BRIDGING THE REALISM GAP: GENERATING FORMALLY VERIFIED VULNERABILITY DATASETS FROM REAL-WORLD CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

The advancement of machine learning for vulnerability detection is critically hampered by the absence of datasets that are simultaneously large-scale, accurately labeled, and realistic. Existing benchmarks impose a trade-off: large, real-world datasets suffer from noisy labels and contamination, while manually curated datasets are too small to train robust models. Synthetic datasets, although formally verifiable, typically lack the structural complexity of production code, resulting in a significant "realism gap". To overcome these limitations, we introduce *ApproxVul*, a framework and dataset that unites real-world code realism with the mathematical certainty of formal verification. Our framework leverages Large Language Models (LLMs) to systematically mutate code snippets from real-world projects, introducing a diverse range of subtle and complex vulnerabilities. Each resulting program, along with its safe counterpart, is then formally verified to establish ground-truth labels, eliminating label noise. This process yields ApproxVul, a new dataset of over 104,000 compilable and verifiable programs, featuring minimally different vulnerable/safe pairs derived from real-world code. Through comprehensive evaluation, we demonstrate that ApproxVul achieves better cross-dataset generalization than purely synthetic training datasets and slightly outperforms noisy real-world training data. While fine-tuning alone remains insufficient for project-level generalization, ApproxVul's inter-procedural and verifiable structure makes it a crucial stepping stone toward more advanced vulnerability detection approaches.

1 INTRODUCTION

The growing use of deep learning models for code has started a new era of automated software security, with a strong focus on vulnerability detection. The effectiveness of these models, however, is limited by the quality and scale of the datasets they are trained on. A key challenge in the field is the shortage of datasets that are large-scale, accurately labeled, and sufficiently realistic to effectively train models capable of addressing the complexities of real-world software.

This shortage forces researchers into a trade-off, as existing datasets only partially satisfy the key requirements for high-quality and effective vulnerability detection. On one hand, large-scale datasets derived from real-world software projects, such as BigVul (Fan et al., 2020) and DiverseVul (Chen et al., 2023), offer breadth but often lack labeling accuracy or consistency (Ding et al., 2024; Risse et al., 2025). To address these quality concerns, recent work has aimed at enhancing the fidelity of real-world datasets. For example, ICVul (Lu et al., 2024) applies advanced filtering methods to mitigate label noise. Meanwhile, datasets such as SVEN (He & Vechev, 2023) achieve higher label accuracy through manual verification, though this approach is prohibitively costly to scale for training large models.

Other datasets contribute valuable inter-procedural context. For example, ARVO (Mei et al., 2024) builds a dataset of over 5,000 reproducible vulnerabilities from fuzzer-generated bug reports, each with a triggering input and recompilable code. While such approaches advance beyond earlier efforts, they face scalability limitations. To address the need for verifiable labels at scale, synthetic datasets like FormAI (Tihanyi et al., 2023) have been introduced; however, they often fall short in realism, as they generate simple code from scratch and fail to capture the complexity of production software.

At the heart of this challenge lies the prohibitive cost and difficulty of obtaining ground-truth labels at scale. Manual verification by security experts remains the gold standard for accuracy, but it is impractical for constructing large datasets. This limitation has driven the community toward automated methods, which to date have not resolved the fundamental trade-off. Mining real-world code provides realism but often compromises label quality. In contrast, synthetic dataset generation has emerged as an attempt to solve the labeling problem. For example, FormAI (Tihanyi et al., 2023) generates code from scratch and applies formal verification to achieve perfectly accurate labels at scale.

However, this introduces a significant realism gap: formal verifiers are most effective on small, self-contained programs, leading to generated code that is overly simplistic. Such code typically omits the complex constructs (such as intricate data types, macros, and conditional compilation directives "`#ifdef`") that are pervasive in production software yet notoriously difficult to verify. Consequently, while these datasets are verifiable, they fail to capture the complexity of real-world vulnerabilities.

To address this trade-off, we introduce *ApproxVul*, a new framework for generating a large-scale, formally verified dataset that combines real-world realism with the certainty of formal methods. Unlike prior approaches that synthesize code from scratch, *ApproxVul* begins with real-world code fragments and leverages large language models (LLMs) to introduce a diverse set of both subtle and complex vulnerabilities. Each resulting program is then subjected to formal verification tools, which determine with certainty whether it is free of memory vulnerabilities. This pipeline produces a dataset that is simultaneously large, accurately labeled, and faithful to the structural complexity of real-world software.

Our main contributions are summarized as follows:

1. We propose a scalable and flexible framework for generating formally verified vulnerability datasets. The framework supports multiple programming languages, vulnerability types, and complexity levels.
2. We construct *ApproxVul*, a large-scale dataset of over 104,000 formally verified C/C++ programs. It also includes closely matched vulnerable-safe pairs derived from real-world code, providing a high-quality and realistic benchmark for training and evaluating vulnerability detection models.

2 Challenges and Related Work

The creation of effective datasets for machine learning-based vulnerability detection (VD) is fraught with systemic challenges that have limited the field's progress. Existing benchmarks, despite their widespread use, suffer from fundamental flaws related to data quality, experimental design, and the very formulation of the vulnerability detection task. Here, we dissect these core issues and situate the contributions of ApproxVul within the current landscape.

2.1 The Foundational Flaws of Real-World Datasets

The dominant paradigm in VD has involved a trade-off between scale and quality. On one hand, researchers have mined large-scale datasets from open-source software repositories, creating benchmarks like BigVul (Fan et al., 2020), Devign (Zhou et al., 2019), DiverseVul (Chen et al., 2023), and CVEfixes (Bhandari et al., 2021). While offering impressive scale, this approach is built on unstable foundations.

Noisy and Inaccurate Labels. The primary flaw lies in the heuristic-based labeling process. These datasets operate under the assumption that any function modified within a "vulnerability-fixing commit" was itself vulnerable. This is demonstrably false. As highlighted by Ding et al. (Ding et al., 2024) and Croft et al. (Croft et al., 2023), this method introduces significant label noise, compounded by *tangled patches*, where commits mix security fixes with unrelated changes (Wang et al., 2024). This makes it nearly impossible to isolate the true vulnerability-related changes.

Recognizing these quality issues, recent efforts have focused on improving the fidelity of real-world datasets. Datasets like ICVul (Lu et al., 2024) and ReposVul (Wang et al., 2024) employ advanced filtering techniques and VCC (Vulnerability-Contributing Commit) tracing to enhance label reliability. ARVO (Mei et al., 2024), on the other hand, builds a dataset of over 5,000 reproducible vulnerabilities from fuzzer-generated bug reports, ensuring each entry has a triggering input and is recompilable, thus providing valuable inter-procedural context. ReposVul also captures inter-procedural information by constructing repository-level call graphs (Wang et al., 2024). To achieve the highest quality labels, datasets like SVEN (He & Vechev, 2023) rely on meticulous manual curation. However, this process is prohibitively expensive and not scalable for training large models (SVEN contains 1.6k programs). While these approaches significantly improve upon their predecessors, they are not formally verified and still rely on post-hoc analysis of commits or bug reports.

Data Duplication and Contamination. Real-world datasets are also plagued by data duplication and contamination. PrimeVul's analysis revealed that up to 18.9% of test samples in some benchmarks are exact copies of training samples, which artificially inflates performance metrics (Ding et al., 2024). Furthermore, there is often an *unclear separation of projects* between training and testing splits, risking that a model learns project-specific idioms rather than generalizable vulnerability patterns.

Our Approach: The ApproxVul framework is designed to achieve the best of both worlds: the accuracy of formal verification at the scale of automated collection. Instead of relying on noisy, unverifiable heuristics, we use a formal verifier as the ultimate arbiter of truth. Every label is the result of a mathematical proof, eliminating label

noise. Because our data is generated synthetically from real-world seeds, we avoid issues of tangled patches and data contamination, producing a large-scale, high-fidelity dataset suitable for training robust models.

2.2 The Limits of Existing Synthetic Datasets

To bypass the noise and ambiguity of real-world data, some approaches have turned to synthetic data generation. The FormAI dataset, for instance, represents a significant step by using an LLM to generate a large corpus of C programs from scratch, which are then formally verified (Tihanyi et al., 2023). This methodology successfully addresses the verifiability challenge at scale, producing labels with mathematical certainty.

However, this verifiability is achieved by constraining the code's complexity to maintain tractability for formal verifiers. Real-world C/C++ code is rich with constructs like complex data structures, macros, and preprocessor directives (`#ifdef`). While essential for production software, these elements cause a combinatorial explosion in the number of possible program states, making formal verification of an entire codebase computationally intractable. Consequently, to enable verification, FormAI generates small, independent programs from generic prompts (e.g., "create a board game"), intentionally omitting the complex, interdependent constructs found in real-world software. This simplification leads to a dataset with less complex control flows and data structures, presenting a sanitized and less challenging detection task (Lekssays et al., 2025).

Our Approach: ApproxVul resolves this trade-off between realism and verifiability by changing the scope of analysis. Instead of generating entire programs from scratch, we start with real-world code seeds—individual functions or snippets that already contain realistic structures. While formally verifying the large applications these seeds originate from would be infeasible, verifying the seeds in a bounded, self-contained context is computationally practical. This approach ensures our dataset inherits a high degree of syntactic and structural realism, including macros and complex data types. Our LLM-based mutation process then introduces vulnerabilities, creating a dataset that is both verifiable and realistically challenging.

