Dependable Code Repair with LLMs: AI-Driven Vulnerability Detection and Automated Patching

Sungmin Han
School of Cybersecurity
Korea University
Seoul, Republic of Korea
sungmin_15@korea.ac.kr

Hyoungshick Kim
Computer Science and Engineering
Sungkyunkwan University
Suwon, Republic of Korea
hyoung@skku.edu

Hojoon Lee
Computer Science and Engineering
Sungkyunkwan University
Suwon, Republic of Korea
hojoon.lee@skku.edu

Hyungon Moon
Computer Science and Engineering
UNIST
Ulsan, Republic of Korea
hyungon@unist.ac.kr

Yuseok Jeon
Computer Science and Engineering
Korea University
Seoul, Republic of Korea
ys_jeon@korea.ac.kr

Ho Bae
Department of Cybersecurity
Ewha Womans University
Seoul, Republic of Korea
hobae@ewha.ac.kr

Donghyun Yeo
Research Planning
AI SPERA Inc.
Seoul, Republic of Korea
dhyeo@aispera.com

Gail-Joon Ahn
Center for Cybersecurity & Trusted Foundations
Arizona State University
Tempe, AZ, USA
gahn@asu.edu

Sangkyun Lee*
School of Cybersecurity
Korea University
Seoul, Republic of Korea
sangkyun@korea.ac.kr

***Abstract*—The rapid proliferation of software vulnerabilities has created an urgent need for intelligent, automated methods to detect and mitigate security flaws at scale. Traditional vulnerability analysis depends heavily on manual inspection and domain-specific expertise, which are increasingly inadequate in the era of generative AI–driven code development. This research proposes an AI-based automated vulnerability detection and secure code generation framework that leverages multi-modal datasets, including source code and binaries, to achieve end-to-end automation across the vulnerability lifecycle: detection, patch generation, and validation. The system integrates explainable AI (XAI)–based vulnerability cause analysis, generative patch synthesis, system-level defensive code generation, Rust-based memory safety transformation, and differential privacy mechanisms for model confidentiality. Developed through a Korea-U.S. joint research initiative, this project aims to establish an internationally deployable platform for trustworthy and privacy-preserving AI-driven software security. The proposed research contributes both foundational methods and operational tools toward self-healing, explainable, and secure-by-design software ecosystems.**

***Index Terms*—AI security, vulnerability detection, secure code generation, large language models, explainable AI**

I. Introduction

The software ecosystem has witnessed a surge in vulnerabilities, publicly reported CVEs numbered well over $29,000$ in 2023 [1], posing serious risks to global digital infrastructures. Attack campaigns from state-sponsored and other advanced groups (for example, the North Korean Lazarus Group exploiting CVE-2023-42793, and the financially-motivated TA505) continue to exploit such vulnerabilities with significant sophistication [2], [3]. As conventional defense approaches rely on manual code review, symbolic execution, and heuristic-based static analysis, they are insufficient to meet the speed and complexity of modern cyber threats – especially in environments where AI-generated code itself may introduce novel attack surfaces.

The emergence of large language models (LLMs) has revolutionized software development by enabling AI-assisted code generation. However, these systems often produce insecure code snippets, such as weak cryptographic implementations or unsanitary input handling, thus amplifying the risk of the software rather than mitigating it [4]. As highlighted in recent research and industry analyzes, even advanced AI-assistants (like GitHub Copilot and Amazon CodeWhisperer) may overlook secure-coding guidelines.

A. Research Gap and Challenges

Existing vulnerability-detection techniques predominantly operate on single-modality data (e.g., source code only) and lack interpretability. Generative-AI-based patching research remains at an early stage, with limited ability to verify or certify the security of generated code. Furthermore, the deployment of such models introduces privacy concerns: training data may leak sensitive code segments or intellectual property via model-inversion and data-extraction attacks on LLMs [5].

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2024-00439819, AI-Based Automated Vulnerability Detection and Safe Code Generation).

