Time Present and Time Past: Analyzing the Evolution of JavaScript Code in the Wild

Dimitris Mitropoulos,*† Panos Louridas,*† Vitalis Salis† and Diomidis Spinellis*
*Department of Management Science and Technology, Athens University of Economics and Business
†Greek Research and Technology Network
{dimitro, louridas, dds}@aueb.gr, vitsalis@noc.grnet.gr

***Abstract*—JavaScript is one of the web's key building blocks. It is used by the majority of web sites and it is supported by all modern browsers. We present the first large-scale study of client-side JavaScript code over time. Specifically, we have collected and analyzed a dataset containing daily snapshots of JavaScript code coming from Alexa's Top 10000 web sites (∼7.5 GB per day) for nine consecutive months, to study different temporal aspects of web client code. We found that scripts change often; typically every few days, indicating a rapid pace in web applications development. We also found that the lifetime of web sites themselves, measured as the time between JavaScript changes, is also short, in the same time scale. We then performed a qualitative analysis to investigate the nature of the changes that take place. We found that apart from standard changes such as the introduction of new functions, many changes are related to online configuration management. In addition, we examined JavaScript code reuse over time and especially the widespread reliance on third-party libraries. Furthermore, we observed how quality issues evolve by employing established static analysis tools to identify potential software bugs, whose evolution we tracked over time. Our results show that quality issues seem to persist over time, while vulnerable libraries tend to decrease.**

***Keywords*-JavaScript, Software Evolution, Bug Persistence, Code Reuse**

I. Introduction

The web is a vast repository of resources. Web sites are based on a variety of elements including HTML markup, style specifications, multimedia content, back-end databases, server-side processing frameworks, and client-side code.

JavaScript is the most commonly encountered client-side computer language for two key reasons. First, it provides a rich interactive experience to web users by supporting browser-provided methods to manipulate the web page's Document Object Model (DOM). Secondly, HTML provides the ability to easily include scripts from arbitrary internet locations. This helps developers reuse existing JavaScript code.

The popularity of JavaScript has led to numerous studies examining several aspects of the language and its use. For instance, such studies include extensive analyses of the dynamic behavior of JavaScript programs [1], [2], [3], examination of programming practices [4], usage patterns of JavaScript APIs [5], data-flow analysis frameworks [6], [7], and construction of JavaScript-specific benchmarks [8].

Due to its nature, JavaScript is also related to various security issues [9] such as Cross-site Scripting (XSS) vulnerabilities [10], [11]. Studies on script inclusion have also indicated that there are thousands of web sites that import at least one vulnerable library [12]. The caveats of the inclusion of scripts hosted on Content Distribution Networks (CDNs) by multiple web sites has also been extensively discussed [13].

There is a certain aspect that JavaScript-related studies do not take into account: *time*. We present the first large-scale study on the use of client-side JavaScript code over an extended period. We specifically focus on the *frequency of change* of JavaScript files that are used by a web site, i.e., their lifespans. We also examine the *popularity* of the various JavaScript libraries provided by CDNs, over a specific period of time. Our research also focuses on the *evolution of bugs* found in JavaScript code and the observation of the *persistence* of vulnerable libraries through time.

Motivating Examples. Our study aims to answer questions that have a common denominator: temporal aspects of the use of JavaScript. The distinct nature of JavaScript and its strong relationship with the web provide an opportunity to shed light to topics related to code update intervals, code reuse, and bug evolution. We identified the following research questions:

- **RQ1: What is the development pace of web applications?** With agile software development being all the rage [14], [15], [16] and the extensive use of continuous integration tools [17] and online configuration management [18], [19], [20], we would like to see how these trends translate in development in popular sites. Continuous changes in the scripts included in the web sites would indicate short deployment cycles.
- **RQ2: How do library dependencies evolve across time?** On many occasions, developers reuse pieces of code or libraries to reduce engineering effort. However, code reuse could lead to unexpected consequences such as the "unpublishing" of an NPM module that led to the breaking of thousands of projects using it [21]. Worse, existing vulnerabilities in CDNs may allow attackers to execute malicious JavaScript on thousands of websites [22]. Also, it would be interesting to see if there are multiple web sites that import libraries with known vulnerabilities for long periods of time.
- **RQ3: How does the quality of client-side code change**

To appear in *MSR '19: Proceedings of the 16th Conference on Mining Software Repositories*, 2019.

over time? There are several studies that examine the evolution of software bugs and quality issues [23], [24], [25]. Specifically, researchers have examined whether bugs increase or decrease over time, the persistence of quality issues, and the relations between bug categories. Security bugs are usually of particular interest [26], [27] because such bugs may introduce a potentially exploitable weakness into a computer system. We wanted to perform similar studies on a large repository containing terabytes of JavaScript code. In this way, we could examine the treatment of quality issues when a new version of a web site is released.

Contributions. To perform our study we built a toolkit that (1) collects scripts from a given set of sites on a daily basis and stores them to a Git repository, (2) generates associated metadata (e.g., the results of diverse static analysis tools), and (3) calculates corresponding metrics. We have collected and analyzed more than 2 TB of script code coming from Alexa's Top 10000 web sites [28], for a nine-month period. Our contributions can be summarized as follows.

1) Our study is the first to examine on a large scale the script change frequency of popular web sites. Our results show an overall high frequency of changes. When we consider third party code though (which in many occasions is automatically imported), change frequency becomes lower. This suggests that a site is released more often than its own developers update their code.
2) Going one step further, we perform a qualitative analysis to examine the nature of the changes. Our findings indicate that changes can relate to either code development or online configuration management [18], [20]. For example, apart from development changes involving the addition of a specific functionality, we also observe entities (e.g., arrays) that are automatically generated to support the handling of different browsers.
3) We identify popular libraries and their families. Our findings indicate that third-party libraries are shared by many sites for long time segments. We point out the caveats behind extensive sharing, complementing previous related work [13], [12].
4) We examine the evolution of the quality issues contained in the scripts and compare our results to existing work on the field [26], [29], [27]. Our findings indicate that the potential bugs found in the scripts tend to persist. In a similar way, we analyze the persistence of vulnerable libraries over time. Specifically, we observe that vulnerable libraries tend to decrease as time progresses.

II. Methods

We discuss the methods we followed to collect scripts on a daily basis and generate metadata including different types of hashes and bug reports coming from two static analyzers.

A. Data Collection and Repository

We collected both *inline* and *external* scripts from Alexa's Top 10000 web sites as of November 2016. In our script collection, we automatically downloaded the contents of each site's main page via `wget` on a daily basis. If we identified a file as an external script (`*.js`), we kept it. If we encountered an HTML file, we examined its encoding, parsed it, extracted all its inline scripts, and saved them as separate files.

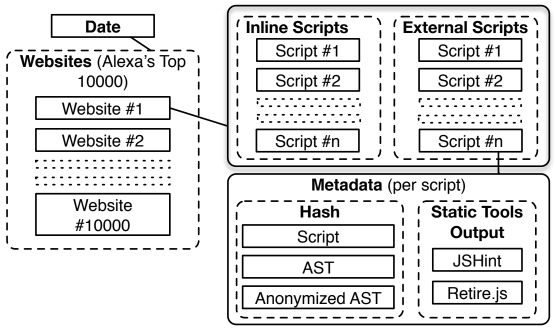


Fig. 1: The hierarchy of the resulting dataset.

