Inspecting JavaScript Vulnerability Mitigation Patches with Automated Fix Generation in Mind

Chapter · September 2020

DOI: 10.1007/978-3-030-58811-3_69

Péter Hegedűs
University of Szeged

Inspecting JavaScript Vulnerability Mitigation Patches with Automated Fix Generation in Mind*

Péter Hegedűs

MTA-SZTE Research Group on Artificial Intelligence `hpeter@inf.u-szeged.hu`

Abstract. Software security has become a primary concern for both the industry and academia in recent years. As dependency on critical services provided by software systems grows globally, a potential security threat in such systems poses higher and higher risks (e.g. economical damage, a threat to human life, criminal activity).

Finding potential security vulnerabilities at the code level automatically is a very popular approach to aid security testing. However, most of the methods based on machine learning and statistical models stop at listing potentially vulnerable code parts and leave their validation and mitigation to the developers. Automatic program repair could fill this gap by automatically generating vulnerability mitigation code patches. Nonetheless, it is still immature, especially in targeting security-relevant fixes.

In this work, we try to establish a path towards automatic vulnerability fix generation techniques in the context of JavaScript programs. We inspect 361 actual vulnerability mitigation patches collected from vulnerability databases and GitHub. We found that vulnerability mitigation patches are not short on average and in many cases affect not just program code but test code as well. These results point towards that a general automatic repair approach targeting all the different types of vulnerabilities is not feasible. The analysis of the code properties and fix patterns for different vulnerability types might help in setting up a more realistic goal in the area of automatic JavaScript vulnerability repair.

Keywords: Security · Vulnerability · JavaScript · Prediction models · Automatic repair

* The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004). Project no. 2018-1.2.1-NKP-2018-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme and partially supported by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary. Furthermore, Péter Hegedűs was supported by the Bolyai János Scholarship of the Hungarian Academy of Sciences and the ÚNKP-19-4-SZTE-20 New National Excellence Program of the Ministry for Innovation and Technology.

1 Introduction

Software security is one of the most striking problems of today's software systems. With the advent of low-cost mobile/IoT devices connected to the Internet, the problem of insecure applications has risen sharply. Large impact security vulnerabilities are explored daily, for example, a serious flaw [18] has been discovered in 'Sudo', a powerful utility used in macOS this February. Security problems can cause not just financial damage [3] but can compromise vital infrastructure, or used to threaten entire countries.

Finding potential security vulnerabilities at the code level automatically is a very popular approach to aid security testing and to help explore potential security issues before they get into the release. However, most of the methods based on machine learning and statistical models stop at listing potentially vulnerable code parts and leave their validation and mitigation to the developers. Automatic program repair could fill this gap by automatically generating vulnerability mitigation code patches. Nonetheless, it is still immature, especially in targeting security-relevant fixes.

In this work, we inspect 361 actual vulnerability mitigation patches collected from vulnerability databases and GitHub to gain some knowledge about them that can help in establishing an automatic vulnerability repair mechanism. We formulated our particular research goals into the following three research questions that we address in this paper:

RQ1: What are the basic properties (the amount of added or removed files and lines) of the vulnerability mitigation code changes in JavaScript programs?

RQ2: How do vulnerability mitigation patches affect test code?

RQ3: How frequent are those small and concise code changes among real-world vulnerability patches that could be targeted by automated code fix generation?

We found that vulnerability mitigation patches are not short on average and in many cases affect not just program code but test code as well. The results point towards that a general automatic repair approach targeting all the different types of vulnerabilities is not feasible. Most of the security fixes containing many lines would be hard to generate automatically, however, some types of vulnerabilities are mitigated with a relatively small commit containing ten lines or less. These are the ones we would first target by an automatic repair tool. The analysis of the code properties and fix patterns for different vulnerability types might help in setting up a more realistic goal in the area of automatic JavaScript vulnerability repair.

The remaining of the paper is structured as follows. Section 2 collects the related works. In Section 3 we give details about the data we analyze. Section 4 contains the discussion of results we got and the answers to the research questions. Section**??** enumerates the possible threats to the validity of our work, while we conclude the paper in Section 5.

2 Related Work

Two lines of research works are very closely related to ours. First, the prediction of software vulnerabilities based on manual or automatic features extracted from the source code or related repositories, and second, automatic program repair with a special focus on security-related bugs.