*Corresponding author: sangkyun@korea.ac.kr

Addressing these challenges requires a holistic integration of multi-modal reasoning, explainable diagnostics, and confidential AI-training methods into a unified automation framework.

B. Research Objectives and Scope

This project introduces a multi-stage research framework titled "AI-Based Automated Vulnerability Detection and Secure Code Generation." The research encompasses four key objectives:

AI-based Vulnerability Detection and Analysis. Development of XAI-driven models capable of identifying root causes of vulnerabilities across heterogeneous code bases, combining program-graph analysis with human-centric reasoning data.

Generative AI for Secure Code Synthesis. Creation of specialized fine-tuned generative models for vulnerability patching, defensive code generation, and automated translation to memory-safe Rust implementations.

Privacy-preserving Learning for Secure Deployment. Integration of differential privacy and adversarial-robustness mechanisms to ensure confidentiality of training datasets and prevent leakage during inference.

Automated Data Collection and Orchestration. Construction of a reinforcement-learning-based orchestration engine that automates the end-to-end data pipeline – from vulnerability discovery to patch validation and model retraining.

The collaborative implementation between Korea University and the Arizona State University's CTF Center (led by Prof. Gail-Joon Ahn) aims to produce a fully automated platform supporting DARPA-grade cyber-reasoning capabilities while aligning with national AI safety and defense objectives.

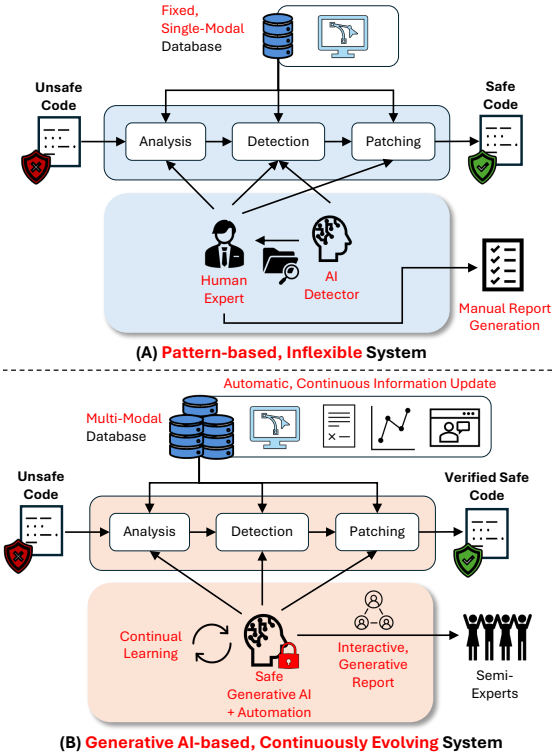

Fig. 1. Comparison between existing semi-automated, single-modal workflows (Panel A) and the envisioned fully automated, multi-modal AI-driven system (Panel B), which unifies vulnerability detection, repair, and verification into a continuous, self-improving pipeline.

II. Related Work

A. Source Code Vulnerability Detection

The goal of source code vulnerability detection is to identify code segments containing security weaknesses exploitable by attackers. This task is typically divided into *static* and *dynamic analysis* [6]. Dynamic analysis examines program behavior during execution, whereas static analysis inspects source code without running it. Because static analysis can detect vulnerabilities early in the development lifecycle and requires fewer computational resources, it is widely preferred for large-scale vulnerability screening.

Traditional static analyzers rely on rule-based techniques such as signature matching and handcrafted policies [7]. Although interpretable and precise when a pattern matches, these approaches suffer from several limitations [8]: (i) high false-positive rates in complex real-world projects, (ii) dependence on human experts, introducing subjectivity and inconsistency in rule design, and (iii) brittleness to coding style and API evolution, which makes rule maintenance costly.