The format of our data repository can be seen in Figure 1. We use a Git repository, where every commit corresponds to a specific day. For every day between November 1st, 2016 to July 31st, 2017 (271 days), there is a constant set of folders corresponding to the sites that we have included in our study. Each site folder contains subfolders that include either the external or the internal scripts. Specifically, there is a standard sub-folder that shares the same name with the parent folder (the site's name) containing the inline scripts of the homepage and the site's own external scripts. The external scripts that are fetched from external hosts are included in sub-folders named after the hosts.

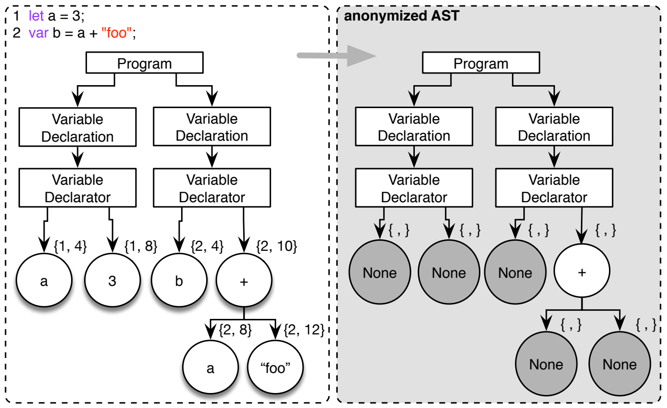
As we mentioned earlier inline scripts are saved as separate files. Their naming convention is as follows: `name.html-jsf-N`, where `name` is the name of the HTML file from which the script was extracted, and `N` its serial number. For example, `index.html-jsf-0` is the script that was found first in the `index.html` file. External scripts are saved exactly as they are downloaded.

The average number of external scripts per day is 49002, and the corresponding number for the inline is 139932. The size of the repository is ∼65 GB while the raw data per day are ∼7.5 GB—approximately 2 TB for all days [30].

B. Static Analysis

To study bug evolution, we employed two well-established static analysis tools: JSHint [31] and Retire.js [32]. JSHint is a lint-like static tool that identifies errors and quality issues in JavaScript code. Note that we did not include all the output of JSHint in our metrics. Instead, we kept warnings that may indicate deeper problems such as bugs due to implicit type conversion and leaking variables. We removed warnings about potentially unsafe line breaking, comma first coding style, and multi-line strings. Retire.js identifies vulnerable libraries based on known file hashes, regular expressions over the contents, and API method fingerprints dynamically evaluated in an empty sandbox environment. Both tools have been used for research purposes several times before [12], [33], [34], [35]. Also, JSHint has been widely accepted by developers and has been used by organizations such as Mozilla and Wikipedia.

Both tools can produce JSON output, which we used for the analysis presented in Section III. Note that, analyzing all the



```
1 let a = 3;
2 var b = a + "foo";
```

Fig. 2: The anonymized AST of a simple program that contains constant values. Note also the removal of the location of each node ({`line, column`}).

scripts of the dataset with JSHint was very time consuming. Hence, we generated metadata that include the output of JSHint for all the scripts used by Alexa's Top 1001 sites.

C. Generating Hashes

The examination of script lifespans can be done in different ways, depending on what we consider as a change. Comparing *script hashes* between different days is one way to study if a script changes. However, there are cases that may distort the picture. For instance, we identified web sites where the only thing that changes in a script is an integer number somewhere in it (e.g., an access ID or a timestamp). Hence, there is no real change in the script's logic and functionality. Similar situations that can effect a trivial change include the addition of comments and white space.

Another take could involve the comparison of the *Abstract Syntax Tree* (AST) *hashes* of the scripts. This does not completely overcome the problems, however. Consider a web site that generates scripts dynamically for its users based on some identity information, e.g., serves personalized advertisements. The generated scripts do not differ in structure but the values of specific class or variables names are unique for each user (the Open Web Analytics framework [36] provides such functionality through its API). This would generate a different AST hash for the same script. We observed the phenomenon in many web sites during our initial tests.

Based on the above observations, we chose to replace all identification information (e.g., function, class, variable names) and all literal values (i.e., any constant value in an expression such as `"red"` in `var color = "red";`) of each script with a standard value, producing *anonymized* AST *hashes*. In this manner, we can better observe substantial changes in the code and leave out trivial changes like the aforementioned ones.

To generate script ASTs we used acorn [37], an open-source JavaScript parser that generates abstract syntax tree objects as specified by the ESTree spec [38]. Our AST-processing steps are as follows. First, we traverse the AST by using a stack, doing a depth-first search. For each node, acorn saves its position within the script (line and column count). As a result, by adding a space, such values in all nodes would change. Thus, we remove this information for all nodes and examine the nature of the node. If it is a literal, we remove its value and if it is either a variable or a function (described as `Identifiers` by the ESTree spec) we replace the name with a neutral value. Figure 2 illustrates the anonymization process of the AST corresponding to a simple code fragment.

TABLE I: Site lifespan medians and KM estimates (in days).

Scripts	**Sites**	**Median**	**KM estimate**
Inline	9190	7	8
External (all)	8218	5	5
Inline and External (all)	8128	2.5	3
External (own)	5909	21	29
External (third party)	6982	5	6
Inline and External (own)	8128	5	5

III. Analysis and Results

The analysis of our dataset had different aspects and spawned various kinds of results. First, we investigated the rate of change of scripts (individually) and the total JavaScript code of web sites. Then, we manually examined web sites to characterize the nature of the changes. We also focused on the popular libraries and the potential threats of sharing such libraries, and we examined the evolution the quality issues and the persistence of vulnerable libraries. Finally, we investigated for how long such scripts were imported by the web sites.

A. Lifespans

From the gathered data we can investigate the changes of individual JavaScript files and the changes of whole sites. We can count the number of changes that happen in files, or sites, in our study period, and thereby derive descriptive statistics, such as the mean and the medium, for the time that elapses between a file or a site is changed. We can also investigate the elapsed time that we expect a file or a site to remain unchanged. Using a metaphor whereby something is alive in the timespan between two successive changes, we want to investigate the *lifespans* of files and sites.

Measuring just the average or median time between changes will not give an accurate estimate of the lifespan, because sites and files continue to exist beyond the end of our data gathering window; we cannot assume that it changes, i.e., in our metaphor, that it dies, when we stopped collecting data. That means that our data are *right-censored*; to get around this problem, we used the Kaplan-Meier (KM) estimator [39] to measure the median lifespan of each item. Without getting too deep into statistics, the Kaplan Meier estimator provides an estimate, at each point in time (daily, in our experiment), of the probability that an individual is alive, taking into account the number of individuals that have died up to, and including, that point. The Kaplan Meier median estimate defines the time at which on average 50% of the population has died (changed, in our experiment).

Our data are also *left-truncated*, in that an item may be born before our gathering window: the web site may have remained unchanged for a period before we started looking and it, and the same goes for an individual file. We do not correct for that

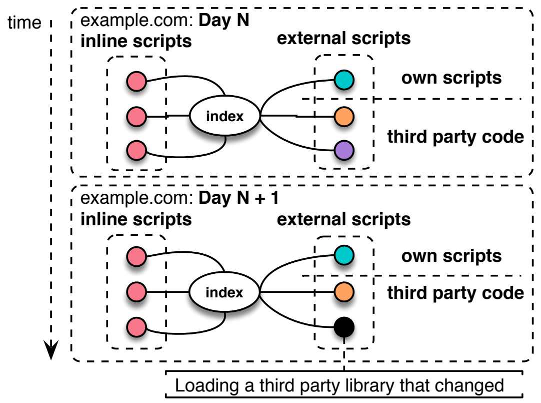


Fig. 3: The external scripts of a file can be either third party code, or libraries written by the developers of the web site (own external). A change on a third party library seems to affect the lifespans of a web site. By not taking into account the third party code, the lifespan increases.

in what follows; we will return to this point when we discuss threats to validity.