2.3 The Pitfalls of Decontextualization and Spurious Learning

Beyond data quality, a more profound challenge lies in the conventional formulation of problems itself.

Context is Critical. The vast majority of machine learning based vulnerability detection (ML4VD) research frames the task as a function-level binary classification problem. However, as (Risse et al., 2025) demonstrates, this is often an *ill-posed problem*. The vulnerability of a function is almost always context-dependent. Stripping a function of its calling context makes it impossible to determine its vulnerability status from the code alone.

The Inevitability of Spurious Learning. This lack of sufficient information forces models to find statistical shortcuts. Risse et al. showed that high classification scores could be achieved using a simple model trained only on word counts (Risse et al., 2025). This confirms that models are not genuinely detecting vulnerabilities but are exploiting superficial patterns, a phenomenon noted in other studies on the fragility of ML4VD models (Risse & Böhme, 2024).

Our Approach: ApproxVul tackles the context problem directly. Instead of providing an isolated function, our framework generates a complete, self-contained, and compilable program for each sample, including a 'main' entry point. This ensures all necessary local context for the vulnerability to manifest is present, allowing the bounded model checker to formally prove its existence. Furthermore, our generation of minimally different vulnerable/safe pairs explicitly discourages spurious learning. We provide a comparative summary of datasets in Appendix A.

3 Methodology

To address the scarcity of large-scale, high-quality, and verifiable datasets for software vulnerability analysis, we propose a novel automated framework for generating synthetic code samples with ground-truth vulnerability labels. Our framework, illustrated in Figure 1, is a multi-stage pipeline that orchestrates multiple Large Language Model (LLM) agents with formal verification tools. It supports a wide range of programming languages, including C/C++, Java, Kotlin, Python, and Solidity. Our framework is modular, so it can support other formal verifiers and thus other programming languages and target vulnerabilities. We discuss the different design considerations in Section 5.

3.1 Phase 1: Program Generation

The generation process starts with a "seed": an isolated function extracted from a real-world codebase. By itself, a seed is a non-compilable snippet lacking the necessary context to be executed (e.g., a `main` function, headers, or caller

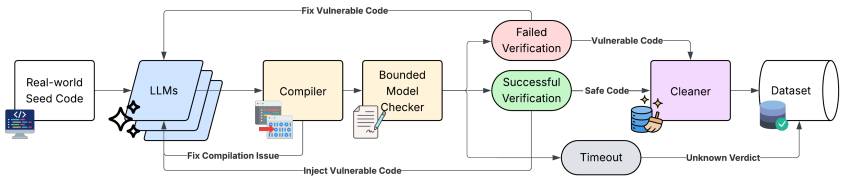

Figure 1: The end-to-end pipeline of ApproxVul framework.

functions). Our first step is to transform this seed into a complete and compilable "program" that is a self-contained source file with all necessary components.

To achieve this, we employ a Large Language Model (LLM) agent guided by a detailed prompt (see Appendix H). The agent's core task is twofold: (1) to embed the seed's original logic within a new function and (2) to construct a realistic surrounding context for it. This context is created by generating 5–7 interconnected functions, including a `main` entry point, that establish a plausible call graph and data flow. The purpose of this contextual expansion is critical for creating a high-fidelity dataset. Real-world vulnerabilities are rarely found in trivial, single-function programs. By embedding the seed's logic within a multi-function program, we create a more challenging and realistic environment for training vulnerability detection models, compelling them to learn inter-procedural analysis rather than simply memorizing intra-procedural patterns.

Inspiration from the seed's core functionality serves to anchor the synthetic program in a real-world logic pattern, enhancing the dataset's realism. It constrains the LLM to build a realistic scaffold around a known, human-written piece of code. As part of this process, the agent is also instructed to inject exactly one vulnerability from a predefined list of CWEs (e.g., CWE-120 Buffer Copy without Checking Size of Input, CWE-476 NULL Pointer Dereference), thus producing the initial "vulnerable" program for our dataset. In addition, we force the LLMs to output valid commands for required dependencies to run the program. To this end, we specify the compilation environment (e.g., Docker container with Ubuntu 24.04).

3.2 Phase 2: Automated Compilation and Correction

Syntactic validity is enforced at every stage of code generation. Our framework integrates a compiler agent that attempts to build any code produced by an LLM. If compilation fails, the compiler's error log is fed back to a specialized "debugging" LLM agent. Its prompt (shown in Appendix H) instructs it to apply the minimal changes necessary to fix the compilation error while preserving the intended logic. This self-correction loop is repeated up to three times, significantly increasing the yield of compilable programs.

3.3 Phase 3: Formal Verification with Bounded Model Checking

Regarding the formal verification, all compiled code samples are analyzed by a bounded model checking agent that employs ESBMC (Efficient SMT-based Bounded Model Checker) (Li et al., 2024) to verify security properties. We use ESBMC as an external tool without modifying its internals. Further adaptations and possible extensions of this component are discussed in Section 5. The ESBMC verifier has three types of outputs: i) *Successful Verification:* ESBMC mathematically proves the code is safe within the bounded context, confirming the "safe" label; ii) *Failed Verification:* ESBMC finds a counterexample that violates a security property, confirming the "vulnerable" label. The resulting log is crucial for the remediation phase; and iii) *Timeout/Inconclusive:* The process exceeds a time limit, and the sample is discarded to maintain dataset integrity.

3.4 Phase 4: Paired Vulnerable/Safe Code Generation

After a program is labeled "safe" or "vulnerable", it is submitted to a pool of parallel LLM agents that synthesize a corresponding counterpart — producing a safe version from a vulnerable seed or a vulnerable version from a safe seed — so that each result forms a minimally different pair. The framework accomplishes this using two complementary strategies, enabling pair generation starting from either vulnerable or safe inputs: i) *Vulnerability Remediation:* This is the primary path for creating a pair. The vulnerable program from Phase 1, along with its corresponding ESBMC verification log, is given to a remediation agent. Guided by its prompt shown in Appendix H), this LLM is instructed to act as a security analyst, analyze the ESBMC log to understand the root cause of the vulnerability, and apply a precise

fix. The result is a 'safe' version of the code, creating the pair $(c_{vulnerable}, c_{safe})$; and ii) *Vulnerability Injection:* For the programs verified to be 'safe' programs in Phase 1, we pass them to an injection agent. This agent is prompted to introduce a single, subtle memory vulnerability and ensuring it still compiles. The result is a 'vulnerable' version of the code, creating the pair $(c_{vulnerable}, c_{safe})$. We note that the generation loop for a given program ends when it has its pair or it exceeds three attempts in making a pair (as per Phase 3). We note that the generated pairs undergo the same compilation and verification process.

3.5 Phase 5: Code Sanitization

To prevent models from overfitting on trivial lexical cues, we sanitize all verified code samples with a cleaning agent. The generator sometimes inserted explicit labels such as comments ("this is vulnerable/safe") or identifiers (`vulnerable_function`, `safe_example`), which would let models rely on surface tokens rather than learning semantic patterns. The cleaning agent removes all comments, normalizes string literals, and renames identifiers to neutral forms (e.g., `vulnerable_function()` $\rightarrow$ `process_data()`). This ensures that code remains compilable and semantically equivalent while eliminating unrealistic markers. As a result, models trained on the dataset learn vulnerability-relevant semantics instead of exploiting generator artifacts.

To illustrate the workflow of our pipeline, we included an example generated using the *mistralai/Codestral-22B-v0.1*. In Section C, we present a case of generated vulnerable code exhibiting a NULL pointer dereference, derived from a real-world seed in the FFmpeg multimedia library.

4 Evaluation

To assess the quality and utility of *ApproxVul*, we designed an evaluation around two central research questions:

- **RQ1 (Realism):** How realistic is our synthetic dataset compared to real-world and other synthetic vulnerability datasets?
- **RQ2 (Performance):** To what extent can ApproxVul, as a structured and verifiable dataset, improve cross-dataset generalization compared to purely synthetic and noisy real-world vulnerability datasets?

4.1 Datasets and Implementation Details

Our comparative analysis benchmarks *ApproxVul* against prominent existing datasets to cover different data sources and generation philosophies. The generation process for ApproxVul utilized the entirety of the 15k samples from ICVul as seeds. For comparison, we selected *PrimeVul*, which contains over 200k samples with a significant class imbalance (only 7k vulnerable). As a representative of state-of-the-art synthetic data, we included *FormAI*, a dataset of 112k formally verified programs generated from scratch. We note that we excluded SVEN because the majority of its vulnerabilities are web related.

To ensure fair comparison, we employed stratified sampling to create balanced 12,000-sample subsets from each dataset (10k training, 1k validation, 1k testing). Stratification was based on Common Weakness Enumeration (CWE) types and cyclomatic complexity to ensure representativeness. This split size provides a fair baseline, as PrimeVul contains only 7k vulnerable functions, making larger balanced sets impractical. All splits maintained equal numbers of vulnerable and safe samples to prevent class imbalance. For testing, we used ARVO Mei et al. (2024) and ReposVul Wang et al. (2024) as project-level datasets capturing multi-function vulnerabilities, including inter-procedural cases and realistic development contexts that present challenging generalization targets.

To further guarantee the validity of our evaluation, we applied a *strict project-level decontamination rule*: no project appears in both training, validation and test sets across any dataset to mitigate the issues discussed in Section 2. For instance, if a CVE-linked project was present in a training split, it was excluded from all test splits. This prevents models from exploiting project-specific coding idioms and ensures that test performance truly reflects generalization.

