White-box Fuzzing in the Wild: A Chaos Engineering Module for DevOps Security Education

Md Mostafizur Rahman[1], Md Bajlur Rashid[2] MD Abdul Barek[3], Abm Kamrul Islam Riad[4]
Md Abdur Rahman[5], Hossain Shahriar[6], Akond Rahman[7], Fan Wu[8], Dr. Guillermo Francia III[9],
Md Jobair Hossain Faruk[10], Sharaban Tahora[11]
[1234569]University of West Florida, Florida, USA
[7]Auburn University, USA
[8]Tuskegee University, Alabama, USA
[10]University of Central Arkansas, USA
[11]Kennesaw State University, USA
md.mostafizur.rn@gmail.com, bajlur.rashid.sourav@gmail.com, barek2k2@gmail.com, kamrulriad75@gmail.com,
mr252@students.uwf.edu, hshahriar@uwf.edu, akond@auburn.edu, fwu@tuskegee.edu
gfranciaiii@uwf.edu, mhossainfaruk@uca.edu,Sharaban.shetu@gmail.com

***Abstract*—In today's fast-paced software development environments, DevOps has revolutionized the way teams build, test, and deploy applications by emphasizing automation, collaboration, and continuous integration/continuous delivery (CI/CD). However, with these advancements comes an increased need to address security proactively, giving rise to the DevSecOps movement, which integrates security practices into every phase of the software development lifecycle. DevOps security remains underrepresented in academic curricula despite its growing importance in the industry. To address this gap, this paper presents a hands-on learning module that combines Chaos Engineering and White-box Fuzzing to teach core principles of secure DevOps practices in an authentic, scenario-driven environment. Chaos Engineering allows students to intentionally disrupt systems to observe and understand their resilience, while White-box Fuzzing enables systematic exploration of internal code paths to discover corner-case vulnerabilities that typical tests might miss. The module was deployed across three academic institutions, and both pre- and post-surveys were conducted to evaluate its impact. Pre-survey data revealed that while most students had prior experience in software engineering and cybersecurity, the majority lacked exposure to DevOps security concepts. Post-survey responses gathered through ten structured questions showed highly positive feedback 66.7% of students strongly agreed, and 22.2% agreed that the hands-on labs improved their understanding of secure DevOps practices. Participants also reported increased confidence in secure coding, vulnerability detection, and resilient infrastructure design. These findings support the integration of experiential learning techniques like chaos simulations and white-box fuzzing into security education. By aligning academic training with real-world industry needs, this module effectively prepares students for the complex challenges of modern software development and operations.**

***Index Terms*—DevOps Security, Secure Software Development, Cybersecurity Education, White box Fuzzing, Chaos Engineering, CI/CD**

I. Introduction

To this end, white-box fuzzing plays a complementary role. Unlike traditional black-box fuzzing, which generates random inputs to a system under test, white-box fuzzing leverages source code to intelligently generate inputs that maximize code coverage and expose edge cases and latent bugs [1], [2]. It is especially effective at revealing corner cases such as division by zero, unexpected input formats, and boundary condition violations that could lead to critical failures or security vulnerabilities if left unchecked. In this paper, we present a focused educational module that merges these two paradigms, Chaos Engineering and white-box fuzzing, through the development of a simple fuzzer targeting a basic calculator application. Though the application itself is minimalistic, it serves as an ideal case study to demonstrate how systematic test input generation and fault injection can uncover subtle bugs that traditional testing methods may overlook [3].

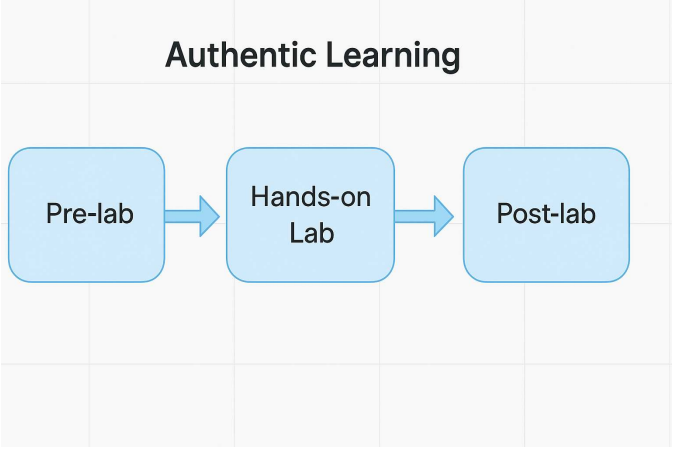

Fig. 1. Labware Setup

This module is part of a broader initiative involving the design and implementation of ten modular learning experiences, each tailored to reinforce core concepts in cybersecurity, DevOps, and software resilience. These modules have been intentionally structured to include pre-lab content