We considered two ways to identify when a change happens in a JavaScript file: a file changes when anything changes in it, or a file changes when its AST changes. We used anonymized ASTs for our analyses because these minimize the effect of trivial changes, as explained in II-C.

The first way will result in more changes, which may be unsubstantial, taking note of changes in whitespace or automatically generated values. It will therefore produce shorter lifetimes than the second way. In general, though, the minimum possible lifetime is one day, where the maximum possible lifetime may be unknown, if no change was detected in an item throughout the whole data collection period.

Although the experiment collection ran for 271 days, we use 261 days for our analysis because we had a data gathering gap on July 20, 2017, which could affect the calculations.

1) File Lifespan: If a site contains n external JavaScript files, $f_1, f_2, \ldots, f_n$, and file f_i had i_k updates at times $t_{i,1}, t_{i,2}, \ldots t_{i,i_k}$, then we performed our analysis for each file using the following set of lifespans:

$$L = \{t_{i,j+1} - t_{i,j},\ \forall i \in 1, 2, \ldots n,\ j \in 1, 2, \ldots, i_{k-1}\}$$

In particular, we calculated the median lifespan and the KM estimate of the median lifespan for each file referenced in each of the sites that we studied using the set of the individual lifespans of each file.

There were 8243 sites with external JavaScript files. Working with all changes, the median lifespan of a JavaScript file in a site was found to be 5 days. The KM estimate of the lifespan was found to be 6 days. About 9% of the sites did not witness any change in any of their files throughout the study period.

If we take only AST changes into account, the lifespans increase somewhat. There were 8218 sites whose ASTs we could process. The median lifespan of a file was found to be 7 days and the KM estimate of the lifespan was found to be 8 days. There were about 10% of sites without any changes in any files.

There is a slight twist concerning inline JavaScript. An inline script at one day may change position with another script the next day: for example, script #1 may appear as script #2, and vice versa (see Subsection II-A). We cannot know whether the change was intentional by a developer, or if the scripts were injected in different places in the DOM tree by a tool or an automated process. Therefore we cannot compare inline JavaScript on a day to day basis. This phenomenon can also appear with external JavaScript, when an external script is simply renamed, and indeed, we found sites where this appears to be the case. When this happens, a file's lifetime is artificially curtailed at each renaming, which means that the contents of the script actually live longer than our method would suggest. We decided to accept that, as file renaming is still an actual change, and anyway the situation is not typical.

2) Web Site Lifespan: A web site changes when any of the files it contains changes. To study the changes of a web site, we took the files on each day. By concatenating the hashes (either of files, or of ASTs) on each day, we get a representation of the state of the web site for that particular day. We can then use the resulting vector of values (one state per day) and calculate the lifespans between different states. With these we can calculate the median lifespan and the KM estimate of the median, as before. We noticed above that a script, internal or external, may move around from day to day. That means that a concatenation of hashes may result in more state changes than happened in reality. For that reason, we sorted the daily hashes. More formally, for each site s with n JavaScript files we have a tuple of sorted hashes for each day i:

$$H_{s,i} = (h_1, h_2, \ldots, h_n)$$

By concatenating the contents of each $H_{s,i}$ we obtain a vector V_s whose entries represent the state of the web site at each day i:

$$V_{s,i} = h_1 : h_2 : \ldots : h_n$$

It is not necessary that a file exists on day i; that means that its hash is null. We represent such hashes with a special unique value (nan). To find the lifespans we then work with the vector V_s, detecting runs of identical elements. If we find:

$$V_{s,i} = V_{s,i+1} = \ldots = V_{s,i+k}$$

then we know the web site stayed unchanged for k days.

Although we analyzed hashes for both files and anonymized ASTs, in what follows we will present the results for the AST hashes only; the file hashes show shortest lifespans, but the anonymized AST hashes take care of trivial changes as we discussed in Section II-C. Table I summarizes the results.

If we work with inline JavaScript, the median is at 7 days and the KM estimate of the median is at 8 days; about 9.5% of the web sites did not change at all. If we move to external JavaScript files, both the median lifespan and the KM estimate is 5 days; at the same time, there are about 7.7% of web sites that did not change at all.

We can work with all changes, both to the inline scripts and to the external JavaScript files, by concatenating the state according to the inline scripts and the external files and

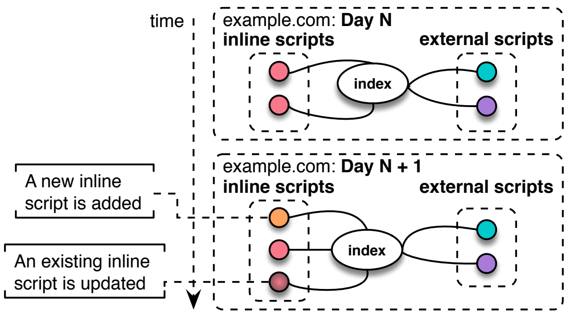


Fig. 4: Developers seem to either add, remove or update inline scripts over time. In this example an inline file is added and another is updated. This actually happens in ndtv.com where developers introduce a new function in a fresh inline file and invoke the function in an existing one.

performing the same analysis. If we do that, we find that the estimates fall: the median is 2 days and the KM estimate is 3 days, while about 3.5% of the sites did not change at all.

Going back to the external files, we can distinguish between those files that are provided by third parties (say, jQuery implementations) and those files that are presumably developed by the web site developers (see Figure 3). To distinguish between the two, we check whether the path for the file contains the domain name of the site. We can partition external files to two groups then, own files and third party files, and see how each group contributes to web site changes.

We found 5909 web sites that contained own external JavaScript files. The median lifespan using them climbed to 21 days and the KM estimate climbed to 29 days, while about 16% of the web sites did not change at all.

Concerning the third party JavaScript files, we found 6982 web sites that contained them. The median lifespan is 6 days and the KM estimate is 5 days; about 6.8% of the sites remained unchanged throughout.

Observing the difference between the own external files and the third party files, we can proceed to see what is the situation if we take into account the inline JavaScript and the own external files, thus excluding those files that we know they are developed elsewhere. The results should be briefer lifespans than taking either of them, as we are performing a logical or. Indeed, we found that the median and the KM estimate are both at 5 days, while 6.2% did not have any related change.

3) Qualitative Analysis: We have manually examined three categories of web sites to see what kind of changes take place over a given period of time. Specifically, we focused on (1) ten sites containing files that change frequently (i.e., they either have a very small change median or include inline or external files with a very small change median), (2) ten popular sites including facebook.com, youtube.com, google.com, and (3) five sites chosen at random. Then, we picked at random 20 consecutive dates for each site and searched for potential changes by examining both inline and external scripts. We have classified the changes that we have identified in two categories: *development* and *configuration*.

Development changes involve the addition or removal of functionalities, function updates, function invocations and others. Consider for instance ndtv.com where, in the course of ten days, two inline scripts were added. Both scripts contained new functions. In addition, six inline scripts were also updated to invoke the new functions. Figure 4 illustrates such a scenario. The situation was similar in tripid.com (17 days) and native-instruments.com (10 days). We found a similar pattern on external files. For instance, in tripid.com we saw changes regarding font-handling, while in campograndenews.com.br (7 days) new pop-up windows were added.

