Automated Cross-Repository Vulnerability Variant Retrieval Using Patch-Weighted Contrastive Learning

Joseph Chen
Department of Management Information Systems
University of Arizona
Tucson, Arizona
josephchen@arizona.edu

Benjamin Ampel
Department of Computer Information Systems
Georgia State University
Atlanta, Georgia
bampel@gsu.edu

Steven Ullman
Department of Information Systems and Cybersecurity
University of Texas at San Antonio
San Antonio, Texas
steven.ullman@utsa.edu

Raúl Y. Reyes
Department of Management Information Systems
University of Arizona
Tucson, Arizona
raulreyes@arizona.edu

Hsinchun Chen
Department of Management Information Systems
University of Arizona
Tucson, Arizona
hsinchun@arizona.edu

***Abstract* — Vulnerabilities propagate across open-source software (OSS) ecosystems through code reuse. However, existing vulnerability management tools often analyze repositories in isolation. Variant analysis can identify semantically similar vulnerable code at scale, but current workflows remain largely manual and expert-driven. To address this, we propose Automated Variant Analysis (AVA), a retrieval model that leverages known vulnerable code snippets and their corresponding patches to identify unremediated variants across OSS repositories. AVA generates vulnerability-preserving variants, represents code with Tree-sitter-based Token Projected Graphs (TPG), and trains embeddings using a patch-weighted supervised contrastive loss. Our results show that supervised contrastive training improves discrimination between vulnerable and patched code, while graph augmentation can reduce separation by pulling structurally similar vulnerable and patched samples closer together. These findings demonstrate that real-world patches can provide scalable supervision for detecting reused or modified vulnerabilities across OSS ecosystems, supporting remediation beyond the repository where a flaw was originally discovered.**

Keywords — Open-source software, variant analysis, code clone detection, contrastive learning, code embeddings.

I. Introduction

Open-source software (OSS) is implemented in 70–90% of modern applications, generating an estimated $8.8 trillion in value by enabling collaborative reuse [1], [2]. However, the widespread adoption of OSS introduces supply chain risks. 87% of commercial codebases contain vulnerabilities due to copied, reimplemented, or modified OSS code [3]. Security teams often rely on vulnerability management tools to identify and mitigate these risks [4].

The vulnerability management lifecycle involves scanning repositories, validating findings, and executing risk-based prioritization (based on a flaw's severity and exploitability) [4], [5]. However, traditional scanners are often limited to a single OSS repository [5]. OSS code can propagate across repositories through copying, adaptation, and reuse, but security patches do not necessarily also propagate [5]-[7]. Consequently, a vulnerability may be fixed in one repository while unpatched instances persist elsewhere. Variant analysis addresses this gap by using a known vulnerability as a seed to retrieve semantically similar flaws, a critical capability given that over 40% of zero-day vulnerabilities are variants of previously reported issues [7]-[9]. However, current variant analysis often remains a manual process requiring expert query construction [10].

To address this gap, we propose Automated Variant Analysis (AVA), a retrieval framework that uses known vulnerabilities and their patches to automatically identify unremediated variants across OSS repositories, removing the need to manually specify retrieval logic per disclosure. This study examines whether real-world patches can provide scalable supervision for variant analysis and whether structural augmentation improves discrimination of vulnerable code from its patch.

II. Literature Review

A. OSS Vulnerabilities

Recent OSS vulnerability research primarily focuses on single-repository detection, framed as a binary classification problem, using standard benchmarks and language-specific datasets [11]-[15]. While research demonstrates that vulnerabilities can extend beyond individual repositories [16], [17], it does not address how vulnerabilities can propagate as variants across OSS. To capture vulnerability semantics, prevailing literature often relies on structural representations, primarily Code Property Graphs (CPGs) [11], [15]. However, CPGs require language-specific parsers and full compilation pipelines [12]. Extant CPG methods do not scale effectively across OSS ecosystems and often fail when analyzing incomplete or uncompilable code snippets. Parallel to these structural approaches, the literature has also heavily explored pre-trained models. These models provide a strong foundation for vulnerability tasks because they have already learned general code representations [18]. Therefore, we review pre-trained

code-embedding models to determine how we can use them for variant analysis.

