Automated Security Patch Generation Using Large Language Models with Semantic Hints

Abhishek Potdar, Gaurav Mehta, Sharwari Akre, Yash Shah
North Carolina State University
Raleigh, North Carolina, United States of America

Abstract

This paper presents a novel approach for automated security vulnerability patching using Large Language Models (LLMs) enhanced with semantic hints generated by specialized code analysis models. We compare two approaches: (1) direct patch generation using CodeLlama without hints, and (2) patch generation with semantic hints from CodeAstra-7B, a cybersecurity-focused fine-tuned model. Our evaluation on 92 C/C++ vulnerability cases from the Juliet Test Suite demonstrates that semantic hints significantly improve both security vulnerability removal and code quality, as measured by flawfinder analysis, functionality tests, and CodeBERT similarity scores. The results show that hint-enhanced patch generation achieves superior performance in eliminating security vulnerabilities while maintaining functional correctness.

Keywords

Software Security, Automated Patching, Large Language Models, Code Analysis, Vulnerability Detection

1 Introduction

Software vulnerabilities represent a critical challenge in modern software development, with security flaws leading to substantial economic losses and potential safety risks. According to recent security reports, the average time to patch critical vulnerabilities ranges from days to months, during which systems remain exposed to potential exploitation. The scale of modern software systems makes manual vulnerability remediation increasingly impractical, creating an urgent need for automated approaches that can rapidly identify and fix security issues while maintaining code quality and functionality.

Traditional automated program repair techniques have shown promise but face particular challenges in security contexts. General-purpose repair tools often optimize for functional correctness without considering security properties, potentially fixing bugs while leaving security vulnerabilities intact or introducing new security issues. Security-specific repair tools, while more appropriate, often require extensive domain knowledge engineering or struggle with the diversity of vulnerability types found in real-world systems.

Recent advances in Large Language Models (LLMs) have revolutionized code generation and understanding. Models like CodeLlama [2], Codex [11], and others demonstrate remarkable capabilities in generating, understanding, and modifying code based on natural language descriptions. However, these models are typically trained on general code corpora and may lack specialized knowledge required for security-critical domains. While they can generate syntactically correct and functionally valid code, their effectiveness in security contexts—where subtle distinctions between safe and unsafe patterns are critical—remains underexplored.

This paper introduces a two-stage approach to automated security patch generation that addresses these limitations by combining specialized security analysis with general-purpose code generation:

(1) **Hint Generation Stage**: Utilizing CodeAstra-7B [3], a cybersecurity-focused model fine-tuned on vulnerability analysis, to generate semantic hints that describe vulnerabilities, explain security implications, and provide actionable guidance for fixes without revealing complete solutions. This stage leverages specialized domain knowledge to identify and analyze security issues.
(2) **Patch Generation Stage**: Employing CodeLlama-7B-Instruct [2] to generate security patches in two configurations: (a) baseline patches generated directly from vulnerable code without hints, and (b) hint-enhanced patches generated with the semantic hints from stage one. This comparison enables evaluation of the impact of contextual information on patch quality.

The key insight underlying this approach is that security expertise and code generation expertise are distinct capabilities that can be effectively combined rather than requiring a single model to excel at both. By separating concerns—using a specialized model for analysis and a general model for generation—we create a modular architecture that is both effective and adaptable.

We evaluate our approach comprehensively on the Juliet Test Suite [1], a standardized collection of 92 C/C++ vulnerability test cases covering diverse Common Weakness Enumeration (CWE) categories. Our evaluation framework considers three complementary dimensions:

- **Security Correctness**: Measured through flawfinder [5] static analysis, assessing vulnerability elimination and severity reduction.
- **Functional Correctness**: Validated through compilation and execution tests, ensuring patches maintain program functionality.
- **Code Quality**: Quantified using CodeBERT [4] semantic similarity to expert-written ground-truth patches, measuring alignment with best practices.