Configuration changes are related to configuration management, such as arrays and dictionaries generated automatically by the server or JSON objects serving as data storage. For instance, in several scripts of mozilla.org there are dictionaries with different object ordering, on different days. Also, in some cases, there are objects that are either added or removed from the dictionary. Such entities can be related to the handling of different browsers as indeed we observed in mozilla.org. Similar entities also appear on facebook.com. Furthermore, in facebook.com we observed that entire functions seemed to be generated automatically (e.g., `envFlush`). In particular, we observed that the same if / else statements, were written with brackets on one day (`if(condition)`{`command;`}) and without on another (`if(condition)command;`), which leads to a different AST. Given that it is highly unlikely that someone adds or removes the brackets day by day we can presume that this happens automatically. Or consider zameen.com, which lists developers, agencies, and properties for sale and rent. Interestingly, all data are stored in a JSON object in an external script instead of a database backend. This object is updated every time an entity is added or removed, something that we observed in our analysis.

Using anonymized AST hashes (rather than using simple AST hashes) turned to be the right choice. In particular, there were sites, such as clubic.com, where function names changed almost every day even if the rest of the code stayed the same (e.g., from `funct05` to `funct05b`). Variable names displayed a similar behavior in citibank.com and staturn.de (note that this is not the saturn.de site, which also belongs in the top 10000). Such cases would produce a distorted image regarding script lifespans, if we relied on simple hashes.

RQ1: What is the development pace of web applications? Overall, our results show that the lifespans of the scripts of a web site are short, which implies a rapid development pace. Change frequency becomes lower, however, when we do not consider third party code (which in many cases is automatically imported). This indicates that a site is released more often than its own developers implement functional changes on it. If sites use continuous integration, the deployment pipeline [40] is very short, but that does not mean that this is the result of the site developers writing and committing new code. Due to the large number of dependencies to third party code, which is tracked in real time, a site's total code content changes more often than its own code base. Furthermore, our qualitative analysis indicates that online configuration management is vividly reflected on client-side code. Indeed, the automatic

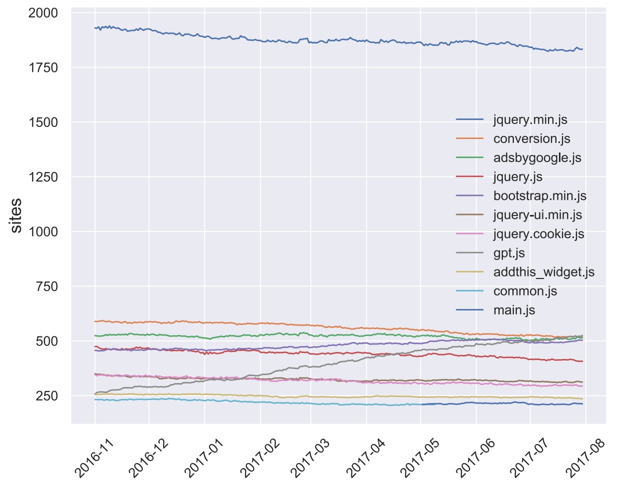


Fig. 5: The top ten most popular JavaScript libraries over time. They are actually 11, because `common.js` overtakes `main.js` in May 2017.

code updates that take place on Facebook scripts are in accordance to the holistic configuration management that the web site employs [18].

B. Library Popularity across Time

To investigate the popularity of JavaScript libraries across time, we counted, for each library, the number of sites that uses it every day. Then we took the top ten libraries in terms of popularity each day. It turned out that they are in fact 11 libraries in the top ten, because `common.js` overtakes `main.js` at the bottom. You can see the evolution in Figure 5.

Four of the libraries in Figure 5 are directly related to jQuery and three of them to Google Adwords: `adsbygoogle.js`, `conversion.js` [41] (conversion tracking tags) and `gpt.js` [42] (Google publisher tags).

The version of each script does not appear in the name of the file in Figure 5, because of the different ways that a developer can include an external library. For instance, if developers select to include jQuery from Google's CDN, they will add the version name as part of the directory tree. If they do so through Microsoft, the version will appear in the file's name (e.g., `jquery-1.11.1.min.js`, found in 150 different sites, not making it into the top libraries).

The popularity of `jquery-1.11.1.min.js` is of particular importance. That is because this library decodes HTML entities in a wrong way which may lead to an XSS attack [43]. There are also other cases of popular libraries that are at the same time vulnerable to XSS (e.g., `jquery-1.7.2.min.js` used by 126 sites and `jquery-1.7.1.min.js` used by 103 sites) [44]. Hence, one vulnerable library can affect hundreds of popular web sites and their corresponding users.

In a similar study that incorporated a one-day snapshot of JavaScript code coming from thousands of sites, Lauinger et al. [12] indicated that the jQuery libraries and Bootstrap are also included in their top 10 list, which matches our findings. However, they missed out `common.js` [45] and Google

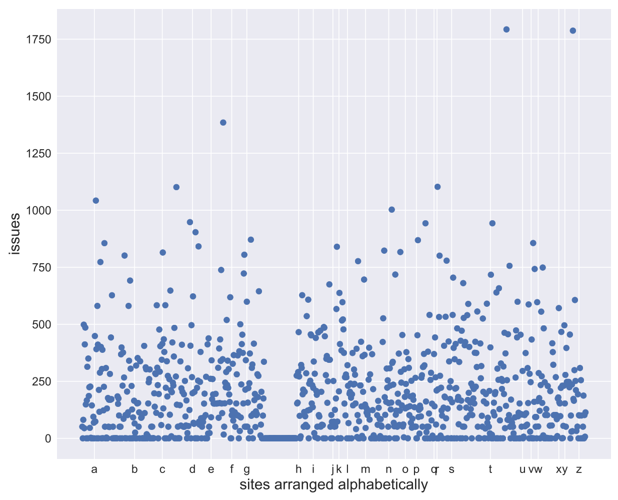


Fig. 6: Mean number of quality issues per day for all sites, arranged alphabetically.

Adwords-related libraries such as `adsbygoogle.js`. Conversely, they included SWFObject [46] and Moment [47] in their list, which we did not.

Heavy dependencies to third party libraries at such a large scale can be worrisome. Consider the case where attackers manage to hijack the jQuery web site and change benign scripts with malicious ones. As a result, all sites that include these scripts will automatically run malicious code—an issue that has also been raised by Nikiforakis et al. [13].

RQ2: How do library dependencies evolve across time? Overall, our results show that there are multiple third-party libraries that are shared by web sites for long periods of time. Vulnerable popular libraries may pose a threat to the security of hundreds of web sites. Also, the dependencies to such libraries can be alarming if we consider that attackers could hijack the infrastructure that provides them. Unfortunately, the vulnerability that was found recently in the popular http://unpkg.com content delivery network [22], indicated that this scenario is not far from reality.

C. Software Quality Issues Evolution

From the generated metadata we can examine the evolution of the potential bugs identified by JSHint and the persistence of the vulnerable libraries detected by Retire.js.

1) Quality Issues Over Time: As we described in Section II-B, we used JSHint to analyze the scripts of the 1001 most popular web sites. Our initial results indicate that 808 of them have at least one issue in one day during the study period, including facebook.com (mean value $\approx$ 101 defects per day), twitter.com (mean value $\approx$ 106 issues per day), and cnn.com (mean value = 204 issues per day, in fact there are constantly 204 issues in each day). Defect types include unnecessary usage of the "use strict" directive, leaking variables and more. Some of these defects (e.g., missing semicolons) can be considered as code smells [48]. Nevertheless, they may indicate deeper problems in the code.