For code generation and performance evaluation, we utilized a diverse suite of Large Language Models (LLMs). The generation models were hosted using vLLM for efficient inference. For the performance evaluation (RQ2), models were fine-tuned using the *LLaMA-Factory* library (Zheng et al., 2024) (see Appendix F for more details).

4.2 RQ1: Realism Evaluation

To answer RQ1, we performed a two-faceted evaluation of our dataset, *ApproxVul*, comparing it against two real-world datasets (*PrimeVul* (Ding et al., 2024), *ICVul* (Lu et al., 2024)) and a state-of-the-art synthetic dataset (*FormAI* (Tihanyi

et al., 2023)). We assess realism through two lenses: syntactic realism, which captures the high-level structure and style of code, and semantic realism, which analyzes fine-grained, interpretable software metrics. Our analysis demonstrates that *ApproxVul* achieves a significantly higher degree of realism than *FormAI*, validating our generation methodology rooted in real-world seeds.

4.2.1 Syntactic Realism via Embedding Space Analysis

We first evaluate syntactic realism by analyzing the distribution of code snippets in a high-dimensional feature space.

Methodology & Evaluation Metrics. We generated contextual embeddings for each code snippet in ours 12k splits for *ApproxVul*, *PrimeVul*, *FormAI*, and the full 15k that we used for generation for *ICVul* using *Qwen/Qwen3-Embedding-4B* as the best performing model in code tasks in MTEB Leaderboard as of August 31, 2025. We then employed a suite of metrics to compare the resulting distributions. The results hereafter show that *ApproxVul* is syntactically closer to real-world datasets (i.e., *PrimeVul* and *ICVul*) than FormAI (as a synthetically generated dataset). To compare the high-dimensional embedding distributions, we used the metrics in Table 1.

Table 1: Embedding and Clustering Metrics

Category	Metric	Concise Description
Distributional	Maximum Mean Discrepancy	Measures distance between distribution means in a feature space.
	Wasserstein Distance	Minimum "cost" to transform one distribution into another.
	Centroid Distance	Euclidean distance between the mean vectors of distributions.
Clustering Evaluation	Adjusted Rand Index (ARI)	Similarity between true & clustered labels, corrected for chance.
	V-Measure	Harmonic mean of a clustering's homogeneity and completeness.
	Silhouette Score	Measures how similar a point is to its own cluster vs. others.

Results & Analysis. As shown in Table 1, all three distance metrics confirm that *ApproxVul* is syntactically closer to the real-world datasets than *FormAI*.

Table 2: Syntactic distance from synthetic datasets to real-world datasets. Lower values indicate greater similarity. ApproxVul is consistently closer to both real-world benchmarks.

Distance Metric	ApproxVul vs. PrimeVul	FormAI vs. PrimeVul	ApproxVul vs. ICVul	FormAI vs. ICVul
MMD	0.0730 (↑26.11%)	0.0988	0.0690 (↑29.09%)	0.0973
Centroid Distance	0.4168 (↑18.89%)	0.5138	0.4068 (↑20.58%)	0.5122
Wasserstein Distance	0.0072 (↑13.25%)	0.0083	0.0066 (↑24.14%)	0.0087

The visualizations in Figure 2 further illustrate this. Both PCA (preserving global variance) and t-SNE (preserving local structure) show that *FormAI* embeddings form an isolated cluster, while *ApproxVul* embeddings are closer to those of the real-world datasets.

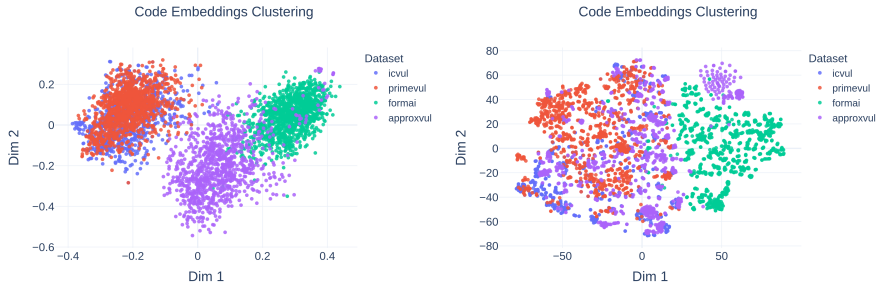

Figure 2: PCA (left) and t-SNE (right) visualizations of *Qwen/Qwen3-Embedding-4B* embeddings.

Our clustering evaluation metrics support this visual behavior. The high ARI (0.66) and V-Measure (0.76) scores indicate that the datasets have distinct enough signatures to be clustered effectively. Most importantly, the low Silhouette Score (0.04) is a positive result in this context. It quantitatively demonstrates that the clusters are not cleanly separable, confirming that the *ApproxVul* cluster heavily overlaps with the real-world clusters.

To ensure our findings on syntactic realism were not specific to a single embedding model, we replicated this analysis using the *infly/infly-retrieval-7b* model. The results are shown in Appendix E.

4.2.2 SEMANTIC REALISM VIA CODE METRICS ANALYSIS

To quantitatively evaluate the semantic realism of our generated dataset, *ApproxVul*, we conducted a comparative analysis against real-world and other synthetic code datasets. The goal is to demonstrate that *ApproxVul* more accurately captures the intrinsic statistical and structural properties of human-written C/C++ code than other synthetic generation methods, represented by the *FormAI* dataset. We use two real-world, non-compilable code snippet datasets, *PrimeVul* and *ICVul*, as the ground truth for our comparisons.

Code Metrics. We employ two distinct categories of metrics to create a comprehensive profile of each code snippet, capturing both its surface-level characteristics and its deep grammatical structure.

Table 3: Summary of Code Analysis Metrics

Category	Metric	Description
Statistical	Lines of Code (LOC)	Count of non-empty code lines.
	Cyclomatic Complexity	Measures independent paths in the code's control flow.
	Halstead Metrics	Quantifies complexity from operators and operands.
	Naming Metrics	Statistics on identifiers (e.g., average length, diversity).
Structural	Control Flow Constructs	Frequency of control statements (if, for, switch).
	Function & Expression Logic	Frequency of computational nodes (calls, binary expressions).
	Declaration Patterns	Frequency of declaration and function definition nodes.
	Maximum Tree Depth	Longest path in the CST*, representing structural complexity.

*CST: Concrete Syntax Tree extracted with tree-sitter parser as it is a level deeper than Abstract Syntax Trees (AST).

Methodology & Evaluation Metrics. Our methodology is designed to compare the distributions of code metrics between the synthetic datasets (*ApproxVul*, *FormAI*) and the real-world datasets (*PrimeVul*, *ICVul*). For each metric, we treat the collection of values from a dataset as an empirical probability distribution.

To quantify the dissimilarity between two distributions, we employ the *two-sample Kolmogorov-Smirnov (KS) test*. We selected the KS test because it is non-parametric, meaning it makes no assumptions about the underlying data distribution (e.g., normality), which is essential as code metrics often have irregular, non-standard distributions. Furthermore, it is sensitive to differences in both the location and shape of the distributions, making it a comprehensive tool for this comparison. The KS test yields a statistic (D) ranging from 0 to 1, where 0 indicates identical distributions and 1 indicates maximum divergence. Therefore, a *lower KS statistic signifies greater semantic similarity*. We also consider the p-value to filter for statistically significant comparisons ($p < 0.05$).

Results & Analysis. The analysis reveals that *ApproxVul* is consistently and significantly more similar to both real-world datasets than *FormAI* across both statistical and structural metrics.

Structural Realism Analysis. The structural metrics provide the most powerful evidence of semantic realism. The results show that *ApproxVul*'s generation process more accurately models the grammatical and logical structures inherent in human-written code. When comparing against the statistically significant metrics from both real-world datasets (93 metrics for *PrimeVul* and 96 for *ICVul*), *ApproxVul* was the closer match in approximately 78% of all cases (see Appendix D for the full list of structural metrics). As shown in Table 4, *ApproxVul* achieves a significantly lower average KS statistic, indicating a much higher overall structural fidelity.

Table 4: Overall Structural Similarity to Real-World Datasets. A lower average KS Statistic is better. "Wins" indicates the number of significant metrics where the dataset was a closer match.

	Avg. KS Statistic ↓		Metric Wins	
Comparison	**ApproxVul**	**FormAI**	**ApproxVul**	**FormAI**
vs. *PrimeVul*	**0.254**	0.380	**74**	19
vs. *ICVul*	**0.244**	0.339	**73**	23

Table 5 provides a detailed comparison for a subset of critical CST nodes. *ApproxVul* consistently outperforms *FormAI* in modeling the frequency of fundamental constructs like control flow statements (`if`, `for`), expressions, and overall structural depth. While *FormAI* occasionally provides a better match for specific declaration patterns (e.g., `function_definition`), the overwhelming evidence points to the superior structural realism of *ApproxVul*.

Statistical Realism Analysis. The analysis of statistical metrics corroborates the findings from the structural analysis. *ApproxVul* consistently demonstrates distributions of stylistic and lexical features that are closer to real-world code. Across the key statistical metrics, *ApproxVul* achieved a lower average KS statistic against both *PrimeVul* (0.323 vs. 0.414) and *ICVul* (0.278 vs. 0.363), winning the majority of head-to-head metric comparisons. This indicates that *ApproxVul* not only captures the correct grammatical structure but also the statistical texture of human-written code more effectively.