To overcome these drawbacks, recent studies have proposed AI-based vulnerability detection approaches leveraging deep learning models to complement or replace traditional rule-based detectors [9]–[12]. Recent LLM-based approaches further showcase the potential of transformer architectures in detecting security flaws beyond traditional deep learning methods [13]. These models learn multi-level representations of source code—capturing syntactic structures, semantic dependencies, and contextual embeddings—to identify complex vulnerability patterns. By leveraging such learned representations beyond handcrafted rules, they generalize more effectively and achieve superior performance on benchmark datasets [11].

Despite these promising advances, most methods still focus solely on source code or its structural representations (e.g., data-flow and dependency graphs) [11], overlooking execution- and compilation-level artifacts that expose vulnerabilities beyond the source layer. In practice, many flaws manifest only after compilation or optimization, making binary-level vulnerability detection an essential complement for identifying issues such as memory corruption, unsafe linking, or compiler-induced inconsistencies. Beyond these technical constraints, existing approaches also neglect human-centric contextual data that reflect developers' intent and software evolution—such as commit histories and issue discussions. Without this cross-layer and human-informed context, models struggle to capture vulnerabilities arising from configuration errors, dependency mismanagement, or runtime-specific behaviors. Furthermore, deep learning–based detectors often lack interpretability and provide limited insight into why a code fragment is deemed vulnerable, which hinders their practical adoption by security analysts.

B. Automated Vulnerability Repair

Automated Vulnerability Repair (AVR) is the task of automatically generating security-critical patches that remove or mitigate exploitable flaws while preserving the intended functionality and minimizing regressions. AVR workflows are typically composed of three main stages: vulnerability localization, patch generation, and patch validation [14].

Patch-generation techniques span search-based and synthesis/template approaches, program transformation strategies, and learning-based methods that leverage pre-trained code models to propose candidate repairs. Recent trends increasingly integrate large language models (LLMs) that leverage their reasoning capability to generate patches from natural-language prompts and iteratively refine them based on validation feedback to improve patch correctness and security [14], [15]. Empirical studies further show that reasoning prompts and feedback-driven iteration can substantially improve patch quality for security-relevant fixes [15].

Despite these advances, current AVR systems remain narrow in scope and limited in automation. Most operate within a code-only context, lacking the capability to autonomously gather and correlate human- and system-level evidence—such as binary artifacts, developer commits, issue discussions, runtime traces, and test cases—that connect development intent with runtime behavior. Without such contextual reasoning, AVR pipelines fail to generalize across projects or infer the causal pathways that link coding decisions to manifested vulnerabilities. Furthermore, despite notable progress in deep learning–based vulnerability detectors, system-level integration of detection, repair, and verification modules remains largely unexplored, leaving the stages loosely coupled and heavily dependent on manual intervention.

Challenges and Key Design Considerations. Current approaches to AI-based source code vulnerability detection and automated vulnerability repair still fall short of full automation, relying on manual oversight, limited transparency, and an excessive dependence on source code while underutilizing multi-modal information such as binary artifacts, developer commits, and issue reports that provide complementary semantic and contextual signals beyond the code itself. These limitations hinder scalability and responsiveness in large-scale, real-world environments that are rapidly evolving in the era of generative AI–driven software development. To address these challenges, it is essential to develop fully automated, multi-modal, and explainable frameworks that can autonomously execute the end-to-end pipeline—from data collection and vulnerability detection to patch generation and verification—while continuously adapting to emerging vulnerabilities with minimal human intervention.

Fig. 1 shows the current landscape in panel (A), where vulnerability detection and repair remain fragmented, semi-automated, and limited to single-modal (source-code-only) analysis. In contrast, panel (B) envisions our target state as a unified, fully automated, multi-modal AI-driven system.

III. Framework

We propose a novel LLM-based framework for automated vulnerability detection and safety-aware secure code generation, which leverages not only source code but also diverse security-related artifacts (e.g., binary artifacts, issue reports, commit logs, and vulnerability descriptions) while operating entirely without human intervention.

