PyResBugs: A Dataset of Residual Python Bugs for Natural Language-Driven Fault Injection

PyResBugs: A Dataset of Residual Python Bugs for Natural Language-Driven Fault Injection

Domenico Cotroneo, Giuseppe De Rosa, Pietro Liguori
University of Naples Federico II
Naples, Italy
cotroneo@unina.it, giuseppe.derosa20@unina.it, pietro.liguori@unina.it

***Abstract*—This paper presents *PyResBugs*, a curated dataset of *residual* bugs, i.e., defects that persist undetected during traditional testing but later surface in production—collected from major Python frameworks. Each bug in the dataset is paired with its corresponding fault-free (fixed) version and annotated with multi-level natural language (NL) descriptions. These NL descriptions enable natural language-driven fault injection, offering a novel approach to simulating real-world faults in software systems. By bridging the gap between Software Fault Injection techniques and real-world representativeness, *PyResBugs* provides researchers with a high-quality resource for advancing AI-driven automated testing in Python systems.**

***Index Terms*—Residual Bugs, Dataset, Python, Fault Injection, Natural Language**

I. Introduction

Modern society's dependence on software systems, powering critical services in healthcare, finance, and transportation, has grown alongside the increasing complexity of software. This complexity makes defect-free systems unattainable, as demonstrated by high-profile failures disrupting economies and services [1]–[5]. While rigorous testing mitigates many issues, fault-tolerance mechanisms are essential to handle the unforeseen faults that inevitably arise in complex systems [6], [7], especially in weakly typed or interpreted languages like Python, which are prone to runtime failures [8].

Software Fault Injection (SFI) has become a critical technique for evaluating system behavior under such fault scenarios [9]–[12]. By injecting controlled *faults* (i.e., *software bugs*), SFI helps identify hidden vulnerabilities and improves system robustness, making it a cornerstone of resilient software design. A critical challenge in SFI lies in addressing the complexity of *residual faults*, i.e., those hidden defects that elude traditional testing and persist in deployed systems. Achieving fault representativeness is key, as an ideal *faultload* (i.e., the set of faults to inject in the system) should emulate these elusive defects with high fidelity. However, accurately modeling residual faults remains daunting, as it requires replicating the diverse and often unpredictable human error patterns that give rise to software defects.

While significant research efforts have aimed at identifying and characterizing standard bug classes to approximate residual fault patterns [13]–[15], current approaches face three pivotal challenges. First, predefined fault models, often applied through pattern-matching techniques, struggle to capture the complexity and variability of real-world failure scenarios, leading to fault loads that inadequately reflect residual faults [16], [17]. Then, fault injectors and models are frequently tailored to specific programming paradigms or architectural styles, reducing their applicability across diverse software systems and diminishing their utility for comprehensive fault analysis [10], [18]–[20]. Finally, the complexity of configuring and using fault injection tools, along with steep learning curves, discourage widespread adoption and limits their effectiveness against residual faults [17], [21]–[23].

To address these challenges and advance research on residual faults, we propose *PyResBugs*, a comprehensive dataset containing $5,007$ residual bugs comprising pairs of fault-free and faulty code from major Python open-source frameworks. The standard driving idea is that faults disclosed by the test cases typically do not represent accurate residual faults [14], [15], as developers would identify and correct these during the development process. We collected faults by analyzing actual software bugs found in the field, using two sources: GitHub to collect a set of hard-to-find bugs (like concurrency, memory corruption, and security issues) in a custom dataset and established datasets from major conferences and journals. Each fault in our collection links to a specific commit where developers fixed a bug that users found after the software was released - meaning these are bugs that testing didn't catch.

To simplify fault generation for testers and developers who want to assess the behavior of systems against unforeseen faults, we annotated the faults with natural language (NL) descriptions of different levels of detail. This dataset enables the specialization of AI-powered solutions (e.g., Large Language Models, LLMs) to understand fault descriptions and automatically generate corresponding code defects. Users can interact with these models using NL instead of dealing with complex fault models, making fault injection accessible without requiring specialized expertise. Therefore, with *PyResBugs*, we aim to bridge the gap between the usage and accessibility of SFI tools while maintaining the complexity of real-world failures by exploiting the potential of NL annotations, a technique widely accepted in the software testing landscape [24].