The mean number of potential bugs per day varies widely between sites, from 0 up to 1793. Most sites have less than

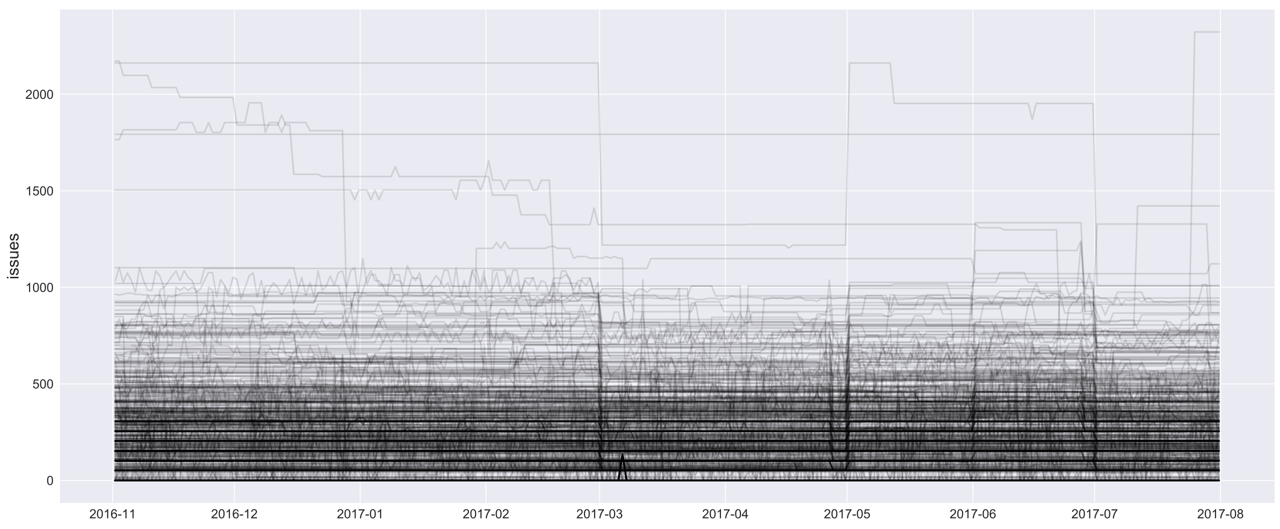


Fig. 7: Evolution of quality issues over time. Each line represents the number of potential bugs of a web site through the study period. The horizontal bands indicate that the number of issues tends to remain stable.

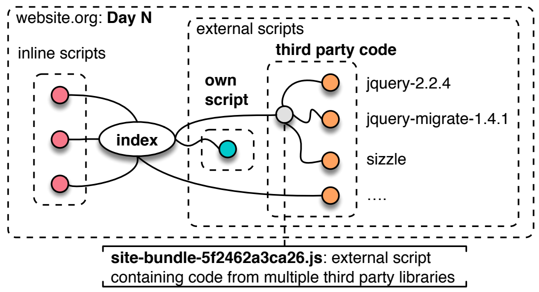


Fig. 8: In some occasions third party code is gathered in external files. Such files are automatically generated by utilities that assign a different name to the file each time (typically by adding the hash of the file to the name to avoid browser caching). In this case, which is based on a real example, we observe that the developers of the web site include two vulnerable libraries in one external file (`jquery-2.2.4`, and `jquery-migrate-1.4.1`).

200 issues per day: the median of the mean is at $\approx$ 135.

Plotting the mean number of issues per day for all sites, arranged alphabetically, we obtain Figure 6. The pattern that can be detected is the paucity of issues in Google sites.

Beyond the mean number of quality issues per day we can see what is going on in time. Figure 7 shows the evolution of all defects for all sites. Each line corresponds to the number of defects for a single site. Lines are semi-transparent. We can discern horizontal bands, which are sites with approximately the same number of defects over time. There are also some sudden peaks around March, May, and July 2017.

2) Evolution and Persistence of Vulnerable Libraries: Retire.js indicated that 5835 sites contained at least one vulnerable library during the covered period. This is a result that complements the findings of Lauinger et al. [12], who have indicated that 37% of 133K sites include at least one library with a known defect. Also, we found that 2158 of the 10K sites include on average more than one vulnerable library per day.

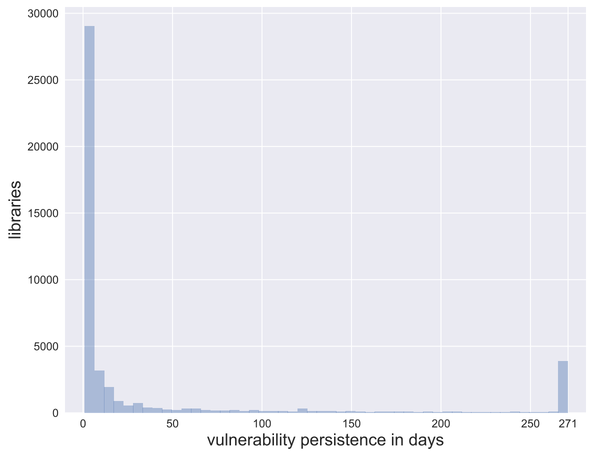


Fig. 9: Vulnerable library persistence histogram. The x axis shows the days that a vulnerability remains unresolved and the y axis the number of libraries that contain vulnerabilities that remain open for that amount of time.

Vulnerable code from different libraries may flock together to a single external file. Apart from simple copy paste, this can also be done automatically by utilities (e.g., `webpack` [49]) that gather the code into a unique file and assign a different name to it (typically by adding the hash of the file to the name to avoid browser caching). Figure 8 shows the case of a web site (based on a real example) that includes the code of different libraries. Two of them `jquery-2.2.4` and `jquery-migrate-1.4.1`, contain vulnerabilities.

There are cases where specific libraries with defects are included by a web site on all 271 days. `jquery-2.2.4` is one of these libraries. This version, in particular, is vulnerable to XSS attacks when a cross-domain Ajax request is performed without the `dataType` option causing *text/javascript* responses to be executed [50].

How long does it take to fix a vulnerability? The sooner the better, but reality may differ. The urgency is attenuated if the detected vulnerability does not in fact render the web

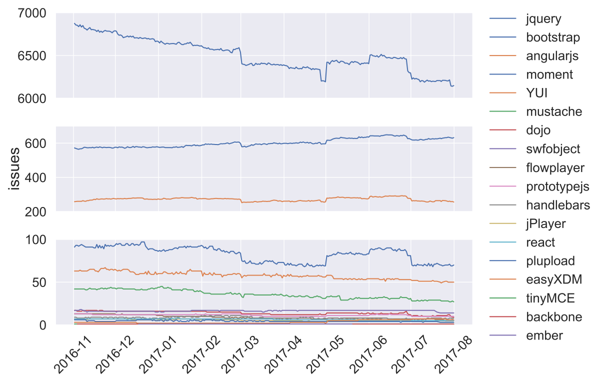


Fig. 10: Vulnerable library families over time.

site vulnerable. For instance, in many occasions where sites include `jquery-2.2.4`, developers use the `dataType` option correctly, thus the site is not vulnerable. However, such constructs can be seen as time-bombs waiting to go off in a potential misuse.

Moreover, in hindsight we do know when a defect occurred, but the developers might become aware of it only later. These factors may explain Figure 9, a histogram of the persistence of all the detected libraries. There are 46,205 libraries that appear as vulnerable—not all of them unique, as the same vulnerable library may appear in multiple sites or at different time intervals in the same site. The mean time that a vulnerable library is used is about 41 days, with the median value at 3 days. The difference between the median and the mean is due to the large number of outliers. There are many thousands that are not fixed soon, either because it is not needed, or out of ignorance; 3849 vulnerable libraries (more than 8%) remain unfixed throughout the whole study period.