Vulnerability prediction and data sets. In their preliminary study, Siavvas et al. [16] investigated if a relationship exists among software metrics and specific vulnerability types. They used 13 metrics and found that software metrics may not be sufficient indicators of specific vulnerability types, but using novel metrics could help.

In their work, Jimenez et al. [8] proposed an extensible framework (VulData7) and dataset of real vulnerabilities, automatically collected from software archives. They presented the capabilities of their framework on 4 large systems, but one can extend the framework to meet specific needs.

Neuhas et al. [12] introduced a new approach (and the corresponding tool) called Vulture, which can predict vulnerable components in the source code, mainly relying on the dependencies between the files. They analyzed the Mozilla codebase to evaluate their approach using SVM for classification.

In their work, Shin et al. [14] created an empirical model to predict vulnerabilities from source code complexity metrics. Their model was built on the function level, but they consider only the complexity metrics calculated by Understand C++.[1] They concluded that vulnerable functions have distinctive characteristics separating them from "non-vulnerable but faulty" functions. They studied the JavaScript Engine from the Mozilla application framework. In another work Shin et al. [13] performed an empirical case study on two large code bases: Mozilla Firefox and Red Hat Enterprise Linux kernel, investigating if software metrics can be used in vulnerability prediction. They considered complexity, code churn, and developer activity metrics. The results showed that the metrics are discriminative and predictive of vulnerabilities.

Chowdhury et al. [5] created a framework that can predict vulnerabilities mainly relying on the CCC (complexity, coupling, and cohesion) metrics [4]. They also compared four statistical and machine learning techniques (namely C4.5 Decision Tree, Random Forests, Logistic Regression, and Naive Bayes classifier). The created model was accurate (with precision above 90%), but the recall was only 20% or lower which means an F-measure of 0.33 at most. The authors concluded that decision-tree-based techniques outperformed statistical models in their case.

Morrison et al. [11] built a model – replicating the vulnerability prediction model by Zimmermann et al [20] – for both binaries and source code at the file level. The authors checked several learning algorithms including SVM, Naive Bayes, random forests, and logistic regression. On their dataset, Naive Bayes and random forests performed the best.

[1] https://scitools.com/

Yu et al. introduced HARMLESS [19], a cost-aware active learner approach to predict vulnerabilities. They used a support vector machine-based prediction model with under-sampled training data, and a semi-supervised estimator to estimate the remaining vulnerabilities in a codebase. HARMLESS suggests which source code files are most likely to contain vulnerabilities. They also used Mozilla's codebase in their case study, with 3 different feature sets: metrics, text, and the combination of text mining and crash dump stack traces. The same set of source code metrics were used than that of Shin et al. [13].

Some works investigate the possible application of general fault prediction models for vulnerability prediction. Zimmermann et al. [20] argued in their work that vulnerabilities cannot be predicted as easily as defects. They used classical metrics widely used in defect prediction to see whether they can work as predictors of vulnerabilities at the binary level. They also planned to leverage specific metrics that are related to software security.

Shin et al. [15] investigated the usability of fault prediction models in the case of vulnerability prediction. They performed an empirical study on the code-base of Mozilla Firefox. The authors created 2 models: a traditional fault prediction model and a specialized, vulnerability prediction model. In their research, both models predicted vulnerabilities with high recall (90%) and low precision (9%) with F-measure around 0.16.

All the above works target file-level vulnerability prediction, while our focus is on identification and automatic repair of vulnerable JavaScript functions.

Automatic vulnerability repair. Automatic program repair research has matured a lot in the last couple of years. A lot of the challenges [9] have been solved but there are still a lot of open questions. Moreover, automatic program repair results targeting software vulnerabilities specifically are very sparse.

To assist developers to deal with multiple types of vulnerabilities, Ma et al. propose a new tool, called VuRLE [10], for automatic detection and repair of vulnerabilities. VuRLE (1) learns transformative edits and their contexts (i.e., code characterizing edit locations) from examples of vulnerable codes and their corresponding repaired codes; (2) clusters similar transformative edits; (3) extracts edit patterns and context patterns to create several repair templates for each cluster. VuRLE uses context patterns to detect vulnerabilities and customizes the corresponding edit patterns to repair them.

