal{U}$ was intact during the polling period if and only if $|H_{\mathcal{U}}| = 1$. Although the polling period allows to only observe a rather short time span, a simple subtraction $|H_{\mathcal{U}}| - 1$ gives the number of changes.

4 Results

In what follows, the main empirical insights are summarized by presenting a few descriptive statistics on the dataset and then discussing the classification results.

4.1 Descriptive Statistics

Temporal integrity changes are relatively common: more than a quarter of the polls indicated at least one integrity change. The shape of the distribution in Fig. 1 is also interesting: it seems that integrity changes may tend to converge toward a bimodal distribution. In other words, the dataset contains a majority class of scripts for which temporal integrity remained intact, and a minority class for which each daily download resulted in a different hash. The two right-hand side plots also tell that the contents downloaded are indeed mostly JavaScripts.

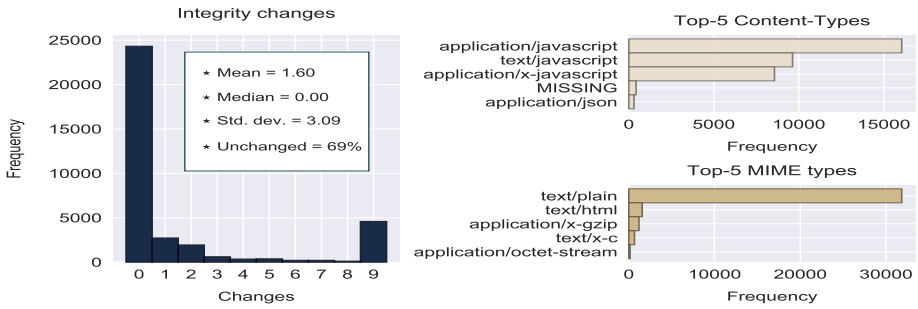

Fig. 1. Temporal Integrity Changes ($|H_{\mathcal{U}}| - 1$) and Buffer Types

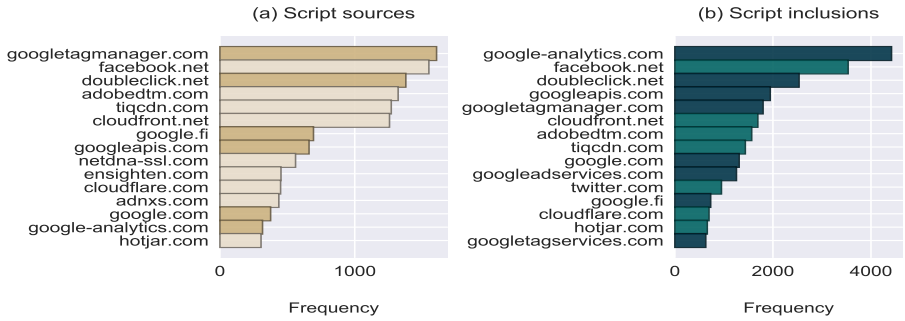

Fig. 2. Top-15 Second-Level Domains (ranks based on (a) the aggregation from the `host` fields of the URLs polled and (b) the number of times the aggregated `host` fields were included by the cross-origin `<script>` tags of the second-level domains sampled)

The temporal integrity changes observed go hand in hand with the lack of integrity checks. There are one-to-many references from the URLs polled to the `<script>` tags that included the sources behind the URLs with cross-origin references. To briefly probe the attributes within these tags, the percentage share of URLs with at least one "back-reference" to a given attribute can be used. Given the few interesting, JavaScript-specific attributes, the shares are: 42.2% for `async`, 3.7% for `defer`, 2.4% for `crossorigin`, and 0.34% for `integrity`. Thus, it is safe to generalize that subresource integrity checks are rarely used in the current Web. Given the integrity changes observed, widespread future adoption of the subresource integrity standard seems also somewhat unlikely.

However, this tentative prediction partially depends on the cloud service and CDN companies who are hosting and distributing popular JavaScripts. For pointing out the main players, Fig. 2 shows the second-level domain names of the fifteen most frequent sources behind the cross-origin scripts observed. The two plots largely confirm the existing wisdom: common locations of remote JavaScript code are extremely concentrated, tracing only to a few companies [21, 26]. In particular, Google continues to be the leading distributor of common JavaScript snippets, although Facebook has recently been catching up.

