Fix that Fix Commit: A real-world remediation analysis of JavaScript projects

Vinuri Bandara[†‡], Thisura Rathnayake[†‡], Nipuna Weerasekara[†‡],
Charitha Elvitigala[§†], Kenneth Thilakarathna[‡], Primal Wijesekera[‡‡], Chamath Keppitiyagama[‡]
[§]Bug Zero, [†]SCoRe Lab, [‡]University of Colombo School of Computing, [‡‡]University of California, Berkeley
{vinurib, thisura, w.nipuna}@scorelab.org,
charitha@bugzero.io, primal@berkeley.edu, {kmt, chamath}@ucsc.cmb.ac.lk

***Abstract*—While there is a large body of work on understanding vulnerabilities in the wild, little has been done to understand the dynamics of the remediation phase of the development cycle. To this end, we have done a timeline analysis on 118K commits from 53 of the most used JavaScript projects from GitHub to understand the provenance and prevalence of vulnerabilities in those projects. We used a vulnerability detector (CodeQL) to filter commits that introduced vulnerabilities and the commits that fixed a prior vulnerability. We found that in 82% of the projects, a commit fixing a prior vulnerability, in turn, introduced one or more new vulnerabilities. Among those projects, on average, 18% of the commits intended to fix vulnerabilities, in turn, introduced one or more new vulnerabilities. We also found that 50% of the total vulnerabilities found in those projects originated from a commit meant to fix a prior vulnerability, and 78% of those vulnerabilities could have been avoided if they were to use proper internal testing. We provide critical insights into how proper internal testing can avoid a significant portion of vulnerabilities, increasing organizations' security posture.**

***Index Terms*—Software Security, Vulnerability Analysis, Vulnerability Remediation, Security Testing**

I. Introduction

Despite recent advancements in vulnerability detection tools and best practices, a stream of data breaches and exploits have been observed. We believe that the real reason relies not only on vulnerability discovery techniques but also on vulnerability remediation's success. Public resources such as MITRE Corporation's CVE [2] database and the National Vulnerability Database (NVD) [6] are providing valuable references and resources for a successful remediation phase. The question, however, remains whether developers are making the most of those public resources.

Recent literature on vulnerability remediation found that there are severe issues in the remediation phase that nullify any vulnerability discovery progress to find security vulnerabilities [10], [16]. Li et al. found that not all vulnerability fixes fix the intended vulnerability; it takes more than one commit to fix the vulnerability. Alomar et al. found that many leading industry security practitioners are worried that a host of reasons impede successful vulnerability remediation, and lack of proper attention from the management and lack of developer knowledge to fix a vulnerability are leading causes.

The literature on vulnerability analysis lacks any work on understanding this issue in the real world, and except for very minimal work, the research community is only waking up to learn these issues in the wild. As a research community, we need a proper measurement study before exploring potential solutions to this issue. To that end, we propose a novel automated vulnerability analysis to filter commits responsible for introducing vulnerable codes and filter commits that fix existing vulnerabilities in the code.

We analyzed 53 JavaScript projects found on GitHub. We based the study on JavaScript because of its popularity among developers[1], but the proposed framework is language agnostic. The proposed pipeline can longitudinally analyze a project's vulnerability introduction and fixing patterns as time-series analysis. This gives a holistic view of how projects manage vulnerabilities and the efficacy of vulnerability remediation.

We found that in 82% of the analyzed projects, a fixing vulnerability has introduced more vulnerabilities to the code one-fifth of the time. Overall, 78% of the found vulnerabilities were publicly disclosed at the introduction; hence a naive internal scan would have prevented them from the code-base. The proposed work contributes to the following:

1) A novel technique to longitudinally analyze vulnerability management in projects.
2) To the best of our knowledge, we provide the first analysis of quantification of vulnerability remediation issues at scale.
3) Outline the importance of proper internal testing to avoid a significant portion of vulnerabilities and increase the overall security of the project.

II. Related Work

In one of the early works, studying vulnerability analysis, Frei et al. discuss the lifecycle of a vulnerability from a security researcher's or a malicious actor's perspective [11]. They state that the vulnerability discovery date is mostly unknown to the public, and the vulnerabilities exist before being discovered. The issue here for the developers is that they have to discover the vulnerabilities before the malicious actors can discover it. However, our data suggest that the landscape has changed significantly now that a sizable portion of issues could have been prevented if they were to use proper testing. Prechelt et al. also suggested that the current state of

[1]https://insights.stackoverflow.com/survey/2020

the art repository mining techniques are not adequate to study vulnerability inducing commits [22].

