Towards Improving Code Review Effectiveness Through Task Automation

Asif Kamal Turzo
Wayne State University
Detroit, USA
asifkamal@wayne.edu

ABSTRACT

Modern code review (MCR) is a widely adopted software quality assurance practice in the contemporary software industry. As software developers spend significant amounts of time on MCR activities, even a small improvement in MCR effectiveness will incur significant savings. As most of the MCR activities are heavily dependent on manual work, there are significant opportunities to improve effectiveness through tool support. To address the challenges, the primary objective of my proposed dissertation is *to improve the effectiveness of modern code reviews with the automation of reviewer selection and bug identification.* On this goal, I propose three studies. The first study aims to investigate the notion of useful MCRs and factors influencing MCR usefulness. The second study aims to develop a reviewer recommendation system that leverages a reviewer's prior history of providing useful feedback under similar contexts. Finally, the third study aims to improve the effectiveness of static analysis tools by leveraging bugs identified during prior reviews.

KEYWORDS

code review, security, software quality, software development

ACM Reference Format:
Asif Kamal Turzo. 2022. Towards Improving Code Review Effectiveness Through Task Automation. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3551349.3559565

1 INTRODUCTION

Modern code review (MCR) is a software quality assurance measure that has been adopted widely in commercial and Open Source Software (OSS) projects. Besides early identification of defects, MCR provides other crucial benefits such as knowledge transfer and building team awareness [3]. Due to widespread adoption and growing importance, recent studies have focused on understanding factors influencing MCR outcomes [3–5, 25], automating reviewer selection [28, 32, 38], and identifying locations of suspicious code segments to assist reviewers [17, 18, 36, 39]. However, practicing MCR mandates a significant investment for an organization, since developers spend on average 10-15% of their time on code review activities [5]. Therefore, even a small improvement in MCR effectiveness can incur significant savings for an organization practicing MCR. Therefore, the primary objective of my proposed dissertation is *to improve the effectiveness of modern code reviews with automation of reviewer selection and bug identification.*

On this goal, my proposed dissertation aims to conduct the following three studies.

(Study 1) Identify the factors that make code review useful to the OSS developers

Motivation: *What is the perspective of OSS developers about the usefulness of review comments and which factors influence a developer to provide useful review comments?* This question has not been investigated in an OSS environment. Developers might ask questions such as: (1) what is the frequency of different categories of review comments?, (2) What are the factors that lead a developer to provide useful review comments?, and (3) which aspects a reviewer should consider while providing a review comment? Insight gathered from the study may help OSS developers author useful review comments and select appropriate reviewers to maximize the likelihood of useful feedback.

Objective: *To perceive the developer's perspective about the usefulness of code review comments and identify factors that lead a review comment to become useful.*

(Study 2) Develop a usefulness-aware reviewer recommendation system

Motivation: Prior studies have proposed numerous approaches for constructing reviewer recommendation systems that used 'history' for assessing the performance of those recommendation systems [28, 32, 38]. Those recommendation systems have overlooked the fact of whether a reviewer has provided useful comments previously or not. Studies also suggest that 'history' can sometimes be overly optimistic and sometimes be overly pessimistic [11]. Incorporating the reviewer's capability factor of providing useful reviews while building a reviewer recommendation system is the motivation behind this study.

Objective: *To build a reviewer recommendation that considers a reviewer's history of useful reviews under similar contexts.*

(Study 3) Automatic identification of buggy code segments

Motivation: Although the primary objective of MCR is bug identification, most of the reviews do not find any bug [3]. As reviewers do not have adequate time to understand the whole code, they often identify minor code issues (e.g. documentation, typos, and refactoring) [3]. Therefore, the identification of bugs within a limited time is a challenge for reviewers. To assist in bug identification, if

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559565

reviewers can be made aware of the problematic code segments under review, they are more likely to investigate those problematic code segments thoroughly and are more likely to identify potential bugs. On this goal, many projects leverage static analysis tools to automate reviews. However, static analysis tools are good at identifying certain types of bugs and will miss other categories such as misunderstood requirements or violations of design constraints. On the other hand, expert human reviewers are better at identifying these categories of defects.