The rest of the paper is structured as follows: Section II discusses the related work; Section III details the process of construction of the dataset and provides statistics; Section IV concludes the paper.

II. Related Work

Various datasets have emerged in the domain of Python complex systems, each addressing specific challenges and research goals. Table I summarizes the Python-focused datasets most relevant to our study. These datasets span a wide range of applications, from defect detection and debugging to vulnerability identification and patch analysis.

Despite their contributions, existing datasets often lack certain features essential for fault injection studies. Most datasets are designed to address general-purpose tasks, such as bug detection or patch analysis, rather than focusing on the challenges posed by residual faults. While less frequently studied, these faults are critical for understanding complex failure modes in real-world software systems. Another limitation of existing datasets is the granularity of metadata, as they lack detailed descriptions of fault behavior, which are crucial for advanced fault analysis and NL-driven fault injection. In contrast, our dataset enriches fault instances with NL descriptions, enabling more effective integration with AI-driven fault analysis tools. Furthermore, prior datasets often target narrowly defined tasks, such as vulnerability detection or framework-specific bug analysis, without offering a comprehensive view of functional fault diversity.

Our dataset addresses these gaps by combining the strengths of existing resources while introducing unique features tailored for fault injection studies. It aggregates diverse residual faults from multiple Python projects, ensuring functional diversity and reproducibility. Moreover, including detailed metadata supports advanced fault analysis, bridging the gap between reproducibility and AI-enabled evaluation. These features position our dataset as a valuable tool for researchers exploring fault injection, debugging, and security in Python systems.

TABLE I
List of Selected Python Datasets.

Author(s) (Year)	Description and Contribution	Focus
Cotroneo *et al.* (2019) [18]	OpenStack bug study: 179 critical fail-stop faults analyzed.	Fault Analysis
Widyasari *et al.* (2020) [26]	BugsInPy: 493 reproducible bugs from Python projects, aiding testing and debugging.	Bug Benchmarking
Akimova *et al.* (2021) [27]	PyTraceBugs: 24k buggy and 5.7M correct code samples from 11k+ repos.	DL Training
Bhandari *et al.* (2021) [28]	CVEFixes: Links 5,365 CVE records to 5,495 vulnerability fixing commits from 1,754 projects.	Vulnerability Analysis
Xu *et al.* (2022) [29]	Tracer: A framework for tracking vulnerability patches with a dataset of 1,295 CVEs.	Vulnerability Analysis
Sun *et al.* (2022) [30]	PySecDB: Python-focused dataset of 1,258 security commits and 2,791 non-security commits.	Vulnerability Analysis
Mahbub *et al.* (2023) [31]	Defectors: Includes 213K Python files with 93K labeled instances for defect prediction.	DL Training
Richter *et al.* (2023) [32]	A 33k Python bug fix dataset achieving 170% better neural bug detection vs artificial data.	DL Training
Akhoundali *et al.* (2023) [33]	MoreFixes: Curates 26,617 validated CVE fixes for security analysis.	Vulnerability Analysis
Du *et al.* (2023) [34]	Analyzes 3,555 bugs from DL-frameworks, with a focus on their classification and characteristics.	Fault Analysis
Tambon *et al.* (2023) [35]	SilentBugs: Analysis and classification of 77 "silent bugs" TensorFlow faults.	Fault Analysis

III. *PyResBugs* Dataset

This section details the creation of *PyResBugs*, a dataset comprising $5,007$ residual faults. These faults escaped detection during testing and release, only to surface later in operation, as evidenced by bug reports. Their occurrence highlights test suite limitations and the need for enhanced fault detection mechanisms [14], [15], [25].

The rest of the section details the steps involved in creating and curating the dataset. We begin by describing the process of fault collection (§ III-A), focusing on identifying and sourcing residual bugs from real-world projects. Next, we elaborate on the filtering and selection criteria applied to ensure the dataset's quality and relevance (§ III-B). Finally, we explain how we documented the faults using multi-level natural language descriptions (§ III-C) and systematically categorized (§ III-D), enabling their use in AI-driven fault injection and analysis.

A. Data Collection

To collect faults and build the *PyResBugs*, we systematically selected corpora from top-tier conferences and journals in the software engineering field, focusing on research published in the last 5 years. A critical insight guiding our approach, based on prior studies [36]–[38], is the recognition that vulnerabilities often represent residual faults—those escaping test suites and manifesting later as zero-day exploits. This assumption led us to incorporate Common Vulnerabilities and Exposures (CVEs) into the dataset, further enriching its utility for fault injection and defect prediction research.

