VulGuard: An Unified Tool for Evaluating Just-In-Time Vulnerability Prediction Models

Duong Nguyen*, Manh Tran-Duc*, Thanh Le-Cong†, Triet Huynh Minh Le‡, M. Ali Babar‡, Quyet-Thang Huynh*
*School of Communication and Information Technology, Hanoi University of Science and Technology, Hanoi, Vietnam
{duong.nd215336, manh.td194616}@sis.hust.edu.vn, thang.huynhquyet@hust.edu.vn
†School of Computing and Information Systems, The University of Melbourne, Melbourne, Australia
congthanh.le@student.unimelb.edu.au
‡School of Computer and Mathematical Sciences, The University of Adelaide, Adelaide, Australia
{triet.h.le, ali.babar}@adelaide.edu.au

***Abstract*—We present VulGuard, an automated tool designed to streamline the extraction, processing, and analysis of commits from GitHub repositories for Just-In-Time vulnerability prediction (JIT-VP) research. VulGuard automatically mines commit histories, extracts fine-grained code changes, commit messages, and software engineering metrics, and formats them for downstream analysis. In addition, it integrates several state-of-the-art vulnerability prediction models, allowing researchers to train, evaluate, and compare models with minimal setup. By supporting both repository-scale mining and model-level experimentation within a unified framework, VulGuard addresses key challenges in reproducibility and scalability in software security research. VulGuard can also be easily integrated into the CI/CD pipeline. We demonstrate the effectiveness of the tool in two influential open-source projects, FFmpeg and the Linux kernel, highlighting its potential to accelerate real-world JIT-VP research and promote standardized benchmarking. A demo video is available at: https://youtu.be/j96096-pxbs.**

I. Introduction

Software vulnerabilities negatively impact the reliability, security, and functionality of software systems to an unignorable degree, leading to severe damage to both users and companies. A notable mention would be the 2024 CrowdStrike outage, a misalignment between expected field and actual input caused a cascade of system failures, affecting millions of devices and disrupting essential services worldwide [1], [2]. This incident highlights the significant financial and operational burdens of post-deployment detected vulnerabilities, as well as the hidden technical risks within software systems.

To mitigate these challenges, Just-In-Time Vulnerability Prediction (JIT-VP) [3] has emerged as a promising approach to improving software quality assurance. At the early stages of the software development life cycle, JIT-VP techniques can identify security-threatening modifications in the software system, allowing developers to take immediate action. As a result, integrating JIT-VP into the development life cycle can improve the security inspection procedure and reduce the costs associated with future remediation.

Despite significant advancements in Just-In-Time Vulnerability Prediction (JIT-VP) [3], [4], [5], real-world adoption remains limited. A primary obstacle lies in the complexity of data curation: extracting, cleaning, and preprocessing commits from heterogeneous and evolving software repositories is often repository-specific, error-prone, and labor-intensive. This challenge results in reduced experimental scale and inconsistent model evaluation [6]. Furthermore, existing research rarely addresses integration with modern development workflows, thereby hindering the delivery of actionable feedback to developers and limiting the practical utility of academic models [7]. To close this gap, there is a need for a unified tool that streamlines the end-to-end JIT-VP pipeline, from data collection and preprocessing to model training and evaluation, while supporting seamless integration into real-world development environments.

To address these challenges, we introduce VulGuard, a unified tool for evaluating JIT-VP techniques. This tool has been employed in the empirical study on JIT-VP presented in our recently accepted paper at ICSME 2025 [7]. VulGuard offers three main features: (1) dataset construction, (2) model training, and (3) model evaluation. For dataset construction, our tool is designed to extract various features from commits, such as expert features [3], [8], property graphs [4], messages, and code changes. It also provides a tool to train and evaluate state-of-the-art JIT-VP techniques. Notably, VulGuard adopts a realistic evaluation setting that incorporates both vulnerability-related and neural commits, in accordance with the findings of our empirical study [7]. The VulGuard pipeline begins by cloning the given GitHub repositories to the local machine, then leveraging *git* application to extract commit data, and the V-SZZ [9] algorithm to trace the vulnerable commits.

Once the dataset is constructed, VulGuard can utilize it to train and evaluate the implemented techniques.