4.2 Classification

It is interesting to examine how systematic the temporal integrity changes are statistically. For this purpose, the conditional probability that a change occurs, $|H_{\mathcal{U}}| > 1$, is a sensible measurement target. As for features potentially explaining a change, a good point of reference is provided by the literature on classifying URLs pointing to malware and phishing websites. This literature typically operates with numerous simple metrics extracted from URLs, DNS, and related sources [1, 23, 34]. To illustrate a few of such metrics, Fig. 3 displays six so-called mosaic plots. Interpretation of the plots is easy: in each plot the area of a rectangle corresponds with the frequency of a cell in a contingency table.

Table 1. Metrics ($\mathcal{D}$ for dichotomous and $\mathcal{C}$ for continuous scale, $f(x) = \log(x + 1)$ applied for all metrics with $\mathcal{C}$ scale, no additional scaling or centering for classification)

Metric	Scale	Description and operationalization
INCL	$\mathcal{C}$	Number of sampled domains that included a script with a $\mathcal{U}$.
SLEN	$\mathcal{C}$	Character count of the buffer during the first download.
BLCK	$\mathcal{D}$	True if a $\mathcal{U}$ would be blocked by a common ad-blocking list [28].
QURL	$\mathcal{D}$	True if a `query` field is present in a $\mathcal{U}$.
QDOM	$\mathcal{D}$	True if any of the domains including a script's $\mathcal{U}$ appears in `query`.
NOJS	$\mathcal{D}$	True if a `path` field of a $\mathcal{U}$ does *not* end to a `.js` character string.
ULEN	$\mathcal{C}$	Character count of a whole $\mathcal{U}$ used for the polling.
UNUM	$\mathcal{C}$	Number of numbers $(0, \ldots, 9)$ appearing in a whole $\mathcal{U}$.
DNUM	$\mathcal{C}$	Number of domains in a `host` field (excl. the top-level domain and IPv4s).
DTOP	$\mathcal{D}$	A dummy variable for each of the domains in the plot (a) in Fig. 2.

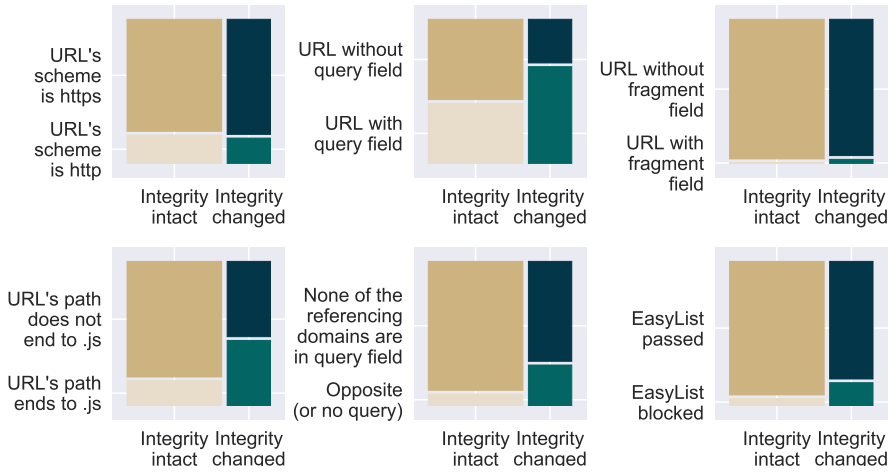

Fig. 3. Temporal Integrity According to a Few Dichotomous Metrics (see main text)

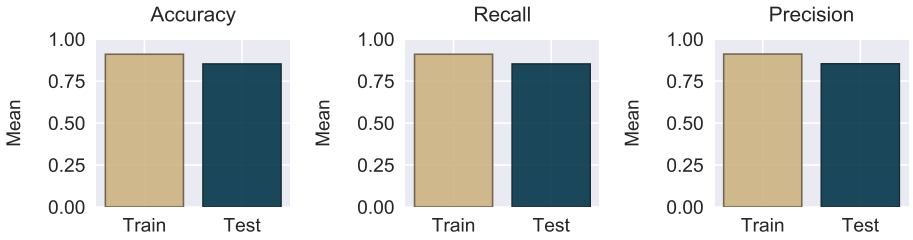

Fig. 4. Classification Performance (35417 script download URLs, 24 metrics, decision tree classifier [27], maximum tree depth restricted to 15, and 10-fold cross-validation for each of the 100 random samples with under-sampling from the majority class)

