Analysis of Security Code Review Effectiveness

Anne Edmundson
Cornell University

Brian Holtkamp
University of Houston-Downtown

Emanuel Rivera
Polytechnic University of Puerto Rico

Adrian Mettler
University of California, Berkeley

David Wagner
University of California, Berkeley

Abstract

With the rapidly increasing number of web applications, developers should be concerned with web security vulnerabilities. It is essential in the development process to detect and correct these vulnerabilities before they are released to the public. This research aims to quantify the effectiveness of software developers at security code reviews as well as determine the variation in effectiveness among web developers. We hired 30 developers to conduct a manual code review of a small web application. The web application supplied to developers had 6 known vulnerabilities, including three different types: Cross-Site Scripting, Cross-Site Request Forgery, and SQL Injection. Our preliminary findings are: 1) none of the subjects found all confirmed vulnerabilities, 2) more experience does not necessarily mean that the review will be more accurate or effective, 3) self-reported understanding of the codebase is not indicative of how well the subjects could find the vulnerabilities, and 4) certain vulnerabilities went unnoticed by all reviewers.

1 Introduction

With the wide spread adoption and reliance on the internet, more people are using web applications for everyday life. With such a large user base, web applications have become a prime target for malicious users with the intent to steal information or damage the application. Unfortunately, it is fairly common for these applications to be susceptible to attacks. These vulnerabilities may arise due to poor programming practice or lack of knowledge about web security. The result is web applications that are susceptible to attack.

In this study we focus on the manual code review of web applications for security. We hire web developers with varying amounts of security experience to conduct a security code review of a simple web application. In this preliminary study, we collect data from ten web developers hired through an out-sourcing website. The developers are asked to perform a line-by-line code review of the application and submit a report of all security vulnerabilities found. Our main contributions are:

- We quantify the effectiveness of developers at security code review.
- The results of our analysis could be used to calculate the optimal number of independent reviewers to hire to achieve a desired level of confidence.
- We measure the extent to which developer demographic information can be used to predict effectiveness at security code review.

These results may help hiring managers and developers in determining how best to allocate resources when securing their web applications.

2 Goals

In this work, we conduct an exploratory analysis of software developers effectiveness in conducting security code review. We have three focus areas: the degree of variation among software developers, if and how we can predict the effectiveness of different software developers based on their demographics, and the optimal number of reviewers.

2.1 Effectiveness

Our research will measure how well developers conduct security code review. The following is one question that will be addressed by this research.

- How effective are developers at security code review?

2.2 Variation in Effectiveness

This study will quantify how developers vary in effectiveness; we want to answer the following questions regarding differences in developers' effectiveness.

- Are some developers significantly more effective than others?
- Is there significant variation in code review effectiveness between developers?
- How much variation is there between developers?

2.3 Optimal Number of Reviewers

Depending on the answers to the questions posed in Section 2.1, we want to determine the best number of reviewers to hire. Intuitively, if more developers are hired, then a larger percentage of vulnerabilities will be found, but we want to determine the point at which an additional reviewer yields insignificant results. This would be useful for determining the best allocation of resources (money) in the development of a web application. Specifically, we will address the following questions to find the optimal number of reviewers.

- What is the degree of overlap between vulnerabilities found by multiple code reviewers?
- Will multiple independent code reviewers be significantly more effective than a single reviewer?
- If multiple independent code reviewers are significantly more effective than a single reviewer, how much more effective?

2.4 Predicting Effectiveness

We also want to discover what affects developers effectiveness. In particular, we study whether there is any relationship between effectiveness and any of the factors in {application comprehension, self-assessed confidence in the review, education level, experience with code reviews, certifications, experience in software/web development, experience in security, confidence as a software/web developer, confidence as a computer/web security expert, most familiar programming languages}. Finding a relationship could lead to predicting which developers will be most effective at security code review.

Knowing if there is a significant difference between hiring developer X and hiring developer Y to conduct a security code review would be helpful to any engineer or manager creating a web application. This information may provide some insight into what criteria or factors to consider when hiring a security code reviewer.