Source code analysis to detect vulnerabilities have been studied in the past [12], [14], [20]. These authors have proposed methods to identify vulnerabilities in the codebase using patterns described in MITRE Corporation's Common Weakness Enumeration. Another related project has discussed the challenges of finding commits pushing vulnerabilities using the mining data from source control repositories [13]. However, our methodology goes one step beyond to understand specific security vulnerabilities introduced in each commit.

Version controlling data is used to analyze the changes in a codebase and find the vulnerability introductions [9], [15], [20]. The commit log and the commit messages are also mined to understand the intention and if the commit has fixed any bugs. The work by Peguero et al. [19] also uses the source code and version controlling data for identifying the cross-site scripting vulnerabilities in JavaScript frameworks. We are using the Git version controlling system and the GitHub platform to mine the commit data such as commit message, commit author, and commit hash of a specific repository. However, we are not focusing on the commit messages to find whether the commit has done any fixing of vulnerability or not. We are focusing on the CodeQL analysis result for a specific commit version to identify whether the commit has fixed a vulnerability or the commit has introduced a vulnerability.

Prior work has looked into the remediation phase both in general software engineering [24], [26] and in specific software security domains [16], [17], [21]. Meneely et al. have studied commits that have introduced vulnerabilities and likely reasons behind those vulnerabilities, such as lack of knowledge [17]. Li et al. have conducted a large scale study on security patches and found that not all patches have fixed the vulnerability, and there is a significant lapse of time between public disclosure of a vulnerability and the time it took to fix the said vulnerability [16]. In both cases, their findings highlight the need for an effective vulnerability remediation.

Recent work on vulnerability pipeline in organizations has also found that many organizations have focused more on vulnerability discovery but less on the remediation, which is a grave concern [10]. They found that management is likely to have a false sense of security without proper remediation. We like to measure this observation's prevalence in the wild by analyzing code and commits in public repositories.

III. Methodology

Our main objective is to understand the flaws in vulnerability remediation through source code and repository analysis. We detail our methodology in this section.

A. Selection of JavaScript repositories

We selected the repositories with at least 500 commits since the analysis needs a substantial list of commits for a particular repository. The minimum threshold enables us to carry out longitudinal analysis of the commits. We randomly selected 53 JavaScript repositories from combining three publicly available lists that list most dependent upon JavaScript packages[2][3][4]. Each of these repositories has more than two authors and has a substantial amount of current users. We are in the process of analyzing all the projects mentioned in those resources, but in this paper, we present the first 53 projects analyzed so far.

B. Analysing GitHub Commits

Vulnerabilities in a codebase can be analyzed either by manual inspection or by automated inspection [12]. For JavaScript applications, detecting vulnerabilities can be done based on specific patterns. Such patterns can be found in Open Web Application Security Project [7] and the Common Weakness Enumeration (CWE) [3]. For instance, by analyzing the codebase, the developers or testers can identify specific Cross-Site Scripting (XSS) attacking vectors, sources, and sinks.

Since the manual analysis for vulnerabilities is inefficient and not scalable, we used CodeQL [1], which is an open-source tool distributed by GitHub. It is an automated variant analysis tool that identifies the variants of security vulnerabilities and generic software bugs in languages such as JavaScript, TypeScript, Python, C, C++, C#, and Go. CodeQL is based on the QL [8] query language, and it uses the known vulnerabilities as seeds to find similar issues in the codebase. Using CodeQL, we can list the vulnerabilities in a repository at a particular point in time. Therefore, we run CodeQL for all the commits found in each of the 53 JavaScript repositories. For each of these commits, we then use the GitHub API [4] to extract its date, commit message, commit hash and the author's details (name, username and email).

C. Security vulnerability extraction

The result from the above step contains all the issues that CodeQL identified by analyzing a given commit and an example of such an issue is shown in Figure 1. The identified issue in Figure 1 is a *Invalid prototype value*. We need to identify which of the CodeQL issues has an associated CWE reference number – only the security vulnerabilities have a CWE reference. The CodeQL references were scraped from its website [5] and stored in a JSON file. Listing 1 shows the respective CWE reference related to *Invalid prototype value*. We also downloaded the CWE database[5] to obtain the published date of a given CWE. From here onwards, vulnerability means any issue CodeQL detected with one or more CWE references.