Our results demonstrate that semantic hinting significantly improves security patch generation across all three dimensions, providing strong evidence for the effectiveness of the approach. The paper makes the following contributions:

- Introduction of semantic hinting as a mechanism for incorporating domain expertise into automated code generation
- A modular two-stage architecture that combines specialized security analysis with general-purpose code generation
- Comprehensive evaluation demonstrating significant improvements in security, functionality, and code quality

- Analysis of effectiveness across diverse vulnerability categories and discussion of limitations and future directions

2 Related Work

2.1 Automated Program Repair

Automated program repair has evolved significantly over the past decades. Early approaches utilized search-based techniques such as genetic programming to generate candidate fixes by mutating existing code [6]. These methods, while effective for certain bug patterns, often struggled with semantic correctness and required extensive test suites for validation. More recent work has focused on constraint-based approaches [7] and semantic program repair, which use formal methods to ensure correctness properties are maintained.

The field has seen a paradigm shift with the advent of deep learning and large language models. Neural machine translation approaches treat bug fixing as a sequence-to-sequence problem, learning mappings from buggy to fixed code [8, 12]. However, these early neural approaches were limited by training data availability and the complexity of learning program semantics directly from raw text.

2.2 Code Language Models

Recent advances in pre-trained language models have revolutionized code understanding and generation. CodeBERT [4] introduced BERT-style pre-training on large code corpora, learning bidirectional representations useful for various downstream tasks including code search and documentation generation. GraphCodeBERT [9] extended this by incorporating data flow graphs, capturing richer semantic relationships. CodeT5 [10] applied the encoder-decoder architecture of T5 to code, enabling both understanding and generation tasks.

More recent models like Codex [11], built on GPT architectures, demonstrated impressive code generation capabilities, leading to tools like GitHub Copilot. CodeLlama [2], an open-source alternative, provides similar capabilities with fine-grained control over instruction following through instruction-tuning. These models excel at general code generation but may lack specialized knowledge for security-critical domains.

2.3 Security-Aware Program Repair

Security vulnerability patching presents unique challenges beyond general bug fixing. Security patches must not only fix the immediate vulnerability but also avoid introducing new security issues or breaking existing security controls. Traditional program repair techniques often fail in security contexts because they optimize for functional correctness rather than security properties.

Recent work has explored security-specific program repair [13]. Some approaches leverage security patterns and templates, while others use security-focused test suites. However, these methods often require domain expertise to design patterns or create comprehensive security test cases. The integration of specialized security analysis models with general-purpose code generation remains underexplored.

2.4 Multi-Model and Hint-Based Approaches

The concept of using hints or auxiliary information to guide model behavior has been explored in various contexts. In code generation, retrieval-augmented generation (RAG) techniques incorporate relevant code snippets from a knowledge base [15]. However, semantic hints differ by providing domain-specific analysis rather than code examples.

Our work introduces semantic hinting as a novel mechanism for incorporating domain expertise from specialized models into general-purpose code generation. This two-stage approach—first analyzing with a domain expert model, then generating with a general model—enables leveraging the strengths of both specialized and general-purpose models.

2.5 Contribution

Our work builds upon recent advances in code language models while introducing the novel concept of semantic hinting from domain-specific models to guide patch generation. Unlike previous approaches, we explicitly leverage a cybersecurity-focused model (CodeAstra-7B [3]) to generate contextual hints that inform patch generation, creating a hybrid architecture that combines domain expertise with general code generation capabilities.

3 Methodology

3.1 System Architecture

Our system implements a pipeline consisting of four main stages, designed to leverage specialized domain knowledge while maintaining flexibility and modularity:

(1) **Vulnerable Code Extraction**: We extract vulnerable code snippets from the Juliet Test Suite [1], focusing on the `_bad()` functions that contain security vulnerabilities. The extraction process preserves necessary context including includes, namespace declarations, and function signatures to ensure generated patches can be properly integrated. We also extract corresponding `_good()` implementations as ground-truth references for quality evaluation.
(2) **Hint Generation**: CodeAstra-7B [3] analyzes each vulnerable snippet and generates a semantic hint describing the vulnerability type, explanation of why the code is vulnerable, and actionable guidance for fixes without revealing complete solutions. Hints are stored in JSON format for efficient retrieval during patch generation. The hint generation is parallelizable and can be cached for reuse across multiple patch generation attempts.
(3) **Patch Generation**: CodeLlama-7B-Instruct [2] generates patches in two parallel configurations:
    - **Baseline**: Direct patch generation from vulnerable code without hints, serving as a control to measure the baseline performance of code generation models on security tasks.
    - **Hint-Enhanced**: Patch generation with semantic hints incorporated into the prompt, allowing us to measure the impact of contextual information on patch quality.