B. Pre-trained Code Embeddings

Pre-trained code embeddings capture syntactic and semantic properties without task-specific supervision [18], [19]. These often include sequence-based models that process flat tokens and structure-aware models that incorporate structural data, such as Abstract Syntax Trees (ASTs) or Data Flow Graphs (DFGs) [18]-[22]. However, prevailing code embedding models are not pre-trained to capture vulnerable code, meaning their representations may not capture the security-relevant patterns for variant analysis. Furthermore, these models often rely on flat sequence inputs while industry tools such as CodeQL and Semgrep demonstrate that structural context is essential for detecting vulnerability semantics [9]. However, these tools rely on rigid pattern matching that is brittle against complex variants [9], motivating our review of clone detection, which targets semantic rather than structural similarity.

C. Vulnerable Code Clone Detection

Code clone detection is widely repurposed in the security context to identify modified vulnerable code [23]-[26]. Rather than looking for exact textual matches (Type-1) or simple variable renaming (Type-2), vulnerable code clone detection specifically targets Type-3 (near-miss structural) and Type-4 (semantic) clones, as these retain vulnerable logic despite syntactic differences [25]. Identifying these variants requires capturing vulnerability semantics to distinguish between a vulnerable snippet and its patched counterpart. This distinction is difficult because vulnerabilities often hide within localized data flows or single lines of a large function. Because vulnerable and patched code exhibit such high similarity, current clone detection methods still rely heavily on traditional static analysis to capture these subtle differences [26]. Furthermore, while general clone detection benefits from large datasets, data curated specifically for vulnerable clones remains scarce due to the extensive manual verification required. To address this limitation, we turn to contrastive self-supervised learning, which we will review next.

D. Contrastive Learning

Contrastive learning mitigates labeled data scarcity by applying semantically preserving augmentations to generate positive training pairs [27]-[29]. These synthetic clones enable the model to derive supervision signals during pre-training. Traditionally implemented via Triplet Loss to evaluate anchors against positive and negative samples, the objective function maps code into a latent space by pulling semantically similar snippets together and pushing different ones apart [30]. Although extant approaches demonstrate that contrastive learning can capture vulnerable code semantics, these methods rely on injecting artificial vulnerabilities inspired by Common Weakness Enumeration (CWE) patterns [29]. While artificial vulnerabilities provide the scale for pre-training, they can fail to capture the complexity of real-world vulnerabilities required for variant analysis.

III. Research Gaps and Questions

We identified three research gaps from our literature review. First, extant code transformations scale poorly across massive OSS repositories. Second, current sequence-based embedding models lack the structural context crucial for detecting vulnerability semantics. Third, existing contrastive learning methods rely on synthetic bug injections that fail to capture the real-world complexity needed for variant analysis. Based on the research gaps, we pose the following questions:

1. How can we develop a scalable framework for variant analysis across OSS?
2. How can we leverage structural data to improve retrieval quality and patch discrimination?
3. How can we create augmentations of existing real-world vulnerabilities at scale to support variant analysis?