To summarize, key features of VulGuard include:

- Construct new datasets for JIT-VP research, which are also extensible for other vulnerability analysis tasks.
- Support multiple programming languages, including C/C++, Java, JavaScript, and Python.
- Integrate multiple JIT-VP techniques to train and evaluate in real-world settings.
- Installable Python package with Command-line interface.

Our tool with manual is available at Github release [10].

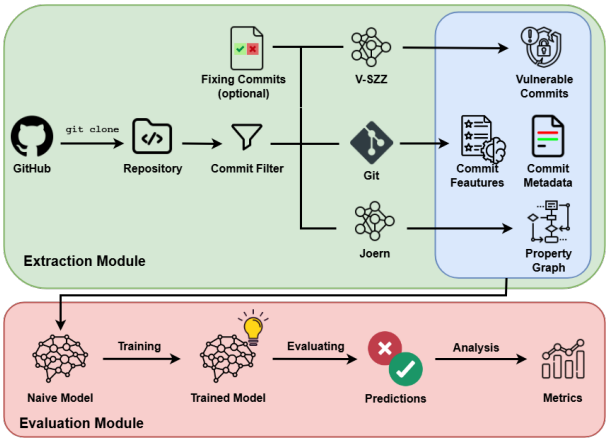

Fig. 1: Architecture of VulGuard

II. Related Works

Zeng et al. [11]'s replication study is the closest to our work, which provides a codebase for extracting commit-level features and metadata. It also includes implementations of several Just-in-Time (JIT) defect prediction baselines: CC2Vec [12], DeepJIT [13], DBN-JIT [14], LR-JIT [8], and their proposed method, LAPredict. We include some models from their work, including DeepJIT, LR-JIT, and LAPredict. However, our tool differs from their replication in multiple aspects. First, we offer user-friendly environments to facilitate adoption by both researchers and practitioners through two primary usage scenarios. (1) Our tool can be installed as a Python library, enabling integration into various software projects. (2) Besides supporting direct usage through Python library imports, we also provide an intuitive command-line interface (CLI), thereby accommodating diverse user preferences and workflows. These features provide both individual usage for research and integration usage for deployment. Second, our tool is designed to address the task of JIT-VP, whereas the focus of Zeng et al.'s study lies within the domain of JIT-DP, resulting in implementation-wise differences. Instead of employing the traditional B-SZZ algorithm [15] to identify bug-inducing commits, we utilize V-SZZ [9], which is designed to improve the accuracy of labeling vulnerability-inducing commits. We offer a greater variety of tools with the implementation of two more JIT-DP approaches, TLEL [16] and SimCom [17], and three state-of-the-art JIT-VP approaches, i.e., VCCFinder [3], JITFine [18], and CodeJIT [4].

JITBot [19] is a GitHub application for users to integrate into their own GitHub Action pipelines [20]. Similarly to our work, JITBot has been created to address the problem of the lack of adoption of JIT-DP tools in CI/CD pipelines. Unfortunately, to the best of our knowledge, JITBot is no longer publicly available on GitHub. Moreover, while JITBot only supports the application phase of JIT-DP models using a specific built-in model, VulGuard facilitates the end-to-end development of JIT VP. This includes data mining, model training, and deployment, all within an accessible environment provided as a Python library.

III. Architecture

VulGuard is built to streamline the data extraction process, as well as to train and evaluate prediction models for vulnerability research. VulGuard has two main modules: Extraction and Evaluation, as shown in Figure 1. We cover these modules in detail in the following subsections.

A. Extraction Module

This module of VulGuard can be divided into four main tasks: **commit collection**; **feature extraction**; **commit annotation** and **data splitting**. We also integrate a **graph builder** module, which generates a graph representation of commits.

1) Commit Collection: VulGuard takes input from a local Git repository. Following practices established in prior work [21], [22], [23], VulGuard filters out merge commits, whitespace-only commits, and comment-only commits to focus on meaningful code modifications. In addition, only commits that involve changes in source code files based on the primary language of the repository are retained. Specifically, the tool considers files with extensions: .c/.h for C, .cpp for C++, .java for Java, .js for JavaScript, and .py for Python.