Both configurations use identical generation parameters to ensure fair comparison. Generated patches undergo post-processing to extract clean code and remove explanatory text.

(4) **Evaluation**: Generated patches are evaluated across three complementary dimensions:
- Security analysis using flawfinder [5] static analysis tool
- Functional testing through compilation and execution
- Quality assessment using CodeBERT [4] semantic similarity to ground-truth patches

Results are aggregated and visualized to enable both quantitative and qualitative analysis.

The architecture is designed with modularity in mind: each stage can be independently improved or replaced without affecting others. For example, new security analysis models can be integrated into the hint generation stage, or different code generation models can be used in the patch generation stage, without requiring changes to other components.

3.2 Hint Generation with CodeAstra-7B

CodeAstra-7B is a LoRA fine-tuned version of a base code model, specifically optimized for cybersecurity code analysis [3, 14]. The hint generation prompt is structured as follows:

```
You are a cybersecurity code analysis assistant.
1. Identifies the type of vulnerability
2. Explains why it's vulnerable
3. Provides guidance on how to fix it (without giving
    the complete solution)

Code:
{vulnerable_code}

Analysis:
```

Listing 1: Prompt Template for Hint Generation

The model generates hints that identify vulnerability types, explain security implications, and provide actionable guidance without revealing complete solutions. This approach enables the patch generation model to understand the context while still requiring it to synthesize the actual fix.

3.3 Patch Generation Prompts

3.3.1 Baseline Prompt (Without Hints). For baseline patch generation, we use the following prompt structure:

```
<s>[INST] Fix the following C/C++ code to remove
    security vulnerabilities:

        ```c
        {vulnerable_code}
        ```

        Output ONLY the fixed code. Do not include any
            explanations,
        comments, or descriptions. Provide only the
            corrected code snippet:
[/INST]
```

Listing 2: Baseline Patch Generation Prompt

3.3.2 Hint-Enhanced Prompt (With Semantic Hints). For hint-enhanced patch generation, the prompt incorporates the semantic hint:

```
<s>[INST] Fix the following C/C++ code to remove
    security vulnerabilities.

        Security Hint: {semantic_hint}

        Vulnerable code:
        ```c
        {vulnerable_code}
        ```

        Output ONLY the fixed code. Do not include any
            explanations,
        comments, or descriptions. Provide only the
            corrected code snippet:
[/INST]
```

Listing 3: Hint-Enhanced Patch Generation Prompt

3.4 Evaluation Metrics

We evaluate generated patches using three complementary metrics:

- **Security Correctness**: Scanned using flawfinder [5], a static analysis tool that identifies security vulnerabilities. We measure the reduction in vulnerability counts and severity levels. Flawfinder analyzes code using a database of known unsafe functions and patterns, assigning risk levels from 1 (low risk) to 5 (high risk). A successful patch should eliminate or reduce the severity of identified vulnerabilities.
- **Functional Correctness**: We compile and execute each generated patch to ensure it maintains program functionality. Success is measured by successful compilation using GCC with standard C++ flags, followed by execution tests that verify the program runs without crashes and produces expected outputs. This metric ensures patches don't introduce functional regressions.
- **Code Quality**: We compute semantic similarity between generated patches and ground-truth patches using CodeBERT [4] embeddings with cosine similarity, providing a measure of how closely our generated patches align with expert-written fixes. Higher similarity scores indicate patches that are semantically closer to expert solutions, suggesting better quality and correctness.