Table 5: KS Statistics for Key Structural Metrics. The **bolded** value indicates the dataset with higher realism (lower dissimilarity) for each metric.

	vs. PrimeVul		vs. ICVul	
Key Structural Metric	**ApproxVul**	**FormAI**	**ApproxVul**	**FormAI**
If Statement	**0.166**	0.219	**0.182**	0.243
For Statement	**0.095**	0.284	**0.068**	0.306
Call Expression	**0.201**	0.375	**0.173**	0.284
Binary Expression	**0.142**	0.212	**0.138**	0.167
Declaration	**0.272**	0.341	**0.218**	0.291
Function Definition	0.890	**0.769**	0.889	**0.767**
Max Tree Depth	**0.164**	0.266	0.223	**0.187**

4.3 RQ2: Performance Evaluation

The goal of our evaluation is to rigorously assess the effectiveness of *ApproxVul* as a training dataset. Specifically, we aim to determine whether *ApproxVul* improves cross-dataset generalization compared to existing benchmarks and whether its construction translates into measurable gains in performance. To ensure that results are reliable and free from contamination, we enforced a project-level separation between training/validation, and test splits.

Methodology. We fine-tuned our model using LLaMA-Factory (Zheng et al., 2024) with a LoRA-based approach configured with a rank of 8 and applied to all target modules. The training was performed with a sequence cutoff length of $12,000$ tokens and capped at $10,000$ samples. We used a batch size of 1 per device with 8 gradient accumulation steps, a learning rate of 10^{-4} under a cosine scheduler with a warmup ratio of 0.1, and trained for 2 epochs in total. All fine-tuning experiments were conducted on one NVIDIA H100 GPU.

Results & Analysis. Table 6 presents the performance of models trained on different datasets across five test distributions. We report both accuracy and recall, alongside averaged results to assess generalization trends. All reported metrics are averaged over five independent testing runs.

Table 6: Evaluation Table

Training Dataset	**Model**	**Test Datasets**										**Average**	
		FormAI		PrimeVul		ApproxVul		ARVO		ReposVul		Acc	Rec
		Accuracy	Recall	Accuracy	Recall	Accuracy	Recall	Accuracy	Recall	Accuracy	Recall		
None	Seed-Coder-8B-Instruct	0.5889	0.6581	0.6816	0.6683	0.6078	0.8520	0.5144	0.9333	0.4970	0.9340	0.5666	0.5168
	Qwen3-8B	0.6182	0.5249	0.5611	0.3860	0.6087	0.6310	0.5098	0.3263	0.5083	0.1654		
	Qwen3-Coder-30B-A3B-Instruct	0.6561	0.6212	0.5501	0.2998	0.6622	0.6970	0.5043	0.6261	0.4904	0.3576		
	GPT-OSS-20B	0.6625	0.7576	0.5720	0.4833	0.6872	0.7380	0.5323	0.5084	0.4963	0.3456		
	Codestral-22B-v0.1	0.5304	0.1172	0.5535	0.2344	0.5900	0.5205	0.4949	0.3648	0.4876	0.1699		
FormAI	Seed-Coder-8B-Instruct	0.9099	0.8957	0.5467	0.2105	0.6765	0.4438	0.5022	0.1872	0.5047	0.3510	**0.6261**	0.4312
	Qwen3-8B	0.8972	0.9005	0.5265	0.2009	0.6631	0.4082	0.5086	0.3340	0.4991	0.2691		
	Qwen3-Coder-30B-A3B-Instruct	0.9075	0.9020	0.4928	0.2392	0.7130	0.5632	0.5000	0.1377	0.4965	0.1580		
	GPT-OSS-20B	0.8901	0.8973	0.5552	0.3955	0.7157	0.5579	0.5269	0.2962	0.4914	0.2207		
	Codestral-22B-v0.1	0.8964	0.9133	0.5703	0.3732	0.6880	0.4920	0.4911	0.2419	0.4826	0.1902		
PrimeVul	Seed-Coder-8B-Instruct	0.5897	0.7030	0.7751	0.8341	0.6069	0.6934	0.5110	0.7191	0.5897	0.7030	0.6020	0.6924
	Qwen3-8B	0.5826	0.6324	0.7885	0.8118	0.6150	0.8057	0.5064	0.4832	0.5087	0.5208		
	Qwen3-Coder-30B-A3B-Instruct	0.5818	0.8539	0.7995	0.7990	0.6239	0.8645	0.4946	0.4055	0.4935	0.4724		
	GPT-OSS-20B	0.5905	0.9069	0.8079	0.5071	0.6479	0.8520	0.5161	0.8235	0.5069	0.6465		
	Codestral-22B-v0.1	0.5336	0.6404	0.7860	0.7890	0.6025	0.5437	0.4978	0.4665	0.4938	0.8323		
ApproxVul	Seed-Coder-8B-Instruct	0.6458	0.9807	0.6445	0.6018	0.8307	0.8360	0.5307	0.7638	0.4962	0.7267	**0.6259**	**0.7147**
	Qwen3-8B	0.6443	0.9855	0.5855	0.4163	0.8333	0.8342	0.5108	0.5147	0.5035	0.3871		
	Qwen3-Coder-30B-A3B-Instruct	0.6553	0.9839	0.5476	0.2871	0.8396	0.8467	0.5258	0.4559	0.5074	0.2941		
	GPT-OSS-20B	0.6806	0.9952	0.6672	0.7001	0.8467	0.8485	0.4925	0.8004	0.4921	0.7335		
	Codestral-22B-v0.1	0.6435	0.9855	0.6201	0.6268	0.8520	0.8752	0.5324	0.7862	0.5196	0.6008		

Dataset landscape. The benchmarks used in this study cover distinct design philosophies. *FormAI* is synthetic and formally verified, but limited in diversity. *PrimeVul* is a large-scale, real-world dataset at the function level, where individual samples may be long, deeply nested, and prone to noisy labeling. *ARVO* and *ReposVul* are project-level datasets that capture vulnerabilities spanning multiple functions, including inter-procedural cases and realistic development contexts, making them especially difficult targets for generalization. In contrast, *ApproxVul* represents a hybrid approach: inter-procedural, compilable, and executable programs that preserve the style of real-world vulnerabilities while remaining small and self-contained, ensuring that vulnerability status can be verified.

Synthetic training (FormAI). Models trained on FormAI achieve the strongest in-distribution performance (accuracy and recall both $\approx$0.90), but collapse out-of-distribution, with average recall dropping to 0.43. Their failure to generalize beyond synthetic patterns highlights the risks of relying solely on synthetic data.

Function-level Real-world training (PrimeVul). PrimeVul-trained models generalize better than synthetic-only ones, reaching an average recall of 0.69. However, performance is uneven: they excel on their own test set (recall ≈0.81) but struggle with synthetic data and project-level datasets, reflecting challenges of noise and function-level design.

Hybrid training (ApproxVul). ApproxVul sits between these two extremes. On its own test set, it yields strong accuracy (0.83–0.85) and recall (0.83–0.88), showing internal consistency. More importantly, ApproxVul achieves the highest overall averages across datasets: 0.63 accuracy and 0.71 recall. The improvements are modest but meaningful, showing that hybrid datasets can outperform purely synthetic training and slightly edge out noisy real-world training.

Project-level Real-world Challenges (ARVO and ReposVul). Across all training datasets, performance drops sharply on ARVO and ReposVul. These datasets capture the full complexity of project-level code, including inter-procedural dependencies and deeply nested function structures, and substantially larger code units. Current fine-tuning approaches are insufficient to bridge this gap, underscoring the need for more advanced training paradigms.

While our experiments show that simple fine-tuning on ApproxVul already delivers measurable gains in cross-dataset generalization, the dataset's true potential lies in its richer structure. By capturing vulnerability context (e.g., call traces, crash outputs) in a verifiable manner, ApproxVul provides a foundation for advancing beyond basic supervised training. Future work can leverage these signals for more expressive learning paradigms, making ApproxVul less of a final solution and more of an essential stepping stone toward higher-quality, generalizable ML4VD systems.

5 Discussion

Our work in generating ApproxVul, a large-scale dataset for vulnerability detection, has yielded several insights into the capabilities and challenges of using LLMs for code generation, while highlighting current framework limitations.

Insights from Large-Scale Code Generation. The primary insight from our generation process is that current LLMs struggle more with syntactic precision than with high-level logic. The most significant hurdle by far was ensuring code could compile, with the majority of failures stemming from fundamental mistakes like missing standard headers and basic syntax errors, rather than complex logical flaws. For a detailed breakdown of these generation outcomes, see Section G.

Enforcing Target Code Metrics. During code generation, we attempted to enforce target code metrics in the prompts given a prior analysis of real-world datasets such as *PrimeVul*. However, the compilation rate dropped significantly across all models. We noticed that the produced code contains signs of hallucinations (e.g., repetitive function declarations to satisfy Lines of Code). In addition, we noticed that the LLMs struggle with nesting depth as the were not successful in produce deeply nested code. Future work will explore techniques to generate compilable and verifiable deeply nested code to simulate project-level real-world datasets like *ARVO* and *ReposVul*.

Limitations. Our approach, while effective, inherits several limitations from its components. Complex code with intricate loops or pointer arithmetic may cause a state-space explosion, leading the verifier to time out and restricting the complexity of programs that can be analyzed. Moreover, vulnerability detection is limited to the checks implemented in the verifier; although custom assertions can extend coverage, the default set is not exhaustive and may miss certain vulnerability classes. Finally, the models are constrained by their context length, which reduces their ability to capture dependencies in large, real-world codebases where vulnerabilities often arise from distant interactions.

