



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

RESEARCH METHODS AND PROFESSIONAL DEVELOPMENT

ACFA7004 Assessment Report

Student Name:
Habibullah Shaikh

Student ID:
2503799

Programme:
MSc Cybersecurity and Digital Forensics

Submission Date:
Friday, 1st May 2026

Table of Contents

1. Introduction	3
1.1 Context and Background	3
1.2 Research Problem	3
1.3 Research Aim and Objectives.....	4
1.4 Why This Research Matters.....	5
2. Literature Review	5
2.1 The AppSec Threat Landscape	5
2.2 Traditional Security Testing Tools	5
2.3 How AI Is Changing Vulnerability Detection	6
2.4 Automated Vulnerability Repair.....	6
2.5 Security Inside CI/CD Pipelines (DevSecOps)	7
2.6 What the Literature Is Missing	7
3. Research Design and Methodology	8
3.1 Research Design	8
3.2 Research Methods.....	8
3.3 System Architecture.....	8
3.4 Data, Tools and Hardware	9
3.4.1 Datasets.....	9
3.4.2 Software Stack.....	9
3.4.3 Hardware Requirements	10
4. Expected Outcomes	10
4.1 What Results Are Anticipated	10
4.2 Broader Impact	10
4.3 Honest Assessment of Limitations	10
5. Project Timeline	10
6. References	12

1. Introduction

1.1 Context and Background

Consider each time you use an app to make a payment, schedule a doctor's appointment online, or log into your bank. Software security is essential to all of these. The field of application security (AppSec) focuses on preventing applications from being compromised, altered, or abused. Secure code, testing, and system monitoring are all covered. To put it simply, AppSec shields your private data from hackers.

The stakes are really high. The average worldwide cost of a data breach was USD 4.88 million, up 10% from the year before, according to IBM's Cost of a Data Breach Report 2024. This increases to more than USD 9.77 million per breach for healthcare organizations. Financial loss, record leaks, and long-term harm to trust are some of these effects. [1]

The problem's scope keeps expanding. Early in 2024, Cloudflare reported processing 57 million HTTP requests per second and filtering 209 billion threats in a single day an 86.6 percent increase over the previous year. [2] Meanwhile, 40,077 vulnerabilities were reported in 2024, a 38 percent increase from 2023, indicating that a new vulnerability was reported almost every 13 minutes. [3]

The most important dangers are highlighted in the OWASP Top 10. Broken Access Control is ranked highest in the 2021 edition, indicating that attackers are taking advantage of lax permission systems. Additionally, it introduced Insecure Design, which reflects the fact that many problems are caused by faulty architecture rather than merely coding mistakes. There are still many frequent hazards, such as SQL Injection, Cryptographic Failures, and Security Misconfiguration. [4]

Both attackers and defenses are now being impacted by artificial intelligence. AI is used by attackers to identify weaknesses more quickly and produce phishing assaults that are more convincing. After being made public, some exploits are released within 22 minutes. [2] By analyzing code more intelligently, AI-powered defensive technologies are enhancing vulnerability detection.

Another significant change is DevSecOps, which incorporates security into the development process instead of viewing it as an afterthought. Resolving problems early in the development process is far less expensive than resolving them later production. Organizations still have to deal with issues including developer resistance, false positives, and tool complexity. [5]

1.2 Research Problem

Application breaches are still increasing despite industry expertise and a variety of sophisticated techniques. The inability of current tools to reliably identify vulnerabilities and provide developers with sufficient guidance on how to repair them is a major contributing factor.

Static Application Security Testing (SAST) is one of the most used tool categories. With the goal of identifying problems early, these tools examine source code without running it. In actuality, though, they produce a large percentage of false positives. 53% of SAST alarms are bogus, which means that more than half of the warnings indicate safe code, according to a 2024 study. [6] According to NIST study, false positive rates for Java might be as high as 78%. [7]

Alert tiredness results from this. Developers start ignoring warnings, including real threats, when they are overloaded with them. According to Black Duck's 2024 research, 60% of enterprises claim that between 21% and 60% of their security warnings are noise, including duplicates and false positives. [8] This diminishes the efficacy of security tools and erodes public confidence in them.