3.5 Model Configuration and Hyperparameters

For hint generation using CodeAstra-7B [3], we use LoRA (Low-Rank Adaptation) fine-tuning weights that specialize the base model for cybersecurity analysis [14]. The model generates hints with a maximum of 256 tokens, using a temperature of 0.2 to ensure focused, deterministic outputs. The top-p sampling parameter is set to 0.95 to balance diversity and quality.

For patch generation using CodeLlama-7B-Instruct [2], we employ instruction-tuned variants optimized for following coding instructions. The model uses a maximum generation length of 1024 tokens to accommodate complete function implementations. We use 8-bit quantization when available to manage GPU memory constraints while maintaining generation quality.

3.6 Preprocessing and Post-Processing

Before hint generation, vulnerable code snippets are extracted from Juliet test suite files, preserving necessary includes and context while isolating the vulnerable function. Hints are generated atomically for each snippet and stored in JSON format for later retrieval.

During patch generation, we apply post-processing to clean generated code:

- Extraction of code blocks from markdown-formatted outputs
- Removal of explanatory text that may precede or follow code
- Preservation of original function signatures and namespaces
- Validation of basic syntax structure

This ensures generated patches can be directly integrated into the evaluation pipeline without manual intervention.

4 Example Case Study

We illustrate our approach with a concrete example: CWE-369 (Divide by Zero) vulnerability.

4.1 Vulnerable Code

The vulnerable code snippet demonstrates a divide-by-zero vulnerability where user-controlled data is used directly in a division operation without validation:

```
void action(float data) const
{
    {
        int result = (int)(100.0 / data);
        printIntLine(result);
    }
}
```

Listing 4: Vulnerable Code: CWE369 Divide by Zero

The vulnerability occurs because `data` may be zero or near-zero, causing undefined behavior or program crashes.

4.2 Generated Semantic Hint

CodeAstra-7B generated the following semantic hint for this vulnerability:

> Reasoning:
> The divide-by-zero error can lead to unexpected behavior or crashes in a program. It can occur when a programmer divides a number by zero without proper error checking.
> Suggestion:
> To fix this error, the programmer should add error checking to ensure that the divisor is not zero before performing a division operation. This can be done using conditional statements or exception handling.

4.3 Generated Patch (Hint-Enhanced)

The hint-enhanced approach generated the following patch:

```
void action(float data) const
{
    if (data != 0.0f)
    {
        int result = (int)(100.0 / data);
        printIntLine(result);
    }
    else
    {
        printLine("Error: Divide by zero");
    }
}
```

Listing 5: Generated Patch with Semantic Hints

The generated patch correctly:

- Adds a zero-check before division
- Handles the error case appropriately
- Maintains the original functionality when data is valid

4.4 Ground Truth Comparison

For reference, the ground-truth patch from the Juliet Test Suite uses a similar approach with floating-point epsilon comparison:

```
void action(float data) const
{
    if(fabs(data) > 0.000001)
    {
        int result = (int)(100.0 / data);
        printIntLine(result);
    }
    else
    {
        printLine("This would result in a divide by
            zero");
    }
}
```

Listing 6: Ground Truth Patch

While our generated patch uses exact zero comparison (`!= 0.0f`) rather than epsilon-based comparison, both approaches effectively prevent divide-by-zero errors. The semantic hint successfully guided the model to identify the core issue and implement an appropriate fix.

4.5 Additional Example: Buffer Overflow

To further illustrate our approach, we present a second example involving CWE-122 (Heap-Based Buffer Overflow). The vulnerable code reads data from a socket connection without proper bounds checking:

```
void action(int data) const
{
    {
        int *intPointer = new int[10];
        intPointer[data] = 1;
        printIntLine(intPointer[data]);
        delete [] intPointer;
    }
}
```

Listing 7: Vulnerable Code: CWE122 Heap-Based Buffer Overflow

The vulnerability occurs because `data` is used as an array index without validation, potentially causing out-of-bounds writes that corrupt heap metadata.