2) Feature Extraction: In this task, VulGuard leverages Git to systematically collect key information from each commit, including commit messages, code changes, file-level metadata, and blame information for line-level authorship tracking. The raw data are then processed to derive relevant features that reflect code quality, developer activity, and temporal patterns [8], [3]. The features extracted in this step can be customized. All output information is serialized in .jsonl format.

Another key feature of VulGuard is its support for parallel execution of `git diff` and `git blame` operations. Since the metadata extraction for each commit is independent and both `git diff` and `git blame` are read-only operations, this parallelization is safe and highly effective. By leveraging concurrent processing, VulGuard significantly accelerates collecting fine-grained code changes and line-level authorship information without compromising repository integrity.

3) Commit Annotation: In this task, each commit is labeled as either vulnerable or non-vulnerable. Specifically, we flag the commits that have changes introducing vulnerabilities as positive, while all other commits are flagged as negative. We implement this practice to mimic the realistic conditions of the software development cycle [7].

However, accurately identifying vulnerability-inducing commits remains a challenging problem. A common practice is to identify fixing commits and then trace back to the vulnerable origins. To expedite the identification of vulnerability-fixing commits, we incorporate the regular expression proposed by Zhou et al. [24] (see Table I). While this technique improves efficiency, it may introduce noises. As a result, we recommend complementing the tool with a manual list of patch commits.

Next, we adopt V-SZZ [9], an enhanced variant of the classic SZZ algorithm [15], which traces the origin of patches to identify their corresponding inducing commits. Among the SZZ family, V-SZZ has demonstrated the highest effectiveness and has been commonly used in vulnerability analysis

TABLE I: Regular expression used to filter patch commits provided by Zhou et al. [24]

Rule name	Regular Expression
strong_vuln_patterns	`(?i)(denial.of.service|\bXXE\b|remote.code.execution|bopen.redirect|OSVDB|\bvuln\b|\bCVE\b|\bXSS\b|\bReDoS\b|\bNVD\b|malicious|x-frame-options|attack|cross.site|exploit|directory.traversal|\bRCE\b|\bdos\b|\bXSRF\b|clickjack|session.fixation|hijack|advisory|insecure|security|\bcross--origin\b|unauthori[z|s]ed|infinite.loop)`
medium_vuln_patterns	`(?i)(authenticat(e|ion)|brute.force|bypass|constant.time|crack|credential|\bDoS\b|expos(e|ing)|hack|harden|injection|lockout|overflow|password|\bPoC\b|proof.of.concept|poison|privelege|\b(in)?secur(e|ity)|de)?serializ|spoof|timing|traversal)`

TABLE II: Summary of approaches studied in this work and our prior work [7], and their utilized data: Expert Features (EF), Commit Messages (CM), and Commit Changes (CC).

Models	Technique	Features		
		EF	CM	CC
VCCFinder [3]	Machine Learning	✓	✓	
CodeJIT [4]	Graph-based Learning			✓
LR [8]	Machine Learning	✓		
TLEL [16]	Machine Learning	✓		
DeepJIT [13]	Deep Learning		✓	✓
LAPredict [11]	Machine Learning	✓		
SimCom [17]	Ensemble Learning	✓	✓	✓
JITFine [18]	Deep Learning	✓	✓	✓

TABLE III: Metrics supported for model evaluation. ED is threshold dependent. ID is threshold independent.

Type	Name	Description
DE	*Accuracy*	Correct predictions out of all predictions.
	Precision	True positives out of all predicted positives.
	Recall	True positives out of all actual positives.
	F1-score	Harmonic mean of precision and recall,
	MCC	Balanced measure of prediction quality factor in class imbalance [31].
ID	*ROC-AUC*	Area under ROC curve.
	PR-AUC	Area under Precision-Recall curve.
Effort	*Recall@20*	Percentage of actual positives found in the top 20% of ranked predictions.
	Effort@20	Percentage of code inspected to find top 20% of actual positives.
	P-opt	Measures effort saved when inspecting files in optimal versus actual order.

(e.g., [25], [26]). We also integrate other SZZ algorithms, such as B-SZZ [15], AG-SZZ [21], and MA-SZZ [27], for comparative evaluations.

4) Data Splitting: The dataset is partitioned using a customizable ratio. By default, all commits are ordered by date to simulate continuous software development [28] and avoid data leakage [29] and then split using a ratio of 75/5/20% for training/validation/testing.