A different strategy is used by Dynamic Application Security Testing (DAST) tools, which test

active applications by simulating threats. They have limits even though they lower false positives. Early-stage testing is challenging because they are unable to access source code, can only identify vulnerabilities brought on by their test cases, and need a deployed product. [9]

Remediation is another significant problem. Developers frequently lack clear instructions on how to address vulnerabilities, even when tools accurately detect them. Chen et al.'s research reveals that many security fixes either create additional problems or don't completely fix the initial vulnerability. [10] The majority of tools are good at spotting issues, but they don't offer many context-aware options for properly resolving them.

The fundamental issue is that remediation and detection are viewed as distinct problems. In contemporary CI/CD pipelines, few solutions or research initiatives integrate intelligent, developer-friendly remediation with precise vulnerability detection. As a result, there is a disconnect between finding a vulnerability and fixing it.

1.3 Research Aim and Objectives

The objective is to reduce false positives, improve detection coverage across OWASP Top 10 categories, and assist developers in fixing vulnerabilities more quickly and accurately by designing, developing, and assessing an AI-powered Application Security framework that combines smarter vulnerability detection with context-aware automated remediation recommendations, delivered inside a CI/CD pipeline.

Obj.	SMART Criteria	What This Research Will Do
1	Specific + Time-bound	Identify research gaps and set performance benchmarks to direct the framework design, which should be finished by the end of the second month, by conducting a comprehensive systematic literature assessment of AI-based vulnerability detection and remediation methodologies.
2	Specific + Measurable	Create and train a hybrid AI model that combines CodeBERT, a top code understanding model, with Graph Neural Networks (GNNs) to categorize vulnerabilities in all OWASP Top 10 categories, with a target precision of 85% or higher and recall of 80% or higher on common benchmark datasets.
3	Specific + Achievable	Create a functional prototype of the framework integrated into a GitHub Actions CI/CD pipeline that, when tested on at least two actual open-source repositories, automatically scans code on each commit and generates a prioritized vulnerability report in less than 60 seconds for codebases up to 100,000 lines.
4	Specific + Measurable	Using GPT-4, provide an automated remediation recommendation engine that produces fix recommendations specific to the code context and has a validated correct-fix rate of at least 70% based on expert assessment.
5	Specific + Measurable + Time-bound	By Month 9, assess the entire framework using precision, recall, F1-score, and false positive rate metrics against three industry tools (SonarQube, Checkmarx, and Semgrep) and three popular benchmark datasets (BigVul, the Juliet Test Suite, and OWASP WebGoat).
6	Specific + Achievable	Conduct a developer usability research with at least ten people to evaluate the framework's usability, the value of the remediation recommendations, and the remaining obstacles to practical implementation.

1.4 Why This Research Matters

Public safety, financial stability, and trust are all directly impacted by software security, which is no longer merely a technical problem. Data theft, monetary loss, and service interruptions can result from breaches in government, banking, or healthcare systems. These mishaps frequently result from vulnerabilities that may have been avoided with better tools.

There are obvious financial advantages to early detection as well. According to IBM's 2024 research, companies who contained breaches within 200 days saved an average of USD 1.12 million as opposed to those that responded more slowly. [1] Thus, there is a cost and security benefit to including AI-driven security early in the development process.

From a research standpoint, integrating AI-based detection and remediation into actual CI/CD workflows is obviously lacking. By offering an open-source implementation and reproducible evaluation methodology, our study seeks to address that issue and contribute to more useful and developer-friendly AppSec solutions.

2. Literature Review

This literature review looks at important research on AI in Application Security. Its goal is to explain what works, what does not, and what problems still need solving. It focuses on studies from 2018 to 2024 and covers key areas: security threats, limits of current testing tools, use of AI for finding vulnerabilities, automated fixing of issues, and how security fits into DevSecOps.