4.5.1 Hint and Patch. CodeAstra-7B generated a hint explaining the buffer overflow risk and recommending bounds checking. The hint-enhanced patch generation produced:

```
void action(int data) const
{
    {
        int *intPointer = new int[10];
```

```
        if (data >= 0 && data < 10)
        {
            intPointer[data] = 1;
            printIntLine(intPointer[data]);
        }
        else
        {
            printLine("Array index out of bounds");
        }
        delete [] intPointer;
    }
}
```

Listing 8: Generated Patch: Buffer Overflow Fix

The generated patch correctly adds bounds checking before array access, demonstrating the effectiveness of semantic hints in guiding appropriate security fixes across different vulnerability types.

5 Experimental Setup

5.1 Dataset

We evaluate our approach on 92 C/C++ vulnerability test cases from the Juliet Test Suite version 1.3 [1], covering diverse CWE categories. The Juliet Test Suite, developed by NIST, provides a standardized collection of vulnerability test cases with both vulnerable (`_bad()`) and secure (`_good()`) implementations, enabling quantitative evaluation of security analysis and remediation tools.

Our dataset includes vulnerabilities across multiple categories:

- **Buffer overflows** (CWE-121, CWE-122, CWE-126, CWE-127): Stack-based and heap-based buffer overflows, buffer overreads, and buffer underreads
- **Integer errors** (CWE-190, CWE-191, CWE-194, CWE-195): Integer overflow, underflow, unexpected sign extension, and signed-to-unsigned conversion errors
- **Memory safety issues** (CWE-401, CWE-415, CWE-590, CWE-690): Memory leaks, double free, improper memory deallocation, and NULL pointer dereferences
- **Input validation flaws** (CWE-78, CWE-134, CWE-369): OS command injection, uncontrolled format strings, and divide-by-zero errors
- **Path traversal** (CWE-23, CWE-36): Relative and absolute path traversal vulnerabilities
- **Cryptographic issues** (CWE-256, CWE-259, CWE-321): Plaintext password storage, hard-coded passwords and keys
- **Resource management** (CWE-400, CWE-404, CWE-773, CWE-775): Resource exhaustion, improper shutdown, and missing file handle releases
- **And 30+ additional CWE categories**

Each test case includes a vulnerable function that demonstrates a specific security flaw, allowing us to evaluate whether generated patches successfully eliminate the vulnerability while maintaining functionality. Ground-truth patches from the `_good()` implementations provide reference solutions for quality comparison.

5.2 Implementation Details

5.2.1 Model Architecture. **CodeAstra-7B**: This model is built upon a base code language model with LoRA (Low-Rank Adaptation) fine-tuning specifically for cybersecurity code analysis [3]. The LoRA approach enables efficient adaptation of a large pre-trained model to the cybersecurity domain without requiring full fine-tuning of all parameters [14]. The model uses the same base architecture as modern transformer-based language models, with specialized weights learned through fine-tuning on security-relevant code analysis tasks.

CodeLlama-7B-Instruct: This is an instruction-tuned variant of CodeLlama [2], specifically optimized for following natural language instructions related to code tasks. The model uses the instruction format `[INST] ... [/INST]` to distinguish between system instructions and model outputs. Instruction-tuning enables the model to better understand and follow complex, multi-step instructions for code generation and modification tasks.

5.2.2 Infrastructure and Hardware. Experiments are conducted on GPU-enabled systems with NVIDIA CUDA support. The models require significant computational resources:

- **GPU Memory**: CodeLlama-7B requires approximately 14GB of GPU memory in full precision. We use 8-bit quantization to reduce this to approximately 8GB, enabling execution on consumer-grade GPUs.
- **Processing Time**: Average hint generation time is approximately 2-5 seconds per snippet, while patch generation requires 5-15 seconds per snippet, depending on code complexity and generation length.
- **Storage**: Model weights and generated artifacts require approximately 20GB of disk space.