By examining vulnerable and at the same time popular libraries we observed that the top ten most popular vulnerable libraries come from the jQuery family. What about other libraries? We bundled related libraries so that we could investigate the most popular vulnerable library families. The result is shown in Figure 10. Vulnerable members of the jQuery family are included in thousands of web sites. Next come Bootstrap and AngularJS, which appear in hundreds of sites. In the lowest tier, going up to a hundred, Moment [47], YUI [51], and mustache [52] are prominent. Overall, the ranking of vulnerable library families remains constant over the period of the study.

Based on our data, it is possible to see the actual evolution of vulnerabilities over time for each site. Putting all sites on a single graph would have too cluttered a result, so Figure 11 shows the evolution of vulnerabilities for sites starting out with 0, 1, up to 9 vulnerable libraries at the start of the study period. As there are many thousands of sites in the first row of the figure, we have used semi-transparent lines to detect the overall pattern. It appears that the number of vulnerable libraries in sites move around in fairly narrow bands. We also see that the overall trend in the panels is to have more movement from the starting point downwards than upwards: that is, vulnerabilities tend to go down rather than up. This is not so in the last two panels, but these are extreme cases anyway.

RQ3: How does the quality of client-side code change over time?

The quality issues contained in the scripts of the various web sites seem to persist over time. Note that studies on the evolution of bugs in other heterogeneous environments such as the Linux file system [29] and the code base of the OpenBSD [26], have also pointed out that the number of bugs does not die out over time. Nevertheless, this is not exactly the case for vulnerable libraries which, in turn, seem to slightly decrease as time progresses. That is an indication that developers seem the treat issues posed by vulnerabilities when they deploy a new version of their web site. This is not the case though with popular, vulnerable libraries because web sites seem to constantly include them.

IV. Threats to Validity

We measured the lifespans of files and sites making the assumption that they are born (i.e., that they are either created or a new version is produced) on the day we started collecting our data. That may not be true. Unless a file or a site changed on the very first day, we miss out its life before we started looking at it; that is the result of left-truncated data, at which we hinted in III-A. Therefore it is possible that our results are somewhat shorter than what would be if we could look back to the past at the most recent previous change. While we could deal with the data from the first change that we see in a file after the start of data gathering onwards, this would result in culling from our data all files that had only a single change in this period, so we decided not to do that. Moreover, the large number of changes from the point we do look at files and sites means that we have a large number of data to work with, so that our results should not be off the mark. Note that we are not interested at when a file or site really comes into existence in the first place: that would take us probably years back. The problem is that we don't know when it last changed before November 1, 2016.

The same library may exist in different forms, as it may come minimized or non-minimized; or it may even be minimized by different tools. As we measure popularity based on the name of the library, this introduces a library at multiple places in the ranking—see `jquery.min.js` and `jquery.js` in Figure 5. A solution is to work with library families, as we did in Figure 10; that said, it is interesting to note the widespread use of both minimized and non-minimized versions.

A threat to the internal validity of our experiment could be the false positives and false negatives of the static tools that we used. For instance, potential false negatives may occur when Retire.js analyzes a library in its empty sandbox environment, and the library has unmet dependencies [12]. In addition, JSHint stops script analysis if it finds two many errors raising a warning and indication the proportion of the script scanned.

As we described in II-A, our tool collects scripts after visiting the main page of each web site only. This is a threat because (1) we do not include all the inline scripts of a web

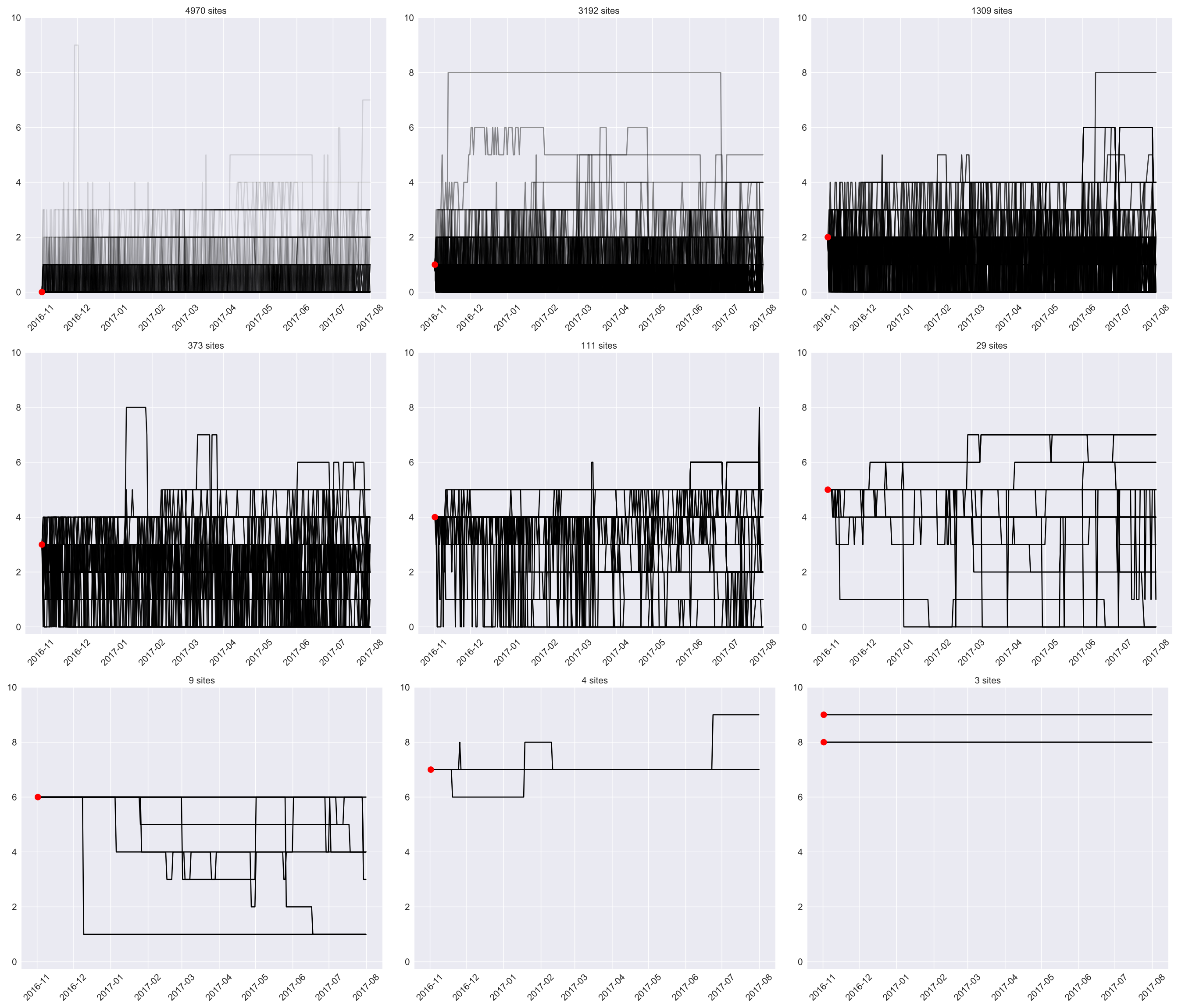


Fig. 11: Persistence timelines. Each panel corresponds to sites that start out with 0, 1, up to 9 vulnerable libraries, the maximum we found. Starting points are indicated with red bullets. The last panel groups together the 8 and 9 starting points. The overall trend is close and downwards from the starting points. When starting from 0 we do see considerable movement to 1, only, but there the only way is up.

site and (2) we may loose other external libraries employed in other pages of the web site. However, best practices dictate that external libraries that are used by a site must be included in all its pages [13].