3 Experimental Methodology

To assess developer effectiveness at security code review, we first identified our application with a number of security vulnerabilities. Then, we hired 30 developers through an outsourcing site and asked them to perform a manual line-by-line security review of the code. After developers computed their review, we asked them to tell us about their experience, qualifications, and answer other questions. We then counted how many of the known vulnerabilities they found.

3.1 Anchor CMS

We had reviewers review an existing web application, Anchor CMS. Anchor CMS is a small content management system. It is an open-source web application, is written in PHP and JavaScript, and uses a MySQL database. There are currently four release versions of Anchor. We chose not to use the latest version. Instead, we had reviewers review the third release, version 0.6. This version had more vulnerabilities while still having comparable functionality to the latest version.

To prepare and anonymize the code for review, we modified the Anchor CMS source code in two ways. First, we removed the Anchor name and all branding. To generate a generic CMS that wouldn't be searchable online, we renamed it TestCMS. We did not want developers to view Anchor CMS's bug tracker or any publicly reported vulnerabilities; we wanted to ensure they reviewed the code from scratch with no preconceptions. Our anonymization included removal of the title "Anchor," all relevant images and logos, and all instances of Anchor in variable names or comments.

Once the code was anonymized, we modified the code in two ways to increase the number of vulnerabilities in it. First, we took one vulnerability from Anchor version 0.5 and forward-ported it into our code. After this modifications, the web application had 3 Cross-Site Scripting vulnerabilities and no Cross-Site Request Forgery protection throughout the application.

Second, we carefully introduced two SQL injection vulnerabilities. To ensure these were representative of real SQL injection vulnerabilities naturally seen in the wild, we found similarly structured CMS applications on security listing websites (SecList.org), analyzed them, identified two SQL injection vulnerabilities in them, adapted the vulnerable code for Anchor, and introduced these vulnerabilites into TestCMS. The result is a web application with 6 known vulnerabilities. Our procedures ensure that these vulnerabilities should be reasonably representative of the issues present in other web applications.

These 6 known vulnerabilities are exploitable by any person who is not a registered user of the web application. Additionally, these vulnerabilities are solely from the PHP source code in the application; for this reason, we do consider problems such as Denial of Service attacks or insecure password choice to be in the scope of this project. While these were not included in the list of 6 known vulnerabilities, we did not classify them as incorrect vulnerabilities; they were placed in their own category. We recorded an incorrect vulnerability as a case where a developer reported a specific vulnerability at a certain location, when in fact there was not a vulnerability at the specified location. Lastly, vulnerabilities in the administrative interface were not considered for this study.

3.2 oDesk

oDesk is an outsourcing site that can be used to hire freelancers to perform many tasks, including web programming, development, and quality assurance. We chose oDesk because it is one of the most popular such sites, and because it gave us the most control over our hiring process; oDesk allows users to post jobs (with any specifications, payments, and requirements), send messages to users, interview candidates, and hire multiple people for the same job. We used oDesk to publicize our study, hire developers that met our requirements, and pay our subjects for their work.

3.3 Subject Population and Selection

We recruited subjects for our experiment by posting our job on oDesk. We specified that respondents needed to be experienced in the PHP scripting language in order to comprehend and work with our codebase. Additionally, they should have basic web security knowledge. We screened all applicants by asking them about how many times they have previously conducted a code review, a security code review, a code review of a web application, and a security code review of a web application. We also asked four multiple-choice quiz questions to test their knowledge of PHP and security. Each question showed a short snippet of code and asked whether the code was vulnerable, and if so, what kind of vulnerability it had. We accepted all respondents who scored 25% or higher on the screening test.

3.4 Task

We gave participants directions on how to proceed with the code review, an example vulnerability entry, and the TestCMS codebase. The instructions specified that no automated code review tools should be used. Also, the developers were told to spend 12 hours on this task; this number was calculated based upon a baseline of 250 lines of code per hour [17]. We designed a template that participants were instructed to use to report each vulnerability. The template has the following sections for the developer to fill in accordingly:

1. Vulnerability Type
2. Vulnerability Location
3. Vulnerability Description
4. Impact
5. Steps to Exploit

The type and location would give us basic information about the vulnerability. The template included Vulnerability Description and Impact sections in order to deter developers from using automated tools; it would be more challenging to successfully fill out these sections if a tool was used as opposed to a manual review. The last section, Steps to Exploit the vulnerability, would help make sure developers do not report only exploitable vulnerabilities as opposed to any possible security errors in the code.