2.1 The AppSec Threat Landscape

Software flaws follow recurrent patterns rather than occurring at random. Organizations like MITRE categorize them in a methodical manner using the Common Weakness Enumeration (CWE), which identifies different kinds of software flaws and identifies the most serious ones in its yearly Top 25 list. [11] In a similar vein, the OWASP Top 10 uses industry data to rank these problems in practical application situations. [4]

Two important discoveries were presented in the 2021 OWASP Top 10. Attackers frequently take advantage of inadequate permission systems, as seen by the top risk being broken access control. Additionally, Insecure Design was introduced, highlighting the fact that many significant problems are caused by architectural defects rather than straightforward code mistakes. This implies that technologies that are solely concerned with code-level errors may overlook important design issues. [4]

These tendencies are supported by research. According to Harer et al. (2018), the majority of high-severity exploits are caused by memory safety and injection flaws. [12] Lipp et al. (2022) have demonstrated that injection and authentication problems often recur even after solutions, emphasizing that successful remediation is just as crucial as detection. [13]

2.2 Traditional Security Testing Tools

Most organisations use three main types of security testing tools: SAST, DAST, and Software Composition Analysis (SCA). Each is useful, but all have important limits, especially in fast-paced development environments.

Static Application Security Testing (SAST) tools examine source code without running it. They can be used early in development, which is an advantage. Popular tools include SonarQube, Checkmarx,

and Fortify. However, studies show they are often inaccurate. A 2020 study found false positive rates between 48% and 79% across six SAST tools on real Java projects. [9] A 2024 study of 815 vulnerable code commits found a single SAST tool detected only half of the real issues, while 76% of its alerts were irrelevant. [6] This means developers miss real problems while being flooded with mostly useless warnings.

Dynamic Application Security Testing (DAST) tools test applications while they are running, sending attack payloads to see how the system responds. They can find issues SAST misses, but they cannot see the code itself. This makes it hard for developers to locate and fix problems. Studies show DAST tools cover less than 40% of complex application flows [14] and can have false positive rates up to 82%. [7]

Software Composition Analysis (SCA) tools check third-party libraries for known vulnerabilities, which became especially important after attacks like Log4Shell in 2021. These tools help prevent supply chain risks but struggle against attacks that do not match known vulnerability signatures. A 2020 study showed SCA tools were mostly ineffective against malicious packages with intent-based attacks. [15]

2.3 How AI Is Changing Vulnerability Detection

Researchers are now using machine learning (ML) and deep learning for vulnerability identification due to the limitations of rule-based techniques. These techniques learn patterns from thousands of samples rather than establishing rules.

Code Property Graphs (CPGs), which integrate code structure (AST), execution logic (CFG), and data dependencies (PDG) into a single graph, were first presented by Yamaguchi et al. (2014). [16] As a result, relationships in code that are missed by basic text scanners can be understood by ML models.

Using bidirectional LSTMs on code slices, Li et al. (2018) developed VulDeePecker, a deep learning system that achieved a mere 3% false positive rate on the NVD/NIST dataset—much better than the 48–79% rates of SAST methods. [17] With Devign, Zhou et al. (2019) enhanced this by employing Graph Convolutional Networks on CPGs for improved generalization across vulnerability kinds. [18]

Large pre-trained code models are used in recent developments. A transformer that may be optimized for vulnerability identification, CodeBERT (Feng et al., 2020) was trained on millions of code lines in six different languages. [19] In 2024, accuracy increased by 8.34% and F1-score by 29.71% when CodeBERT and CNNs were combined. [20] CodeBERT in conjunction with enhanced Program Dependency Graphs works better than either strategy alone, according to a different 2024 study. [21]

But there are still holes. The majority of models are not evaluated in actual CI/CD pipelines; instead, they are primarily tested on benchmark datasets. Few integrate vulnerability detection with advice on how to remedy them. According to a 2024 arXiv paper, CodeBERT can still miss the majority of genuine vulnerabilities under difficult circumstances, with false negative rates exceeding 80% on the more difficult PrimeVul dataset. [22]