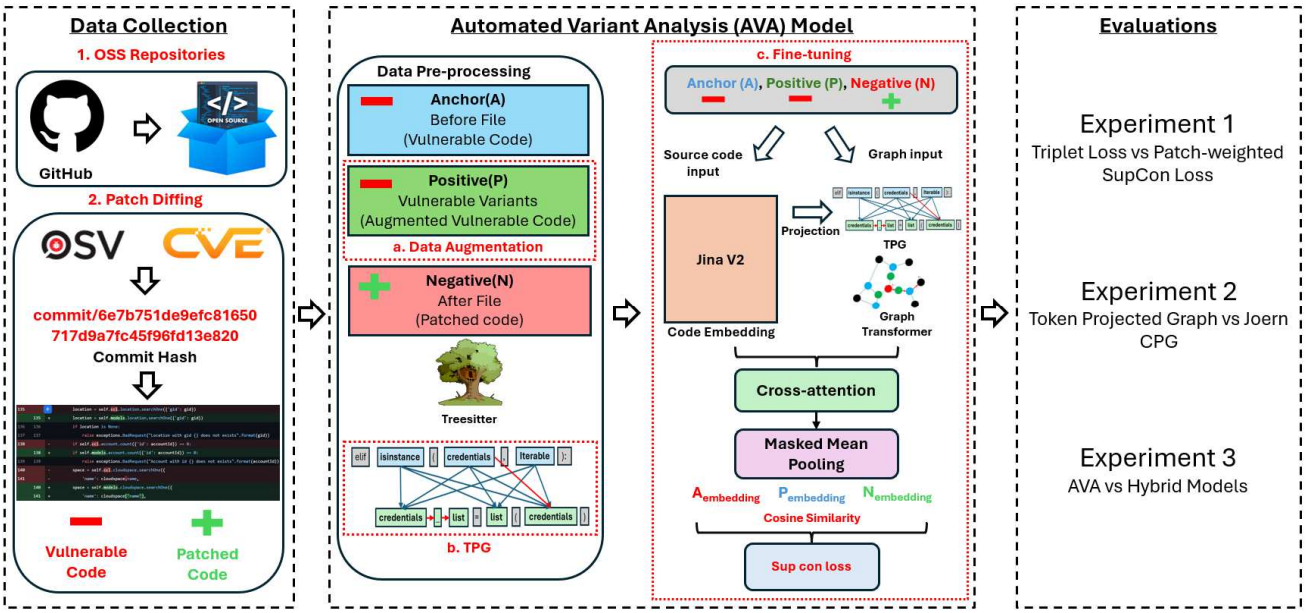

Fig. 1. Proposed Research Design

IV. Research Design

To address the research questions, we propose a three-phase research design (Fig. 1). Phase 1 encompasses data collection, extracting vulnerable–patched code pairs from real-world patch commits. Phase 2 demonstrates the AVA framework, which generates data to learn representations of vulnerable code and their remediated counterparts for variant retrieval. Phase 3 evaluates AVA by comparing alternative loss functions, structural representations, and model configurations. The following sections describe each phase.

A. Phase 1. Data Collection

To obtain real-world vulnerable and patched code, we collected vulnerability records from the Open Source Vulnerabilities (OSV) and Common Vulnerabilities and Exposures (CVE) databases. We then retrieved patch commits for each CVE using the GitHub API and the associated commit URL. Each patch commit contains a patch diff (line-level changes between vulnerable and patched code), where deleted lines are labeled as vulnerable code and added lines as patched code. The patch diffs directly reflect the state of the codebase before and after the patch was applied, offering ground-truth labels. We retained vulnerabilities from the five most prevalent language groups: PHP, Java, JavaScript, Python, and C-family languages, yielding 60,758 patch diffs spanning 15,692 CVEs.

B. Phase 2. Automated Variant Analysis (AVA) Model

AVA is a retrieval model designed to identify unremediated vulnerability variants across OSS repositories and operates in three phases:

1) ***Data Augmentation.*** Vulnerable code clone detection suffers from a scarcity of verified variant data [25]. To generate positive training pairs at scale, we implement an LLM-driven augmentation pipeline that creates syntactically diverse but vulnerability-preserving variants of known vulnerabilities (shown in Fig. 2).

To generate variants of each vulnerable snippet, we used Devstral-Small-2, a 24-billion-parameter instruction-tuned model with a 256,000-token context window selected for its software engineering benchmark performance [31]. Because a syntactically valid transformation may inadvertently eliminate the original vulnerability, we integrate Semgrep into the generation process to provide a scalable approximation for preserving vulnerability.

The vulnerable snippet and its patched counterpart are provided jointly to the language model to generate a vulnerability-specific Semgrep rule. The rule is retained only if it flags the vulnerable version but not the patch. The vulnerable snippet is then transformed using five augmentation operators adapted from extant contrastive code-learning research [29]: (1) identifier permutation, (2) control-flow restructuring, (3) data-flow reordering, (4) dead-code injection, and (5) logic mirroring. Each generated variant is reinserted into its source file, checked with a language-specific linter to remove syntactically invalid outputs, and evaluated using the validated Semgrep rule. Variants not flagged by the rule are removed. This procedure yields 12,461 variants for AVA training.