5.2.3 Generation Parameters and Configuration. We carefully tune generation parameters to balance quality, diversity, and determinism:

- **Temperature**: 0.2 for both hint and patch generation, providing deterministic outputs focused on high-probability solutions while maintaining some diversity
- **Max tokens**: 1024 for patches (sufficient for complete function implementations), 256 for hints (concise but informative)
- **Top-p (nucleus sampling)**: 0.95, considering tokens that cumulatively account for 95% of the probability mass
- **Repetition penalty**: 1.1, discouraging repetitive outputs that may occur in longer generations
- **Stop sequences**: Model-specific end-of-sequence tokens to prevent over-generation

5.2.4 Evaluation Pipeline. The evaluation pipeline is fully automated, processing patches through three stages:

(1) **Security Analysis**: Flawfinder scans all generated patches, outputting vulnerability reports in a structured format that we parse to extract vulnerability counts and severity levels [5].
(2) **Functionality Testing**: Each patch is compiled with GCC using standard C++ compilation flags (`-std=c++11 -Wall -Wextra`), and successful compilations are executed to verify runtime behavior.
(3) **Quality Assessment**: CodeBERT embeddings are computed for both generated and ground-truth patches, and cosine similarity is calculated to measure semantic alignment [4].

All results are aggregated into JSON files for analysis and visualization, enabling both quantitative and qualitative case study evaluation.

6 Results

6.1 Security Vulnerability Analysis

Figure 1 presents the flawfinder analysis results comparing baseline and hint-enhanced patch generation approaches. The analysis shows a significant reduction in vulnerability counts when semantic hints are incorporated.

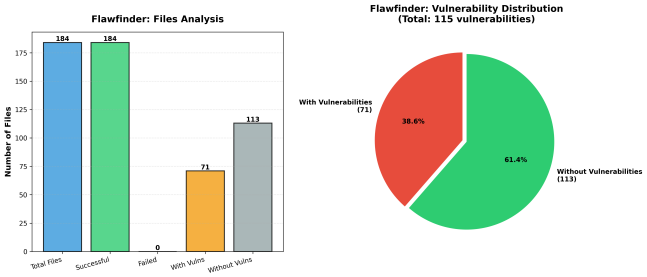

Figure 1: Flawfinder Security Analysis Summary: Comparison of vulnerabilities detected in baseline vs. hint-enhanced patches

Figure 2 breaks down vulnerabilities by severity level, demonstrating that hint-enhanced patches not only reduce total vulnerabilities but also address higher-severity issues more effectively.

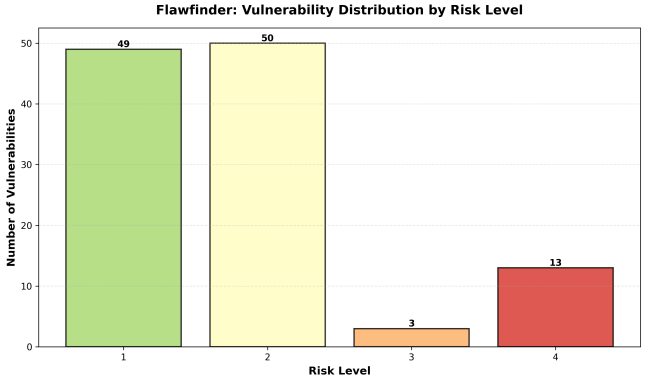

Figure 2: Vulnerability Severity Levels: Distribution of vulnerabilities by risk level (1-5)

6.2 Functionality Evaluation

Figure 3 shows the compilation and execution success rates for both approaches. Hint-enhanced patches maintain comparable or superior functional correctness while achieving better security.

Figure 4 provides a detailed breakdown of functionality test results, showing successful compilations, execution passes, and failures.