5) Graph Builder: Many recent JIT-VP approaches utilize structural representations for prediction. As a result, VulGuard incorporates a graph builder module that is built on the artifacts provided by CodeJIT [4]. In their study, Nguyen et al. [4] leverage Joern [30] to generate code property graphs. This package is integrated into VulGuard with CLI.

B. Evaluation Modules

VulGuard provides a framework with model-level customization. Currently, we support eight prominent vulnerability prediction models: VCCFinder [3], CodeJIT [4], Logistic Regression (LR) [8], LAPredict [11], TLEL [16], DeepJIT [13], SimCom [17], and JIT-Fine [18]. These models represent a diverse set of techniques ranging from classical machine learning to deep learning and graph-based approaches. A summary of the methodology of each model is provided in Table II. Upon evaluation, VulGuard automatically computes standard classification metrics outlined in Table III.

IV. Usage

This section covers requirements and usages of VulGuard. Example commands are available at tool release [10].

A. Requirements

VulGuard is designed to operate on Linux-based systems equipped with GPU acceleration. For Linux users, we support installation via the Python library and Conda environment. For other platforms, it is recommended to follow the instructions in our package to build and construct your own Docker image.

B. Data Mining

Use case: Extract relevent commit data for JIT-VP. In addition, automatically identify vulnerability-fixing commits, and vulnerable introducing commits.

Preparation: VulGuard's input of mining process is local Git repository with main languages include C/C++, Java, JavaScript, and Python. By default, Vulguard identify patch commits using regular expression (Table I). However, you can provide the tool with customize patch commits by using jsonl file with each line following this format.

```
{
    "commit_id": <commit_id>,
    "Repository": <repo_name>
}
```

Command:

```
python -m vulguard.cli mining  \
    -dg_save_folder <save_folder> \
    -mode local \
    -repo_name <repository_name> \
    -repo_path <path/to/repository> \
    -repo_language <main_language_of_project> \
```

C. Model Evaluation

Use case: Train and evaluate implemented JIT-VP approaches. The trained models can be utilized for inference.

Preparation: To train and evaluate JIT-VP approaches, VulGuard leverages data extracted through a structured data mining process. By default, the dataset includes vulnerability-introducing, vulnerability-fixing, and unrelated commits. Users

TABLE IV: Results of the idealized setting experiment from our empirical study [7]. The highest values are in bold.

	Metric	VCCFinder	LAPredict	LR	TLEL	SimCom	DeepJIT	JITFine	CodeJIT	Average
FFmpeg	PR-AUC	0.895	0.558	0.780	0.850	0.921	0.906	**0.959**	0.798	0.833
	MCC	0.746	0.337	0.574	0.701	0.770	0.638	**0.864**	0.579	0.651
	F1-score	0.832	0.373	0.671	0.800	0.847	0.759	**0.909**	0.716	0.738
	ROC-AUC	0.948	0.620	0.865	0.918	0.954	0.946	**0.980**	0.838	0.884
Linux	PR-AUC	0.809	0.610	0.788	0.829	**0.892**	0.823	0.885	—	0.805
	MCC	0.447	0.358	0.558	0.627	0.658	0.613	**0.716**	—	0.568
	F1-score	0.664	0.458	0.695	0.752	0.786	0.735	**0.818**	—	0.701
	ROC-AUC	0.867	0.699	0.825	0.881	0.913	0.873	**0.915**	—	0.853

TABLE V: Results of the realistic setting experiment from our empirical study [7]. The highest values are in bold.

	Metric	VCCFinder	LAPredict	LR	TLEL	SimCom	DeepJIT	JITFine	CodeJIT	Average
FFmpeg	PR-AUC	0.071	0.041	0.093	0.112	**0.134**	0.082	0.111	0.079	0.091
	MCC	0.122	0.067	0.192	0.169	**0.226**	0.138	0.161	0.135	0.151
	F1-score	0.132	0.086	0.176	0.130	**0.231**	0.150	0.156	0.142	0.150
	ROC-AUC	0.688	0.591	0.769	0.795	**0.809**	0.746	0.790	0.721	0.738
Linux	PR-AUC	0.013	0.010	0.023	0.027	**0.031**	0.005	0.005	—	0.016
	MCC	0.030	0.028	0.070	**0.081**	0.073	0.000	0.000	—	0.040
	F1-score	0.036	0.025	**0.039**	0.038	0.034	0.011	0.011	—	0.028
	ROC-AUC	0.588	0.591	0.746	**0.787**	0.779	0.497	0.497	—	0.641

who wish to train or evaluate on a different dataset should provide a data file of which each line's format is:

```
{
    "commit_id": <commit_id>,
    "feature 1": <value_1>,
    ...
    "feature k": <value_k>,
    "label": <0 or 1>
}
```