2) ***Token Projected Graph (TPG)****.* Vulnerability detection frameworks often rely on CPGs to capture structural context [11], [23]. However, CPG construction requires complete, compilable code, causing traditional tools to fail on the fragmented code snippets common in real-world patch diffs [13]. To overcome this, we propose the TPG, a lightweight alternative inspired by recent literature [32]. We use Tree-sitter (a fault-tolerant parser that generates ASTs from raw, incomplete source code across over 40 languages) [33] as our structural backbone, from which we derive control- and data-flow edges using three lightweight heuristics. The TPG is constructed in four stages, illustrated in Fig. 3. First, we tokenize our collected source code into a flat sequence of subword tokens, mapping each token's character offset to a byte-to-token dictionary. Second, Tree-sitter builds an AST over the same source code in parallel to annotate every leaf node with its start and end byte positions and a type label (e.g., identifier, type_identifier, call, etc.). Third, we implement byte alignment to unify the two representations, mapping AST nodes directly to token indices. Fourth, graph assembly applies three heuristics to construct the final heterogeneous graph (1) every token that is identified as a leaf by Tree-sitter becomes a node, (2) control-flow edges (denoted in blue arrows) are drawn from each identifier token in a condition branch to every identifier token in the corresponding body branch, encoding that the body only runs when the condition holds, and (3) data-flow edges

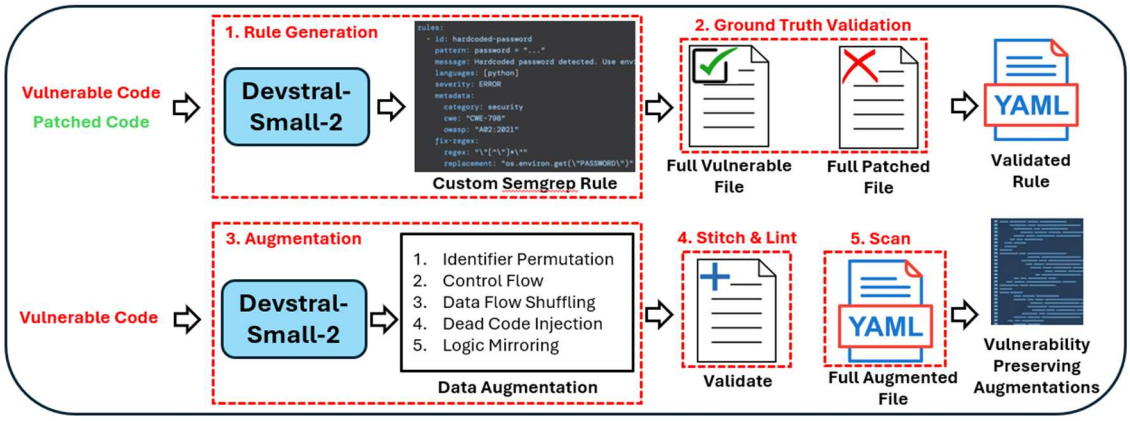

Fig. 2. Data Augmentation Pipeline

(denoted in red arrows) chain together all tokens belonging to the same variable, linking fragmented subword pieces of the same identifier and connecting repeated occurrences of the same variable across the snippet. The resulting TPG is a structural representation that bypasses the computational overhead of traditional CPG generation.