Smirnov and Chiueh introduce a program transformation system called DIRA [17] that can automatically transform an arbitrary application into a form that (1) can detect a control-hijacking attack when the control-sensitive data structure it tampers with is activated; (2) can identify the network packets that lead to the control-hijacking attack, and send these packets to a front-end content filter to prevent the same attack from compromising the application again, and (3) can repair itself by erasing all the side effects of the attack packets as if it never received them.

Gao et al. [7] present BovInspector, a tool framework for automatic static buffer overflow warnings inspection and validated bugs repair. Experimental results on real open source programs show that BovInspector can automatically inspect on average of 74.9% of total warnings, and false warnings account for about 25% to 100% (on average of 59.9%) of the total inspected warnings. In addition, the automatically generated patches fix all target vulnerabilities.

All of the above tools, however, focus primarily on general techniques and/or evaluate the approach on a limited number of particular security issues for general-purpose OO languages (Java and C++, respectively). We aim to explore the vulnerability fixes for JavaScript programs and to discover the possibilities of an automated vulnerability repair mechanism that could target many types of security issues.

3 Background

To inspect real-world JavaScript vulnerability mitigation patches, we had to collect them using data mining. We reused the processing tool-chain, data sources, and intermediate data files published in our previous work [6], where we built vulnerability prediction models based on the collected data using static source code metrics as predictors. We were able to collect 361 actual vulnerability fixing code changes from over 200 different JavaScript projects. The high-level process of collecting actual vulnerability patches is depicted in Figure 1.

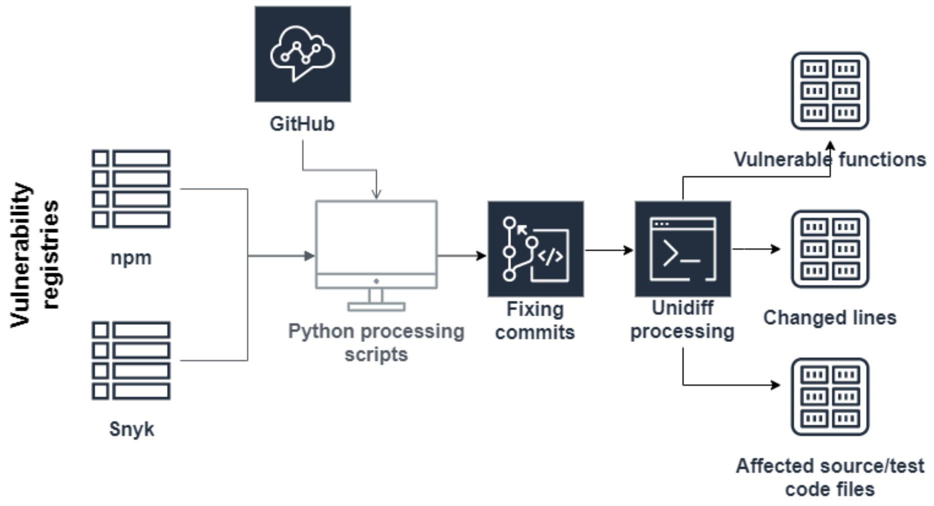

Fig. 1. Overall process of the data collection

We leveraged two publicly available vulnerability databases, nsp (the Node Security Platform, which is now part of npm) [1] and the Snyk Vulnerability Database [2]. Both of these projects aim to analyze programs for vulnerable third-party module usages. The main issue with these extracted raw vulnerability sources is that they contain unstructured data. The entries include a human-readable description of vulnerabilities with URLs of fixing commits, pull requests, or issues in GitHub or other repositories. However, these URLs are somewhat arbitrary, they can appear on multiple places within the entries and any of them might be missing entirely. To handle this, we wrote a set of Python scripts to process these vulnerability entries and create an internal augmented and structured representation of them.

Once we collected all the vulnerability fixing commits, we analyzed these unidiff format patches to extract added, removed, and modified files as well as added/removed program and test lines of code. We applied a simple heuristic to find test code, we searched for the term "test" in the path or name of the file or in the name of JavaScript function a code change affects. We also apply sophisticated mapping to be able to tell which functions are changed by the vulnerability fix. Therefore, we are not only able to tell how many lines of code is changed, but also how many and what particular functions are affected by the security fix. All the data we used for answering our research questions is available online.[2]

4 Results

4.1 RQ1: Basic Properties of Vulnerability Fixing Patches

Table 1 displays the descriptive statistics of the 361 real vulnerability mitigation patches we analyzed.