Objective: *To improve the effectiveness of static analysis tools by leveraging bugs identified during prior reviews.*

The remainder of the paper is organized as the following. Section 2 provides a brief overview of the research context and prior related works. Finally, Section 3 and Section 4 describe the three proposed studies and concludes the paper respectively.

2 BACKGROUND

Modern Code Review: Modern code review is a practice of reviewing a code by peers before merging it to the main codebase. MCR has been adopted by most commercial and OSS projects as a quality assurance gateway [6]. MCR is asynchronous, lightweight, and tool-based [3]. In an MCR process, an author invites their peer(s) for reviewing. If reviewer(s) find any concern with the change, they inform the author with review comments. If all the concerns are resolved and the code is of sufficient quality, reviewers approve the change for merging to the main branch.

Related Works: While earlier related works [26, 27] primarily focused on analyzing formal Fagan-style software inspection process [9], recent studies have shifted focus on modern code reviews [30] as those have been widely adopted [3, 5]. Bacchelli and Bird conducted a study where they uncovered multiple benefits that code review provides, e.g. defect finding, create bonding among team members, knowledge transfer among team members [3]. Mantyla *et al.* proposed a classification of defects identified in code review process [19], which was later reused and updated by Bosu *et al.* [6]. Rahman *et al.* [29] proposed a study for classifying review comments based on their textual characteristics. Gousios *et al.* conducted a study where they identified getting timely and constructive feedback is challenging for OSS contributors [12]. The study which closely relates to Study 1 was proposed by Bosu *et al.* [6], where they identified characteristics of useful code review and proposed a classifier for categorizing review comments into useful and non-useful categories. The study of Bosu *et al.* was conducted for a commercial project, whereas I conducted Study 1 for an OSS project and used different research instruments.

Prior works have focused on automating reviewer selection task. Thongtanunam *et al.* proposed a reviewer recommendation system based on file path characteristics [32]. Zanjani *et al.* proposed a reviewer recommendation system where they considered the contribution of a reviewer in their prior review [38]. Ouni *et al.* proposed a search-based reviewer recommendation system [24]. Rahman *et al.* [28], Chouchen *et al.* [8] also proposed reviewer recommendation systems for assisting a developer by finding suitable reviewers. Prior proposed studies have considered history as a benchmark and overlooked reviewer's prior ability to provide useful feedback.

In the field of vulnerability detection, several research works have been proposed. Li *et al.* proposed a Bidirectional LSTM model for identifying software vulnerabilities using code segments [18]. Sysevr [17] used localized code segments and argued that specific code portions are important for identifying defects. Russel *et al.* proposed a model which converts the source code into minimum intermediate representation and used machine learning to detect vulnerability [31]. Zhou *et al.* [39] and Yamaguchi *et al.* [36] proposed graph-based machine learning model for detecting bugs within code. Despite the effort of the researchers, Chakraborty *et al.* argued that the reported performance of prior models drops by 50% when experimented with real-world vulnerability dataset [7].

3 PROPOSED RESEARCH

This section presents the approaches of my proposed studies. I am expecting feedback from the respected symposium members on improvement suggestions for the recommendation system in Study 2 and for the automation of bug localization task in Study 3.

3.1 Study 1: Identify the factors that make code review useful to the OSS developers

For understanding the developer's perspective on useful code review and identifying what are the factors that influence usefulness, this study is designed to focus on two research questions.

Research Questions: The code review process is a significant investment for a software company both from the economic perspective and from the developer's time perspective. If the code review process is not useful to the developers, then not only the investment is unnecessary, but also there is the chance of potential vulnerabilities within the software product. So, we need to understand what is the developer's perspective about useful code review. Although Bosu *et al.* investigate the usefulness of code review at Microsoft [6], I focused my effort on OSS developers and followed different research approaches. So, my first research question is:

RQ1: What makes a code review comment useful to the OSS developers?