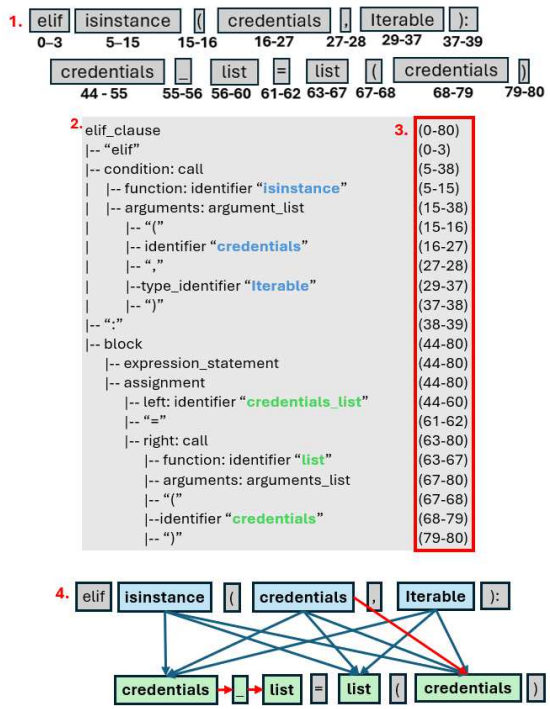

Fig. 3. Token Projected Graph Example

3) ***Fine-tuning***. Prior work demonstrates that combining sequential encoders with structural graph representations produces richer code embeddings than either alone [29]. AVA therefore adopts a hybrid architecture that fuses a pre-trained sequence encoder with TPG-derived structural signals, fine-tuned using patch-weighted contrastive loss. The intuition is that the model has to learn vulnerability semantics by recognizing the same flaw across diverse variants while pushing the patch away. Representations grounded in vulnerability semantics rather than token overlap will naturally cluster variants together and push the patch away. First, the encoder (jina-embeddings-v2-base-code) processes the full source code to produce contextual token embeddings capturing syntax. Second, these embeddings are projected onto the TPG as initial nodal features. A Heterogeneous Graph Transformer (HGT) updates these nodes by propagating signals along data-flow and control-flow edges, enriching representations with dependencies. Third, the updated graph embeddings are mapped back to token positions in a zero-filled tensor of the same length. A cross-attention layer fuses these representations, using Jina token embeddings as queries and the graph tensor as keys and values, producing a hybrid embedding based on linguistic and structural context. Fourth, these are aggregated into a single embedding through masked mean pooling across all non-padding tokens. Fifth, the model is trained using a patch-weighted contrastive loss that assigns a weight multiplier for patched samples in the denominator.

$$\mathcal{L} = -\frac{1}{|P(i)|}\sum_{j\in P(i)} log\frac{\exp(\mathrm{sim}(z_i, z_j)/\tau)}{\sum_{k\neq i} w_k \cdot \exp(\mathrm{sim}(z_i, z_k)/\tau)}$$

where $P(i)$ denotes the set of positive indices for anchor i (i.e., vulnerability-preserving variants of the same CVE), z represents the normalized embedding, τ is the temperature parameter, and w_k is a weight multiplier applied to patched samples in the denominator. By inflating w_k for patched negatives relative to unrelated files (i.e., code from another CVE), the loss explicitly increases the penalty for embedding a vulnerability near its corresponding fix. This teaches the model the boundary between a flaw and its patch. Finally, the embeddings are indexed using Facebook AI Similarity Search (FAISS) to enable nearest-neighbor retrieval. When a vulnerable embedding is queried, the model's retrieval quality and patch discrimination are evaluated.

C. Phase 3. Evaluation

We evaluate AVA through three experiments, summarized in Table 1. First, to assess the contribution of the training objective, we compare the patch-weighted supervised contrastive loss with a triplet loss using sequence-based code-embedding models. Second, we evaluate structural effectiveness by comparing the proposed TPG against Joern CPG , a widely established tool for CPG generation, using graph-only models [11], [23]. Third, because AVA integrates both sequential and structural representations, we benchmark its end-to-end performance against prevailing hybrid code embedding models. We benchmarked AVA against baseline models utilized in the literature for code clone detection and vulnerability detection [29].

To assess retrieval quality, we rely on standard Information Retrieval (IR) metrics prevalent in code clone detection literature [24], [29]: HitRate@1 (top-1 recall), R-Precision (proportion of relevant variants in top-R results), NDCG@5 (rank-sensitive quality), and Mean Average Precision (MAP). However, standard IR metrics only reward retrieval of a vulnerable variants, they do not penalize the retrieval of a patched version, which would introduce false positives in a real-world security environment. Therefore, we introduce three security-specific metrics: Mean Margin (the average distance between a variant and its patch in the embedding space), Patch@5 (the frequency of patched code erroneously appearing in the top-5 results), and Discrimination Accuracy (the model's overall accuracy in separating vulnerable from patched code).

1) ***Experiment 1.*** We compared our patch-weighted Supervised Contrastive Loss (SupCon) against a standard Triplet Loss across four state-of-the-art code embedding models (CodeBERT, CodeT5+ , CODESAGE, and Jina V2). Our results indicate that patch-weighted SupCon consistently outperforms Triplet Loss across all four models. The data reveals a performance gap driven by the optimization strategies: while standard Triplet Loss optimizes only local pairwise relationships (failing to push near-identical patched code far