Although the first plot indicates that HTTPS does not explain temporal integrity changes, it is still noteworthy—and troublesome—that many of the scripts were included by the websites sampled with the `http` scheme. As was discussed in Subsection 2.2, subduing the use of plain HTTP is a prerequisite for sound integrity checks. The two remaining plots on the upper row foretell about an association between integrity changes and the presence of `query` fields but not `fragment` fields. These observations are reinforced by the three plots on the second row. In particular, a `path` ending to `.js` is associated with integrity changes, the domains sampled often appear in the `query` fields of those URLs whose content changed during the polling, and, finally, the probability of a temporal integrity change is slightly higher for URLs blocked by a common ad-blocking list [28] according to an offline parser [19]. These observations hint that temporal integrity changes are typical with respect to scripts used for advertisement and tracking purposes. While this conclusion may seem unsurprising, it is important in terms of future adoption of integrity checks for cross-origin

JavaScript code. Because many websites rely on advertisements and analytics for business reasons, but the corresponding scripts tend to violate temporal integrity premises, it seems that many websites are simply unable to enforce subresource integrity checks—even when these would be widely endorsed by web developers.

After empirically reviewing over 25 metrics, the ten metrics enumerated in Table 1 turned out to be relevant for statistical prediction. The metrics that did not improve prediction include all of the standardized `<script>` attributes, all of the `Content-Type` and MIME types present in the sample, top-level domain names extracted from the URLs, and numerous dummy variables such as whether a `host` field refers to an Internet protocol (IPv4) address. Given the limited amount of information used for predicting whether a temporal integrity change occurs, the results summarized in Fig. 4 are even surprisingly good. The average classification accuracy is 0.85. It can be concluded that the temporal integrity of cross-origin JavaScripts vary systematically, and that it is possible to predict whether a change occurs to a reasonable degree even with limited information.

5 Discussion

This paper presented the first empirical study on temporal integrity of remote, cross-origin JavaScript code commonly used in the current Web. According to the empirical results, temporal integrity changes—or, depending on the viewpoint, temporal integrity violations—are relatively common. Given over 35 thousand URLs observed in a short polling period of ten days, about 31% of the JavaScript content behind the URLs witnessed at least one temporal integrity change. One way to digest this result is to simply state that arbitrary code is commonly executed on the client-side of the current Web. Because temporal integrity is not guaranteed, a cryptomining script [13], for instance, can easily replace an existing legitimate script without any alerts for the clients executing the script.

There are many potential solutions but all of these contain limitations. The simplest solution would be to block all cross-origin content on the client-side, but this would severely impact functionality and user experience. Another solution would be to transform cross-origin `script` tags to `iframe` tags [5], but this solution has performance implications, and it cannot solve the privacy problems. Analogously: using code clones solves the reliance on dynamically loaded third-party code, but at the expense of maintenance and the security risks entailed by in-house maintenance of third-party code [26, 32]. The subresource integrity standard offers a further option. As was discussed and empirically demonstrated, widespread adoption of the standard faces many practical obstacles, however. One obstacle affecting the standard—as well as this paper—is the lack of context behind the temporal integrity changes. In other words, a different hash will result upon fixing a vulnerability in a third-party JavaScript library or making a cosmetic change to such a library. Deducing about the nature of temporal changes would be a good topic for further research, although the commonplace obfuscation of JavaScript code makes the topic challenging to say the least.

To put technical details aside, it might be also possible to refine the underlying ideas presented in the standard. The standard leaves the enforcement of integrity checks to the server-side, but there are no theoretical reasons why clients could not enforce the checks themselves based on a trusted collection of scripts. After all, code signing has a long history elsewhere in the software industry [6]. Given that both web developers and the JavaScript library ecosystem are still taking their first steps toward systematic dependency management and rigorous vulnerability tracking [22], code signing seems like a good long-term goal rather than an immediately applicable solution, however. But for large CDNs and companies such as Google and Facebook, signing the JavaScript code included by hundreds of millions of websites might be possible even today. Another question is whether temporal integrity is in the interests of these companies—if clients would no longer blindly execute arbitrary code, user tracking would be more difficult. In this sense, there exists a classical trade-off between security and privacy, but the current balance that violates privacy undermines also security.