Intro commit is the commit CodeQL observes a vulnerability for the first time in a codebase. It is the commit responsible for the vulnerable code. We define *Fix commit* as the commit responsible for fixing a given vulnerability. We developed an Automated Analysis Pipeline, which integrates all the steps starting from getting all git commits up to analyzing using

[2]https://www.npmjs.com/browse/depended
[3]https://gist.github.com/anvaka/8e8fa57c7ee1350e3491
[4]http://bit.ly/hackernoon-36
[5]https://cwe.mitre.org/data/csv/699.csv.zip

Name	Invalid prototype value
Description	An attempt to use a value that is not an object or 'null' as a prototype will either be ignored or result in a runtime error.
Severity	error
Message	Values of type number cannot be used as prototypes.
Path	/index.js
Start Line	100
Start Column	15
End Line	100
End Column	30

Fig. 1: An example output from CodeQL analysis

```
{"link": "https://help.semmle.com/wiki/display
    /JS/Invalid+prototype+value",
"name": "Invalid prototype value",
"description": "An attempt to use a value that
     is not an object or 'null' as a prototype
     will either be ignored or result in a
    runtime error.",
"id": "js/invalid-prototype-value",
"kind": "problem",
"severity": "error",
"precision": "high",
"recommendation": "Fix the prototype
    assignment by providing a valid prototype
    value.",
"references": ["CWE-704"]}
```

Listing 1: A CodeQL vulnerability with its CWE reference

CodeQL and finally saving the results as CSV files. The algorithm is shown in Algorithm 1.

Algorithm 1: Analysis Pipeline

Data: JavaScript repositories
Result: CSV output file of CodeQL analysis
1. Select JavaScript repository
2. Get all the Git commit hashes of the selected repository
3. Get all the commit author data from the selected repository
for *commit in Commits* **do**
    1. Checkout the commit
    2. Create CodeQL database for the checked out version of the repository
    3. Analyse the created CodeQL database using the CodeQL tool
    4. Select all CodeQL issues with a CWE reference
    5. Get the published date for the given CWE from the CWE database
    6. Write the result file into a CSV file

IV. Analysis

We have analyzed 53 JavaScript projects using our automated analysis pipeline. The pipeline has processed 118,023 commits from those 53 projects. CodeQL found 5,046 security vulnerabilities among those 118K commits. CodeQL categorizes severity of vulnerabilities into errors, warnings and recommendations. Out of the commits analyzed, we found 541 *errors*, 4,231 *warnings*, and 274 *recommendations*. At the time of analysis, on average, there were 10 unresolved security weaknesses in all of the projects – there were, on average, 95 security weaknesses found in each project. We observed that, on average, developers had taken 88 days to fix a security weakness. In the remainder of this section, we present a finer-grained analysis highlighting inefficacies found in vulnerability remediation on JavaScript projects.

The focus of this study is to highlight the importance of source code analysis for understanding the efficacy and current status of vulnerability remediation. Prior work has filtered security patches based on commit messages and analyzed higher-level factors such as their success rate, time taken to react to a vulnerability, etc [16]. Our novel methodology, however, lets us collect data at a finer-grained level to understand better insights and gain a holistic view.

A. Bad Fixes

At the heart of the vulnerability remediation is fixing a vulnerability. The success of the entire vulnerability analysis of an organization depends on how quickly and successfully a given vulnerability is fixed. Alomar et al. [10] found that many industry experts have voiced concerns about the lack of success in fixing bugs due to a variety of reasons such as the lack of knowledge on fixing a given vulnerability, staffing or budget issues, lack of proper attention to remediation, and third-party dependencies.

To quantify this argument, we used our novel automated pipeline to filter out commits that have fixed a vulnerability and to understand the success of the remediation. *Fix commit* is a commit that is responsible for removing a vulnerability from the code base; hence we believe that one of the main objectives of that commit is to fix a prior vulnerability. Li et al. found that in some cases, there multiple attempts to fix a vulnerability [16]. In our analysis, we only count the final attempt that fixed the vulnerability ignoring prior unsuccessful attempts. Hence, the numbers we present could be a lower bound to the issues we raise.

B. Developer's Role

A critical question in this issues is to understand factors that push developers to introduce new vulnerabilities in the process of fixing another vulnerability. Answering that question requires significant working hours to analyze code changes. Understanding why a fix went wrong is also answering the question of developers' role in this eco-system. There is a rich line of work on understanding developers, security testers, security advice on the internet [23], [25]. Literature lacks work on understanding why developers fail to fix vulnerabilities.