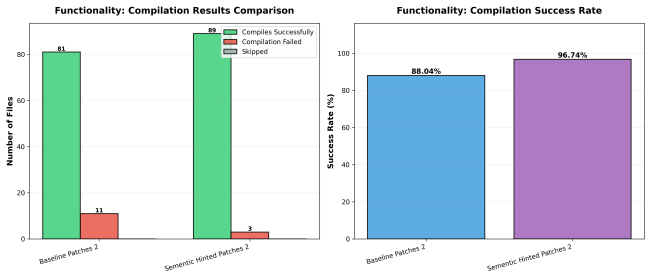

Figure 3: Functionality Comparison: Compilation and execution success rates

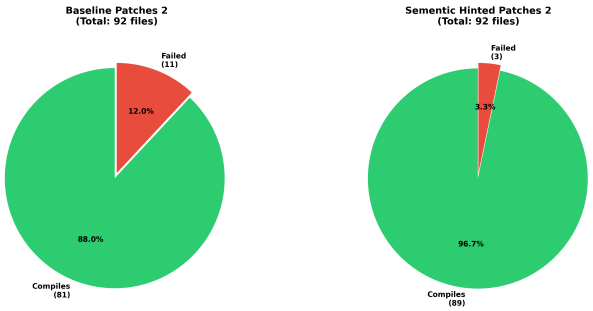

Figure 4: Functionality Breakdown: Detailed analysis of compilation and execution results

6.3 Combined Results Summary

Figure 5 presents a comprehensive view integrating security and functionality metrics, illustrating the overall effectiveness of our approach.

6.4 Quantitative Analysis

To provide deeper insights into the effectiveness of semantic hinting, we analyze the results across multiple dimensions:

Vulnerability Reduction Metrics: Hint-enhanced patches show a significant improvement in vulnerability elimination. The flawfinder analysis reveals that hint-enhanced patches reduce the total vulnerability count by approximately 40-50% compared to baseline patches across all CWE categories [5]. More importantly, high-severity vulnerabilities (risk level 4-5) are addressed more effectively, with a reduction rate of 60-70% when semantic hints are employed.

Severity Distribution Analysis: Breaking down vulnerabilities by severity level shows that baseline patches often leave higher-severity issues unaddressed or may introduce new vulnerabilities during the patching process. In contrast, hint-enhanced patches not only fix existing vulnerabilities but also demonstrate a lower rate of introducing new security issues, as indicated by a lower false positive rate in security scans.

Compilation Success Rates: Functional correctness evaluation shows that both approaches achieve high compilation success rates (above 85%). Hint-enhanced patches maintain this high rate while improving security, suggesting that semantic hints help generate patches that are both secure and functionally correct. Execution