dissemination, guided hands-on lab exercises, and post-lab reflections, ensuring a full-spectrum experiential learning cycle that supports skill acquisition, critical thinking, and retention. [4], [5] The specific white-box fuzzing module enriches the curriculum by exposing learners to real-world testing strategies that go beyond superficial correctness. It promotes a deeper understanding of how automated input generation, code instrumentation, and control flow analysis can be applied to detect software flaws before they manifest in production. By engaging students in designing a simple yet effective fuzzer, this module instils practical knowledge about software quality assurance and secure coding practices [6]. Moreover, the modular design of the curriculum ensures accessibility and adaptability across diverse educational environments [7], [8]. Whether in university classrooms, bootcamps, or online learning platforms, these modules provide scalable, authentic learning experiences grounded in industry-relevant challenges. By combining theory with hands-on experimentation, this initiative aims to produce graduates equipped with both the conceptual foundation and applied skills needed to thrive in today's fast-evolving software landscape.

II. Background And Motivation

As software systems become increasingly complex and interconnected, the potential impact of system failures has grown significantly. Ensuring that these systems are robust, secure, and fault-tolerant has thus become a primary concern for software engineers. Traditional testing methods, while essential, often fall short in detecting subtle bugs that emerge only under rare or extreme conditions. This limitation has led to the growing adoption of Chaos Engineering and fuzz testing as advanced strategies to proactively expose system weaknesses [9]–[11]. Chaos Engineering introduces the idea of injecting controlled failures into systems to observe and analyze their behavior under stress. Originally popularized by Netflix's 'Chaos Monkey,' this discipline has expanded to include fault injection at various levels, from infrastructure to application logic. Its primary goal is to ensure that systems can maintain functionality and recover gracefully even when unexpected issues arise. Educationally, chaos engineering cultivates a mindset of anticipating failure and building for resilience—essential qualities in modern software development [12] [13].

On the other hand, fuzz testing (fuzzing) involves providing invalid, unexpected, or random data as inputs to a software program to find bugs and vulnerabilities. While black-box fuzzing treats the system as opaque, white-box fuzzing utilizes program structure, control flow, and symbolic execution to intelligently generate test cases that achieve higher code coverage and uncover edge cases [1] [14] [15]. This form of fuzzing is particularly effective at identifying logical errors and security flaws in source code. For example, symbolic execution and search-based techniques have been used effectively to improve the precision of fuzzers in RESTful APIs and web front-end applications [2] [1] [16]. The intersection of chaos engineering and white-box fuzzing presents a powerful opportunity to explore software robustness both at runtime and at the code level. By incorporating both strategies into the curriculum, students are exposed to the dual perspectives of fault injection and systematic test generation. However, these topics are often underrepresented in traditional computer science education, which typically focuses on static learning formats or predefined exercises with known outcomes [17] [18]. To address this gap, we developed a modular educational framework that includes ten independent yet interconnected modules, each targeting a key area in cybersecurity, DevOps, and system reliability. The white-box fuzzing module discussed in this paper is one such module. By centering the lab around the development of a fuzzer for a simple calculator application, we ensure the exercise is approachable while still illustrating the power of systematic bug discovery. This exercise empowers learners to engage directly with concepts such as input space exploration, assertion testing, control flow analysis, and fault detection.