References

[1] Abdelhamid, N.: Multi-Label Rules for Phishing Classification. Applied Computing and Informatics 11(1), 29–46 (2015)

[2] Barth, A.: The Web Origin Concept (RFC 6454) (2011), Internet Engineering Task Force (IETF). Available online in February 2018: https://www.ietf.org/rfc/rfc6454.txt

[3] Berners-Lee, T., Fielding, R.T., Irvine, U., Masinter, L.: Uniform Resource Identifiers (URI): Generic Syntax (RFC 2396) (1998), Internet Engineering Task Force (IETF). Available online in June 2017: https://www.ietf.org/rfc/rfc2396.txt

[4] Bielova, N.: Survey on JavaScript Security Policies and Their Enforcement Mechanisms in a Web Browser. The Journal of Logic and Algebraic Programming 82(8), 243–262 (2013)

[5] Bugliesi, M., Calzavara, S., Focardi, R.: Formal Methods for Web Security. Journal of Logical and Algebraic Methods in Programming 87, 110–126 (2017)

[6] Catuogno, L., Galdi, C.: Ensuring Application Integrity: A Survey on Techniques and Tools. In: Proceedings of the 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2015). pp. 192–199. IEEE, Blumenau (2015)

[7] Cheung, W.T., Ryu, S., Kim, S.: Development Nature Matters: An Empirical Study of Code Clones in JavaScript Applications. Empirical Software Engineering 21(2), 517–564 (2016)

[8] Cisco Systems, Inc.: Umbrella Popularity List (2018), Available online in January 2018: http://s3-us-west-1.amazonaws.com/umbrella-static/index.html

[9] Conti, M., Dragoni, N., Lesyk, V.: A Survey of Man in the Middle Attacks. IEEE Communications Surveys & Tutorials 18(3), 2027–2051 (2016)

[10] Cucurull, J., Guasch, S., Galindo, D.: A Javascript Voting Client for Remote Online Voting. In: Obaidat, M.S. (ed.) Proceedings of the 13th International Conference on E-Business and Telecommunications (ICETE 2016), Communications in Computer and Information Science (Volume 764). pp. 266–290. Springer (2016)

[11] De Ryck, P., Decat, M., Desmet, L., Piessens, F., Joosen, W.: Security of Web Mashups: A Survey. In: Aura, T., Järvinen, K., Nyberg, K. (eds.) Proceedings of

the Nordic Conference on Secure IT Systems (NordSec 2010), Lecture Notes in Computer Science (Volume 7127). pp. 223–238. Springer, Espoo (2010)
[12] Dong, X., Hu, H., Saxena, P., Liang, Z.: A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) Proceedings of the European Symposium on Research in Computer Security (ESORICS 2013), Lecture Notes in Computer Science (Volume 8134). pp. 75–93. Springer, Egham (2013)
[13] Eskandari, S., Leoutsarakos, A., Mursch, T., Clark, J.: A First Look at Browser-Based Cryptojacking. In: Proceedings of the 2nd Workshop on Security & Privacy on the Blockchain (IEEE S&B). pp. 1–9. IEEE, London (2018), Available online in March 2018: https://arxiv.org/abs/1803.02887v1
[14] Mozilla Foundation, et al.: Same-Origin Policy (2018), Available online in January 2018: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
[15] Mozilla Foundation, et al.: Subresource Integrity (2018), Available online in January 2018: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
[16] Geihs, M., Demirel, D., Buchmann, J.: A Security Analysis of Techniques for Long-Term Integrity Protection. In: Proceedings of the 14th Annual Conference on Privacy, Security and Trust (PST 2016). pp. 449–456. IEEE, Auckland (2016)
[17] Jayaraman, K., Lewandowski, G., Talaga, P.G., Chapin, S.J.: Enforcing Request Integrity in Web Applications. In: Foresti, S., Jajodia, S. (eds.) Proceedings of the IFIP Annual Conference on Data and Applications Security and Privacy (DBSec 2010), Lecture Notes in Computer Science (Volume 6166). pp. 225–240. Springer, Rome (2010)
[18] Jia, Y., Chen, Y., Dong, X., Saxena, P., Mao, J., Liang, Z.: Man-in-the-Browser-Cache: Persisting HTTPS Attacks via Browser Cache Poisoning. Computers & Security 55, 62–80 (2015)
[19] Korobov, M.: adblockparser (2018), Available online in March 2018: https://github.com/scrapinghub/adblockparser
[20] Krueger, T., Rieck, K.: Intelligent Defense against Malicious JavaScript Code. Praxis der Informationsverarbeitung und Kommunikation 35(1), 54–60 (2012)
[21] Kumar, D., Ma, Z., Durumeric, Z., Mirian, A., Mason, J., Halderman, J.A., Bailey, M.: Security Challenges in an Increasingly Tangled Web. In: Proceedings of the 26th International Conference on World Wide Web (WWW 2017). pp. 677–684. International World Wide Web Conferences Steering Committee, Perth (2017)
[22] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., Kirda, E.: Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In: Proceedings of the the Network and Distributed System Security Symposium (NDSS 2017). Internet Society, San Diego (2017), Available online in March 2018: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/ndss2017_02B-1_Lauinger_paper.pdf
[23] Ma, J., Saul, L.K., Savage, S., Voelker, G.M.: Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2009). pp. 1245–1254. ACM, Paris (2009)
[24] Magazinius, J., Hedin, D., Sabelfeld, A.: Architectures for Inlining Security Monitors in Web Applications. In: Jürjens, J., Piessens, F., Bielova, N. (eds.) Proceedings of the 6th International Symposium on Engineering Secure Software and Systems (ESSoS 2014), Lecture Notes in Computer Science (Volume 8364). pp. 141–160. Springer, Munich (2014)