Overall, we selected 11 datasets that span diverse domains, including software fault injection, automatic program repair, defect prediction, controlled testing, and vulnerability analysis, described in TABLE I

To further enhance the dataset's relevance and utility, we enriched it with additional residual fault categories critical for system testing yet underrepresented in existing datasets. These fault types, often particularly challenging for developers to identify, debug, and fix [39], [40], include Mandelbugs (e.g., aging-related bugs), Heisenbugs (e.g., concurrency issues), memory corruption bugs, high-priority or critical bugs, and security vulnerabilities. These categories were selected due to their significant impact on software reliability and the inherent difficulty in addressing them, making their inclusion essential for a comprehensive representation of real-world faults.

We carefully selected faults from GitHub repositories to ensure accurate representation by leveraging metadata such as bug reports, commit messages, and linked discussions to validate the fault's nature. This enrichment resulted in 67 additional faults, further broadening the scope and applicability of *PyResBugs* for fault injection and automated testing research.

To standardize data, we designed a composite key with commit hash, project URL, faulty code, and fixed code, ensuring traceability, reproducibility, and seamless analysis.

python/cpython commit 3b959dbcaf

Fault Free Code

```python
def put(self, item, block=1):
    # Same as before self.mutex.acquire()
    release_fsema = True
    try:
        was_empty = self._empty()
        self._put(item)
        if was_empty:
            self.esema.release()
        release_fsema = not self._full()
    finally:
        if release_fsema:
            self.fsema.release()
        self.mutex.release()
```

Faulty Code

```python
def put(self, item, block=1):
    if block:
        self.fsema.acquire()
    elif not self.fsema.acquire(0):
        raise Full
    self.mutex.acquire()
    was_empty = self._empty()
    self._put(item)
    if was_empty:
        self.esema.release()
    if not self._full():
        self.fsema.release()
    self.mutex.release()
```

Modify the put method to introduce a wrong algorithm small sparse modifications (WASM) fault. The function should fail due to removing the try block and the condition variable release_fsema = True, potentially causing deadlocks or inconsistent queue states in multi-threaded environments.

Implementation-Level Description

Modify the put method to introduce a wrong algorithm small sparse modifications (WASM) fault. The function should fail due to changing the release mechanism, potentially causing deadlocks or inconsistent queue states in multi-threaded environments.

Contextual-Level Description

Modify the put method to introduce a wrong algorithm small sparse modifications (WASM) fault.

High-Level Description

Fig. 1. Example of a deadlock fault in CPython: faulty and fault-free code versions, with three NL descriptions for bug injection in the *put* method

B. Data Filtering

We applied a rigorous filtering process to ensure that *PyResBugs* meets its primary objective of representing residual bugs. We guided this process by two core requirements, specifically designed to identify and include faults that align with the definition of residual bugs:

1) The fault must originate from the source code, specifically within a method implementation, and exclude issues related to configuration files, documentation, or build scripts to include only actual bugs encountered during software development.
2) The fault must have a corresponding bug report (identified via commit hashes, pull requests, issues, or formal bug reports) documenting its faulty and faulty-free version (i.e., the bug fix). This criterion ensures the representativeness of residual faults and supports reproducibility by focusing on verifiable defects encountered in real-world scenarios.

We focused on faults from widely adopted, complex Python projects to ensure dataset quality and relevance. We selected projects from diverse domains, including prominent frameworks and libraries such as pandas, CPython, Django, Ansible, Apache Airflow, scikit-learn, NumPy, Black, OpenStack, and Scrapy. We based project selection on community adoption metrics, with repository star counts ranging from 2.5k (Pycripto) to 187k (TensorFlow). While we primarily included faults from these high-impact projects, we made strategic exceptions for critical security vulnerabilities from less popular repositories to ensure comprehensive coverage of representative fault examples.

Then, we extracted each faulty code snippet and its corresponding patch at the method level. Our goal was to capture self-contained bugs that can affect multiple parts of the program through a single function or method call, making them particularly relevant for software fault injection. Although we acknowledge that many residual faults may arise from configuration errors or issues outside the code, we opted to focus on Python source code to train generative models tailored to code-centric fault injection. Consequently, we excluded non-code artifacts such as configuration files to consistently emphasize code-level defects. We employed specialized GitHub scraping tools to automate data collection, and to enhance clarity and reproducibility, we also extracted metadata such as commit messages, bug type classifications, CVE IDs (if available), associated test cases, and the Python version used.