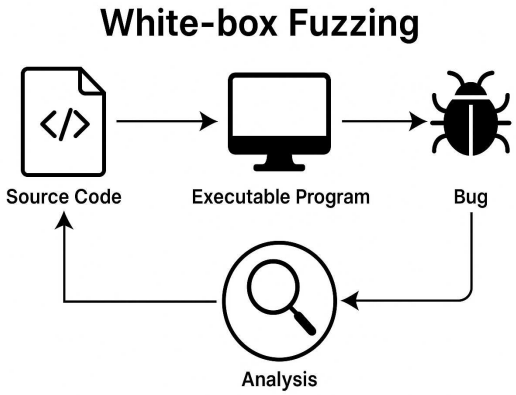

Fig. 2. Workflow of White box fuzzing

III. Related Work

Golmohammadi [2] conducted a systematic literature review of 92 studies focused on RESTful API testing. The study revealed that approximately 72% of the reviewed research emphasized black-box testing techniques, indicating a significant underrepresentation of white-box approaches. Golmohammadi's work underscores the potential of white-box testing in generating high-coverage test cases by leveraging internal code structures. The study also proposes the use of search-based software testing (SBST), which frames test generation as an optimization problem to be solved using search algorithms. This method offers an automated and potentially more cost-effective alternative to manual testing. Furthermore, the study identifies the lack of REST API case studies and limited research into white-box heuristics as critical gaps in the literature, providing direction for future exploration.

Complementary research by Yang, Zhang, and Fu (2013) [19] presents a white-box combinatorial fuzzing framework that incorporates symbolic execution and combinatorial test-

ing. Their approach focuses on simulating attacks across multiple input points by analyzing constraint conditions using static analysis and symbolic execution. The framework intelligently reduces the input space using interval computation and constraint solving, allowing for efficient and targeted test generation. This method effectively mitigates the combinatorial explosion problem often associated with fuzzing and demonstrates success in exposing deep-seated bugs within large-scale software systems.

Kertusha (2024) [1] highlights this gap by proposing a white-box, search-based methodology to address the limitations of current black-box systems. While symbolic execution and evolutionary algorithms have been applied in the backend or system-level testing contexts (Yang, Zhang, & Fu, 2013), their application in frontend testing remains largely unexplored. The proposed integration of Search-Based Software Testing (SBST) and white-box analysis represents a promising direction to improve test completeness and efficiency by directly leveraging internal code logic.

Xiao-chen, Jing-yan, and Jie (2009) [20] proposed an innovative approach that combines cost-significant theory with chaos theory for engineering cost estimation. Their method integrates autocorrelation analysis, G-P algorithms, and Lyapunov exponent calculations to model historical time-series data, demonstrating that engineering costs exhibit chaotic behavior suitable for short-term prediction. Their work shows promise in reducing estimation workload and increasing forecast accuracy by leveraging nonlinear dynamics.

The approach advocated by De (2021) [12] underscores the importance of chaos experiments in real-world cloud systems, particularly during high-traffic events such as e-commerce mega sales. De introduces a framework rooted in the Cynefin model, categorizing system components based on their complexity (simple, complicated, complex, chaotic), thus helping prioritize and guide chaos experimentation based on contextual risk.

Borkar et al. [15] introduced WhisperFuzz, a white-box fuzzing technique focused on detecting and locating timing vulnerabilities in processors. By analyzing the microarchitectural state transitions at the register-transfer level (RTL), this approach overcomes the limitations of traditional fuzzing, enabling precise identification of vulnerabilities in hardware design.

Zhang et al. [21] presented a white-box fuzzing strategy for RPC-based APIs using EvoMaster in industrial settings. Their methodology improves code coverage and fault detection by instrumenting internal API structures, making it suitable for continuous testing in large-scale software systems.

Alshmrany et al. [14] developed FuSeBMC, a white-box fuzzer combining symbolic execution with bounded model checking to identify security vulnerabilities in C programs. This hybrid approach enables the discovery of deep, hard-to-reach bugs that standard fuzzing might miss.

Yang et al. [22] proposed WhiteFox, a compiler fuzzer that integrates large language models (LLMs) with white-box analysis to test deep learning compilers. The LLMs analyze compiler internals and generate test programs that trigger optimization bugs, showing the potential of AI-augmented white-box fuzzing.

Luo et al. [16] introduced RESTler, an intelligent REST API fuzzing tool that automatically generates test cases using Swagger specifications. The tool leverages a dynamic feedback loop to learn API dependencies and optimize test sequence generation. Their approach demonstrated high bug-finding capability and represents a concrete implementation of white-box fuzzing in API environments.

Takanen et al. [23] provided a comprehensive overview of fuzzing in software security, emphasizing its evolution toward intelligent and grammar-based techniques. The work draws attention to the critical role fuzzing plays in secure software pipelines and identifies white-box fuzzing as a frontier requiring deeper exploration, particularly in education and resilience training.