The developers were asked to review only a subset of the code given to them. In particular, we had them review everything but the adminstrative interface and the client-side code. They reviewed 3500 lines of code in total. We specified our interest only in exploitable vulnerabilities.

3.5 Data Analysis Approach

Before the study, we scoured public vulnerability databases, Anchor's bug tracker, and other sources to identify all known vulnerabilities in TestCMS. This gives us ground truth that we use to evaluate responses from the participants. We analyze each participant's report and evaluate the accuracy and correctness of all bugs they reported. No participant found any valid, exploitable vulnerability that was missing from our ground truth. The results of this analysis enable us to assess if the population could properly review the code and find security vulnerabilities.

3.6 Threats to Validity

oDesk Population. As stated previously, we hired developers through the oDesk outsourcing website. This limited our population to registered oDesk users, instead of the desired population of all web developers. Using oDesk may have biased our results if the population on oDesk is not representative of the population of web developers. On the contrary, oDesk does provide a service that is used when hiring freelancers in the web development community; therefore, we assume it is reasonable to believe a web developer hired on an outsourcing site is comparable to a web developer hired through other methods.

Artificial Vulnerabilities. The original source code of Anchor version 0.6 had only three vulnerabilities, two XSS and one CSRF. As mentioned in Section 3.2, we introduced two SQL Injection vulnerabilities. Adding these artificial vulnerabilities creates a flawed codebase where the developer didn't make a flaw. These artificial vulnerabilities can bias the results since it may make the code review easier or harder than reviewing the original application. The changes made to the codebase were modeled after vulnerabilities found in other CMSs. This minimizes the artificiality of the codebase by ensuring that web developers have naturally made this mistake before.

Security Experts vs Web Developers. We hired 30 reviewers for this study, where some were specialized in security while others were purely web developers. Despite the use of a screening test, it is possible that web developers guess correctly or that the questions used vulnerabilities that were significantly easier to detect than those in an application. In this case, web developers would be at a disadvantage. Security experts have a better understanding of the attacks, how they work and how attackers can use them. On the other hand a web developer may have some understanding of how the attacks work, but they may not be as prepared as security experts. This could bias our results when measuring variability; on the contrary, this is a realistic scenario when hiring someone to conduct a security code review.

Difficulties of Anchor Code and Time Frame. The design of Anchor's source code proves to be challenging to understand. It is neither documented well nor is it structured well. With the 12 hours that the reviewers had, results might not be the same as if the code was designed in a different way. The developers may not find all the vulnerabilities or they may give us many false-positives. It has to be assumed that each reviewer dedicated the amount of time indicated and that this was a sufficient amount of time to analyze the code.

4 Results

With our population of 30 subjects, we were able to determine the relative effectiveness of their review by taking the ratio of reported vulnerabilities with our confirmed vulnerabilities. With this we were able to quantify their reports and compare it to demographics in question and provide us with a convenient score for how well the report covered our codebase's vulnerabilities.

4.1 Effectiveness

Our initial findings provided us with the opportunity to find how many vulnerabilities were found and what specific vulnerabilities were found most and least commonly. One out of the three Cross-Site Scripting vulnerabilities were found by the majority of subjects, while fewer subjects found the the lack of Cross-Site Request Forgery protection. Table 1 shows the fractions of developers that reported the corresponding vulnerabilities. The average number of correct vulnerabilities found were 1.497 with a standard deviation of 2.033.

Cross-Site Scripting 1	.40
Cross-Site Scripting 2	.70
Cross-Site Scripting 3	.20
SQL Injection 1	.37
SQL Injection 2	.20
Cross-Site Request Forgery	.17

Table 1: The fraction of developers that reported the corresponding vulnerability.