The framework consists of three tightly integrated phases: (1) *Vulnerability detection and analysis*, powered by explainable AI (XAI)–based root-cause attribution, which not only identifies suspicious code regions but also provides token-, line-, and region-level (i.e., code-block–level) explanations of why the code is considered vulnerable; (2) *Safe and secure code generation*, which encompasses automated patching, system-level defense synthesis, Rust-based code translation, and verification; and (3) *Orchestration and automation*, which enables end-to-end coordination, monitoring, data collection, and continual learning–based updates to keep all constituent models and underlying data continuously up to date.

Fig. 2 illustrates the overall architecture of the proposed framework.

A. Vulnerability Detection and Analysis

1) Code Representation for Vulnerability Analysis: Large language models (LLMs) based on the transformer architecture [16] operate on token-level numerical inputs. Thus, source code must be tokenized and converted into embeddings (vector representations) that capture not only lexical features but also deeper syntactic, semantic, and contextual dependencies.

To meet this requirement, we design a hybrid embedding architecture that fuses pretrained code models with graph neural networks. Specifically, we employ code embedding models such as CodeBERT [17], a transformer-based language model pretrained on large-scale bimodal corpora of source code and natural language (e.g., GitHub repositories and documentation), to tokenize and vectorize the source code, thereby providing strong lexical and syntactic priors. In parallel, we incorporate graph-enhanced code representation models such as GraphCodeBERT [18], which extend transformer-based encoders with mechanisms to capture structural dependencies, enriching the model's understanding of program structure. The structural information itself (e.g., abstract syntax tree (AST), control-flow graph (CFG), and enriched program dependency graph (ePDG) [11]) is explicitly constructed by our analysis pipeline and provided as relational input to the model encoder. The resulting code-structure-aware embeddings jointly capture fine-grained token semantics and high-level program structure.

2) Multi-modal Evidence Integration: While the code embeddings produced by pretrained code and graph-augmented embedding models (e.g., CodeBERT [17], GraphCodeBERT [18]) capture the internal semantics and structural dependencies of source code, relying solely on these representations is often insufficient for comprehensive vulnerability detection. Certain classes of vulnerabilities are context dependent and only manifest under specific development histories, configuration states, or runtime conditions [19], [20].