2.4 Automated Vulnerability Repair

Identifying a vulnerability is only half the problem; appropriately correcting it is much more difficult. For many years, Automated Program Repair (APR) has investigated this. Although they could provide syntactically valid patches, early systems like Prophet (Long and Rinard, 2016) had poor correct-fix rates and concentrated on functional rather than security flaws. (23)

One of the first to use neural techniques particularly to address security flaws was Chen et al.'s SecurityRepair (2023). On 1,000 real-world vulnerabilities, it achieved a 22.3% correct end-to-end patch rate, which is a helpful baseline but insufficient for production. (10)

This has been improved with the emergence of Large Language Models (LLMs). In a zero-shot context, Pearce et al. (2023) evaluated GPT-4 on CWE-classified vulnerabilities, i.e., it was asked to patch code without any particular training instances. GPT-4 significantly outperformed earlier techniques by correctly fixing 65.2% of vulnerabilities. [24] The investigation was conducted outside of actual development workflows, and performance still differed depending on the type of vulnerability.

There is still a significant gap: almost all studies handle remediation and detection independently. Few solutions integrate context-aware AI repair with AI-driven detection in a single end-to-end pipeline. The goal of this study is to close that gap.

2.5 Security Inside CI/CD Pipelines (DevSecOps)

DevSecOps is crucial since all security concepts and tools are only practical if they are integrated into actual development processes. DevSecOps is defined by NIST SP 800-204C as incorporating security into DevOps so that it is an ongoing, automated, shared responsibility rather than a last stage prior to delivery. [25]

In reality, a lot of organizations fall well short of this goal. Only 9% of the 16 open-source CI/CD pipelines that Rahman and Williams (2020) examined had any security scanning, primarily SCA for dependencies rather than complete AppSec testing. [26]

Although tool use is growing, according to the 2024 Black Duck Global State of DevSecOps Report, 82% of firms currently utilize 6–20 security solutions, which causes issues for developers like noise, conflicts, and complexity. Although more than 90% of people utilize AI tools, protecting code produced by AI is still difficult. [8]

Mohan and Othmane (2016) noted that inadequate developer feedback, integration complexity, and scan time are common obstacles to DevSecOps adoption. [27] These, along with problems like false positives, opaque AI choices, incompatibility with legacy systems, and developer aversion to slow tools, are still significant in 2024. [5]

2.6 What the Literature Is Missing

Having surveyed the state of research, three connected gaps emerge clearly. First, while AI-based detection models have made impressive progress on benchmark datasets, they have not been systematically evaluated in real CI/CD pipeline contexts at production speed and scale. Second, the combination of AI-driven detection with AI-driven, context-aware remediation in a single integrated system has not been studied. Third, the practical developer experience of using AI-powered AppSec tools remains largely underexplored, leaving a gap between what the research community has built and what developers would actually find useful. This research proposal directly addresses all three of these gaps.

3. Research Design and Methodology

3.1 Research Design

The Design Science Research (DSR) methodology described by Hevner et al. (2004) is used in this study. [28] DSR is appropriate since the objective is to create a novel solution more precisely, an AI-powered AppSec framework—rather than merely research current issues. It also calls for the solution to be thoroughly tested and to offer practitioners and researchers important information.

There are three primary stages to the project. The problem definition and literature review come first. The AI models, remediation system, and CI/CD integration are developed in design and development, which comes in second. The third step is assessment and reflection, during which the framework is put to the test and input is gathered. The design phase will include several cycles of training, testing, and improvement prior to final review because AI development is iterative.

3.2 Research Methods

A mixed-methods strategy is used in this study. The framework is tested on benchmark datasets through controlled experiments, and it is compared with current tools using standard metrics in the quantitative portion. A developer usability research is included in a smaller qualitative section to determine how well the framework fits actual processes and what needs to be improved.

Benchmarking along with experimental software development is the approach. This is necessary to verify the main assertion that, in comparison to SAST techniques, the AI framework may increase detection accuracy and decrease false positives. Three tools are compared across three standard datasets to validate the results.