Finally, we deduplicated using `[faulty code, fixed code]`, filtered invalid entries, and excluded comment-only changes to ensure consistency and remove redundancy. This process retained $5,007$ fault pairs.

C. Fault Description

We developed a comprehensive approach to describing faults to support the generation of faults directly from NL descriptions, hence streamlining the SFI process. To help model training across various prompt formats and simulate human tester behavior, we generated three levels of NL descriptions for each fault: *Implementation-Level Description*, *Contextual-Level Description*, and *High-Level Description*.

To understand the differences in the three NL descriptions, Fig. 1 illustrates the progression of fault descriptions for a bug in CPython's `put` method, showcasing the three levels of natural language. The figure visually ties these descriptions to the associated code changes in the Faulty Code (right) and Fault-Free Code (left), demonstrating how each level provides progressively less technical detail while retaining relevance for different fault modeling and testing purposes.

The Implementation-Level Description (bottom left) provides the most detailed and technical view, specifying the code changes needed to introduce the fault. It highlights the removal of the try block and the condition `release_fsma = True`, explaining how these changes cause deadlocks or inconsistent queue states in a multi-threaded environment.

Instead, the Contextual-Level Description (center bottom) slightly abstracts the fault by focusing on its general mechanism and potential impact rather than specific code modifications. It mentions modifying the put method and altering the release mechanism, leading to potential issues such as deadlocks or inconsistent states but avoids specifying exact code lines. This level provides testers with a broader understanding of the fault's behavior and consequences.

In the High-Level Description (bottom right), we make the description entirely abstract and omit technical or contextual details about the specific fault. Modifying the put method introduces a "*wrong algorithm small sparse modifications fault*" in the fault-free function. This description suits scenarios where a conceptual understanding of the fault type is sufficient without providing implementation specifics.

A team of six researchers specialized in computer engineering and cybersecurity created and validated the fault descriptions, under the coordination of a full professor with extensive expertise in software testing and fault injection. The professor established the description style, while the postdoctoral researcher, with a PhD in information technologies and background in AI and fault injection, provided ongoing reviews and feedback. The team, which also included a PhD student in cybersecurity and four M.Sc. thesis students, worked together over six months to generate and validate the descriptions. Finally, the professor, postdoc, and PhD student conducted a comprehensive review to ensure consistency and quality.

We involved multiple researchers in generating fault descriptions to ensure quality, reduce workload, and maintain accuracy. We preferred human-generated descriptions over AI-based solutions to avoid biases and reliance on repetitive phrasing. Moreover, the team's diverse linguistic styles and technical expertise introduced natural variability, enhancing the dataset's robustness and improving the generalization capabilities of downstream models.

The final dataset integrates the $5,007$ collected faults with their corresponding three levels of NL descriptions. Each sample in *PyResBugs* includes the three NL descriptions (Implementation-Level, Contextual-Level, and High-Level) detailing how to inject the fault, the corresponding fault-free code (the base for injecting the fault), and the resulting faulty code generated by applying the described fault.

We analyzed the complexity of tokens, the NL descriptions, and both the faulty and fault-free functions. The NL descriptions accompanying the faults add an extra layer of richness to the dataset. High-level descriptions, which are the most abstract, average 14 tokens (SD: 3.39), while Contextual-Level Descriptions average 32 tokens (SD: 5.6). Implementation-level descriptions, the most detailed, average 34 tokens (SD: 4.8). This progression reflects the growing detail and technicality of the descriptions, meeting diverse research needs.

The analysis of fault-free code and its faulty code counterparts shows minimal variation in token counts. Fault-free code snippets average 74 tokens (SD: 104), while Faulty Code snippets average 71 (SD: 103). This balance highlights that the faults are typically well-localized and do not result in excessive or extraneous code changes, maintaining the dataset's focus on precise and meaningful fault examples.