TABLE I. EXPERIMENTS

	Model	R-Precision	HitRate@1	NDCG@5	MAP	Mean Margin	Patch@5	Disc Acc
Experiment 1								
Triplet Loss	**CodeBERT**	77.94	86.52	90.21	86.48	0.2123	56.59	60.48
	CodeT5+	87.24	91.14	95.35	92.55	0.3364	47.23	74.88
	CODESAGE	87.48	90.76	94.83	92.37	0.4875	35.81	78.51
	Jina V2	**94.05**	**95.22**	**97.42**	**96.15**	0.4532	42.38	93.3
Sup Con	**CodeBERT**	86.74	91.91	93.9	93.35	0.1938	54.82	69.72
	CodeT5+	89.38	93.37	95.46	94.03	0.4723	33.46	79.04
	CODESAGE	90.77	92.76	94.01	93.49	0.5642	19.90	92.14
	Jina V2	92.95	93.99	95.16	94.83	**0.6763**	**19.00**	**94.14**
Experiment 2								
TPG	**GraphSAGE**	79.31	81.58	88.74	87.37	0.0107	83.82	66.72
	GAT	67.79	68.65	78.50	77.26	0.0112	84.13	66.02
	Graph Transformer	77.83	79.88	87.87	86.27	0.0092	85.06	64.71
	GCN	80.08	81.27	88.89	87.65	0.0103	84.60	65.09
	GIN	**81.11**	**82.74**	**89.60**	**88.41**	**0.0127**	83.51	**67.88**
	GGNN	80.17	81.42	89.26	87.78	0.0101	84.37	66.02
Joern CPG	**GraphSAGE**	63.02	64.09	72.93	71.55	0.0027	74.92	38.78
	GAT	58.30	48.37	67.85	65.67	0.0028	75.00	40.02
	Graph Transformer	59.96	50.93	69.27	67.55	0.0025	75.46	40.79
	GCN	59.60	54.10	69.68	67.99	0.0023	75.93	39.63
	GIN	59.26	44.04	68.30	66.46	0.0049	**72.99**	40.40
	GGNN	59.53	44.04	68.48	66.47	0.0049	75.15	38.85
Experiment 3								
Hybrid Models	**UniXcoder**	82.05	65.02	86.69	84.86	0.2537	64.64	62.87
	GraphCodeBERT	83.85	72.57	89.10	87.38	0.2950	54.30	70.11
	AVA (TPG)	**93.96**	**94.74**	96.23	**95.96**	0.4622	40.79	94.04
	AVA(Joern CPG)	93.41	93.19	**97.39**	95.49	0.4378	44.1	90.87
	Jina V2 (baseline)	92.95	93.99	95.16	94.83	**0.6763**	**19.00**	**94.14**

enough away), SupCon's batch-wide objective simultaneously clusters variants of the same vulnerability while repelling patched code. In vulnerability variant analysis, this performance gap is critical; because a vulnerability and its patch share token and syntactic overlap, a model that cannot separate them will surface remediated code as a vulnerable variant, triggering false positives. This trade-off is best illustrated by Jina V2. Under Triplet Loss, Jina V2 achieves high raw retrieval rates but poor patch isolation, finding variants but failing to reliably exclude their fixes. Under SupCon, Jina V2 trades a small amount of retrieval volume for substantially greater separation between the variant and the patch. In practical code clone detection, this outcome means security analysts are presented with a cleaner, actionable list of unpatched code that genuinely requires remediation, preventing alert fatigue and saving valuable hours.

*2) **Experiment 2**.* To assess the effectiveness of TPGs in capturing vulnerability semantics, we compared our proposed TPG against the industry-standard Joern CPG using graph-only models. Our results show that TPGs yield superior retrieval metrics by preserving semantic similarity through token-level granularity. However, because a vulnerability and its patch share nearly identical topologies, the TPG struggles to discriminate between patches. Although Joern CPG appears to discriminate patches better, this does not reflect an understanding of vulnerability. Joern CPG's strict parser is sensitive to minor changes, causing it to push all samples apart regardless of relatedness. Consequently, this experiment demonstrates that purely structural, graph-only representations are not suited for the highly localized task of distinguishing vulnerabilities from their patches.

*3) **Experiment 3**.* To measure end-to-end performance, we benchmarked the complete AVA framework against prevailing graph-augmented hybrid models. The data revealed two major architectural findings. First, graph structure cannot replace raw sequence capacity; structure-aware baselines with shorter context windows (UniXcoder and GraphCodeBERT) truncate critical code regions and consistently underperform Jina-based configurations, proving that structural augmentation cannot compensate for a restricted token window. Second, adding graph structure to Jina V2 yields only marginal retrieval gains while actively worsening patch discrimination. Because the graph transformer propagates signals based on topology, and a patch shares a nearly identical structural layout with its vulnerable version, the graph layer inadvertently pulls patched code back into the vulnerable clusters, partially undoing SupCon's separation. Consequently, the sequence-only Jina V2 with patch-weighted SupCon remains the most effective and practical configuration for variant analysis. For practitioners, this outcome demonstrates that cross-repository variant retrieval does not require complex graph representations; a robust contrastive objective applied to a strong sequence encoder is sufficient to locate unpatched vulnerable clones while reliably excluding patches from retrieval results.