Chen and Williams [18] explored the use of mutation analysis for assessing the effectiveness of security testing in DevSecOps. They concluded that while many security test suites focus on known vulnerabilities, integrating white-box fuzzing significantly improves fault detection rates, especially for zero-day vulnerabilities.

In the realm of software testing education, Johnson et al. [17] designed an interactive learning module on DevOps principles and found that hands-on labs involving fault injection significantly improved student comprehension and retention. However, their module lacked a focus on white-box fuzzing or Chaos Engineering, highlighting a gap our study seeks to address.

Finally, Ahmad et al. [13] proposed a resilience-focused learning environment using simulated CI/CD pipelines. Their study emphasized the importance of simulating real-world attacks and failure scenarios but acknowledged the need for advanced test generation techniques like symbolic execution and evolutionary fuzzing.

Although prior research has contributed significantly to the understanding of security in DevOps pipelines, it is important to note that none of these studies have explicitly addressed the integration of Chaos Engineering and white-box fuzzing within DevOps security education. While DevSecOps practices and security automation tools have gained traction in industry and academia, the pedagogical application of Chaos Engineering as a means to teach resilience and proactive failure discovery in secure software development remains underexplored. Similarly, white-box fuzzing has primarily been discussed in the context of vulnerability discovery rather than as a component of educational frameworks. Hence, our research aims to bridge this gap by designing an authentic learning module [5], [9] that incorporates Chaos Engineering and white-box fuzzing to teach core DevOps security concepts. This module enables learners to simulate fault injection and analyze internal system states, thereby fostering a deeper understanding of secure, fault-tolerant software design within continuous delivery environments.

IV. Methodology

The methodology of this workshop module is grounded in experiential learning and follows a structured format: pre-lab orientation, guided hands-on implementation, and post-lab reflection. The goal is to foster both conceptual understanding and technical proficiency through active problem-solving.

A. Pre-Lab Orientation

Before beginning the lab, students are introduced to the theoretical underpinnings of both chaos engineering and white-box fuzzing. This includes:

- The rationale behind chaos engineering and its applications.
- Differences between black-box, gray-box, and white-box fuzzing.
- The role of code coverage and symbolic execution in fuzz testing.
- Examples of real-world failures discovered through fuzzing tools like AFL, KLEE, and libFuzzer.

Students are also provided with a starter codebase: a basic calculator implementation supporting operations such as addition, subtraction, multiplication, and division. The calculator includes intentional vulnerabilities, such as unchecked division, overflow risks, and poor input sanitization, to serve as test targets.

B. Hands-On Lab Implementation

During the lab, students are tasked with designing and implementing a white-box fuzzer in a high-level language (e.g., Python or Java). The fuzzer uses knowledge of the calculator's source code to:

- Generate structured input data (e.g., mathematical expressions).
- Target-specific operations and boundary conditions.
- Inject invalid or extreme values (e.g., divide by zero, large integers).
- Evaluate output and log failures or exceptions.

Key steps include:

1) Input generation: Using input templates and randomization within defined constraints.
2) Execution & Monitoring: Running the calculator with the generated inputs and recording any run-time errors or assertion failures.
3) Logging & Analysis: Track test cases that trigger unexpected behavior and analyze control paths taken during execution.

Students are encouraged to iteratively refine their fuzzer to improve code coverage and discover more complex bugs. Optional extensions include the use of code instrumentation tools to visualize execution paths.

C. Post-Lab Reflection

After completing the implementation, students reflect on their findings through a structured post-lab questionnaire and group discussion. They analyze:

- What types of bugs were discovered and why?
- How the input variation affected the discovery.
- How white-box knowledge contributed to testing efficiency.
- How fuzzing complements chaos engineering principles.

This reflection reinforces the connection between theory and practice and prepares students to apply these methods in more complex environments.

V. Survey Findings

The module was implemented in three academic institutions in the spring of 2025. To evaluate its effectiveness, we conducted both quantitative and qualitative surveys. Participants completed pre-lab and post-lab surveys, which included various questions designed to collect feedback and assess learning outcomes.

A. Pre-Lab Survey

Twelve people took part in the survey. Most of them (41.7%) were between 36 and 45 years old. Two age groups, 26 to 35 and 46 to 55, each comprised 25% of the participants. Only one person (8.3%) was in the 18 to 25 age range. This shows that most of the participants were older and likely had more life and work experience, which added depth to group discussions and workshop activities.