D. Fault Classification

To systematically categorize these residual faults, we adopted the extended version of the Orthogonal Defect Classification (ODC) framework developed by Duraes *et al.* [15], with further refinements from subsequent studies [36], [41]. This classification schema, tailored for SFI, is designed to identify and emulate realistic software faults. It categorizes defects based on their characteristics (e.g., whether the code is missing, incorrect, or extraneous) and the nature of the fault (e.g., assignments, conditionals, function calls, or algorithms), containing a total of 61 different fault categories.

To comprehensively understand the dataset, we analyzed the distribution of fault types across the integrated software systems. The dataset encompasses 56 distinct fault types out of a possible 61 categories, with the remaining five categories absent due to their association with C-style programming patterns, which are uncommon in Python.

We also analyzed the most frequent fault categories to highlight prevalent challenges in Python software development (we limited the discussion to the top 3 due to space limit). Three key fault types dominate the dataset: Missing Parameter in Function Call (MPFC), accounting for 10.78% of the dataset; Missing IF Construct plus Statements (MIFS), contributing 10.11%; and Wrong Algorithm - Large Modifications (WALL), representing 7.92%. According to the ODC, these categories address critical aspects of software faults. MPFC faults, categorized as Interface issues, arise when required parameters are omitted in function calls, often causing runtime errors. MIFS faults, under Checking, involve the absence of conditional logic (e.g., if statements), leading to incorrect execution paths. WALL faults, classified as Algorithm issues, represent significant errors in algorithm implementation, such as flawed logic or design structure. Other fault categories in our dataset include Wrong Function Called with Different Parameters (WFCD, 6.64%), Wrong Variable Used in Parameter of Function Call (WPFV, 5.77%) and Wrong Function Called with Same Parameters (WFCS, 5.57%).

IV. Conclusion and Future Work

We introduced *PyResBugs*, the first dataset specifically designed for SFI in Python applications, focusing exclusively on residual faults. The dataset comprises $5,007$ curated pairs of faulty and fault-free code snippets extracted from real-world Python projects, each enriched with three levels of NL descriptions and detailed metadata.

The dataset is publicly available on GitHub [1]. Future work will focus on developing an SFI tool powered by a model

[1]https://github.com/dessertlab/PyResBugs

trained on *PyResBugs*, showcasing the dataset's effectiveness and advancing the field of automated fault injection.

V. Acknowledgments

Work supported by the *IDA—Information Disorder Awareness* Project, funded by the EU–Next Generation EU under the SERICS program (Grant PE00000014, MUR NRRP).

References

[1] B. I. Staff, "Citigroup accidentally sent $500 million to revlon's lenders because of a poorly designed user interface in oracle flexcube," *Business Insider*, February 2021. Accessed: 2024-11-21.

[2] C. Staff, "Uber self-driving crash in arizona leaves pedestrian dead," March 2018. Accessed: 2024-11-21.

[3] C. Staff, "Crowdstrike outage causes financial and operational impact: The cost and root cause revealed," *CNN*, July 2024. Accessed: 2024-11-21.

[4] I. S. Staff, "How the boeing 737 max disaster looks to a software developer," *IEEE Spectrum*, 2019. Accessed: 2024-11-21.

[5] H. Dolfing, "Project failure case study: Knight capital group," *Henrico Dolfing Blog*, June 2019. Accessed: 2024-11-21.

[6] NASA, "Nasa software safety guidebook," Tech. Rep. NASA-GB-8719.13, NASA, 2004. Accessed: 2024-11-21.

[7] I. O. for Standardization, "Product development: Software level," 2011. Accessed: 2024-11-21.

[8] S. Nanz and C. A. Furia, "A comparative study of programming languages in rosetta code," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 778–788, 2015.

[9] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using Context-Sensitive software fault injection," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2595–2612, USENIX Association, Aug. 2020.

[10] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Tensorfi: A flexible fault injection framework for tensorflow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pp. 426–435, IEEE, 2020.

[11] D. Cotroneo and P. Liguori, "Neural fault injection: Generating software faults from natural language," in *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024 - Supplemental Volume*, pp. 23–27, IEEE, 2024.

[12] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, 2016.

[13] T. Titcheu Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, pp. 434–487, 2020.

[14] R. Natella, D. Cotroneo, J. A. Duraes, and H. S. Madeira, "On fault representativeness of software fault injection," *IEEE TSE*, vol. 39, no. 1, pp. 80–96, 2012.

[15] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, 2006.