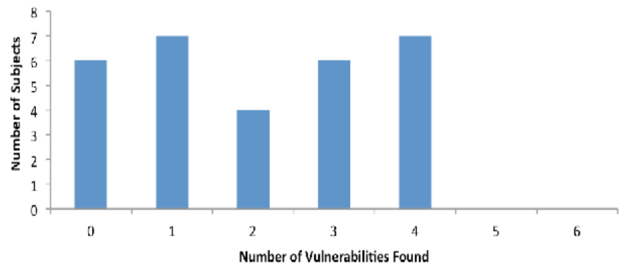

Figure 1: The number of vulnerabilities found by individual developers. Note that none found all six vulnerabilities.

While the average number of correct vulnerabilities found is relatively low, this is not indicative of the total number of vulnerabilities reported by each developer. The average number of reported vulnerabilities is 10.822 with a standard deviation of 8.7. Figure 2 shows the fractions of vulnerabilites reported that were correct.

4.2 Optimal Number of Reviewers

In order to determine the optimal number of reviewers, we simulated hiring various numbers of reviewers. For a single trial, if the combination of hired developers reports all vulnerabilities, then this trial is assigned a value of 1.0. If the combination of hired developers does not

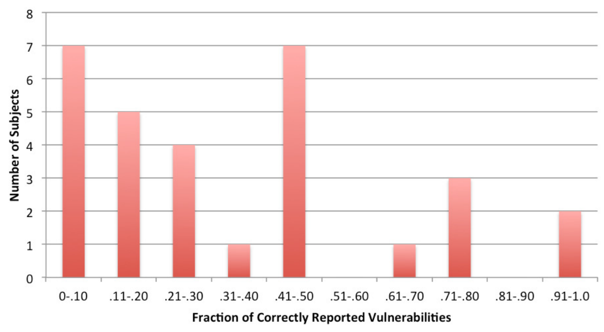

Figure 2: The fraction of reported vulnerabilities that were correct.

report all vulnerabilities, then this trial is assigned a value of 0.0. We conducted 1000 trials and take the average of all trials; this yields the probability of finding all vulnerabilities with a certain number of reviewers. Figure 3 shows the trend of finding all vulnerabilities based on the number of developers hired.

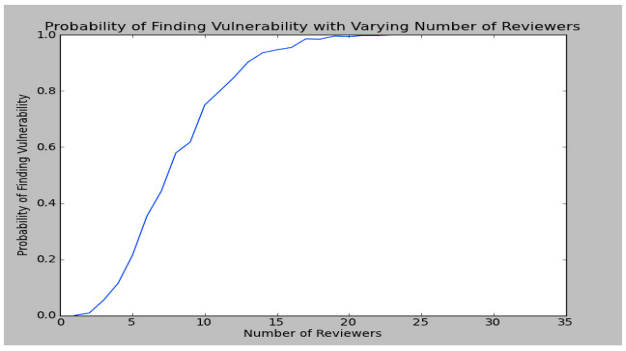

Figure 3: A graph showing the probability of finding all vulnerabilities depending on the number of developers hired.

4.3 Demographic Relationships

Figures 4 - 6 show the relationships between the number of correct vulnerabilities reported and developer experience with software development, web development, and computer security, respectively.

4.4 Limitations of Statistical Analysis

A major limitation of our statistical data is that the experiment was performed with only 30 subjects. A small sample, such as this, may not be representative of an entire population.

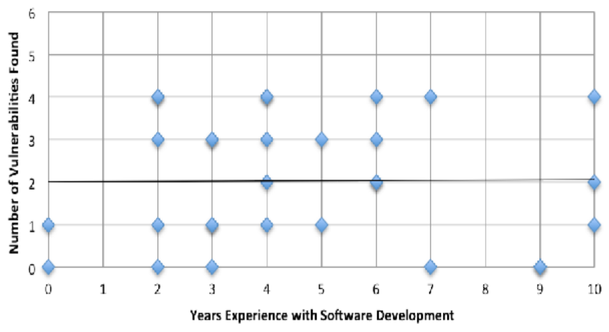

Figure 4: A scatterplot of the relationship between effectiveness and years of experience in software development.