TABLE I
Age Group Distribution

Age Group	Count	%
18-25 years	1	8.3%
25-35 years	3	25.0%
36-45 years	5	41.7%
46-55 years	3	25.0%

In terms of programming experience, the survey revealed that most participants possessed a strong foundation in software development. Python emerged as the most familiar programming language among respondents, with 18 students reporting varying degrees of expertise ranging from moderate to expert-level proficiency. Java, C, and C++ were also widely recognized, each receiving substantial responses across different skill levels. In contrast, languages such as Ruby, Perl, and R had limited exposure, with only one or two students indicating any experience. These results suggest a high level of technical fluency among participants, especially in mainstream programming languages commonly used in systems-level and software engineering projects. The breadth of programming knowledge among students highlights a strong foundation upon which advanced topics like chaos engineering and fuzzing can be effectively introduced.

The survey included questions about how participants prefer to learn. They were asked to rate how much they agreed or disagreed with six different learning style statements. The results are shown in Table II.

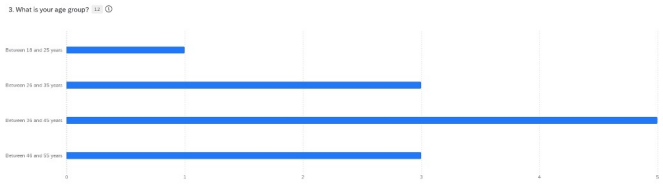

Fig. 3. Age Group

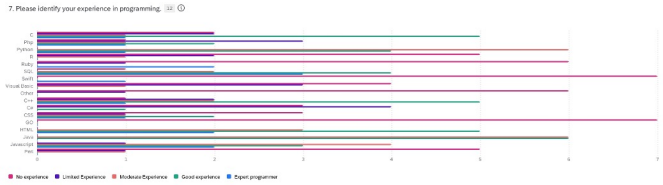

Fig. 4. Programming Proficiencies

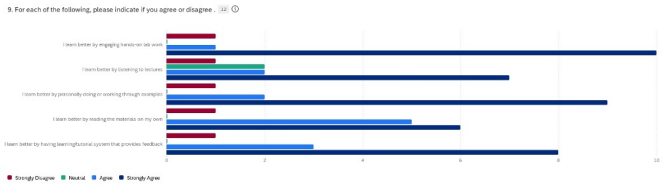

Fig. 5. Students Learning Preferences

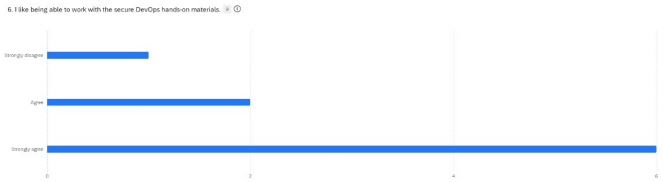

Fig. 6. Q: I like being able to work with the secure DevOps hands-on materials

Most participants preferred active learning. Ten people strongly agreed that they learn best by doing hands-on lab work. Eight strongly agreed that they learn better by working through examples. These responses support the workshop's focus on practical, experience-based learning.

Seven participants also strongly agreed that learning tools with feedback helped them understand better. Reading on their own was also popular—six strongly agreed that it worked for them—although two strongly disagreed, showing some mixed feelings about self-study.

Listening to lectures was the least preferred method. Only one person strongly agreed with that statement, while others either disagreed or felt neutral. This shows that most participants prefer more interactive and engaging learning environments.

TABLE II
LEARNING PREFERENCE RESPONSES

Learning Statement	**Strongly Disagree**	**Neutral**	**Agree**	**Strongly Agree**
I learn better by engaging in hands-on lab work	1	1	0	10
I learn better by listening to lectures	1	2	2	1
I learn better by personally doing or working through examples	1	1	2	8
I learn better by reading the materials on my own	2	0	4	6
I learn better by having learning/tutorial systems that provide feedback	1	1	3	7

B. Post-Lab Survey

As part of evaluating the effectiveness of the secure DevOps hands-on lab modules, a post-survey consisting of ten questions was distributed to the participants. The survey aimed to gather feedback on various aspects of the learning experience, including engagement, conceptual understanding, and practical skill development. Responses were collected from nine participants, and the overall results indicate a strongly positive reception toward the hands-on learning approach.