Identifying defects is the primary expectation of performing code reviews. But defect identification requires a high codebase understanding and in reality, very few review comments are focused to identify defects [3]. So, there is a mismatch between expectation and reality. While selecting reviewers, a developer might wonder what are the factors that help the reviewers provide useful feedback. If developers select reviewers considering those factors, they can maximize their chance of getting useful feedback. To identify the factors that dictate the usefulness of code review, my second research question is:

RQ2: Which factors influence the usefulness of review comments?

Research Methodology: To answer the two research questions, I conduct the study on the OpenStack Nova[1] project. I along with one of my peers collected 2500 code review comments randomly and categorized the comments based on the categorization of code review comments suggested by Beller *et* al. [4]. I designed a survey of OpenStack developers where the objective was to capture developer's perspectives about the usefulness of review comments. Then

[1] https://docs.openstack.org/nova/latest/

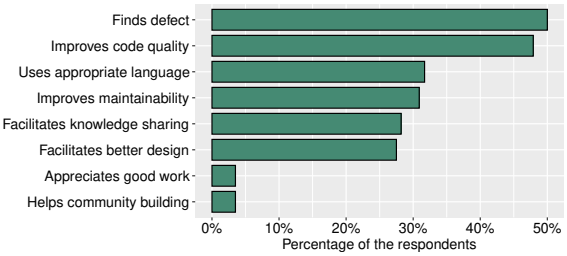

Figure 1: What makes a code review useful to an author?

I conducted a qualitative analysis of the 237 responses gathered from the survey.

I designed a quantitative study for identifying those factors that have an influence on making review comments useful. For the analysis, 17 reviewer's and code factors were calculated that might have an influence on making review comments useful. To finally identify which among those 17 factors are really influential, I constructed a Generalized Linear Model (GLM). The dependent variable is the usefulness rating obtained from the developer's survey for each category of review comments. The factors that obtain a $p - value < 0.05$, are actually influential in obtaining a higher usefulness rating (a higher usefulness rating makes a code review comment more useful). I have also constructed a Multinomial Logistic Regression (MLR) model where the dependent variable is the category in which the comment belongs. The purpose of constructing MLR is to identify the factors that separate a review comment category from the others (e.g. separate review comments that identify functional defects from those comments that identify documentation issues). For constructing the regression models, I followed the approach proposed by Harrell Jr [13, 14]

Research Progress: From the survey, I collected the developer's rating for each category of review comment. I also analyzed the developer's perception about the usefulness of review comments and Figure 1 presents the findings. From the survey analysis, my key finding is *the usefulness of a code review comment not only depends on the technical contribution (e.g. bug finding, improve architecture) but also depends on providing the review in a polite, constructive, and understandable manner.*

For identifying factors that have a significant role in making a review comment useful, I fit the data into a GLM. I found four factors that have a significant role in making review comments useful. The factors are review interval, mutual review, reviewer coding experience, and prior file review count. For identifying factors that separate one category of review comment from another, I constructed an MLR model. I measured the goodness of fit using Nagelkerke *Pseudo* R^2 [23], where I achieved a *Pseudo* R^2 value of 0.429. The four factors that play important role in making review comment useful, is also significant in separating one category of review comments from other. Moreover, comment volume is significant in separating one category from another. Currently, this paper is under submission for review.

Expected Contributions: The expected contributions of this study include: i) a better understanding of OSS developers' perception about the usefulness of a review comment; ii) identification factors influencing code review usefulness; and iii) a set of recommendations for practitioners to improve code review effectiveness.

3.2 Study 2: Develop a usefulness-aware reviewer recommendation system

In OSS and commercial projects, an author's code needs to be verified by a number of reviewers before merging the code into the main repository. The newcomers find it challenging to receive timely and useful feedback [16, 21]. The challenges can be solved by selecting appropriate reviewers who can provide useful feedback. Not only newcomers but also long-time contributors face the challenge of selecting appropriate reviewers for reviewing a code [32, 38].