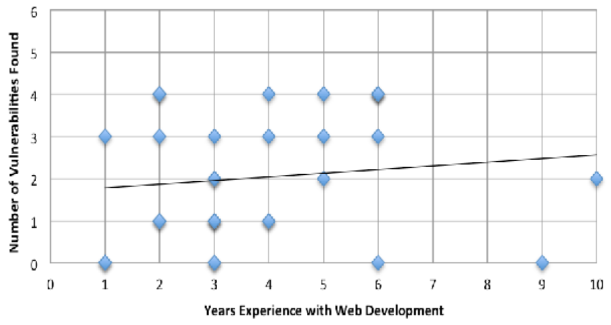

Figure 5: A scatterplot of the relationship between effectiveness and years of experience in web development.

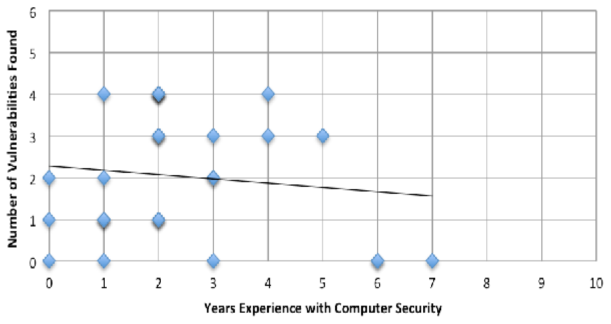

Figure 6: The relationship between effectiveness and years of experience in computer security.

5 Related Work

While there has been no previous research studying the effectiveness of developers at security code review, there have been many studies regarding the evaluation and effectiveness of code inspections. Our discussion of related work falls into three categories: code review effectiveness, methods of detecting web security vulnerabilities, and a comparison of manual code reviews to static analysis tools.

Code Review Effectiveness. One of the largest drawbacks to conducting code inspections is the time-consuming and cumbersome nature of the task, therefore there have been many studies investigating general code inspection performance and effectiveness [5, 4, 16]. Les Hatton conducted two different experiments; the first found a relationship between the total number of defects in a piece of code and then number of defects found in common by teams of inspectors [14]. The authors gave the inspectors a program written in C with 62 lines of code; the inspectors were told to find any parts of the code that would cause the program to fail. From this research, the authors were able to predict the total number of defects. While this differs from our study, future work that applies Hatton's experiment to code reviews for security vulnerabilities would be useful. Hatton's second experiment found that checklists had no significant effect on the efficiency of a code inspection [7]. Similarly, an interesting experiment would test this relationship on code inspections solely for security vulnerabilities. Other studies have discovered if there are relationships between specific factors, such as review rate or the presence of maintainability defects, and the performance of code inspections [1, 6]. These experiments were carried out on software and the inspectors were not limited to security vulnerabilities; our work includes a more specific task as well as different set of code.

Detection of Web Security Vulnerabilities. Most current techniques for detecting web security vulnerabilities are automated tools for static analysis. Despite this, there has been work that has compared different tools due to their differences. One study showed that black-box web application vulnerability scanners do not perform well when detecting advanced and second-order forms of Cross-Site Scripting and SQL Injection [3]. While it is more time-consuming, this may be remedied by manual code inspection. Additionally, there have been many publications evaluating and proposing new automated tools for detecting web security vulnerabilities [18, 13, 12, 10, 8, 11]. Our work focuses less on detecting all of the web security vulnerabilities in a web application and more on how effectively code reviewers can detect vulnerabilities, the variation, and what factors affect this.

Manual Code Review vs. Static Analysis Tools. While ongoing research studies are advancing code inspection efficiency and static analysis tool accuracy, some studies are comparing the different techniques. Capers Jones showed that no single method from {formal design inspection, formal code inspection, formal quality assurance, formal testing} was efficient in detecting and removing defects; the combination of all four methods yielded the highest efficiency. When only one method was used, the highest efficiency for removing defects was due to formal design inspection followed by formal code inspection [9]. Another study compared the effectiveness of other methods of code review: code reading by stepwise abstraction, functional testing, and structural testing. They found that when the experiment was performed with professional programmers, code reading detected more faults than either functional or structural testing; this was applied to software written in a high-level language, but not a web application [2]. When we conducted our experiment we did not specify how the developers should review the code as long as they did not use any automated tools. More recent research has investigated this comparison in the context of web application; they also limited their scope to security vulnerabilities. While manual source code review was found to be more effective than automated black-box testing, black-box testing discovered vulnerabilities not found through the manual source code review [15].