Table 1. Descriptive statistics of the vulnerability fixing patches

	Min	**Max**	**Median**	**Average**	**Std.dev.**
Added files	0	93	0	1.02	8.61
Modified files	0	132	2	3.88	12.72
Removed files	0	7	0	0.11	0.58
Added lines	0	7 521	10	138.53	771.55
Removed lines	0	3 205	4	74.17	315.70
All changed lines	0	9 179	15	212.69	978.27
Affected functions	0	466	2	10.48	40.01

At the file level, security fixes in JavaScript programs mostly modify existing files. The mitigation patches we analyzed contained almost 4 file modifications on average, but only around 1 file addition. File removal was very rare.

The number of line additions is twice as large as line removals. Note, that the unidiff file format marks line modifications as a line removal plus a line addition, thus some of the additions and removals together form line modifications. However, the average of the added lines by a patch is almost 140, while the removed line average is less than 75. Therefore, security fixes contain a lot of added lines

[2] https://doi.org/10.5281/zenodo.3767909

together with some modifications and removals. It is also important to note that the average changed lines (any additions, removals, and modifications altogether) per vulnerability fixes are 212.69. So quite many lines are affected by fixes on average. However, as we can see, the standard deviation of these values is extra large as well, almost one thousand. Additionally, the median of the values is 15 only (half of the patches change 15 lines or less only). These suggest that far fewer lines are changed in most of the fix patches, but several very huge changes affect the average.

We also mapped the code change patches to actual JavaScript functions to see how many different functions are affected in the vulnerability fixes at the logical level. The same trend is true for the changed lines. On average, more than 10 functions are changed in the course of a vulnerability fix. However, the median of such values is only 2. Therefore, most of the patches affect a very small number of functions, with some patches containing an extremely large number of modified functions.

4.2 RQ2: Vulnerability Fixing Patches and Test Code

Table 2 contains the same descriptive statistics for the patches from the test code's perspective. The trends are similar, but the magnitude of values is one fourth that of the program code.

Table 2. Descriptive statistics of the vulnerability fixing patches

	Min	**Max**	**Median**	**Average**	**Std.dev.**
Added test files	0	4	0	0.15	0.55
Modified test files	0	14	0	0.68	1.40
Removed test files	0	1	0	0.01	0.12
Added test lines	0	1 027	0	32.81	108.35
Removed test lines	0	242	0	7.91	32.04
All changed test lines	0	1 269	0	40.72	135.60
Affected test functions	0	63	0	4.55	9.76

Figure 2 provides further details on the distribution of the changed program and test code lines. We can see the average number of changed code and test lines by the vulnerability fixing patches grouped by the CWE categories of the vulnerabilities. The vulnerability databases enumerated the associated CWE categorizations together with the vulnerability descriptions, therefore we could mine this information as well. It is noticeable that there are some vulnerability types where fixes do not introduce test code changes at all. However, where fixing patches do contain test code modifications, their magnitude is comparable to that of the program code's modifications on average.

4.3 RQ3: Possible Targets for Automated Fix Generation

In our previous work [6], we showed that it is possible to use machine learning models to predict JavaScript vulnerabilities effectively based on source code