V. Related Work

Our work falls under two different categories of related work: measurement studies focusing on the characteristics and the behavior of JavaScript, and the evolution of software faults.

A. *Measurement Studies on JavaScript*

The common denominator for almost all JavaScript-related studies is that they focus on web snapshots without taking into account the time factor.

We already had occasion to mention the work of Lauinger et al. [12], who conducted a study on client-side JavaScript code using a snapshot of 133K web sites, half of which were the top 75K Alexa web sites and the other half a random snapshot of the web. In order to identify vulnerabilities they used static analysis and dynamic analysis. They, too, used extensively Retire.js. In our work we have also focused on libraries with defects, but focusing on the evolution over time.

Nikiforakis et al. [13], who we have also met above, did take time into account in their study regarding JavaScript inclusions. They underline that when developers include a third party library into their web site, they should not fully trust the third party provider. Their results show that such providers can be be successfully compromised by attackers by exploiting specific types of vulnerabilities. Their dataset consists of a large scale crawl of the top 10000 Alexa web sites, visiting at most 500 pages per site. The crawl was not continuous though, with months passing in between.

Kumar et al. [9] conducted a study on the top 1M Alexa web sites, showing that each page loads a median of 73 resources, 23 of which are external. Their methodology involved visiting the root pages of each site and analyzing the resources loaded by it. They noted that a small number of third parties serve resources for a large fraction of sites. Such third parties include Google, Facebook, CDNs, and cloud providers. Furthermore, they showed that 33% of the top million sites load unknown content indirectly through at least one third-party, exposing users to resources that site operators have no relationship with.

On the library popularity front, Libscore [53] is a popular web-based tool that scans the 1 million most popular web sites to collect statistics related to JavaScript library usage. Libscore can detect modules loaded either via RequireJS, jQuery plugins, window variables and external scripts. This is accomplished partly by visiting each site and running heuristics related to the window variables to determine if they were the result of a third party library.

Focusing on JavaScript related vulnerabilities, Lekies et al. [54] showed cases of insecure usage of browsers' local storage for code caching purposes using the Alexa Top 500K web sites. Meanwhile, Son et al. [55] wrote about the defects arising from unsafe uses of the `postMessage()` function using the Alexa Top 10K web sites. Moving to XSS attacks, Lekies et al. [10] detected and validated DOM-based XSS vulnerabilities on the Alexa Top 5K web sites. The different (and at times dangerous) usage patterns of the `eval()` function have also been extensively studied [5]. Yue et al. [4] showed insecure practices related to JavaScript, including cross-domain inclusion and the execution and rendering of dynamically generated JavaScript and HTML. They find that over 66.4% of the 6,805 web sites in their dataset include JavaScript from external domains into the root documents of their web sites, and 44.4% use `eval()`. Finally, the error messages printed by JavaScript and their root causes have been studied [56] using 50 of the top 100 Alexa web sites. The corresponding results indicated that both non-deterministic and deterministic errors occur depending on the time spent on the web site or the speed of testing.

B. Studies on Bug Evolution

Ozment and Schechter [26] have examined the evolution of the defects found in the code base of OpenBSD. Specifically, they measured the rate at which new code has been introduced and the rate at which bugs have been identified and reported over a 7 year period and 15 releases. The authors indicated that there is a small decrease in the rate at which vulnerabilities are being reported. However, defects seemed to be persistent for a period of at least 3 years.

Lu et al. [29] examined the evolution of filesystem code. In particular, they analyzed the changes of Linux filesystem patches to identify and extract bug patterns and categorize bugs based on their impact. Their findings indicated that the number of bugs does not die down over time. We observed a similar situation with the bugs contained in the various scripts.

Mitropoulos et al. [27] have studied the evolution and the persistence of potential security bugs found in different versions of software libraries contained in the Maven Repository [57]. Their results show that it is not clear whether across projects defect counts increase or decrease over time. Regarding persistence, security bugs seem to be persistent in the same manner as other bugs (e.g., performance bugs).

Massacci et al. [23] analyzed the evolution of bugs by examining six releases of Firefox. To achieve this, they created a database that contained information coming from Bugzilla entries, the MFSA (Mozilla Firefox-related Security Advisories) list [58], and others. Their findings show that bugs are indeed persistent over time.

The evolution of software artifacts has also been examined to reduce the false positives of static tools. To do so, Spacco et al. [59] attempted to pair sets of bugs between versions to find similar patterns. In their research they examined the evolution of 116 builds of the JDK. Their findings showed that high priority bugs are fixed as time progresses.

VI. Conclusions and Future Work

Our findings show that the lifespans of the scripts of a web site are short, which indicates a high development pace (RQ1)—certainly much higher than traditional methods such as the waterfall model. This can be the result of continuous integration tools, as well as widespread adoption of agile development methods. At the same time, many changes are not really changes in functionality, but configuration changes. That may reflect the fact that, in JavaScript, configuration is often given by JavaScript objects, which are indistinguishable from code. In a certain sense, configuration *is* code.

We found that there are many popular third-party libraries shared by multiple sites over time, and the popularity of libraries seems to remain constant over time (RQ2). A worrying finding involves the usage of vulnerable libraries by hundreds of web sites. The quality issues contained in the scripts of the different web sites seem to persist as time progresses. However, this is not the case for vulnerable libraries which, in turn, seem to decrease over time (RQ3).

We mentioned the problem of false positives emitted by static analyzers in Section IV. A possible way to address false positives would be to take a random sample of JSHint reported issues and manually check their status. Then we could get confidence intervals on the actual number of bugs.

Further studies based on our dataset may examine how web site client code conforms to Lehman's laws of software evolution [60], and specifically, "continuing change", "increasing complexity" and "declining quality". It is also possible to study the evolution of issues separately for third party libraries and site-specific code. This will provide us with details on how third party code affects the web sites that include it. The popularity of web sites can also be taken into account in the measurements, for instance investigating how the persistence of vulnerable libraries varies in sites with different numbers of visitors.

Data Availability. Our dataset is publicly available through the Zenodo service [30].

Acknowledgements. We would like to thank the reviewers for their insightful comments and constructive suggestions. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825328. This work was also supported by the European Union's Connecting Europe Facility (CEF) Work Program 2016 under grant agreement INEA/CEF/ICT/A2016/1332498 and by computational time granted from the Greek Research & Technology Network (GRNET) in the National HPC facility - ARIS - under project ID: Panos Louridas – JS Evolution.

References

[1] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of JavaScript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 1–12.

[2] S. Wei, "Blended analysis for JavaScript: A practical framework to analyze dynamic features," in *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 101–102.

[3] S. Wei, F. Xhakaj, and B. G. Ryder, "Empirical study of the dynamic behavior of JavaScript objects," *Softw. Pract. Exper.*, vol. 46, no. 7, pp. 867–889, Jul. 2016.

[4] C. Yue and H. Wang, "A measurement study of insecure JavaScript practices on the web," *ACM Trans. Web*, vol. 7, no. 2, pp. 7:1–7:39, May 2013.

[5] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in JavaScript applications," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ser. ECOOP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–78.

[6] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 50–62.

[7] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 270–283.

[8] G. Richards, A. Gal, B. Eich, and J. Vitek, "Automated construction of JavaScript benchmarks," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 677–694.