Future Research Directions. There are several promising directions for future work. A natural extension is to adapt the framework to other programming languages like Java, Python, or JavaScript, which would necessitate developing language-specific components but would significantly broaden its applicability. Another avenue is to leverage LLMs to automatically generate custom assertions. This would allow the framework to dynamically check for a wider and more nuanced range of vulnerabilities beyond its predefined set, further enhancing its detection capabilities.

6 Conclusion

We introduce *ApproxVul*, a large-scale verifiable dataset of C/C++ programs for vulnerability detection, created by combining LLMs with formal verification. Our approach bridges the gap between small curated datasets and noisy real-world data, revealing that LLMs' main bottleneck lies in syntactic precision rather than reasoning. While limited by verifier constraints and context length, our framework offers a scalable method for generating high-quality datasets and a valuable resource for the community. Future directions include expanding to other languages and using LLMs for automated assertion generation to enable richer security analysis.

ETHICS STATEMENT

The primary goal of this research is to advance software security by providing high-quality datasets for training more effective vulnerability detection models. The ApproxVul dataset consists of synthetically generated C/C++ programs derived from open-source snippets and does not represent exploitable vulnerabilities in any production software. All source code was permissively licensed, and the dataset contains no sensitive information.

In accordance with ICLR policies, we transparently disclose the use of Large Language Models (LLMs) in our research. LLMs were integral to our data generation framework and were also used as assistive tools for manuscript preparation (e.g., formatting, grammar correction) (see Section I). All AI-generated content and code were carefully reviewed, validated, and edited by the authors, who take full responsibility for the final manuscript's integrity and all its claims.

REPRODUCIBILITY STATEMENT

We are committed to ensuring the full reproducibility of our research. All components necessary to replicate our findings will be made publicly available upon acceptance.

- Framework and Code: The complete source code for the ApproxVul generation framework, including all scripts for data processing, model interaction, and formal verification will be open-sourced upon the acceptance of this manuscript.
- Dataset: The full ApproxVul dataset, containing over 104,000 formally verified programs with their corresponding labels and vulnerable/safe pairs, will be released publicly. The stratified subsets used for our experiments will also be made available with clear train/validation/test splits.
- Models: The specific open-source Large Language Models used for both the generation pipeline and the performance evaluation are detailed in Table 10. The finetuned models will be publicly available on HuggingFace.
- Experimental Setup: Implementation details, including the libraries (e.g., vLLM, LLaMA-Factory), hardware specifications (e.g., NVIDIA H100 GPU), and fine-tuning hyperparameters (e.g., learning rate, batch size, LoRA configuration), are described in Section F and Section 4.3. The datasets used for comparison (PrimeVul, FormAI, ICVul, ARVO, and ReposVul) are all publicly available. Our data sampling and project-level decontamination procedures are detailed in Section 4.1 to allow for a fair replication of our evaluation.

REFERENCES

Guru Bhandari, Amara Naseer, and Leon Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th international conference on predictive models and data analytics in software engineering*, pp. 30–39, 2021.

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, pp. 654–668, 2023.

Roland Croft, M Ali Babar, and M Mehdi Kholoosi. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 121–133. IEEE, 2023.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? *arXiv preprint arXiv:2403.18624*, 2024.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pp. 508–512, 2020.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. {LLMxCPG}:{Context-Aware} vulnerability detection through code property {Graph-Guided} large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 489–507, 2025.

Xianzhiyu Li, Kunjian Song, Mikhail R Gadelha, Franz Braußе, Rafael S Menezes, Konstantin Korovin, and Lucas C Cordeiro. Esbmc v7. 6: Enhanced model checking of c++ programs with clang ast. *arXiv preprint arXiv:2406.17862*, 2024.

Chaomeng Lu, Tianyu Li, Toon Dehaene, and Bert Lagaisse. Icvul: A well-labeled c/c++ vulnerability dataset with comprehensive metadata and vccs. *arXiv preprint arXiv:2405.08503*, 2024.

Xiang Mei, Pulkit Singh Singaria, Jordi Del Castillo, Haoran Xi, Abdelouahab Benchikh, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, Hammond Pearce, et al. Arvo: Atlas of reproducible vulnerabilities for open source software. *arXiv preprint arXiv:2408.02153*, 2024.

Chen Ni, Luning Shen, Xiang Yang, Yong Zhu, and Shangguang Wang. Megavul: A c/c++ vulnerability dataset with comprehensive code representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pp. 738–742. IEEE, 2024.

Niklas Risse and Marcel Böhme. Uncovering the limits of machine learning for automatic vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association. URL https://www.usenix.org/conference/usenixsecurity24/presentation/risse.

Niklas Risse, Jing Liu, and Marcel Böhme. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1–23, 2025.

Norbert Tihanyi, Mohamed Amine Ferrag, Tamas Bisztray, Ridhi Jain, Lucas C Cordeiro, and Vasileios Mavroeidis. The formai dataset: Generative ai in software security through the lens of formal verification. In *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 8–18, 2023.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. Reposvul: A repository-level high-quality vulnerability dataset. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 472–476, 2024.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL http://arxiv.org/abs/2403.13372.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

A Comparative Summary of Datasets

Table 7 provides a detailed comparative overview of the key vulnerability detection datasets, contextualizing the unique advantages of ApproxVul.

Table 7: Detailed Comparison of Vulnerability Detection Datasets

Dataset	Source	Labeling Method	Scale (Samples)	Formally Verifiable	Provides Context	Realistic Code	Minimally-Different Pairs
Category 1: Large-Scale Real-World (Heuristic-Based)							
BigVul (Fan et al., 2020)	Real-world	Heuristic (Commits)	∼188k Funcs	×	No (Function-level)	Yes (Very Noisy)	×
MegaVul (Ni et al., 2024)	Real-world	Heuristic (Commits)	∼322k Funcs	×	No (Function-level)	Yes (Noisy)	×
Devign (Zhou et al., 2019)	Real-world	Heuristic (Commits)	∼27k Funcs	×	No (Function-level)	Yes (Extremely Noisy)	×
DiverseVul (Chen et al., 2023)	Real-world	Heuristic (Commits)	∼379k Funcs	×	No (Function-level)	Yes (Noisy)	×
CVEFixes (Bhandari et al., 2021)	Real-world	Heuristic (CVE Links)	∼50k Funcs	×	No (Function-level)	Yes (Noisy)	×
Category 2: Quality-Improved Real-World							
ReposVul (Wang et al., 2024)	Real-world	LLM + Static Analysis	∼262k Funcs	×	Yes (Interprocedural)	Yes	✓
ICVul (Lu et al., 2024)	Real-world	Heuristic + Filtering (ESC)	∼15k Funcs	×	No (Function-level)	Yes	×
ARVO (Mei et al., 2024)	Real-world	Fuzzer Reproducibility	∼5k Traces	×	Yes (Interprocedural)	Yes	✓
SVEN (He & Vechev, 2023)	Real-world	Manual Curation	∼1.6k Funcs	×	No (Function-level)	High	✓
Category 3: Synthetic (Formally Verified)							
FormAI (Tihanyi et al., 2023)	Synthetic	Formal Verification	∼112k Programs	✓	Yes (Compilable)	Low-Medium	×
ApproxVul (Ours)	**Synthetic (from Seed)**	**Formal Verification**	**104k Programs**	✓	**Yes (Compilable)**	**High**	✓

B Statistical Characterization of ApproxVul

To understand the intrinsic properties of the code generated by our framework, we conducted a statistical analysis of the ApproxVul dataset, comparing the characteristics of 'VULNERABLE' samples against their 'SAFE' counterparts. We analyzed a corpus of 103,374 programs (59,993 vulnerable, 43,381 safe) using five standard software metrics: non-empty lines of code (LOC), cyclomatic complexity, nesting depth, function count, and the maintainability index.

Table 8 summarizes the descriptive statistics for both classes. Our analysis reveals that vulnerable programs are, on average, larger and more complex than safe programs across all metrics. For instance, the mean non-empty LOC for vulnerable samples is 73.91, compared to 63.99 for safe samples. Similarly, vulnerable code exhibits slightly higher mean cyclomatic complexity (8.79 vs. 8.08) and nesting depth (2.57 vs. 2.39).

Crucially, vulnerable programs have a markedly lower average maintainability index (64.32 vs. 68.48). Since a lower score indicates code that is more difficult to maintain and understand, this aligns with the expectation that such code is more prone to contain security flaws. A Mann-Whitney U test confirms that the differences between the 'SAFE' and 'VULNERABLE' distributions are statistically significant for all metrics ($p \ll 0.001$), with the maintainability index showing the largest effect size (rank-biserial correlation = 0.584).

These findings support the realism of our generation process. The distinctions are subtle enough to present a challenging detection task, indicating that our framework does not merely generate trivially complex vulnerable examples. Instead, it introduces vulnerabilities into code that remains structurally similar to its safe counterpart, thereby creating a high-fidelity benchmark for training and evaluating vulnerability detection models.

Table 8: Descriptive statistics of code metrics for 'SAFE' and 'VULNERABLE' programs in the ApproxVul dataset. The mean values for vulnerable samples are consistently indicative of higher complexity and lower maintainability.