We, however, manually inspected ten randomly picked *intro+fix commits*. When manually inspecting these ten commits, we identified seven commits where the author has fixed the

vulnerability, and, part of that fix has introduced another vulnerability because the author has either renamed the file containing the vulnerability or refactored the vulnerable code to another file. Upon inspecting two other commits, which from the random sample of intro+fix commits, we identified an instance where the author tried to fix it by moving the vulnerable piece of code from the initially identified vulnerable file to another file as a modified function. Which, in return, resulted in another vulnerability. The remaining commit contained no relationship between its fix and the newly introduced vulnerability. While these commits won't give us a the full picture, this manual analysis shows that inability to properly fix a vulnerability could be tip of the ice among host of security related developer issues.

Our automated pipeline can delve into more developer centric statistics on this issue. Based on our analysis, we found that in some projects, as much as 5% of developers are responsible for *intro+fix commits* introducing more vulnerabilities. Out of all the *intro+fix commits*, 80% of the time, the *intro commit* and the *intro+fix commit* that was supposed to fix the original vulnerability were pushed by the same developer. This is a good sign in terms of developers taking ownership of the code that caused the vulnerability in the first place. However, further investigation is required to better understand why they could not fix the vulnerable code properly, fixing their own mistakes. Among the developers responsible for *intro+fix commits*, 6.8% of them have pushed such code changes to multiple projects – it is imperative to take necessary steps to understand this phenomenon and find approaches to help developers to write secure code.

C. Security Posture

Alomar et al. found that many organizations lack a properly defined vulnerability remediation framework [10]. Such a framework would help testers and triagers of vulnerability reports to prioritize new vulnerabilities to fix them based on business requirements and predefined set of rules. They have also mentioned that the lack of proper internal testing before production or, bug bounty program, would make testing approach less effective.

We compared the published data of CWEs and the date of the *intro commit* that pushed the vulnerable code to check whether the vulnerable code could have been avoided. We found that 78% of the time, developers have pushed a publicly-disclosed vulnerability to the codebase. We also found that, on average, projects have repeated 75% of vulnerabilities. This reiterates the above fact that even naive internal testing could have avoided a significant portion of vulnerabilities. From a software engineering perspective, proper development tools could have also helped the developer find those weaknesses before pushing the changes to avoid costly security testing or costlier security breaches.

We also found that 50% of the time, developers pushed a fix while there were other high severity weaknesses in the codebase. While this could happen due to many legitimate reasons such as high severity weaknesses could take longer to fix or they might have followed the chronological order of the weaknesses. However, if none of the above two reasons are applicable, then the organizations should have a mechanism to prioritize weaknesses to fix.

V. Discussion

This study's main objective is to showcase the feasibility of using the source code analysis to answer some of the critical security questions at the intersection of software engineering and software security. Based on the results presented in this paper, we show that issues recently raised in the literature on vulnerability remediation can be quantified. This quantification has many implications on software security, software development tools, and internal security testing. We present a brief discussion on future research avenues along those lines.

Recently Alomar et al. found that vulnerability remediation not only lacks proper attention but also can mute any progress made in vulnerability discovery techniques [10]. Their study participants mentioned that fixing a vulnerability could be as hard as finding them for reasons such as lack of knowledge, lack of details with the vulnerability report, etc. Our quantification of the remediation process has shown that fixing a given vulnerability one-fifth of the time has gone from bad to worse by introducing more vulnerabilities to the codebase. The literature lacks any quantifications on this issue, and lack of data makes it hard to prove this issue and encourage developers to be more secure in development.

Our novel source code analysis pipeline can provide interesting insights into other aspects of the above-mentioned vulnerability remediation issue. Source code analysis will reveal more insights into hidden patterns behind failed vulnerabilities fixes. While we only manually examined ten *intro+fix commits*, more work is needed to understand patterns and possibly figure out the reasons behind those failures. In seven out of ten manually-analyzed *intro+fix commits*, the code supposed to fix a prior vulnerability has directly contributed to a new set of vulnerabilities.

In the manual analysis, we found out that seven out of ten intro+fix commits have been created because the commit author has changed the file name or has refactored the code. The vulnerability in one file is identified as fixed, and a new vulnerability is introduced in the new file. Moving forward, we have to devise ways to detect code refactoring automatically. That will reduce incorrect labeling of commit fixes.

Understanding finer grained failure patterns in the code will help to understand developer failure. Prior work has looked into writing secure code and getting security to advise on the web [23], [25]. Our work brings out a different avenue on understanding why developers fail to fix vulnerabilities. It is an important question to investigate to find a practical solution to the remediation issues. Source code analysis will be at the center in understanding current failure patterns before reaching out to developers to understand their side of the stories.