V. CONCLUSION

This work introduced the AVA model to address scalability and data issues in detecting vulnerability variants in OSS ecosystems. AVA includes an LLM-driven data augmentation pipeline and a TPG for scalable structural extraction. Experiments show that while structural context preserves semantic similarity, it hampers patch discrimination due to similar graph topology, pulling patched code back into the vulnerable cluster. Results indicate Jina V2 + patch-weighted Supervised Contrastive Loss outperforms graph-augmented models. The study

demonstrates that learning vulnerability semantics from real patch diffs offers a scalable basis for variant analysis. Future directions include developing a rigorous validation pipeline for data augmentation to ensure variants retain flaws, and exploring multi-task learning by adding an auxiliary classifier for better patch discrimination.

VI. Acknowledgement

This work was supported in part by the National Science Foundation under grant numbers No. DGE-2336109 (SFS). Additionally, Gemini was utilized for code generation and debugging across the research design and evaluation stages.

References

[1] M. Eslamimehr, "A multi-faceted analysis: The unseen costs and latent risks of open source software," Quandary Peak Research, Dec. 1, 2025. [Online]. Available: https://quandarypeak.com/2025/12/unseen-costs-and-latent-risks-of-oss/

[2] M. Hoffmann, F. Nagle, and Y. Zhou, "The value of open source software," Harvard Business School Strategy Unit Working Paper No. 24-038, 2024. [Online]. Available: https://doi.org/10.2139/ssrn.4693148

[3] Black Duck Software, "2026 open source security and risk analysis report," Feb. 25, 2026. [Online]. Available: https://www.blackduck.com/resources/analyst-reports/open-source-security-risk-analysis.html

[4] S. Elder, M. R. Rahman, G. Fringer, K. Kapoor, and L. Williams, "A survey on software vulnerability exploitability assessment," ACM Comput. Surv., vol. 56, no. 8, Art. no. 205, 2024, doi: 10.1145/3648610.

[5] T. H. M. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," ACM Comput. Surv., vol. 55, no. 5, Art. no. 100, 2022, doi: 10.1145/3529757.

[6] SentinelOne, "Securing the supply chain: Managing the risk of open source software," May 10, 2023. [Online]. Available: https://www.sentinelone.com/blog/software-management-a-guide-for-every-business-using-open-source-in-2023/

[7] W. Chabbott and J. Fletcher, "Multi-repository variant analysis: A powerful new way to perform security research across GitHub," The GitHub Blog, Mar. 9, 2023. [Online]. Available: https://github.blog/security/vulnerability-research/multi-repository-variant-analysis-a-powerful-new-way-to-perform-security-research-across-github/

[8] E. Lim, "Finding more zero days through variant analysis," Semgrep, Jul. 10, 2025. [Online]. Available: https://semgrep.dev/blog/finding-more-zero-days-through-variant-analysis

[9] M. Stone, "The ups and downs of 0-days: A year in review of 0-days exploited in-the-wild in 2022," Google Security Blog, Jul. 27, 2023. [Online]. Available: https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html

[10] GitHub, "About CodeQL," CodeQL documentation, 2025. [Online]. Available: https://codeql.github.com/docs/codeql-overview/about-codeql/

[11] A. Lekssays, H. Mouhcine, K. Tran, T. Yu, and I. Khalil, "LLMxCPG: Context-aware vulnerability detection through code property graph-guided large language models," in Proc. 34th USENIX Secur. Symp. (SEC '25), 2025, Art. no. 26, pp. 489–507.

[12] Y. Yang, B. Xu, X. Gao, and H. Sun, "Context-enhanced vulnerability detection based on large language models," ACM Trans. Softw. Eng. Methodol., 2025. [Online]. Available: https://doi.org/10.1145/3779222

[13] Y. Jiang, Y. Zhang, X. Su, C. Treude, and T. Wang, "StagedVulBERT: Multigranular vulnerability detection with a novel pretrained code model," IEEE Trans. Softw. Eng., vol. 50, no. 12, pp. 3454–3471, 2024, doi: 10.1109/TSE.2024.3493245.