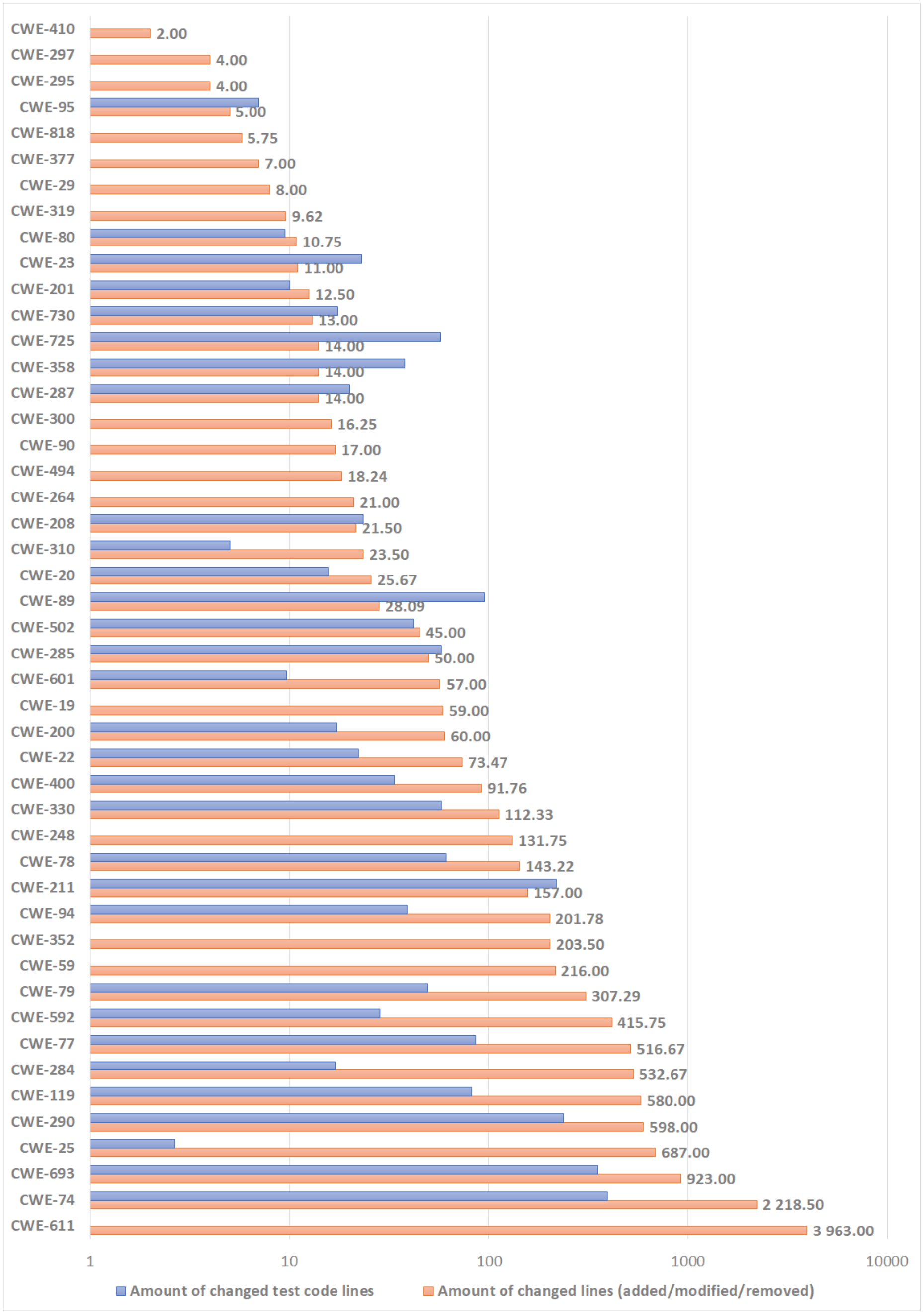

Fig. 2. The average number of changed lines in program and test code within the mitigation patches, grouped by CWE types

analysis results. In this work, we investigated how feasible it would be to generate also fixes for the detected vulnerabilities in an automated way, i.e. applying automated program repair techniques. Since the state-of-the-art of such techniques is still immature and efficient automatic patch generation works only for small code fragments, we inspected how many of the real-world vulnerability fixes are small and concise, thus being a good candidate for automatic repair.

According to Figure 2, there are only 8 different CWE groups where the vulnerabilities belonging to them have fixes with 10 or less changed lines. Vulnerabilities of these types are definitely to best candidates for building automated repair techniques for fixing them. However, it is also evident, that there are much more short vulnerability fixes, so grouping them based on their CWE categories might not be the best strategy. Therefore, we plotted the number of vulnerabilities by CWE groups together with the number of fixes containing 10 or fewer lines of code changes. As can be seen in Figure 3, a considerable proportion of each CWE group contains vulnerabilities with short fixing patches. Thus, automated patch generation could be extended to such fixes.

Going a bit deeper, we also examined a couple of vulnerability fixing patches with such short code-changes manually. Our goal was to evaluate the feasibility of automatic fix generation at the semantic level. It is not enough that a code we need to generate is short, but it should be such that its semantics is not overly complicated to apply automatic repair techniques for it.