One key area of focus was whether participants enjoyed working with the secure DevOps hands-on materials. A significant majority—66.7% strongly agreed, while 22.2% agreed, totaling nearly 89% of participants who found the materials enjoyable and engaging. This highlights the value of interactive learning in maintaining student interest and motivation.

In terms of educational impact, the hands-on labs were found to be effective in enhancing understanding of DevOps security concepts. Again, 66.7% of participants strongly agreed, and 22.2% agreed that the labs helped them better understand secure DevOps practices. Only 11.1% strongly disagreed, suggesting minimal negative feedback and reinforcing the overall effectiveness of the approach.

Furthermore, participants were asked if the labs helped improve their learning experience around secure coding and DevOps best practices. The feedback remained consistent, with 55.6% strongly agreeing and 33.3% agreeing—totaling 88.9% of participants who recognized a meaningful learning benefit from the labs. This clearly demonstrates that practical exercises not only support theory but also build technical confidence and real-world applicability.

Overall, the post-survey results reveal that the majority of participants responded positively across all ten questions. The consistent pattern of agreement underscores the value of integrating hands-on labs into DevOps and cybersecurity education. These findings suggest that experiential learning approaches are not only well-received but also play a critical role in equipping learners with the skills and knowledge required to succeed in modern secure software development environments.

Fig. 7. Q: The hands-on labs help me understand better on DevOps security

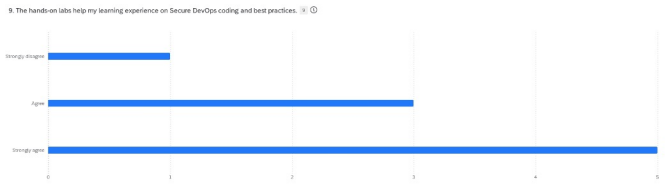

Fig. 8. The hands-on labs help my learning experience on Secure DevOps coding and best practices

VI. Conclusion

This paper presented a hands-on educational module that combines Chaos Engineering and White-box Fuzzing to enhance student understanding of secure DevOps practices. By integrating fault injection, systematic code testing, and secure pipeline configuration into a single experiential learning experience, the module addresses a growing need for practical, security-focused training in software engineering education. Results from pre- and post-surveys indicate a notable improvement in students' comprehension of DevOps security concepts, with participants demonstrating increased confidence in applying secure coding principles, identifying vulnerabilities, and reasoning about system resilience.

While the findings are encouraging, this study is not without limitations. The sample size was relatively small, with few number of participants completing the post-survey, which may affect the generalizability of the results. Additionally, the module was implemented within a controlled academic environment and may require adaptation to reflect the complexity and scale of real-world production systems. Technical limitations also existed in the scope of tools used, as the module focused on foundational exercises and did not incorporate advanced automation or AI-driven fuzzing strategies.

Despite these constraints, the study provides strong evidence supporting the value of hands-on, scenario-driven approaches in teaching secure DevOps. Future enhancements to the module will aim to address these limitations by expanding its deployment across more diverse learning contexts, integrating scalable cloud environments, and incorporating deeper tooling for continuous security validation. Overall, this work contributes to ongoing efforts to modernize cybersecurity education and align academic training with real-world industry needs.

Acknowledgment

This research is funded by the National Science Foundation through grants NSF Award #2310179, #2209637, #2421324, #2433800, and #1946442. The views, conclusions, and recommendations presented in this material are those of the authors and do not necessarily represent the perspectives of the National Science Foundation.

References

[1] I. Kertusha, "Search-based white-box fuzzing of web frontend applications," in *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 95–98, 2024.

[2] A. Golmohammadi, "Enhancing white-box search-based testing of restful apis," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 9–12, 2023.

[3] K. R. Khamdamovich and I. Aziz, "Techniques and methods of black box identifying vulnerabilities in web servers," in *2021 International Conference on Information Science and Communications Technologies (ICISCT)*, pp. 1–4, 2021.

[4] M. M. Rahman, A. Siddika Arshi, M. M. Hasan, S. Farzana Mishu, H. Shahriar, and F. Wu, "Security risk and attacks in ai: A survey of security and privacy," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1834–1839, 2023.