Research questions: To assist the code author in selecting appropriate reviewers, researchers have proposed reviewer recommendation systems such as RevFinder [32], cHRev [38], CORRECT [28], RevRec [24], and WhoReview [8]. The main challenge these reviewer recommendation systems face is that most reviewer recommendation systems focus only particular type of history (e.g. RevFinder considers prior review history). Moreover, the existing reviewer recommendation systems do not consider the reviewer's ability to provide useful feedback in prior reviews. Let's consider a scenario where Reviewer A provided useful feedback in a previous review but Reviewer B raised an invalid concern in another review. Existing recommendation systems assess Reviewer A and B both as appropriate reviewers despite the fact that Reviewer B provided invalid feedback. To address the reviewer's prior ability of providing useful reviews, our first research question is:

RQ1: Can we incorporate reviewer's prior ability of providing useful review while building a reviewer recommendation system?

Gauthier *et al.* reported that historical data based on previous reviews can be overly optimistic since there might be other appropriate reviewers than only the selected set or historical data can be overly pessimistic since prior incorrect recommendations are considered equivalent to correct recommendations [11]. They also announced that history-based recommendation systems act far more pessimistically than optimistically. In the proposed recommendation system in RQ1, I assessed the reviewer's prior ability to provide useful reviews. So, we are attempting to correct the history by only selecting those reviewers who provided useful comments in prior reviews, which raises the question of how accurately such a recommendation system can perform. To address the issue, my second research question is:

RQ2: How accurately can the reviewer recommendation system recommend reviewer on real OSS projects?

Research Methodology: To build a reviewer recommendation system considering the reviewer's prior ability to provide useful reviews, I collected 2,500 code review comments from the oVirt[2] project. oVirt uses Gerrit as their primary code review tool. In the manual labeling step, each review comment was categorized into 'useful' or 'not useful' category. I calculated 12 historical attributes based on reviewer and file characteristics. The 12 factors were identified either from the understanding of Study 1 or came from prior literature. I am planning to collect code context vectors extracted from code using the approach proposed by Alon *et* al. [1].

Usefulness Density (UD) is the ratio of code review comments that are perceived as useful [6]. For each reviewer r and review

[2]https://www.ovirt.org/

comment c, the UD value can be calculated using the equation:

$$UD_{r,c} = \alpha_0 + \alpha_1 V_1^{rc} + \alpha_2 V_2^{rc} + \alpha_3 V_3^{rc} + ... + \alpha_n V_n^{rc} + \epsilon_i$$

ϵ_i indicates the error term, α_0 represents the intercept term, and each of $V_1^{rc}, V_2^{rc}, V_3^{rc}, ..., V_n^{rc}$ represents either a historical attribute or one dimension code context vector, and $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ represents the coefficient of historical attribute or context vector respectively. For each reviewer r, the model would compute the UD score. Then the available reviewers would be ranked based on higher UD scores and the topmost reviewers would be recommended.

Research Progress: The data collection, manual labeling, and attribute calculation are already completed. In model fitting, the regression model is showing poor performance. The main reason I identified behind such performance is, 87% of code review comments discuss trivial issues such as documentation, refactoring, and variable naming-related issues. Providing such comments requires less expertise and codebase understanding than finding defects [3]. So, the historic attributes (such as reviewer coding experience, and reviewing experience) fail to provide the expected result. Code review is a multi-objective approach as modeled by Mirsaeedi and Rigby [22]. I am experimenting now to incorporate code context and other objectives (such as reviewer workload factor).

Expected Contributions: The expected contributions of Study 2 include: i) A manually labeled dataset of 2500 code review comments categorized as 'useful' or 'not useful'; ii) A novel approach for constructing reviewer recommendation systems based on the reviewer's prior ability to provide useful review comments; and iii) An empirical evaluation for assessing the performance and practical usability of the proposed recommendation system in OSS projects.

3.3 Study 3: Automatic identification of buggy code segments

Bugs are software defects that can be a serious security threat. The longer a bug takes to be identified, the more cost it will incur in the development process [20]. Despite code going through a rigorous review process, bugs are identified in the later development phase and even after the release of the product. So the central idea which propels the study is *if we highlight a suspicious code segment, then reviewers can concentrate their focus on those suspicious code segments*. The objective of this study is to improve the effectiveness of static analysis tools by locating buggy code segments and by suggesting potential solutions to fix those bugs.