[16] P. Costa, J. G. Silva, and H. Madeira, "Practical and representative faultloads for large-scale software systems," *Journal of Systems and Software*, vol. 103, pp. 182–197, 2015.

[17] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Profipy: Programmable software fault injection as-a-service," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 364–372, 2020.

[18] D. Cotroneo, L. De Simone, P. Liguori, R. Natella, and N. Bidokhti, "How bad can a bug get? an empirical analysis of software failures in the openstack cloud computing platform," in *ESEC/FSE 2019*, p. 200–211, 2019.

[19] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Enhancing the analysis of software failures in cloud computing systems with deep learning," *Journal of Systems and Software*, vol. 181, p. 111043, 2021.

[20] D. Cotroneo, L. De Simone, P. Liguori, and R. Natella, "Fault injection analytics: A novel approach to discover failure modes in cloud-computing systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1476–1491, 2022.

[21] F. Huang, B. Liu, and B. Huang, "A taxonomy system to identify human error causes for software defects," in *The 18th international conference on reliability and quality in design*, pp. 44–49, 2012.

[22] F. Huang, "Human error analysis in software engineering," *Theory and Application on Cognitive Factors and Risk Management: New Trends and Procedures*, p. 19, 2017.

[23] D. Cotroneo, L. De Simone, and R. Natella, "Thorfi: a novel approach for network fault injection as a service," *Journal of Network and Computer Applications*, vol. 201, p. 103334, 2022.

[24] O. A. Nielsen and M. G. Lange, "The use of data in bus fleet management: A systematic literature review," *Transportation Research Part C: Emerging Technologies*, vol. 116, p. 102664, 2020.

[25] X. Zhang, X. Teng, and H. Pham, "Considering fault removal efficiency in software reliability assessment," *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, vol. 33, no. 1, pp. 114–120, 2003.

[26] R. e. a. Widyasari, "Bugsinpy: A database of existing bugs in python programs," in *Proc. ESEC/FSE 2020*, pp. 1556–1560, 2020.

[27] E. N. Akimova, A. Y. Bersenev, A. A. Deikov, K. S. Kobylkin, A. V. Konygin, I. P. Mezentsev, and V. E. Misilov, "Pytracebugs: A large python code dataset for supervised machine learning in software defect prediction," in *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 141–151, 2021.

[28] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, (New York, NY, USA), p. 30–39, Association for Computing Machinery, 2021.

[29] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vulnerabilities," in *ESEC/FSE 2022*, pp. 860–871, ACM, 2022.

[30] S. Sun, S. Wang, X. Wang, Y. Xing, E. Zhang, and K. Sun, "Exploring security commits in python," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 171–181, 2023.

[31] P. Mahbub, O. Shuvo, and M. Masudur Rahman, "Defectors: A large, diverse python dataset for defect prediction," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 393–397, 2023.

[32] C. Richter and H. Wehrheim, "How to train your neural bug detector: Artificial vs real bugs," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1036–1048, 2023.

[33] J. e. a. Akhoundali, "Morefixes: A large-scale dataset of cve fix commits mined through enhanced repository discovery," in *Proceedings of the 20th International Conference on Predictive Models and Data Analytics in Software Engineering*, p. 42–51, 2024.

[34] X. Du, Y. Sui, Z. Liu, and J. Ai, "An empirical study of fault triggers in deep learning frameworks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 4, pp. 2696–2712, 2023.

[35] F. Tambon, A. Nikanjam, L. An, F. Khomh, and G. Antoniol, "Silent bugs in deep learning frameworks: an empirical study of keras and tensorflow," *Empirical Softw. Engg.*, vol. 29, Nov. 2023.

[36] J. Fonseca and M. Vieira, "Mapping software faults with web security vulnerabilities," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pp. 257–266, 2008.

[37] J. Fonseca, N. Seixas, M. Vieira, and H. Madeira, "Analysis of field data on web security vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 2, pp. 89–100, 2014.

[38] F. Cerveira, R. Barbosa, M. Mercier, and H. Madeira, "On the emulation of vulnerabilities through software fault injection," in *2017 13th European Dependable Computing Conference (EDCC)*, pp. 73–78, 2017.

[39] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 178–187, 2013.

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

[41] G. Pereira, R. Barbosa, and H. Madeira, "Practical emulation of software defects in source code," in *2016 12th European Dependable Computing Conference (EDCC)*, pp. 130–140, 2016.