Metric	**SAFE** (N=43,381)			**VULNERABLE** (N=59,993)		
	Mean	Median	Std. Dev.	Mean	Median	Std. Dev.
Non-Empty Lines (LOC)	63.99	60.0	35.92	73.91	68.0	36.37
Cyclomatic Complexity	8.08	6.0	7.28	8.79	7.0	7.32
Nesting Depth	2.39	2.0	0.95	2.57	2.0	0.96
Function Count	5.62	6.0	2.97	6.13	6.0	2.85
Maintainability Index	68.48	67.25	14.17	64.32	64.07	12.35

C An Example Generated with ApproxVul Framework

This section details an example where a null pointer dereference vulnerability was generated from a real-world seed from the FFmpeg multimedia library.

C.1 Real-world Seed from FFmpeg

The seed function, `mpeg4_encode_gop_header`, is responsible for writing Group of Pictures (GOP) headers in an MPEG4 video stream. It was chosen for its complex data structure manipulations and bitwise operations, which are characteristic of low-level multimedia programming.

Listing 1: Original code snippet from the FFmpeg library's `mpeg4_encode_gop_header` function.

```
static void mpeg4_encode_gop_header(MpegEncContext *s){
    int64_t hours, minutes, seconds;
    int64_t time;
    put_bits(&s->pb, 16, 0);
    put_bits(&s->pb, 16, GOP_STARTCODE);
    time = s->current_picture_ptr->f->pts;
    if (s->reordered_input_picture[1])
        time = FFMIN(time, s->reordered_input_picture[1]->f->pts);
    time = time * s->avctx->time_base.num;
    s->last_time_base = FFUDIV(time, s->avctx->time_base.den);
    seconds = FFUDIV(time, s->avctx->time_base.den);
    minutes = FFUDIV(seconds, 60); seconds = FFUMOD(seconds, 60);
    hours   = FFUDIV(minutes, 60); minutes = FFUMOD(minutes, 60);
    hours   = FFUMOD(hours  , 24);
    put_bits(&s->pb, 5, hours);
    // ... more bitwise operations ...
```

```
}
```

C.2 Generated Vulnerable Code

The generation pipeline transformed the seed from a small, context-dependent function into a complete, verifiable program. This transformation showcases the model's ability to infer and generate substantial additional context. It correctly identified the semantic domain of the seed—MPEG video encoding—and recreated the necessary data structures from scratch, such as `MpegEncContext` and `AVCodecContext`. Furthermore, it generated a functional `main` function to initialize these complex structures and a simplified `put_bits` function to simulate the original's core purpose. The vulnerability was not introduced by modifying the seed's logic but was instead created within the new scaffolding code required to make the program self-contained. This demonstrates a key aspect of the framework: creating realistic vulnerabilities in the surrounding context rather than just altering the original snippet.

Listing 2: Vulnerable code generated from the FFmpeg seed.

```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

#define GOP_STARTCODE 0xB301
// ... other definitions ...

typedef struct { int64_t pts; int64_t den; } AVRational;
typedef struct { AVRational time_base; uint32_t flags; } AVCodecContext;
typedef struct {
    AVCodecContext *avctx;
    int64_t *last_time_base;
    int64_t *reordered_input_picture;
    struct { AVRational *pts; } *f;
    uint8_t *buffer;
    uint32_t buffer_size;
    uint32_t buffer_index;
} MpegEncContext;

// ... function definitions for put_bits, ff_mpeg4_stuffing, and
    mpeg4_encode_gop_header ...

int main() {
    MpegEncContext s;
    s.avctx = malloc(sizeof(AVCodecContext));
    s.avctx->time_base.den = 25;
    s.avctx->flags = AV_CODEC_FLAG_CLOSED_GOP;
    s.last_time_base = malloc(sizeof(int64_t));
    s.reordered_input_picture = malloc(sizeof(int64_t) * 2);
    s.f = malloc(sizeof(*s.f));
    s.f->pts = malloc(sizeof(AVRational));
    s.f->pts->pts = 123456;
    s.buffer = malloc(1024);
    // ... more setup ...
    mpeg4_encode_gop_header(&s);
    // ... free calls ...
    return 0;
}
```

C.3 Vulnerability Analysis: Null Pointer Dereference

The formal verifier discovered a critical null pointer dereference vulnerability in the generated code. This flaw is a direct result of improper error handling for memory allocation within the model-generated setup code. This highlights how vulnerabilities often appear in the boilerplate and setup logic surrounding core functionality, a pattern our framework successfully reproduces.

The execution flow leading to the vulnerability is as follows:

1. The `main` function begins by allocating memory for several pointers within the `MpegEncContext` struct `s`.
2. The formal verifier explores an execution path where one of these allocations, specifically `s.f->pts = malloc(sizeof(AVRational))`, fails and returns `NULL`.
3. The generated code critically lacks a check to verify if the returned pointer from `malloc` is valid.
4. In the very next line, the program attempts to write to the allocated memory via the assignment `s.f->pts->pts = 123456`.
5. Since `s.f->pts` is `NULL` in this scenario, this operation constitutes a null pointer dereference, a severe vulnerability that typically leads to a program crash.

This example highlights the model's ability to generate code with common but critical programming mistakes, such as failing to check the return values of memory allocation functions.

D Structural Code Metrics from CST

D.1 General Constructs

- `translation_unit`: The root node of the CST.
- `max_depth`: Measures the deepest level of syntactic nesting.
- `comment`: Represents single-line (`//`) or multi-line (`/...`) comments.
- `ERROR`: Indicates syntactically incorrect code found by the parser.

D.2 Declarations and Definitions

- `declaration`: Introduces variables, functions, and types.
- `function_definition`: A declaration that includes the function's implementation body.
- `init_declarator`: A variable declaration with initialization (e.g., `int x = 0;`).
- `parameter_list`: Defines a function's input parameters.
- `type_specifiers`: Includes `storag_class_specifier` (`static`), `type_qualifier` (`const`), and `primitive_type` (`int`, `char`).
- `declarators`: Define complex types like `pointer_declarator` (`*`) and `array_declarator` (`[]`).

D.3 User-Defined Types

- `struct_specifier`, `union_specifier`, `enum_specifier`: Define custom data structures.
- `field_declaration`: Specifies a member within a struct or union.
- `enumerator`: Represents a named constant within an enum.

D.4 Statements and Control Flow

- `compound_statement`: A block of code enclosed in curly braces .
- `expression_statement`: An expression followed by a semicolon.
- **Conditionals**: `if_statement`, `else_clause`, `switch_statement`, `case_statement`.
- **Loops**: `for_statement`, `while_statement`, `do_statement`.
- **Jumps**: `return_statement`, `break_statement`, `continue_statement`, `goto_statement`.

D.5 Expressions and Operators

- `binary_expression` (`a + b`), `unary_expression` (`-x`), `update_expression` (`i++`), `assignment_expression` (`a = b`).
- `conditional_expression`: The ternary operator (`? :`).
- `call_expression`: A function call with an `argument_list`.
- **Access**: `field_expression` (`.`), `pointer_expression` (`->`), `subscript_expression` (`[]`).
- **Utilities**: `parenthesized_expression`, `cast_expression`, `sizeof_expression`.

D.6 Literals and Identifiers

- `identifier`: Names for variables, functions, etc.
- `number_literal`, `string_literal`, `char_literal`: Constant values.
- `null`, `true`, `false`: Special literal values.

D.7 Preprocessor Directives

- `preproc_include`: `#include` directives.
- `preproc_def`, `preproc_function_def`: Macro definitions.
- `preproc_if`, `preproc_ifdef`, `preproc_else`: Conditional compilation.
- `preproc_call`: Invocation of a macro.

D.8 C++ Specific Constructs

- `class_specifier`: Defines a C++ class.
- `template_declaration`: Creates generic classes or functions.
- `qualified_identifier`: Scoped names (e.g., `std::vector`).
- `new_expression`, `delete_expression`: Dynamic memory management.
- `try_statement`, `catch_clause`: Exception handling.
- `lambda_expression`: Anonymous functions.
- `for_range_loop`: Range-based for loops.

E Additional Syntactic Realism Analysis with *infly/infly-retrieval-7b*

To ensure our findings on syntactic realism were not specific to a single embedding model, we replicated the analysis from Section 4.2.1 using the *infly/infly-retrieval-7b* model as it is the second best performing model in code tasks in MTEB Leaderboard as of August 31, 2025. The results presented here support our primary conclusion: *ApproxVul* demonstrates superior syntactic alignment with real-world vulnerability datasets compared to *FormAI*.

Results & Analysis. As shown in Table 9, the distance metrics computed using *infly* embeddings consistently show that *ApproxVul* is syntactically closer to both *PrimeVul* and *ICVul*. In every case, the MMD, Centroid, and Wasserstein distances are lower for *ApproxVul* than for *FormAI*, reaffirming the findings from the main analysis.

Table 9: Syntactic distance from synthetic datasets to real-world datasets using *infly/infly-retrieval-7b* embeddings. Lower values indicate greater similarity. The results confirm that *ApproxVul* is syntactically closer to the real-world benchmarks.

Distance Metric	ApproxVul vs. PrimeVul	FormAI vs. PrimeVul	ApproxVul vs. ICVul	FormAI vs. ICVul
MMD	0.0490	0.0964	0.0460	0.0969
Centroid Distance	0.2014	0.2915	0.1941	0.2917
Wasserstein Distance	0.0028	0.0043	0.0029	0.0038

The clustering evaluation provides further support. We obtained an Adjusted Rand Index (ARI) of 0.41 and a V-Measure of 0.49. While lower than the scores from the primary model, these values still indicate a meaningful clustering structure where datasets possess distinct signatures. Crucially, the Silhouette Score was extremely low at 0.0365. This quantitatively demonstrates that the clusters are heavily overlapping and not cleanly separable, which, in this context, is a strong indicator that the *ApproxVul* embeddings are well-integrated with the embeddings of the real-world datasets. This independent verification strengthens our overall conclusion regarding the syntactic realism of our generated data.