Static analysis tools are effective in identifying programming practice violations and certain types of bugs such as null pointer dereference, and overflow of array [2]. Several existing solutions have already proposed the integration of static analysis tools in the code review pipeline. However, the novelty of my proposed research lies in the development of this tool. In a prior study, we find that existing static analysis tools are better at finding certain classes of defects, while human reviews are at others [25]. I plan to enhance existing static analysis tools with human knowledge disseminated during code reviews. I can use the large code review corpus that identifies bugs to train a machine learning model and later incorporate the model with static analysis tools. If we train a supervised machine learning-based static analysis tool with bugs identified by human reviewers, then the efficiency of static analysis tools can be improved. Thus, the proposed research can augment the effectiveness of the static analysis tools by incorporating a trained model.

Research Methodology: To identify buggy code segments within a code file, I plan to conduct a three-stage study. In the first stage of the study, I will construct an automated model which will analyze code review comments and corresponding code to identify bug-finding review comments. There are existing models for classifying review comments, but they use commit messages for classification [15, 37] or use code analysis metrics (e.g., Lines Of Code, or Cyclomatic Complexity) [10]. My approach differs from previous approaches as I am planning to construct the model considering code characteristics such as number of operators, number of brunches, and number of keywords.

In the second stage, I would use the automated model developed in the first stage to construct a large-scale dataset of the buggy and benign methods. Each buggy method is the pre-review version and the benign method is the corresponding post-review version. My dataset will contain only code reviews where bugs have been identified. In the third stage, I will use the dataset to train a machine learning model for identifying buggy and benign methods. Leveraging the automated approach of the first stage, I would collect a large dataset with buggy and corresponding benign methods. The approach can be considered as a sequence-to-sequence translation, where a buggy sequence needs to be converted into a benign sequence. In such sequence-to-sequence translation, transformers models [35] work well [33, 34]. I will experiment with different transformers architectures. So my hypothesis is *if transformers can be trained using a large code review corpus and integrated with static analysis tools, the effectiveness of static analysis tools can be augmented.* Using transformer models will provide benefits not only by identifying bugs but also by providing potential improvements.

Research Progress: The data collection, manual labeling, and model building is yet to be done for Study 3.

Expected Contributions: The expected contributions of Study 3 include: i) A novel automatic approach to identify bug-finding code review comments; ii) A large-scale dataset containing buggy (pre-review version) methods and corresponding benign (post-review version) methods; and iii) An empirically evaluated machine learning model to identify buggy code segments within a code file.

4 CONCLUSION

In this doctoral dissertation, my primary focus is to improve the effectiveness of peer review by automating some code review tasks. On this goal, I have conducted Study 1 to gather the perspective of developers about the usefulness of code review comments. Then utilizing the concept of useful code review from the first study, in Study 2, I have proposed a reviewer recommendation system for automating the reviewer selection task. The reviewer recommendation system would consider a reviewer's prior ability to provide useful review comments while suggesting appropriate reviewers. In Study 3, I would focus on automating bug localization task and automatic code change suggestions for fixing those bugs in the code review process. I believe the proposed studies would improve the effectiveness of code review by augmenting existing approaches and fostering future research in automating code review tasks.

ACKNOWLEDGMENTS

Work conducted by Asif Kamal Turzo for this research is partially supported by the US National Science Foundation under Grant No. 1850475. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[2] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[3] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 712–721.

[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *Proceedings of the 11th working conference on mining software repositories*. 202–211.

[5] Amiangshu Bosu, Jeffrey C Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. 2016. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering* 43, 1 (2016), 56–75.

[6] Amiangshu Bosu, Michaela Greiler, and Christian Bird. 2015. Characteristics of useful code reviews: An empirical study at microsoft. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 146–156.

[7] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[8] Moataz Chouchen, Ali Ouni, Mohamed Wiem Mkaouer, Raula Gaikovina Kula, and Katsuro Inoue. 2021. WhoReview: A multi-objective search-based approach for code reviewers recommendation in modern code review. *Applied Soft Computing* 100 (2021), 106908.