[14] C. Thapa, S. I. Jang, M. E. Ahmed, S. Camtepe, J. Pieprzyk, and S. Nepal, "Transformer-based language models for software vulnerability detection," in Proc. 38th Annu. Comput. Secur. Appl. Conf. (ACSAC '22), 2022, pp. 481–496, doi: 10.1145/3564625.3567985.

[15] Y. Wu, D. Zou, S. Dou, W. Yang, D. Xu, and H. Jin, "VulCNN: An image-inspired scalable vulnerability detection system," in Proc. 44th Int. Conf. Softw. Eng. (ICSE '22), 2022, pp. 2365–2376, doi: 10.1145/3510003.3510229.

[16] B. Lazarine, S. Pulipaka, S. Samtani, and R. Venkataraman, "Collecting, linking, and assessing machine learning open-source software: A large scale collection and vulnerability assessment pipeline," in Proc. 58th Hawaii Int. Conf. Syst. Sci. (HICSS), 2025, pp. 398–405, doi: 10.24251/HICSS.2025.048.

[17] S. Ullman, S. Samtani, H. Zhu, B. Lazarine, H. Chen, and J. F. Nunamaker, "Enhancing Vulnerability Prioritization in Cloud Computing Using Multi-View Representation Learning," J. Manage. Inf. Syst., vol. 41, no. 3, pp. 708–743, 2024, doi: 10.1080/07421222.2024.2376384.

[18] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "CodeT5+: Open code large language models for code understanding and generation," in Proc. Conf. Empir. Methods Nat. Lang. Process. (EMNLP), Singapore, 2023, pp. 1069–1088.

[19] D. Zhang et al., "Code representation learning at scale," in Proc. 12th Int. Conf. Learn. Represent., 2024. [Online]. Available: https://openreview.net/forum?id=vfzRRjumpX

[20] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in Proc. 60th Annu. Meet. Assoc. Comput. Linguistics, Dublin, Ireland, 2022, pp. 7212–7225.

[21] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in Findings Assoc. Comput. Linguistics: EMNLP 2020, Online, 2020, pp. 1536–1547.

[22] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learn. Represent., 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[23] S. Wi, S. Woo, J. J. Whang, and S. Son, "HiddenCPG: Large-scale vulnerable clone detection using subgraph isomorphism of code property graphs," in Proc. ACM Web Conf. 2022, 2022, pp. 755–766.

[24] T. Bui et al., "VulCoCo: A Simple Yet Effective Method for Detecting Vulnerable Code Clones," arXiv preprint arXiv:2507.16661, 2025.

[25] S. Woo, H. Hong, E. Choi, and H. Lee, "MOVERY: A precise approach for modified vulnerable code clone discovery from modified open-source software components," in Proc. 31st USENIX Secur. Symp. (USENIX Security 22), 2022, pp. 3037–3053.

[26] H. W. Alomari, C. Vendome and H. Gyawali, "A Slicing-Based Approach for Detecting and Patching Vulnerable Code Clones," 2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC), Ottawa, ON, Canada, 2025, pp. 60-72, doi: 10.1109/ICPC66645.2025.00015.

[27] H. Liu, J. Zhan, and Q. Zhang, "Uncertainty-aware contrastive learning with hard negative sampling for code search tasks," in Proc. AAAI Conf. Artif. Intell., vol. 39, no. 18, pp. 18807–18815, 2025, doi: 10.1609/aaai.v39i18.34070.

[28] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal., 2022, pp. 519–531, doi: 10.1145/3533767.3534371.

[29] Y. Ding et al., "Concord: Clone-aware contrastive learning for source code," in Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Test. Anal., Jul. 2023, pp. 26–38.

[30] X. Liu et al., "Self-supervised learning: Generative or contrastive," IEEE Trans. Knowl. Data Eng., vol. 35, no. 1, pp. 857–876, 2023, doi: 10.1109/TKDE.2021.3090866.

[31] A. Rastogi et al., "Devstral: Fine-tuning language models for coding agent applications," arXiv preprint arXiv:2509.25193, 2025

[32] Z. Liu and W. Xie, "Litevul: A lightweight vulnerability detection via directed PPMI-weighted token graphs," Cybersecurity, vol. 9, Art. no. 140, 2026, doi: 10.1186/s42400-026-00570-x

[33] M. Brunsfeld et al., "Tree-sitter," GitHub repository, 2018. [Online]. Available: https://github.com/tree-sitter/tree-sitter