Command:
Training:

```
python -m vulguard.cli training  \
    -dg_save_folder <save_folder> \
    -model <model_name> \
    -repo_name <project_name> \
    -repo_language <main_language_of_project> \
    -epochs <epochs>
```

Testing:

```
python -m vulguard.cli evaluating\
    -dg_save_folder <save_folder> \
    -model <model_name> \
    -repo_name <project_name> \
    -repo_language <main_language_of_project>
```

D. Model Inference

Use case: Utilized trained models to predict new commits.
Preparation: New commits must be extracted and provided in the same format as model evaluation before inference.
Command:

```
python -m vulguard.cli inference  \
    -dg_save_folder <save_folder> \
    -model <model_name> \
    -repo_name <project_name> \
    -repo_language <main_language_of_project>
```

V. DEMONSTRATION

VulGuard has been employed in the empirical study on JIT-VP presented in our recently accepted paper at ICSME 2025 [7]. The following section summarizes our experiments and findings, and showcases VulGuard's potential application.

TABLE VI: Commit distribution overview for FFmpeg and the Linux kernel. The table reveals the number of vulnerability-introducing commits (#VIC), vulnerability-fixing commits (#VFC), vulnerability-neural commits (#VNC), and the total number of commits in each data split.

Project	Split	#VIC	#VFC	#VNC	#Total
FFmpeg	Training	3,826	2,519	39,823	43,650
	Validation	255	330	3,242	3,827
	Testing	1,020	1,903	37,778	40,701
Linux Kernel	Training	3,461	1,735	796,965	800,426
	Validation	231	616	35,086	35,317
	Testing	922	1,691	157,039	157,961
Total		9,715	8,996	1,069,933	1,081,882

A. Data Mining

We apply VulGuard to mine commits from two widely used and actively maintained open-source projects: FFmpeg and the Linux kernel. These repositories are selected due to their extensive contribution histories. We collect all commits from the master branch as of September 24, 2024. The total number of commits from the two projects after the filtering process is 1,081,882. To accelerate the data extraction phase, we utilize parallel processing with 50 concurrent processes. It takes approximately **1 hour** for FFmpeg and roughly **12 hours** for Linux to complete data extraction. A detailed summary of the curated datasets is presented in Table VI.

B. Model Evaluation

Using VulGuard, we have conducted an empirical evaluation of the implemented JIT-VP models under two distinct settings: `ideal`, which includes only vulnerabilities and their corresponding fixing commits, and `realistic`, which also includes security-unrelated changes, resulting in data enlargement. The findings reveal a consistent and substantial decline in model performance when transitioning from the `ideal` to the `realistic` scenario across various evaluation metrics, i.e., PR-AUC, MCC, F1-score, and ROC-AUC. The detailed

results are shown in Tables IV and V for the Ideal and Realistic settings, respectively. For comprehensive analyses and discussions, please refer to our full research paper [7].

VI. Conclusion and Future Work

We introduced VulGuard, a unified and extensible tool that automates the end-to-end process of mining, processing, and analyzing software commits for JIT-VP research. Our empirical evaluation of two influential projects, FFmpeg and Linux kernel, demonstrates the tool's practical utility and effectiveness in real-world scenarios. Looking ahead, our goal is to enhance VulGuard by incorporating ensemble learning techniques and large language models like GPT-3/4, which have been shown to work well for function-level vulnerability prediction [32], to further boost JIT-VP predictive performance as well as extend it to other vulnerability tasks [33], [34].

References

[1] CrowdStrike, "Crowdstrike outage report," 2024. [Online]. Available: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf

[2] TechTarget, "Crowdstrike outage damage," 2024. [Online]. Available: https://www.techtarget.com/whatis/feature/Explaining-the-largest-IT-outage-in-history-and-whats-next

[3] 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 *the 22nd ACM SIGSAC conference on computer and communications security*, 2015, pp. 426–437.