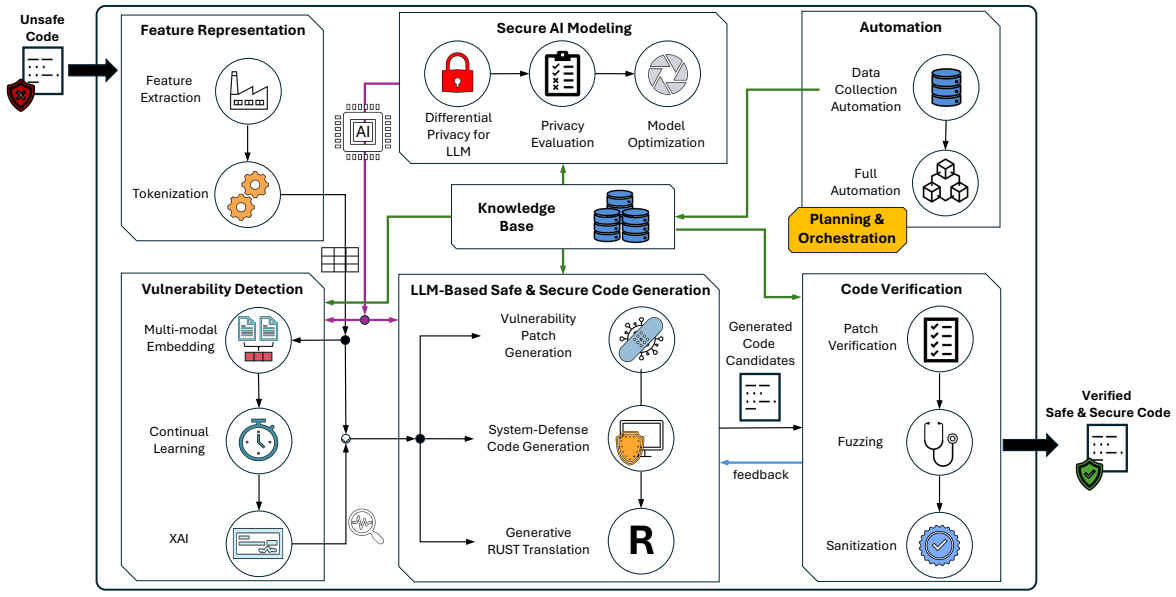

Fig. 2. The schematic of our proposed framework.

To address this limitation, our framework integrates human- and system-centric evidence (e.g., binary artifacts, commit histories, issue reports, and CVE descriptions) to enrich code-level representations with contextual cues. Each code fragment is automatically linked to its relevant external artifacts, from which separate artifact embeddings are derived. The resulting code and artifact embeddings are then fused into unified multi-modal representations that jointly encode code semantics and contextual evidence, enabling more comprehensive and context-aware vulnerability reasoning.

3) Vulnerability Detection: Building upon these multi-modal embeddings, our vulnerability detector identifies potentially vulnerable code segments. The detector is implemented as an LLM-based classifier fine-tuned on labeled vulnerability datasets, which models $p_\theta(y = \text{vulnerable} \mid \mathbf{h}_{\text{multi}}(x))$, where $\mathbf{h}_{\text{multi}}(x)$ denotes the fused multi-modal representation of code x, and θ represents the learnable parameters of the detector. The output p_θ provides the estimated probability that a given code fragment is vulnerable.

4) Analysis of Detection Causes: To make vulnerability detection interpretable, our framework incorporates an explainable AI (XAI) component that analyzes the underlying causes of each detected vulnerability. Rather than returning only a binary decision (vulnerable or safe), the XAI module computes the contribution of individual tokens, lines, and regions to the model's prediction—quantifying which parts of the input code most strongly influence the vulnerability classification.

Specifically, this module employs attribution-based techniques [21], [22] to quantify the contribution of each modal feature (e.g., code tokens, binary patterns, and textual artifacts) to the model's vulnerability prediction. The resulting attribution scores are visualized as heatmaps that highlight the most influential regions across modalities, providing detailed explanations of why the model considered the code vulnerable. To make these insights actionable, the framework further utilizes the attribution maps to present explanations and suggest repair directions tailored to each developer's specific code context.

B. Safe and Secure Code Generation

1) Patch Generation: Once vulnerable code segments are identified, the patch generation component automatically synthesizes corresponding repairs. Our approach advances the LLM-driven AVR paradigm [14] by conditioning generation on multiple complementary signals drawn from vulnerability detection, XAI-derived explanations, and multi-modal contextual information. These complementary cues guide the model to focus its reasoning and editing behavior on semantically relevant regions, minimizing hallucinated or functionally irrelevant changes.

Patch synthesis proceeds through a multi-stage process that first leverages the LLM's reasoning ability to comprehend the underlying vulnerability and then generates candidate repairs accordingly. In particular, multi-modal contextual cues and XAI-derived attributions help the model infer root causes, exploit conditions, and intended repair objectives. Based on this understanding, the system performs in-context prompting, providing the LLM with compact exemplars of vulnerable–fixed code pairs enriched with XAI-derived explanations that localize the intended edit scope, through which the model produces candidate patches. This process allows the model to synthesize patches that are semantically consistent and functionally aligned with the identified vulnerability context.

2) System-level Defense Code Generation: Beyond local patch synthesis, our framework also generates and, when necessary, automatically injects system-level defense code to secure the broader software environment against recurring or systemic vulnerabilities. Specifically, this module targets two primary security tasks: (i) preserving memory safety through

spatial and temporal protection mechanisms such as bounds checking; and (ii) enforcing privilege separation via kernel-level isolation and process credential management.