[25] Mayer, W., Schmiedecker, M.: Turning Active TLS Scanning to Eleven. In: De Capitani di Vimercati, S., Martinelli, F. (eds.) Proceedings of the 32nd IFIP TC 11 International Conference on ICT Systems Security and Privacy Protection (IFIP SEC 2017). pp. 3–16. Springer, Rome (2017)
[26] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., Vigna, G.: You Are What You Include: Large-Scale Evaluation of Remote JavaScript Inclusions. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS 2012). pp. 736–747. ACM, Raleigh (2012)
[27] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., V. Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12, 2825–2830 (2011)
[28] Petnel, R., et al.: EasyList (2018), Available online in February 2018: https://easylist.to/easylist/easylist.txt
[29] Prokhorenko, V., Choo, K.R., Ashman, H.: Web Application Protection Techniques: A Taxonomy. Journal of Network and Computer Applications 60, 95–112 (2016)
[30] Ruohonen, J., Leppänen, V.: Whose Hands Are in the Finnish Cookie Jar? In: Proceedings of the European Intelligence and Security Informatics Conference (EISIC 2017). pp. 127–130. IEEE, Athens (2017)
[31] Ruohonen, J., Salovaara, J., Leppänen, V.: Crossing Cross-Domain Paths in the Current Web. In: Proceedings of the 16th Annual Conference on Privacy, Security and Trust (PST 2018). IEEE, Belfast (2018)
[32] Somé, D.F., Bielova, N., Rezk, T.: Control What You Include! Server-Side Protection Against Third Party Web Tracking. In: Bodden, E., Payer, M., Athanasopoulos, E. (eds.) Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS 2017), Lecture Notes in Computer Science (Volume 10379). pp. 115–132. Springer, Bonn (2017)
[33] Varghese, S.: UK Researcher Says One Line of Code Caused Ticketmaster Breach (2018), iTWire, available in July 2018: https://www.itwire.com/security/83416-uk-researcher-says-one-line-of-code-caused-ticketmaster-breach.html
[34] Vasek, M., Moore, T.: Empirical Analysis of Factors Affecting Malware URL Detection. In: Proceedings of the eCrime Researchers Summit (eCRS 2013). pp. 1–8. IEEE, San Francisco
[35] W3C: Cross-Origin Resource Sharing, W3C Recommendation (2014), World Wide Web Consortium (W3C). Available online in February 2018: https://www.w3.org/TR/cors/
[36] W3C: Subresource Integrity, W3C Recommendation (2016), World Wide Web Consortium (W3C). Available online in May 2017: https://www.w3.org/TR/SRI/
[37] Zalewski, M.: Browser Security Handbook, Part 2 (2009), Google, Inc. Available online in March 2018: https://code.google.com/archive/p/browsersec/wikis/Part2.wiki