Vulnerability fix in the Node.js WebSocket project.[3] A vulnerability of type CWE-410: Insufficient Resource Pool is fixed in one line (see Listing 4.1), changing the value of *maxPayload* from *null* to *100 * 1024 * 1024*. This is a simple, straightforward change and the occurrance of “null” is easy to detect. However, to come up with a proper payload value automatically would be a hard task for an automatic repair tool.

```
--- lib/WebSocketServer.js
+++ lib/WebSocketServer.js

@@ -37,7 +37,7 @@ function WebSocketServer(options, callback)
     {
    disableHixie: false,
    clientTracking: true,
    perMessageDeflate: true,
-   maxPayload: null
+   maxPayload: 100 * 1024 * 1024
  }).merge(options);

  if (!options.isDefinedAndNonNull('port') && !options.
      isDefinedAndNonNull('server') && !options.value.
      noServer) {
```

Listing 4.1. Vulnerability fix in Node.js WebSocket

[3] npm:ws:20160624

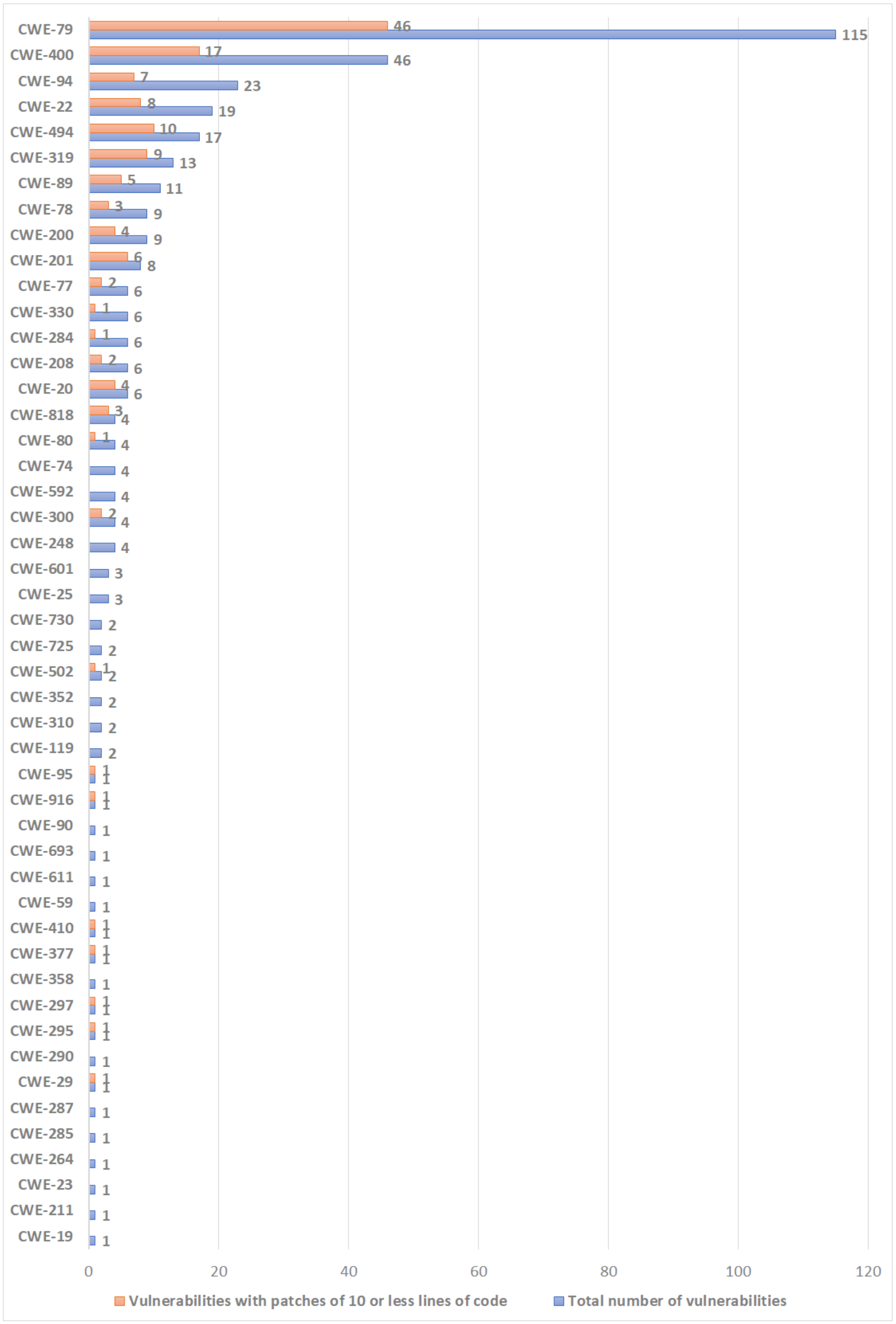

Fig. 3. Number of total vulnerabilities and mitigation patches with 10 lines or less, grouped by CWE categories

Vulnerability fix in the LinkedIn dust.js project.[4] A vulnerability of type CWE-95: Improper Neutralization of Directives in Dynamically Evaluated Code ('Eval Injection') is fixed by extending a simple conditional check with an additional condition (see Listing 4.2). These types of fixes are common in automatic repair for regular software bugs, thus they should be adapted to the vulnerability domain as well.

```
--- lib/dust.js
+++ lib/dust.js

@@ -851,7 +851,10 @@
      SQUOT  = /\'/g';

  dust.escapeHtml = function(s) {
-   if (typeof s === 'string') {
+   if (typeof s === "string" || (s && typeof s.toString === "function")) {
+     if (typeof s !== "string") {
+       s = s.toString();
+     }
      if (!HCHARS.test(s)) {
        return s;
      }
```

Listing 4.2. Vulnerability fix in LinkedIn dust.js

Interestingly, this fix includes changes in the test code. The added condition is tested with a new test case as shown in Listing 4.3.

```
--- test/jasmine-test/spec/coreTests.js
+++ test/jasmine-test/spec/coreTests.js

@@ -688,6 +688,13 @@ var coreTests = [
              },
    expected: "1",