[4] S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, and H. D. Vo, "Code-centric learning-based just-in-time vulnerability detection," *Journal of Systems and Software*, vol. 214, p. 112014, 2024.

[5] L. Yang, X. Li, and Y. Yu, "Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *GLOBECOM 2017-2017 IEEE Global Communications Conference*. IEEE, 2017, pp. 1–7.

[6] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi, "Just-in-time software vulnerability detection: Are we there yet?" *Journal of Systems and Software*, vol. 188, p. 111283, 2022.

[7] D. Nguyen, T. Le-Cong, T. Huynh Minh Le, M. A. Babar, and Q.-T. Huynh, "Toward realistic evaluations of just-in-time vulnerability prediction," in *the 41st International Conference on Software Maintenance and Evolution*. IEEE, 2025.

[8] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2012.

[9] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[10] D. Nguyen, M. Tran-Duc, T. Le-Cong, T. Huynh Minh Le, M. A. Babar, and Q.-T. Huynh, "Vulguard: An unified framework for evaluating just-in-time vulnerability prediction models," 2025. [Online]. Available: https://github.com/AI4Code-HUST/VulGuard

[11] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: how far are we?" in *the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 427–438.

[12] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 518–529.

[13] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "Deepjit: an end-to-end deep learning framework for just-in-time defect prediction," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 34–45.

[14] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," in *2015 IEEE International Conference on Software Quality, Reliability and Security*. Vancouver, BC, Canada: IEEE, Aug. 2015, pp. 17–26. [Online]. Available: http://ieeexplore.ieee.org/document/7272910/

[15] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[16] X. Yang, D. Lo, X. Xia, and J. Sun, "Tlel: A two-layer ensemble learning approach for just-in-time defect prediction," *Information and Software Technology*, vol. 87, pp. 206–220, 2017.

[17] X. Zhou, D. Han, and D. Lo, "Simple or complex? together for a more accurate just-in-time defect predictor," in *the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 229–240.

[18] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: integrating semantic features with expert features for defect prediction and localization," in *the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 672–683.

[19] C. Khanan, W. Luewichana, K. Pruktharathikoon, J. Jiarpakdee, C. Tantithamthavorn, M. Choetkiertikul, C. Ragkhitwetsagul, and T. Sunetnanta, "JITBot: an explainable just-in-time defect prediction bot," in *the 35th IEEE/ACM International Conference on Automated Software Engineering*. Virtual Event Australia: ACM, Dec. 2020, pp. 1336–1339. [Online]. Available: https://dl.acm.org/doi/10.1145/3324884.3415295

[20] GitHub, "Github action." [Online]. Available: https://github.com/features/actions

[21] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *the 21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[22] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," in *the 40th international conference on software engineering*, 2018, pp. 560–560.

[23] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[24] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *the 11th joint meeting on foundations of software engineering*, 2017, pp. 914–919.

[25] T. H. M. Le, X. Du, and M. A. Babar, "Are latent vulnerabilities hidden gems for software vulnerability prediction? an empirical study," in *the 21st International Conference on Mining Software Repositories*, 2024, pp. 716–727.

[26] S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, X. Liu, X. Lin, and W. Liu, "Snopy: Bridging sample denoising with causal graph learning for effective vulnerability detection," in *the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 606–618.

[27] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[28] A. K. Arani, T. H. M. Le, M. Zahedi, and M. A. Babar, "Systematic literature review on application of learning-based approaches in continuous integration," *IEEE Access*, 2024.

[29] T. H. M. Le, B. Sabir, and M. A. Babar, "Automated software vulnerability assessment with concept drift," in *the 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 371–382.

[30] Joernio, "Joern." [Online]. Available: https://github.com/joernio/joern

[31] T. H. M. Le and M. Ali Babar, "Mitigating data imbalance for software vulnerability assessment: Does data augmentation help?" in *the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2024, pp. 119–130.

[32] T. H. M. Le, M. A. Babar, and T. H. Thai, "Software vulnerability prediction in low-resource languages: An empirical study of codebert and chatgpt," in *the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 679–685.

[33] T. H. M. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–39, 2022.

[34] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.