To achieve these goals, the framework prompts the LLM with security task-specific instructions tailored for memory-safety preservation and privilege-separation enforcement. Guided by these prompts and contextual cues from detected vulnerabilities and validated patches, the LLM generates system-level countermeasures that generalize across similar attack surfaces and strengthen overall code security.

3) Rust-based Code Translation: Unlike languages such as C and C++, Rust is a memory-safe systems language that enforces strict ownership, lifetime, and type-safety rules at compile time, effectively eliminating common sources of vulnerabilities such as buffer overflows, dangling pointers, and data races [23]. Accordingly, the Rust translation component is designed to convert existing code into Rust to enhance security while preserving functionality and structural dependencies.

To accomplish this, we fine-tune a pretrained LLM using prompt templates specialized for code translation tasks. These prompt templates are carefully crafted to emphasize safety-critical semantics—such as pointer ownership, memory allocation, and concurrency synchronization—enabling the LLM to generate idiomatic Rust code that faithfully replicates the original program's behavior while eliminating unsafe patterns.

Because large-scale codebases often exceed the input context length of current LLMs, we employ a LangChain-based transformation pipeline [24] that preserves global program context across segmented translation units. The system performs dependency-aware clustering of source files and functions using static program analyses to construct semantically coherent transformation units. Each unit is incrementally converted while retaining metadata describing inter-unit dependencies (e.g., variable scopes, function calls, and linker relationships). This metadata is embedded and passed through a LangChain controller, which maintains continuity and consistency of variable bindings, types, and ownership semantics throughout the transformation process.

4) Code Verification: The code verification component serves as a unified evaluation and feedback engine for all AI-generated code, encompassing patch generation, system-level defense implementation, and Rust translation. It ensures both functional correctness and security soundness by validating code through static analysis, directed fuzzing, cross-language interface testing, and policy compliance checks, while providing structured feedback for the continual improvement of generation models.

Verification Targets and Strategy. Instead of exhaustively testing entire codebases, the verifier focuses on regions that are most likely to contain or be affected by vulnerabilities introduced during generation. Static and graph-based analyses first locate these high-impact regions—such as modified functions, security-critical code blocks, and cross-language interfaces. Guided by these analyses and XAI-derived explanations, directed fuzzing generates targeted inputs that stress vulnerability-prone paths with high coverage. During execution, runtime feedback such as sanitizer reports and coverage metrics is collected and used to iteratively refine the test inputs and improve the robustness of subsequent verification rounds.

Policy Compliance and Sanitization. Beyond functional correctness, the verifier detects policy violations (e.g., insecure API usage or missing access checks) using LLM-assisted reasoning and chain-of-thought justification. For confirmed violations, it automatically synthesizes and inserts sanitization code—such as input validation or credential checks—while balancing security gain against performance overhead.

Concretely, when a policy violation is detected, the system prompts the LLM with the violating code region and a brief natural-language description (e.g., "unvalidated pointer dereference"). The model then synthesizes a minimal sanitizer snippet that enforces the required check, which is inserted in place and immediately re-verified.

Continuous Feedback and Refinement. Verification outcomes, including fuzzing coverage and sanitizer results, are organized into structured feedback artifacts. Negative results (failed patches, ineffective defenses, or unsafe translations) are propagated to corresponding modules to guide prompt tuning, model updates, and rule refinement. Each component, including patch generation, system-level defense generation, and Rust-based code translation, refines its behavior based on these feedback signals to correct prior errors. This closed feedback loop ensures that every generation cycle improves code synthesis precision and produces verified, deployable code that meets security and functionality requirements.

C. Orchestration and Automation

This phase serves as the control layer that integrates and coordinates the preceding modules into a unified, continuously operating pipeline. At its core lies the *orchestrator*, which coordinates modules, manages data, and adaptively allocates tasks to ensure that detection results, generated patches, and verification outcomes flow efficiently through the system.