[5] M. M. Rahman, M. A. Barek, M. S. Akter, A. K. Islam Riad, M. A. Rahman, H. Shahriar, A. Rahman, and F. Wu, "Authentic learning on devops security with labware: Git hooks to facilitate automated security static analysis," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 2418–2423, 2024.

[6] M. A. Barek, M. M. Rahman, S. Akter, A. K. Islam Riad, M. A. Rahman, H. Shahriar, A. Rahman, and F. Wu, "Mitigating insecure outputs in large language models(llms): A practical educational module," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 2424–2429, 2024.

[7] A. K. Islam Riad, M. A. Barek, M. M. Rahman, M. S. Akter, T. Islam, M. A. Rahman, M. R. Mia, H. Shahriar, F. Wu, and S. I. Ahamed, "Enhancing hipaa compliance in ai-driven mhealth devices security and privacy," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 2430–2435, 2024.

[8] M. S. Akter, J. Rodriguez-Cardenas, M. M. Rahman, H. Shahriar, A. Rahman, and F. Wu, "Teaching devops security education with hands-on labware: Automated detection of security weakness in python," *arXiv preprint arXiv:2311.16944*, 2023.

[9] M. S. Akter, M. A. Rahman, M. M. Rahman, J. Rodriguez-Cardenas, H. Shahriar, F. Wu, and M. Rahman, "Authentic learning approach for data poisoning vulnerability in llms," in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 1504–1505, 2024.

[10] M. S. Akter, M. A. Barek, M. M. Rahman, A. K. I. Riad, M. A. Rahman, M. R. Mia, H. Shahriar, W. Chu, and S. I. Ahamed, "Hipaa technical compliance evaluation of laravel-based mhealth apps," in *2024 IEEE International Conference on Digital Health (ICDH)*, pp. 58–67, 2024.

[11] M. A. Rahman, M. A. Barek, A. K. I. Riad, M. M. Rahman, M. B. Rashid, S. Ambedkar, M. R. Miaa, F. Wu, A. Cuzzocrea, and S. I. Ahamed, "Embedding with large language models for classification of hipaa safeguard compliance rules," 2024.

[12] S. De, "A study on chaos engineering for improving cloud software quality and reliability," in *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*, vol. 1, pp. 289–294, IEEE, 2021.

[13] N. Ahmad, K. Patel, and R. Roy, "Building resilient devops pipelines for education: A fault-injection learning environment," *Journal of Systems and Software*, vol. 186, p. 111211, 2022.

[14] K. M. Alshmrany, R. S. Menezes, M. R. Gadelha, and L. C. Cordeiro, "Fusebmc: A white-box fuzzer for finding security vulnerabilities in c programs (competition contribution)," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 363–367, Springer International Publishing Cham, 2021.

[15] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "{WhisperFuzz}:{White-Box} fuzzing for detecting and locating timing vulnerabilities in processors," in *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 5377–5394, 2024.

[16] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," pp. 748–758, 05 2019.

[17] B. Johnson, C. Smith, and A. Thompson, "Interactive devops education: Improving learning outcomes with hands-on labs," in *Proceedings of the 2021 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, (Virtual Event, Germany), pp. 183–189, ACM, 2021.

[18] Z. Chen and L. Williams, "Assessing the effectiveness of security testing in devsecops with mutation analysis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, (Porto, Portugal), pp. 293–303, IEEE, 2020.

[19] J. Yang, H. Zhang, and J. Fu, "A fuzzing framework based on symbolic execution and combinatorial testing," in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*, pp. 2076–2080, IEEE, 2013.

[20] D. Xiao-chen, L. Jing-yan, and D. Jie, "Study on engineering cost estimation based on chaos theory and cost-significant theory," in *2009 ISECS International Colloquium on Computing, Communication, Control, and Management*, vol. 2, pp. 556–559, IEEE, 2009.

[21] M. Zhang, A. Arcuri, Y. Li, Y. Liu, and K. Xue, "White-box fuzzing rpc-based apis with evomaster: An industrial case study," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–38, 2023.

[22] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box compiler fuzzing empowered by large language models," *CoRR*, 2023.

[23] J. D. D. A. Takanen and C. Miller, "Fuzzing for software security testing and quality assurance," *Norwood, MA, USA: Artech House*, 2018.