    message: "should test using a multilevel reference as a key in array access"
+       },
+       {
+   name: "Outputting an array calls toString and HTML-encodes",
+   source: "{array}",
+   context: { "array": ["You & I", " & Moe"] },
+   expected: "You &amp; I, &amp; Moe",
+   message: "should HTML-encode stringified arrays referenced directly"
    }
  ]
},
```

Listing 4.3. Added test case for the vulnerability fix in LinkedIn dust.js

[4] npm:dustjs-linkedin:20160819

Therefore, we should think of automatically generating new test cases upon automatic vulnerability fix generation. During future research, we have to put some effort into finding out which fixes are those that require such test case extensions.

Vulnerability fix in the Chrome driver project.[5] A vulnerability of type CWE-319: Cleartext Transmission of Sensitive Information is fixed by changing "http" to "https" (see Listing 4.4). The CWE-319 group of vulnerabilities appears to be a perfect candidate for generating fixes automatically. If we could find a method that identifies occurrences of "http" in contexts that are used as actual addresses, we could easily propose a fix automatically. In our future research, we will focus first on these vulnerability groups.

```
--- install.js
+++ install.js

@@ -15,7 +15,7 @@ var url = require('url')
  var util = require('util')

  var libPath = path.join(__dirname, 'lib', 'chromedriver')
- var cdnUrl = process.env.npm_config_chromedriver_cdnurl ||
    process.env.CHROMEDRIVER_CDNURL || 'http://chromedriver.
    storage.googleapis.com'
+ var cdnUrl = process.env.npm_config_chromedriver_cdnurl ||
    process.env.CHROMEDRIVER_CDNURL || 'https://chromedriver.
    storage.googleapis.com'
  // adapt http://chromedriver.storage.googleapis.com/
  cdnUrl = cdnUrl.replace(/\/+$/, '')
  var downloadUrl = cdnUrl + '/%s/chromedriver_%s.zip'
```

Listing 4.4. A vulnerability fix in Chrome driver

5 Conclusions

In this paper, we inspected 361 real-world fixes of vulnerabilities explored in JavaScript programs. Our focus was on analyzing the various properties of such security fixing patches to aid further research on automatically generating such patches.