Reinforcement-Learning–Driven Orchestration. Rather than acting as a static scheduler, the orchestrator is modeled as a policy agent that dynamically allocates analysis and verification effort across modules. It observes key pipeline signals—such as detector confidence, sanitizer alerts, fuzzing coverage, and patch success rates—and learns a policy that prioritizes which vulnerabilities to escalate for patching, fuzz testing, or Rust translation. This process is implemented as a reinforcement-learning loop whose reward reflects downstream verification quality and patch acceptance.

Continual Learning and Incremental Updating. The orchestrator maintains a continual learning loop in which new evidence—newly discovered vulnerabilities, failed patches, sanitizer violations, and successful fixes—is appended to a curated high-quality dataset. Rather than retraining all modules from scratch, fine-tuning is performed incrementally on the most drifted submodules.

Privacy-Preserving and Secure Deployment. To ensure privacy-preserving and secure deployment, the orchestrator

applies safe AI modeling principles that enforce privacy-aware data handling and training. All collected data are processed through anonymization and access-controlled layers. In addition, model training and continual-learning updates are performed under differential privacy guarantees [25] to prevent the leakage of sensitive information, ensuring that the framework remains secure and compliant even in large-scale or collaborative environments.

By unifying all components under a fully integrated and privacy-aware orchestration layer, the system enables continuous, self-improving, and securely deployable operation in real-world settings.

IV. Conclusion

This paper presented an integrated research effort toward AI-driven vulnerability detection and secure code generation. By uniting explainable AI, generative patching, and privacy-preserving learning, the framework lays the groundwork for self-improving software security. The Korea–U.S. team is developing four deliverables: (1) an AI vulnerability analysis engine, (2) a generative patch and defense code generator, (3) a security verification suite, and (4) a fully automated orchestration platform.

Our collaborative research (e.g., [15], [22], [26]–[31]) has introduced methodologies surpassing prior benchmarks in vulnerability detection and LLM-based patch generation, ultimately advancing secure, explainable, and autonomous AI that shifts cybersecurity from reactive protection to proactive, self-healing defense.

References

[1] CVE Details, "Browse vulnerabilities by date – 2023 summary," https://www.cvedetails.com/browse-by-date.php, 2024.

[2] Microsoft Threat Intelligence, "Microsoft warns of north korean attacks exploiting jetbrains teamcity flaw," *The Hacker News*, 2023. [Online]. Available: https://thehackernews.com/2023/10/microsoft-warns-of-north-korean-attacks.html

[3] Canadian Centre for Cyber Security, "Profile: Ta505 / cl0p ransomware – threat actor overview," https://www.cyber.gc.ca/en/guidance/profile-ta505-cl0p-ransomware, 2024.

[4] J. S. Research, "Analyzing common vulnerabilities introduced by code-generative ai," https://jfrog.com/blog/analyzing-common-vulnerabilities-introduced-by-code-generative-ai/, 2024.

[5] N. Carlini, F. Tramer, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. B. Brown, D. Song, U. Erlingsson, A. Oprea, C. Raffel, R. Shokri, A. Thakurta, and T. Steinke, "Extracting training data from large language models," in *Proceedings of the 32nd USENIX Security Symposium*, 2023.

[6] J. Wang, M. Huang, Y. Nie, and J. Li, "Static analysis of source code vulnerability using machine learning techniques: A survey," in *2021 4th International Conference on Artificial Intelligence and Big Data*, 2021.

[7] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav, "A survey of static analysis methods for identifying security vulnerabilities in software systems," *IBM Systems Journal*, vol. 46, no. 2, p. 265–288, 2007.

[8] R. Croft, D. Newlands, Z. Chen, and M. A. Babar, "An empirical study of rule-based and learning-based approaches for static application security testing," in *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2021.

[9] S. Han, H. Nam, J. Kang, K. Kim, S. Cho, and S. Lee, "CODE-SMASH: Source-code vulnerability detection using siamese and multi-level neural architecture," *IEEE Access*, vol. 12, 2024.