[9] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey, "Security challenges in an increasingly tangled web," in *Proceedings of the 26th International Conference on World Wide Web*, ser. WWW '17. International World Wide Web Conferences Steering Committee, 2017, pp. 677–684.

[10] S. Lekies, B. Stock, and M. Johns, "25 million flows later: Large-scale detection of DOM-based XSS," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1193–1204.

[11] D. Mitropoulos, K. Stroggylos, D. Spinellis, and A. D. Keromytis, "How to train your browser: Preventing XSS attacks using contextual script fingerprints," *ACM Transactions on Privacy and Security*, vol. 19, no. 1, pp. 2:1–2:31, Jul. 2016.

[12] 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 the 21st Network and Distributed System Security Symposium (NDSS '17')*, 2017.

[13] 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 *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 736–747.

[14] I. Jacobson, I. Spence, and E. Seidewitz, "Industrial-scale agile: From craft to engineering," *Commun. ACM*, vol. 59, no. 12, pp. 63–71, Dec. 2016.

[15] K. Kuusinen, P. Gregory, H. Sharp, and L. Barroca, "Strategies for doing agile in a non-agile environment," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:6.

[16] D. K. Rigby, J. Sutherl, and A. Noble, "Agile at scale," https://hbr.org/2018/05/agile-at-scale, 2018, [Online; accessed 24-July-2018].

[17] S. Vöst, "Vehicle level continuous integration in the automotive industry," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 1026–1029.

[18] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl, "Holistic configuration management at Facebook," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 328–343.

[19] A. Deng, J. Lu, and J. Litz, "Trustworthy analysis of online A/B tests: Pitfalls, challenges and solutions," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, ser. WSDM '17. New York, NY, USA: ACM, 2017, pp. 641–649.

[20] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for Puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 416–430.

[21] C. Williams, "How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript," https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/, 2016, [Online; accessed 25-November-2018].

[22] M. Justicz, "Compromising thousands of websites through a CDN," https://justi.cz/security/2018/05/23/cdn-tar-oops.html, 2018, [Online; accessed 21-December-2018].

[23] F. Massacci, S. Neuhaus, and V. H. Nguyen, "After-life vulnerabilities: a study on Firefox evolution, its vulnerabilities, and fixes," in *Proceedings of the Third international conference on Engineering secure software and systems*, ser. ESSoS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 195–208.

[24] N. Edwards and L. Chen, "An historical examination of open source releases and their vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 183–194.

[25] S. Zaman, B. Adams, and A. E. Hassan, "Security versus performance bugs: a case study on Firefox," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR '11. New York, NY, USA: ACM, 2011, pp. 93–102.

[26] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?" in *Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006.

[27] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "Dismal code: Studying the evolution of security bugs," in *Proceedings of the LASER 2013 (LASER 2013)*. USENIX, 2013, pp. 37–48.

[28] "Alexa: The top 500 sites on the Web," https://www.alexa.com/topsites, 2018, [Online; accessed 06-March-2019].

[29] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, "A Study of Linux File System Evolution," in *Proceedings of the 11th Conference on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[30] D. Mitropoulos, P. Louridas, V. Salis, and D. Spinellis, "All Your Script Are Belong to Us: Collecting and Analyzing JavaScript Code from 10K Sites for 9 Months," https://doi.org/10.5281/zenodo.2593266, Mar. 2019.

[31] "JSHint, a static code analysis tool for JavaScript," http://jshint.com/, 2016, [Online; accessed 25-November-2018].

[32] E. Oftedal, "Retire.js: What you require you must also retire," http://retirejs.github.io/retire.js/, 2014, [Online; accessed 25-November-2018].

[33] S. Karkalas and S. Gutiérrez-Santos, "Enhanced JavaScript learning using code quality tools and a rule-based system in the FLIP exploratory learning environment," in *Proceedings of the 2014 IEEE 14th International Conference on Advanced Learning Technologies*, ser. ICALT '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 84–88.

[34] A. L. Santos, M. T. Valente, and E. Figueiredo, "Using JavaScript static checkers on GitHub systems: A first evaluation," in *Proccedings of the 3rd Workshop on Software Visualization, Evolution and Maintenance (VEM)*, 2015, pp. 33–40.

[35] N. C. Zakas, *Maintainable JavaScript*. O'Reilly Media, Inc., 2012.

[36] "The open web analytics: Web analytics, open source," http://www.openwebanalytics.com/, 2018, [Online; accessed 21-December-2018].
[37] "A small, fast, JavaScript-based JavaScript parser," https://github.com/acornjs/acorn, 2018, [Online; accessed 21-December-2018].
[38] "The ESTree spec," https://github.com/estree/estree, 2018, [Online; accessed 21-December-2018].
[39] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. 457–481, 1958.
[40] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Upper Saddle River, NJ: Addison-Wesley, 2011.
[41] "Add a conversion tracking tag to your website," https://support.google.com/google-ads/answer/6331314?co=ADWORDS.IsAWNCustomer, 2018, [Online; accessed 06-December-2018].
[42] "Overview of Google publisher tags," https://support.google.com/admanager/answer/181073, 2018, [Online; accessed 06-December-2018].
[43] "Exploiting jQuery HTML encoding XSS," https://stackoverflow.com/questions/31282274/exploiting-jquery-html-encoding-xss, 2018, [Online; accessed 06-July-2018].
[44] "Selector interpreted as HTML," https://bugs.jquery.com/ticket/11290, 2018, [Online; accessed 06-December-2018].
[45] "CommonJS: specifying an ecosystem for JavaScript outside the browser," https://requirejs.org/docs/commonjs.html, 2018, [Online; accessed 06-December-2018].
[46] "An open source JavaScript framework for detecting the Adobe Flash Player plugin and embedding flash files." https://github.com/swfobject/swfobject, 2016, [Online; accessed 25-November-2018].
[47] "Moment.js: Parse, validate, manipulate, and display dates and times in JavaScript," https://momentjs.com/, 2018, [Online; accessed 06-December-2018].
[48] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158–173, 2018.
[49] "Webpack: Pack your scripts," https://webpack.js.org/, 2018, [Online; accessed 06-December-2018].
[50] "SNYK XSS vulnerability report on jQuery 2.2.4," https://snyk.io/test/npm/jquery/2.2.4, 2018, [Online; accessed 06-December-2018].
[51] "YUI is a free, open source JavaScript and CSS library for building richly interactive web applications," https://yuilibrary.com/, 2018, [Online; accessed 06-December-2018].
[52] "Minimal templating with {{mustaches}} in JavaScript," , 2018, [Online; accessed 06-December-2018].
[53] J. Shapiro, J. Chase, and J. Chen, "Libscore: a web-based tool that collects statistics on JavaScript library usage," http://libscore.com/, 2014, [Online; accessed 31-December-2017].
[54] S. Lekies and M. Johns, "Lightweight integrity protection for web storage-driven content caching." Google Patents, Nov. 28 2013, uS Patent App. 13/478,991.
[55] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in HTML5 websites." 2013.
[56] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "JavaScript errors in the wild: An empirical study," in *Proceedings of the 2011 IEEE 22Nd International Symposium on Software Reliability Engineering*, ser. ISSRE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 100–109.
[57] "The Maven repository," https://mvnrepository.com/, 2018, [Online; accessed 06-December-2018].
[58] "Known vulnerabilities in Mozilla products," https://www.mozilla.org/en-US/security/known-vulnerabilities/, 2018, [Online; accessed 06-December-2018].
[59] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 133–136.
[60] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proc. IEEE*, vol. 68, no. 9, pp. 1060–1076, September 1980.