Understanding the impact of using proper tools to avoid pushing buggy code is important. The literature on software security is filled with new security vulnerabilities, but little has

been studied from the software development side on how to deploy these techniques to help developers to write secure code and avoid vulnerabilities in the first place. We found that 78% of all discovered security vulnerabilities were publicly known at the time of the commit. If the developers had proper tools to scan the code changes, costly release fixes or data breaches could have been avoided.

Another critical aspect of the source code analysis is understanding and quantifying the security posture of an organization. The capability maturity model in software engineering defines different levels based on the maturity of the software development processes [18]. We envision quantifiable security issues we uncover from source code analysis can be used to define a maturity model for an organization. Observations such as lack of proper internal security testing, lack of processes to avoid the repeated occurrence of the same vulnerability, and proper developer training can be used to understand and potentially rate the organizations for their maturity.

We believe that the public has much to gain from our automated analysis; hence we have opened our analysis to the public through a real-time dashboard displaying our analysis[6]. We envision this to grow to cover a significant portion of public repositories; hence, the public can be informed about many public repositories' security posture. We also envision incorporating developer statistics per user and project, giving developers the incentive to be conscious in writing code.

We presented a novel automated pipeline to uncover *Intro commits*, and *Fix Commits*. This approach can be used to answer many timely-needed research questions such as quantifying vulnerability remediation issues, understanding remediation failures, understanding developer failures in fixing vulnerabilities, etc. We believe most of these new directions will bring the research community a step closer to help organizations to have a secure development culture.

REFERENCES

[1] "CodeQL for research." [Online]. Available: https://securitylab.github.com/tools/codeql

[2] "Common Vulnerabilities and Exposures." [Online]. Available: https://cve.mitre.org/index.html

[3] "Common Weakness Enumeration." [Online]. Available: https://cwe.mitre.org/

[4] "GitHub REST API." [Online]. Available: https://docs.github.com/en/rest

[5] "JavaScript queries." [Online]. Available: https://help.semmle.com/wiki/display/JS/

[6] "National Vulnerability Database." [Online]. Available: https://nvd.nist.gov/

[7] "OWASP top ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[8] "QL language." [Online]. Available: https://help.semmle.com/QL/ql-handbook/about-the-ql-language.html

[9] M. Alohaly and H. Takabi, "When do changes induce software vulnerabilities?" in *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 2017, pp. 59–66.

[10] N. Alomar, P. Wijesekera, E. Qiu, and S. Egelman, ""you've got your nice list of bugs, now what?" vulnerability discovery and management processes in the wild," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, Aug. 2020, pp. 319–339. [Online]. Available: https://www.usenix.org/conference/soups2020/presentation/alomar

[11] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense*, ser. LSAD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 131–138. [Online]. Available: https://doi.org/10.1145/1162666.1162671

[12] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 144–156. [Online]. Available: https://doi.org/10.1145/2950290.2950308

[13] K. Hogan, N. Warford, R. Morrison, D. Miller, S. Malone, and J. Purtilo, "The challenges of labeling vulnerability-contributing commits," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 270–275.

[14] N. Imtiaz, B. Murphy, and L. Williams, "How do developers act on static analysis alerts? an empirical study of coverity usage," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, 2019, pp. 323–333.

[15] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '06. USA: IEEE Computer Society, 2006, p. 81–90. [Online]. Available: https://doi.org/10.1109/ASE.2006.23

[16] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[17] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 65–74.

[18] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability maturity model, version 1.1," *IEEE software*, vol. 10, no. 4, pp. 18–27, 1993.

[19] K. Peguero, N. Zhang, and X. Cheng, "An empirical study of the framework impact on the security of javascript web applications," in *Companion Proceedings of the The Web Conference 2018*, ser. WWW '18. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2018, p. 753–758. [Online]. Available: https://doi.org/10.1145/3184558.3188736

[20] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[21] V. Piantadosi, S. Scalabrino, and R. Oliveto, "Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 68–78.

[22] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study," *Information and Software Technology*, vol. 56, no. 10, pp. 1377–1389, 2014.

[23] E. M. Redmiles, S. Kross, and M. L. Mazurek, "How i learned to be secure: a census-representative survey of security advice sources and behavior," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 666–677.

[24] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. Germán, and J. M. Gonzalez-Barahona, "How bugs are born: a model to identify how bugs are introduced in software components," *Empirical Software Engineering*, pp. 1–47, 2020.

[25] D. Votipka, K. R. Fulton, J. Parker, M. Hou, M. L. Mazurek, and M. Hicks, "Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 109–126. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-understanding

[26] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 26–36.

[6]https://apd.niweera.gq