6 Conclusion

We hired 30 subjects to perform a security code review of a web application that has been confirmed to have security vulnerabilities. The subjects analyzed the code and completed a report based off of a vulnerability report template provided to them. This ensured their understanding on how to recreate the exploit and proof of where it is found in the code. A post-completion survey offers us data about their personal experience in web programming and security and their self-efficacy of the report they submitted.

Our results revealed that years experience offered little in the way of determining how well a subject was able to complete the code review and that their own opinion of how well they performed showed no correlation of how effective their report was.

In future work, we plan to expand the experiment to more subjects allowing for a bigger sample to further refine our statistics and provide more data to increase confidence in our results.

7 Acknowledgments

This work was supported in part by TRUST (Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422). Special thanks to Aimee Tabor and the TRUST program staff.

References

[1] ALBAYRAK, ÖZLEM AND DAVENPORT, DAVID. Impact of maintainability defects on code inspections. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (2010), ESEM '10, pp. 50:1–50:4.

[2] BASILI, V., AND SELBY, R. Comparing the effectiveness of software testing strategies. *Software Engineering, IEEE Transactions on SE-13*, 12 (dec. 1987), 1278 – 1296.

[3] BAU, J., BURSZTEIN, E., GUPTA, D., AND MITCHELL, J. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (may 2010), pp. 332 –345.

[4] BIFFL, S. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific* (2000), pp. 136 –145.

[5] FAGAN, M. E. Design and code inspections to reduce errors in program development. *IBM Systems Journal 15*, 3 (1976), 182 –211.

[6] FERREIRA, A.L. AND MACHADO, R.J. AND COSTA, L. AND SILVA, J.G. AND BATISTA, R.F. AND PAULK, M.C. An approach to improving software inspections performance. In *Software Maintenance (ICSM), 2010 IEEE International Conference on* (sept. 2010), pp. 1 –8.

[7] HATTON, L. Testing the Value of Checklists in Code Inspections. *Software, IEEE 25*, 4 (july-aug. 2008), 82 –88.

[8] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web* (2004), WWW '04, pp. 40–52.

[9] JONES, C. Software defect-removal efficiency. *Computer 29*, 4 (apr 1996), 94 –95.

[10] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on* (may 2006), pp. 6 pp. –263.

[11] KALS, S., KIRDA, E., KRUEGEL, C., AND JOVANOVIC, N. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web* (2006), WWW '06, pp. 247–256.

[12] KIEYZUN, A., GUO, P., JAYARAMAN, K., AND ERNST, M. Automatic creation of sql injection and cross-site scripting attacks. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on* (may 2009), pp. 199 –209.

[13] LAM, M. S., MARTIN, M., LIVSHITS, B., AND WHALEY, J. Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2008), PEPM '08, pp. 3–12.

[14] LES HATTON. Predicting the total number of faults using parallel code inspections. `http://www.leshatton.org/wp-content/uploads/2012/01/Inspect2005.pdf`, May 2005.

[15] MATTHEW FINIFTER AND DAVID WAGNER. Exploring the Relationship Between Web Application Development Tools and Security. In *Proceedings of the 2nd USENIX Conference on Web Application Development* (June 2011), USENIX.

[16] MCCARTHY, P., PORTER, A., SIY, H., AND VOTTA, L.G., J. An experiment to assess cost-benefits of inspection meetings and their alternatives: a pilot study. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International* (mar 1996), pp. 100 –111.

[17] OWASP FOUNDATION. Code Review Metrics. `https://www.owasp.org/index.php/Code_Review_Metrics`, 2010.

[18] WASSERMANN, G., AND SU, Z. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI '07 Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (June 2007), vol. 42, pp. 32–41.