[9] Michael Fagan. 2002. Design and code inspections to reduce errors in program development. In *Software pioneers*. Springer, 575–607.

[10] Enrico Fregnan, Fernando Petrulio, Linda Di Geronimo, and Alberto Bacchelli. 2022. What happens in my code reviews? An investigation on automatically classifying review changes. *Empirical Software Engineering* 27, 4 (2022), 1–43.

[11] Ian X Gauthier, Maxime Lamothe, Gunter Mussbacher, and Shane McIntosh. 2021. Is Historical Data an Appropriate Benchmark for Reviewer Recommendation Systems?: A Case Study of the Gerrit Community. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 30–41.

[12] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work practices and challenges in pull-based development: the contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 285–296.

[13] Frank E Harrell Jr, Kerry L Lee, Robert M Califf, David B Pryor, and Robert A Rosati. 1984. Regression modelling strategies for improved prognostic prediction. *Statistics in medicine* 3, 2 (1984), 143–152.

[14] Frank E Harrell Jr, Kerry L Lee, David B Matchar, and Thomas A Reichert. 1985. Regression models for prognostic prediction: advantages, problems, and suggested solutions. *Cancer treatment reports* 69, 10 (1985), 1071–1077.

[15] Abram Hindle, Daniel M German, Michael W Godfrey, and Richard C Holt. 2009. Automatic classication of large changes into maintenance categories. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 30–39.

[16] Amanda Lee, Jeffrey C Carver, and Amiangshu Bosu. 2017. Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 187–197.

[17] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021).

[18] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[19] Mika V Mäntylä and Casper Lassenius. 2008. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering* 35, 3 (2008), 430–448.

[20] Gary McGraw. 2008. Automated code review tools for security. *Computer* 41, 12 (2008), 108–111.

[21] Christopher Mendez, Hema Susmita Padala, Zoe Steine-Hanson, Claudia Hilderbrand, Amber Horvath, Charles Hill, Logan Simpson, Nupoor Patil, Anita Sarma, and Margaret Burnett. 2018. Open source barriers to entry, revisited: A sociotechnical perspective. In *Proceedings of the 40th International conference on software engineering*. 1004–1015.

[22] Ehsan Mirsaeedi and Peter C Rigby. 2020. Mitigating turnover with code review recommendation: balancing expertise, workload, and knowledge distribution. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1183–1195.

[23] Nico JD Nagelkerke et al. 1991. A note on a general definition of the coefficient of determination. *Biometrika* 78, 3 (1991), 691–692.

[24] Ali Ouni, Raula Gaikovina Kula, and Katsuro Inoue. 2016. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 367–377.

[25] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. 2021. Why security defects go unnoticed during code reviews? a case-control study of the chromium os project. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1373–1385.

[26] Adam Porter, Harvey Siy, Audris Mockus, and Lawrence Votta. 1998. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 1 (1998), 41–79.

[27] Adam Porter, Harvey Siy, and Lawrence Votta. 1996. A review of software inspections. *Advances in Computers* 42 (1996), 39–76.

[28] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *Proceedings of the 38th international conference on software engineering companion*. 222–231.

[29] Mohammad Masudur Rahman, Chanchal K Roy, and Raula G Kula. 2017. Predicting usefulness of code review comments using textual features and developer experience. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 215–226.

[30] Peter C Rigby and Christian Bird. 2013. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 202–212.

[31] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[32] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. 2015. Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 141–150.

[33] Rosalia Tufan, Luca Pascarella, Michele Tufanoy, Denys Poshyvanykz, and Gabriele Bavota. 2021. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 163–174.

[34] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[36] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[37] Meng Yan, Ying Fu, Xiaohong Zhang, Dan Yang, Ling Xu, and Jeffrey D Kymer. 2016. Automatically classifying software changes via discriminative topic model: Supporting multi-category and cross-project. *Journal of Systems and Software* 113 (2016), 296–308.

[38] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. 2015. Automatically recommending peer reviewers in modern code review. *IEEE Transactions on Software Engineering* 42, 6 (2015), 530–543.

[39] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).