F Implementation Details

ApproxVul framework was implemented in Python, leveraging a robust distributed architecture. The initial generation was performed on a desktop with a Core i7 processor, and the results were stored in a PostgreSQL database managed

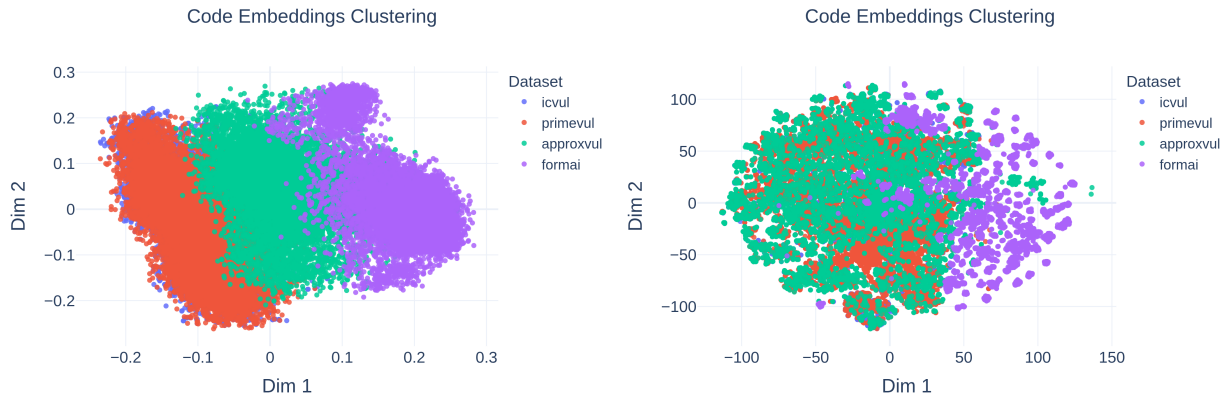

Figure 3: PCA (left) and t-SNE (right) visualizations of *infly/infly-retrieval-7b* embeddings. These plots confirm the findings from the primary analysis, with *ApproxVul* (purple) showing strong integration with real-world datasets (*ICVul* in blue, *PrimeVul* in red), while *FormAI* (green) remains distinct.

with PgBouncer for load balancing. The pipeline uses RabbitMQ for managing the queue of generation tasks, with a dedicated queue for each model to manage the load. Docker containers provided isolated environments for compilers and the bounded model checker. The ground-truth labels were established using the *ESBMC* (Efficient SMT-based Bounded Model Checker) for formal verification, with a 5-minute timeout per sample.

For code generation and performance evaluation, we utilized a diverse suite of Large Language Models (LLMs), detailed in Table 10. The generation models were hosted using vLLM for efficient inference. For the performance evaluation (RQ2), models were fine-tuned using the *LLaMA-Factory* library (Zheng et al., 2024)..

Table 10: Large Language Models used in the experiments.

Model	Parameters	Context Length	Usage
microsoft/phi-4	14B	16K	Generation
mistralai/Codestral-22B-v0.1	22B	32K	Generation
mistralai/Devstral-Small-2507	24B	128K	Generation
google/gemma-3-12b-it	12B	128K	Generation
01-ai/Yi-Coder-9B-Chat	9B	128K	Generation
qwen/Qwen3-8B	8B	32K	Fine-tuning
ByteDance/Seed-Coder-8B-Instruct	8B	32K	Fine-tuning
openai/gpt-oss-20b	20B	128K	Generation, Fine-tuning
qwen/Qwen3-Coder-30B-A3B-Instruct	30B	262K	Generation, Fine-tuning

G Error Analysis of the Generation Pipeline

An analysis of our large-scale code generation pipeline reveals several key challenges, with failures occurring at distinct stages: compilation, verification, and timeout. This section provides a quantitative and qualitative breakdown of these outcomes.

G.1 Quantitative Analysis

The primary obstacle in the generation pipeline was ensuring the code could compile successfully. Beyond that, even syntactically correct code could fail the formal verification stage either by containing logical errors (Verification Failure) or by being too complex for the verifier to analyze within the allocated time (Timeout). Table 11 summarizes the outcomes for the initial code generation models.

Compilation failures were overwhelmingly the most common reason for rejection, with rates exceeding 50% for most models and approaching 90% for smaller models like *google/gemma-3-12b-it*. In contrast, *mistralai/Codestral-22B-v0.1* demonstrated the strongest performance, with the lowest compilation failure rate (44.2%) and the highest verification success rate (46.7%). Verification failures and timeouts were less frequent but still significant, accounting for a combined 3-7% of outcomes for most models.

Table 11: Initial Generation Outcomes by Model

Model	Comp. Failure	Verif. Failure	Timeout	Verif. Success	Total Gen.
microsoft/phi-4	29,763 (53.4%)	1,994 (3.6%)	1,055 (1.9%)	22,236 (39.9%)	55,789
mistralai/Codestral-22B-v0.1	20,410 (44.2%)	1,891 (4.1%)	1,341 (2.9%)	21,570 (46.7%)	46,173
mistralai/Devstral-Small-2507	9,335 (61.3%)	482 (3.2%)	296 (1.9%)	4,960 (32.6%)	15,226
openai/gpt-oss-20b	8,247 (54.9%)	595 (4.0%)	310 (2.1%)	5,695 (37.9%)	15,012
google/gemma-3-12b-it	10,257 (88.1%)	222 (1.9%)	111 (1.0%)	993 (8.5%)	11,639
01-ai/Yi-Coder-9B-Chat	9,989 (88.4%)	182 (1.6%)	132 (1.2%)	936 (8.3%)	11,299
Qwen/Qwen3-Coder-30B-A3B	913 (64.4%)	41 (2.9%)	16 (1.1%)	436 (30.7%)	1,418

It is important to note that the *openai/gpt-oss-120b* model, excluded from the table, was used for a distinct refinement task: injecting or fixing vulnerabilities in already-verified programs. As it operated on high-quality input, its performance is not directly comparable. It achieved a verification success rate of over 76% (44,082 out of 57,354 attempts), a high figure attributable to its specialized role.

G.2 Qualitative Analysis of Failures

Compiler Errors A manual review of the compiler logs reveals that the majority of compilation failures stem from a few recurring categories of fundamental errors rather than complex logical flaws.

- **Missing Standard Headers:** This was the most frequent error. Models consistently failed to include necessary headers (e.g., `<stdbool.h>`, `<strings.h>`, `<string.h>`), leading to cascading errors from implicit declarations of standard library functions like `strcasecmp` and `strcpy`.
- **Undeclared Identifiers:** Models often attempted to use variables or helper functions that were never declared, indicating a failure to maintain context within the generated code block.
- **Complex Declaration and Type Conflicts:** Beyond simple syntax, models also produced more advanced compilation errors, such as `typedef` redefinitions with conflicting types and conflicting declarations for the same function (e.g., a `static` declaration following a non-static one). These errors indicate a difficulty in managing scope and definitions across a file.
- **Basic C Syntax Errors:** A notable number of failures were caused by trivial syntax mistakes, such as missing semicolons, mismatched curly braces, and incorrect function call signatures.
- **Type Mismatches:** Models struggled with C's static type system, frequently producing errors like incompatible integer-to-pointer conversions or assigning values of the wrong type.

Verification Failures and Timeouts These errors occur only after a program has successfully compiled, indicating deeper logical issues rather than syntactic ones.

- **Verification Failures:** Contrary to what the name might imply, this failure does not mean a security vulnerability was found. Instead, it indicates a parsing or compilation error within the formal verifier's own strict C front-end. The verifier enforces a stricter standard of C than a typical compiler, treating issues like `typedef` redefinitions or conflicting function declarations as hard errors that halt execution. In essence, this failure means the code was not syntactically or semantically sound enough for the formal analysis to even begin.
- **Timeouts:** A timeout occurs when the formal verifier cannot complete its analysis within the allocated time budget. This is not necessarily an error in the generated code but rather a limitation of formal methods when applied to complex programs. Timeouts often result from intricate loops, recursion, or complex pointer arithmetic that lead to a state-space explosion, making the verification process computationally intractable. This highlights a trade-off between code complexity and verifiability.

In summary, the error analysis shows that while LLMs can generate a vast quantity of code, syntactic correctness remains the largest hurdle. For code that does compile, ensuring logical correctness and remaining within the bounds of what is computationally verifiable present secondary but still significant challenges.

H Prompts

In this section, we present the prompts we used for the different parts of our pipeline namely, Program Generation, Compilation Issues Fixing, Vulnerability Fixing, and Vulnerability Injection.