3.3 System Architecture

The proposed framework has four main components that work together as a pipeline. The diagram below summarises how they connect.

Component	What It Does and Why
Code Representation Layer	Captures syntax, control flow, and data dependencies in a single structure by converting source code into a Code Property Graph (CPG). This aids the AI in identifying more serious flaws that are missed by basic text scanning. The open-source Joern tool was used in its construction.
Vulnerability Detection Module	An AI engine that looks for vulnerabilities in source code and CPG. It blends a refined CodeBERT model (for code meaning) with a Graph Neural Network (for structure patterns). For improved performance, an ensemble technique is used to combine their results. Goal: recall $\geq 80\%$, precision $\geq 85\%$.
Remediation Recommendation Engine	This module leverages GPT-4 to produce explicit repair recommendations after a vulnerability is discovered. The advice is specific rather than general since it takes into account the code environment, CWE type, and framework features. Aim for a correct-fix rate of at least 70%.
CI/CD Integration Layer	The framework is automatically executed on every push or pull request using a GitHub Actions plugin. It provides fix recommendations, indicates problems directly in the code, and generates results in SARIF format. The goal is to scan up to 100,000 lines of code in less than 60 seconds.

Standard metrics are used for evaluation, including Precision, Recall, F1-score, AUC-ROC, False Positive Rate, and False Negative Rate, which are reported both overall and for each type of vulnerability. Expert ratings on a five-point scale and comparison with known proper fixes are used to evaluate the remediation module. Paired t-tests are used to examine statistical differences between the framework and baseline tools.

3.4 Data, Tools and Hardware

3.4.1 Datasets

Dataset	Where It Comes From	What It Contains	Why It Was Chosen
BigVul	Fan et al. (2020) GitHub [29]	3,754 real-world C/C++ vulnerability commits with CVE and CWE IDs from 348 open-source projects.	Extensively utilized benchmark dataset for direct comparison with Devign and VulDeePecker.
Juliet Test Suite v1.3	NIST / NSA [30]	118 CWE categories with known ground truth are covered by synthetic C/C++ and Java test cases.	Allows for accurate assessment of detection accuracy for particular types of vulnerabilities.
OWASP WebGoat	OWASP Foundation [4]	Java web application that is purposefully vulnerable and covers the whole OWASP Top 10	supports complete pipeline testing rather than isolated snippets on realistic multi-vulnerability applications.
NIST NVD CVE Database	NIST NVD [31]	Includes more than 200,000 known vulnerabilities along with CWE mappings, CVSS scores, and impacted versions.	Provides severity-weighted performance evaluation and offers external validation ground truth.

Ethics: No personal information is present in any of the public databases. With written informed consent, anonymized data, secure storage, and adherence to university ethics and the UK GDPR, the developer research is voluntary. Prior to hiring, complete ethics approval will be acquired.

3.4.2 Software Stack

The following tools and frameworks will be used throughout the project:

- Python 3.11 Main development language for all data processing, modelling, and evaluation scripts
- PyTorch 2.x with PyTorch Geometric (PyG) Building and training the Graph Neural Network component
- Hugging Face Transformers Fine-tuning and running the CodeBERT vulnerability detection model
- Joern Open-source tool for generating Code Property Graphs from C/C++ and Java source code
- OpenAI GPT-4 API Powering the remediation recommendation engine
- GitHub Actions CI/CD pipeline automation and framework integration
- Docker Containerising the framework components for consistent, portable deployment

- SonarQube, Checkmarx, Semgrep Baseline comparison tools against which the framework's performance is measured
- scikit-learn, pandas, numpy, matplotlib, seaborn Statistical analysis, metric calculation, and result visualisation

3.4.3 Hardware Requirements

- GPU: NVIDIA A100 80GB (available via Google Colab Pro+) or equivalent cloud GPU instance (AWS p3.2xlarge) required for GNN and CodeBERT model training
- Cloud compute: Google Cloud Platform or AWS for scalable CI/CD pipeline testing across multiple repository configurations
- Local development machine: Standard modern laptop or workstation for coding, scripting, and results analysis.