We found that security fixes in JavaScript programs mostly modify existing files and that the number of line additions and/or modifications overweight line removals. We also investigated if the test code is changed during vulnerability fixes. Although the overall amount of test code changed is significantly smaller than that of the program code, we observed that if a patch contains test code modifications, its magnitude is similar to the program code changes. That is, there are types of security issues that are fixed without test code adjustments, but for some security types, test code modifications are really important.

[5] npm:chromedriver:20161208

Finally, we examined the patches from the perspective of automatic repair techniques. In general, it would be hard to generate potential vulnerability fixes as the size of code changes and their types vary significantly. However, we showed that at least half of the fixes contain 15 or fewer code lines, which can be handled by automatic repair tools, but the semantic complexity of the changes is also important. We identified a couple of good candidates for starting with automatic security fix generation, like that of the vulnerabilities from the CWE-319 group.

A lot of further research is needed in this area, but we think our small scale empirical investigation contains some useful insights and guidance of further directions. We plan to develop automatic repair techniques for the abovementioned vulnerability categories and evaluate their performance.

References

1. Node Security Platform - GitHub. https://github.com/nodesecurity/nsp, Accessed: 2018-10-16
2. Vulnerability DB — Snyk. https://snyk.io/vuln, Accessed: 2018-10-16
3. Anderson, R., Barton, C., Böhme, R., Clayton, R., van Eeten, M.J.G., Levi, M., Moore, T., Savage, S.: Measuring the Cost of Cybercrime, pp. 265–300. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39498-0_12
4. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Transactions on software engineering **20**(6), 476–493 (1994)
5. Chowdhury, I., Zulkernine, M.: Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. Journal of Systems Architecture **57**(3), 294–313 (2011)
6. Ferenc, R., Hegedűs, P., Gyimesi, P., Antal, G., Bán, D., Gyimóthy, T.: Challenging machine learning algorithms in predicting vulnerable javascript functions. In: Proceedings of the 7th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. pp. 8–14. IEEE Press (2019)
7. Gao, F., Wang, L., Li, X.: BovInspector: automatic inspection and repair of buffer overflow vulnerabilities. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 786–791 (2016)
8. Jimenez, M., Le Traon, Y., Papadakis, M.: Enabling the Continous Analysis of Security Vulnerabilities with VulData7. In: IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 56–61 (2018)
9. Le Goues, C., Forrest, S., Weimer, W.: Current challenges in automatic software repair. Software quality journal **21**(3), 421–443 (2013)
10. Ma, S., Thung, F., Lo, D., Sun, C., Deng, R.H.: Vurle: Automatic vulnerability detection and repair by learning from examples. In: Foley, S.N., Gollmann, D., Snekkenes, E. (eds.) Computer Security – ESORICS 2017. pp. 229–246. Springer International Publishing, Cham (2017)
11. Morrison, P., Herzig, K., Murphy, B., Williams, L.A.: Challenges with applying vulnerability prediction models. In: HotSoS (2015)
12. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 529–540 (01 2007)

13. Shin, Y., Meneely, A., Williams, L., Osborne, J.A.: Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. IEEE Trans. Softw. Eng. **37**(6), 772–787 (Nov 2011)
14. Shin, Y., Williams, L.: An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement. pp. 315–317. ACM (2008)
15. Shin, Y., Williams, L.A.: Can traditional fault prediction models be used for vulnerability prediction? Empirical Software Engineering **18**, 25–59 (2011)
16. Siavvas, M., Kehagias, D., Tzovaras, D.: A preliminary study on the relationship among software metrics and specific vulnerability types. In: 2017 International Conference on Computational Science and Computational Intelligence – Symposium on Software Engineering (CSCI-ISSE) (12 2017)
17. Smirnov, A., Chiueh, T.c.: Dira: Automatic detection, identification and repair of control-hijacking attacks. In: NDSS (2005)
18. Sudo vulnerability in macOS (2020), https://www.techradar.com/news/linux-and-macos-pcs-hit-by-serious-sudo-vulnerability
19. Yu, Z., Theisen, C., Sohn, H., Williams, L., Menzies, T.: Cost-aware vulnerability prediction: the HARMLESS approach. CoRR **abs/1803.06545** (2018)
20. Zimmermann, T., Nagappan, N., Williams, L.: Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In: 2010 Third International Conference onSoftware Testing, Verification and Validation (ICST). pp. 421–428. IEEE (2010)