[10] S. Han, M. Kim, J. Kang, K. Kim, S. Lee, and S. Lee, "Similarity-based source code vulnerability detection leveraging transformer architecture: Harnessing cross-attention for hierarchical analysis," *IEEE Access*, vol. 12, 2024.

[11] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "VulChecker: Graph-based vulnerability localization in source code," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, 2023.

[12] S. Kim, J. Choi, M. E. Ahmed, S. Nepal, and H. Kim, "VulDeBERT: A vulnerability detection system using bert," in *Proceedings of the 33rd International Symposium on Software Reliability Engineering (Industry Track)*, 2022.

[13] L. Huynh, Y. Zhang, D. Jayasundera, W. Jeon, H. Kim, T. Bi, and J. B. Hong, "Detecting code vulnerabilities using llms," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2025, pp. 401–414.

[14] Y. Li, F. H. Shezan, B. Wei, G. Wang, and Y. Tian, "SoK: Towards effective automated vulnerability repair," in *Proceedings of the 34th USENIX Conference on Security Symposium*, 2025.

[15] Y. Kim, S. Shin, H. Kim, and J. Yoon, "Logs in, patches out: Automated vulnerability repair via tree-of-thought llm analysis," in *Proceedings of the 34th USENIX Conference on Security Symposium*, 2025.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017.

[17] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics*, 2020.

[18] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[19] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Computing Surveys*, vol. 54, no. 2, 2021.

[20] S. Ul Haq, Y. Singh, A. Sharma, R. Gupta, and D. Gupta, "A survey on iot & embedded device firmware security: Architecture, extraction techniques, and vulnerability analysis frameworks," *Discover Internet of Things*, vol. 3, no. 1, 2023.

[21] S. Lee and S. Han, "Libra-CAM: An activation-based attribution based on the linear approximation of deep neural nets and threshold calibration," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, 2022.

[22] S. Han, J. Lee, and S. Lee, "Contrast-CAT: Contrasting activations for enhanced interpretability in transformer-based text classifiers," in *Proceedings of the Forty-first Conference on Uncertainty in Artificial Intelligence*, 2025.

[23] P. Larsen, "Migrating c to rust for memory safety," *IEEE Security and Privacy*, vol. 22, no. 4, 2024.

[24] H. Chase, "Langchain," https://github.com/hwchase17/langchain, 2022, accessed: 2025-10-19.

[25] C. Dwork, "Differential privacy," in *International colloquium on automata, languages, and programming*, 2006.

[26] W. Han, Y. Kwak, M. Yu, K. Kim, Y. Lee, H. Moon, and Y. Paek, "Rethinking the capability of fine-tuned language models for automated vulnerability repair," in *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering*, Apr 2026, in press.

[27] K. D. Duy, J. Kim, H. Lim, and H. Lee, "INCOGNITOS: A practical unikernel design for full-system obfuscation in confidential virtual machines," in *2025 IEEE Symposium on Security and Privacy*, 2025.

[28] H. Lim, J. Kim, and H. Lee, "uMMU: Securing data confidentiality with unobservable memory subsystem," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[29] J. Kim, S. Cheon, S. Han, and S. Lee, "TokenROC: A roc-based evaluation method for xai techniques in software vulnerability token detection," *The Transactions of the Korea Information Processing Society*, vol. 14, no. 9, 2025.

[30] J. Hong, W. Jang, M. Kim, L. Yu, Y. Kwon, and Y. Jeon, "CMASan: Custom memory allocator-aware address sanitizer," in *2025 IEEE Symposium on Security and Privacy*, 2025.

[31] S. Jeong, M. Choi, H. Cho, S. Choi, H. Kim, and Y. Jeon, "Intent-aware fuzzing for android hardened application," in *Proceedings of the 2025 on ACM SIGSAC Conference on Computer and Communications Security*, 2025.