4. Expected Outcomes

4.1 What Results Are Anticipated

The hybrid GNN-CodeBERT model is expected to detect vulnerabilities with over 85% precision and 80% recall on BigVul and Juliet benchmarks, much better than typical SAST tools (50–60% F1-score). [9] False positives should drop below 20%, compared to 48–79% for SAST tools and 53% for current alerts. [6][9]

The remediation engine aims to correctly fix at least 70% of vulnerabilities, slightly better than GPT-4's 65.2% zero-shot results, [24] by using code context, CWE type, and framework info. The CI/CD pipeline should scan up to 100,000 lines in under 60 seconds.

4.2 Broader Impact

If successful, the framework helps three groups. Developers get quick, accurate vulnerability checks with clear fix advice, making security easier to use. Researchers get open-source code, model weights, and a way to test tools for future studies. Everyone else benefits because safer software reduces financial losses and personal harm from hacks.

4.3 Honest Assessment of Limitations

This project has some limits. It only works on C/C++ and Java for now, so other languages like Python or JavaScript need more training. Benchmark datasets might not match real-world code perfectly. The fix suggestions use GPT-4, which costs money and might not always be available. The developer study has only ten participants, so the results give useful ideas but can't represent all developers. These limits will be clearly explained in the final report.

5. Project Timeline

The table below lays out a 12-month project schedule. The timeline is designed to be realistic it builds in iteration time during model development and includes a two-week buffer between the end of development (Month 8) and the start of formal evaluation (Month 9) to absorb any delays. Supervisor review meetings are planned at the end of each phase.

Month	Phase	Main Activities	Milestone
1	Phase 1: Foundation	Review the literature, finalize research plan and goals, list needed datasets and tools, and submit the ethics form.	Literature review complete; ethics application submitted
2	Phase 1: Foundation	Download and prepare datasets, set up Joern for CPGs, and configure baseline tools (SonarQube, Checkmarx, Semgrep) for testing.	Working CPG pipeline; baseline tools configured (O1 achieved)
3 to 4	Phase 2: Model Development	Train GNN on Juliet Test Suite, fine-tune CodeBERT on BigVul, test each model on benchmarks, and review errors for improvements.	Individually trained GNN and CodeBERT models with initial performance metrics
5	Phase 2: Model Development	Design and train ensemble fusion architecture combining GNN and CodeBERT; hyperparameter optimisation; validate against target precision and recall metrics	Hybrid model achieving $\geq 85\%$ precision and $\geq 80\%$ recall (O2 achieved)
6 to 7	Phase 3: Framework Build	Develop GitHub Actions plugin; implement SARIF output generation; integrate detection module into CI/CD pipeline; test on two open-source repositories	Working CI/CD-integrated prototype with sub-60-second scans (O3 achieved)
8	Phase 3: Framework Build	Build GPT-4 remediation engine with structured context prompting; design fix evaluation methodology; expert review panel recruitment	Remediation module with $\geq 70\%$ verified correct-fix rate (O4 achieved)
9	Phase 4: Evaluation	Full benchmark evaluation across BigVul, Juliet, and WebGoat; head-to-head comparison with three baseline tools; statistical significance testing	Evaluation report with comparative metrics (O5 achieved)
10	Phase 4: Evaluation	Developer usability study: participant recruitment, informed consent, study sessions, data collection; qualitative and quantitative data analysis	Usability study complete; developer feedback report (O6 achieved)
11	Phase 5: Write-Up	Write methodology, results, and discussion chapters; produce visualisations; circulate draft to supervisor for review	Full research report draft submitted for supervisor feedback
12	Phase 5: Write-Up	Revise based on supervisor feedback; prepare final submission; package and publish open-source code and evaluation framework	Final research report submitted; public GitHub repository released