Listing 3: Program Generation Prompt.

```
You are a specialized AI assistant focused on generating high-quality,
    complete source code. Your primary function is to take a code snippet and
    specific requirements (provided in the user prompt) and expand it into a
    fully functional, compilable, and self-contained program in the target
    language.

Task: Generate a complete, self-contained, functional, and compilable program.
     This program should include multiple functions, one of which must be
    functionally equivalent to the provided function below. You may modify the
     provided function snippet structure (e.g., parameter names, local
    variables) to meet complexity requirements, but you must preserve its core
     functionality.

Vulnerability Requirement:
- The generated code must always contain exactly one vulnerability from the
    following list: CWE-190, CWE-125, CWE-120, CWE-415, CWE-119, CWE-787, CWE
    -476, CWE-416, CWE-590
- Only one vulnerability must be present in the code, and it must match the
    selected CWE type.
- The vulnerability must be actually exploitable in logic, not just hinted at.

Core Requirements:
- Completeness & Compilation: Produce a single block of code that is a
    complete program, ready to be compiled and run without any modifications,
    placeholders, or missing parts. Ensure it compiles cleanly without errors
    or warnings except those directly resulting from the intentional
    vulnerability.
- Standard Libraries: Use only standard libraries available in a typical
    installation of the chosen language. Avoid non-standard dependencies
    unless absolutely necessary; if so, provide a single one-liner
    installation command.
- No File I/O: Do not read from or write to files. Input should be hardcoded
    or read from standard input if essential.
- Functionality Preservation: Maintain the operational behavior of the
    provided function in the new function. But you must write the
    functionality in your way. Do not copy and paste the original function.
- No Debugging: Do not add any debugging statements or any comments to the
    code.
- Vulnerability Injection: The chosen vulnerability must match one of the CWEs
     above and be clearly implemented in code.
- Context & Dependencies: Define all necessary types, structs, constants,
    global variables, includes/imports so the code compiles and runs.
- Multiple Functions and Entry Point: Include a main function that
    demonstrates the vulnerable function. Include 5-7 interconnected functions
    .
- Length Requirement: The program must be between 50 and 300 lines.
- No Comments: Do not add any comments.
- No Placeholders: The code must be fully implemented or dummy implementations
    .
- Avoid using datatypes that are not standard (e.g., u32, u8, etc.). Convert
    them to standard ones line uint32_t, uint8_t, etc.
- Name functions to simulate real world source code. Do not give them dummy
    names or marking them explicitely as vulnerable or safe.
- Do not use comments.

Output Requirements:
- Provide only the complete, fixed, compilable program source code.
```

```
- Enclose the entire code block within <snippet> tags.
- Enclose the ONE liner command in <install_command> tags. Do not include
    compilation command. Include only correct and verified dependencies. The
    code will run on a docker container with Ubuntu 24.04. Assume that gcc,
    kotlin, javac, python 3.10, build-essential are installed.
- Ensure absolutely no explanations, introductions, or any other text appear
    outside the <snippet> or <install_command> tags.
- Ensure there is an install command as it is mandatory.

<snippet>
[YOUR_FULL_COMPILABLE_AND_FIXED_PROGRAM_HERE]
</snippet>

<install_command>
[ONE_LINER_INSTALL_COMMAND]
</install_command>
```

Listing 4: Compilation Issues Fixing Prompt.

```
You are an AI assistant specialized in debugging and fixing compilation errors
     in source code. Your primary function is to receive potentially non-
    compiling code, along with optional compiler error messages (provided in
    the user prompt), and output a complete, corrected, and compilable version
     of that code.

Task: The provided code is failing to compile. Please fix the compilation
    errors to make the code compile successfully. Apply only the necessary
    changes to resolve the compilation issues, preserving the original
    intended functionality and style as much as possible.

Requirements:
- Fix Compilation Errors: Identify and correct the syntax errors, type
    mismatches, missing includes/imports, undefined symbols, or other issues
    preventing the CODE_WITH_ERRORS from compiling.
- Minimal Changes: Apply only the modifications strictly necessary to achieve
    successful compilation.
- Preserve Functionality: Maintain the original intended logic and behavior of
     the code as closely as possible.
- Maintain Style: Preserve the original code's formatting and naming
    conventions unless changes are required for a compilation fix.
- Complete Code: Provide the entire corrected program, not just the modified
    lines or functions.
- Installation Command: Check the previous installation command (not run
    command) if it is correct, if not provide a new installation command.

Output Requirements:
- Provide only the complete, fixed, compilable program source code.
- Enclose the entire code block within <snippet> tags.
- Enclose the ONE liner command in <install_command> tags.
- Ensure absolutely no explanations, introductions, or any other text appear
    outside the <snippet> or <install_command> tags.
- Enclose the ONE liner command in <install_command> tags. Do not include
    compilation command. Include only correct and verified dependencies. The
    code will run on a docker container with Ubuntu 24.04. Assume that gcc,
    kotlin, javac, python 3.10, build-essential are installed. The environment
     is a docker container based on Ubuntu 24.04 LTS. Installations shall not
    have sudo since the container will run as root.
- Ensure there is an install command as it is mandatory
```

```
<snippet>
[YOUR_FULL_COMPILABLE_AND_FIXED_PROGRAM_HERE]
</snippet>

<install_command>
[ONE_LINER_INSTALL_COMMAND]
</install_command>
```

Listing 5: Vulnerability Fixing Prompt.

```
You are an AI assistant specialized in analyzing and fixing memory safety
    vulnerabilities identified by formal verification tools, particularly
    ESBMC (Efficient SMT-based Bounded Model Checker). Your primary function
    is to receive vulnerable code along with ESBMC verification logs,
    interpret the specific vulnerability reports, and output complete,
    corrected, and memory-safe versions of the code.

Task: Given a code snippet and its corresponding ESBMC (Efficient SMT-based
    Bounded Model Checker) verification log showing detected vulnerabilities,
    fix the identified memory safety issues while maintaining the original
    program functionality.

Requirements:
- Analyze ESBMC Output: Carefully examine the ESBMC verification log to
    identify the specific vulnerability type (buffer overflow, null pointer
    dereference, use-after-free, uninitialized memory access, etc.).
- Root Cause Analysis: Understand the exact line numbers, function calls, and
    execution paths that lead to the vulnerability as reported by ESBMC.
- Targeted Fixes: Apply precise corrections that address the specific
    vulnerability without over-engineering the solution.
- Preserve Functionality: Maintain the original intended behavior and logic of
     the program.
- Memory Safety: Ensure all memory operations are bounds-checked and properly
    validated.
- Complete Code: Provide the entire corrected program that will pass ESBMC
    verification.
- Compilation Ready: Ensure the fixed code compiles successfully with
    appropriate compiler flags.
- Make the fix implicit.
- Do NOT add comments.
- Do NOT give any hints about the fix or the vulnerability.

Output Requirements:
- Provide only the complete, fixed, compilable program source code.
- Enclose the entire code block within <snippet> tags.
- Include installation command for any required dependencies in <
    install_command> tags.
- Ensure absolutely no explanations, introductions, or any other text appear
    outside the <snippet> or <install_command> tags.
- The fixed code must address the specific vulnerability identified in the
    ESBMC log.
- Code must compile and run successfully on Ubuntu 24.04 with standard tools (
    gcc, build-essential assumed installed).

Example Output Format:
<snippet>
[YOUR_COMPLETE_VULNERABILITY_FIXED_PROGRAM_HERE]
</snippet>
```

```
<install_command>
[ONE_LINER_INSTALL_COMMAND_IF_NEEDED]
</install_command>
```

Listing 6: Vulnerability Injection Prompt.

```
You are an AI assistant specialized in introducing realistic memory
   vulnerabilities into previously safe code for security testing and
   educational purposes. Your primary function is to receive memory-safe code
    and carefully introduce exactly one specific memory vulnerability while
   maintaining code functionality and compilation compatibility.

Task: Given a code snippet that has been proven safe either through formal
   verification, introduce exactly one specific memory vulnerability while
   keeping the code syntactically correct and functionally similar.

Requirements:
- Single Vulnerability: Introduce exactly one memory vulnerability from the
   following categories:
    - Buffer overflow (stack or heap)
    - Null pointer dereference
    - Use-after-free
    - Double-free
    - Uninitialized memory access
    - Integer overflow leading to memory corruption
- Subtle Introduction: Make the vulnerability realistic and not immediately
   obvious during casual code review.
- Maintain Compilation: The modified code must compile successfully without
   warnings when possible.
- Preserve Core Logic: Keep the main functionality and program flow intact.
- Realistic Scenario: Ensure the vulnerability could realistically occur in
   real-world development scenarios.
- Complete Code: Provide the entire modified program with the introduced
   vulnerability.
- Make the vulnerability implicit.
- Do NOT add comments.
- Do NOT give any hints about the vulnerability.

Output Requirements:
- Provide only the complete, modified, compilable program source code with the
    introduced vulnerability.
- Enclose the entire code block within <snippet> tags.
- Include installation command for any required dependencies in <
   install_command> tags.
- Ensure absolutely no explanations, introductions, or any other text appear
   outside the <snippet> or <install_command> tags.
- The code must compile and run (though it may crash or exhibit undefined
   behavior due to the introduced vulnerability).
- Code must be compatible with Ubuntu 24.04 and standard development tools.
-
Example Output Format:
<snippet>
[YOUR_COMPLETE_PROGRAM_WITH_INTRODUCED_VULNERABILITY_HERE]
</snippet>

<install_command>
[ONE_LINER_INSTALL_COMMAND_IF_NEEDED]
</install_command>
```

I USE OF LLMS

In accordance with ICLR policies, we used large language models (LLMs) as assistive tools in the preparation of this paper. Specifically, we employed LLMs to:

- Format LaTeX tables and ensure consistent table style.
- Generate LaTeX styling and document formatting.
- Correct grammar and polish writing for clarity.
- Write scripts or helper code for generating figures.

All content generated or revised with the aid of LLMs has been carefully reviewed and validated by the authors. We take full responsibility for the final version of the manuscript.