6. References

- [1] IBM Security, "Cost of a Data Breach Report 2024," IBM Corporation, July 2024.
- [2] Cloudflare, "Application Security Report: 2024 Update," Cloudflare Blog, July 2024.
- [3] SiteWALL, "2024 Web Application Security Statistics," sitewall.net, 2025.
- [4] OWASP Foundation, "OWASP Top Ten 2021," Open Web Application Security Project, 2021.
- [5] B. Singh, "Automating Security Testing in CI/CD Pipelines using DevSecOps Tools A Comprehensive Study," World Journal of Advanced Research and Reviews, vol. 22, no. 1, pp. 2083–2093, 2024.
- [6] Endor Labs, "False Negatives in SAST: Hidden Risks Behind the Noise," 2024.
- [7] Contrast Security, "AppSec Noise and Fatigue by the Numbers," Contrast Security Infographic, 2023
- [8] Black Duck, "2024 Global State of DevSecOps Report," Black Duck Software, October 2024
- [9] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "SAST tools: Are we there yet?" in Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 489–500. doi: 10.1109/SANER48275.2020.9054817
- [10] Y. Chen, S. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in C programs," IEEE Transactions on Software Engineering, vol. 49, no. 1, pp. 159–178, Jan. 2023.
- [11] MITRE Corporation, "2023 CWE Top 25 Most Dangerous Software Weaknesses," MITRE, 2023.
- [12] J. A. Harer et al., "Automated software vulnerability detection with machine learning," arXiv preprint arXiv:1803.04497, 2018.
- [13] Lipp, Banescu & Pretschner (2022) — An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection (ACM ISSTA 2022)
- [14] A. Doupé, M. Cova, and G. Vigna, "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners," in Proc. DIMVA, 2010, pp. 111–131. doi: 10.1007/978-3-642-14215-4_7
- [15] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in Proc. DIMVA, 2020, pp. 23–43.
- [16] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in Proc. IEEE Symposium on Security and Privacy, 2014, pp. 590–604. doi: 10.1109/SP.2014.44
- [17] Z. Li et al., "VulDeePecker: A deep learning-based system for vulnerability detection," in Proc. NDSS, 2018.
- [18] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in Proc. NeurIPS, 2019.
- [19] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings of ACL: EMNLP 2020

- [20] R. S. Mim, A. Satter, T. Ahammed, and K. Sakib, "Automated Software Vulnerability Detection Using CodeBERT and Convolutional Neural Network," in Proc. ENASE 2024, pp. 156–167.
- [21] Anonymous, "Code vulnerability detection based on augmented program dependency graph and optimized CodeBERT," Scientific Reports, March 2024. doi: 10.1038/s41598-025-23029-4
- [22] Anonymous, "Vulnerability Detection with Code Language Models: How Far Are We?" arXiv preprint. arXiv:2403.18624, 2024
- [23] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2016.
- [24] H. Pearce et al., "Examining zero-shot vulnerability repair with large language models," in Proc. IEEE Symposium on Security and Privacy (SP), 2023, pp. 2339–2356.
- [25] M. Souppaya, J. Morello, and K. Scarfone, "NIST SP 800-204C: Implementation of DevSecOps for a Microservices-based Application with Service Mesh," NIST, 2022.
- [26] M. M. Rahman and L. Williams, "Security smells in Ansible and Chef scripts: A replication study," ACM TOSEM, vol. 29, no. 4, Aug. 2020.
- [27] V. S. Mohan and L. B. Othmane, "SecDevOps: Is it a marketing buzzword? Mapping research on security in DevOps," in Proc. IEEE SecDevOps, 2016. doi: 10.1109/SecDev.2016.024
- [28] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," MIS Quarterly, vol. 28, no. 1, pp. 75–105, 2004. doi: 10.2307/25148625
- [29] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ code vulnerability dataset with code changes and CVE summaries," in Proc. MSR 2020, pp. 508–512.
- [30] T. Boland and P. E. Black, "Juliet 1.3 test suite: Changes from 1.2," NIST, 2012.
- [31] NIST, "National Vulnerability Database (NVD)," 2024.