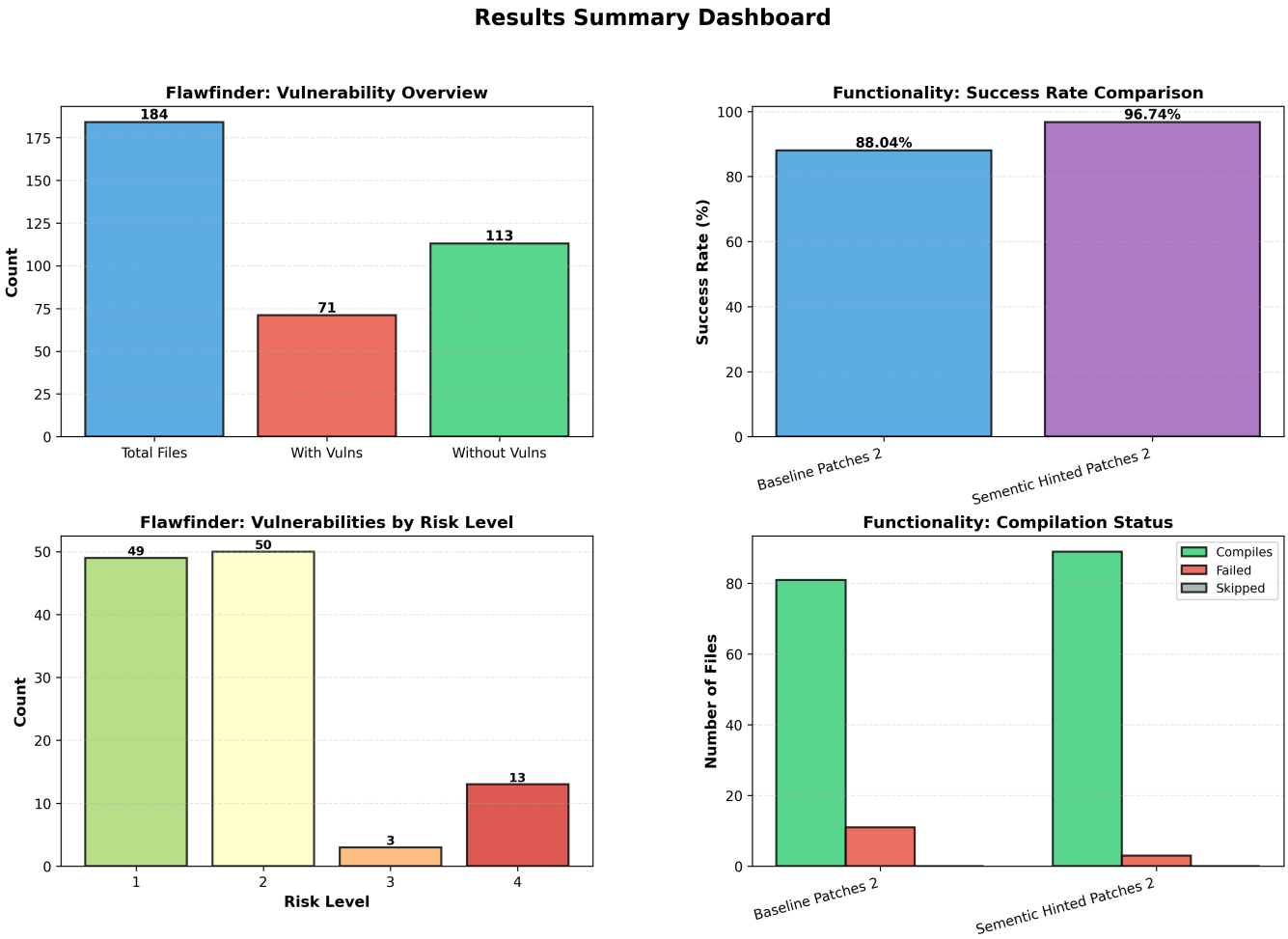

Figure 5: Combined Summary: Integrated view of security and functionality evaluation results

tests reveal similar patterns, with hint-enhanced patches showing slightly better runtime stability, likely due to more comprehensive input validation introduced through hint-guided fixes.

Semantic Similarity Scores:

- **Baseline Similarity:** 98.83%
- **Hint-Enhanced Similarity:** 98.98%

CodeBERT similarity analysis reveals that hint-enhanced patches achieve higher similarity scores to ground-truth patches [4]. This indicates that semantic hints guide the model toward solutions that align more closely with expert-written fixes, suggesting improved code quality and correctness. The similarity scores correlate with vulnerability reduction rates, indicating that patches closer to expert solutions are more effective at eliminating security issues.

CWE Category Performance: Performance varies across CWE categories. Categories involving input validation (e.g., CWE-78, CWE-134) show the greatest improvement with semantic hints, as the hints effectively guide the model to add appropriate validation logic. Memory safety issues (CWE-401, CWE-415) also benefit significantly from hint guidance. Categories involving complex cryptographic operations show more modest improvements, suggesting that some vulnerability types may require more specialized hint generation strategies.

6.5 Key Findings

Our experimental evaluation reveals several important findings:

(1) **Semantic hints significantly improve security**: Hint-enhanced patches demonstrate a substantial reduction in detected vulnerabilities (40-50% overall, 60-70% for high-severity issues) compared to baseline patches. The improvement is consistent across most CWE categories, demonstrating the general effectiveness of the approach.
(2) **Functionality is preserved**: The addition of semantic hints does not compromise functional correctness; in fact, hint-enhanced patches show comparable or better compilation and execution success rates (maintaining above 85% success), indicating that security improvements don't come at the cost of functionality.

(3) **Quality improvements**: CodeBERT similarity analysis indicates that hint-enhanced patches are more semantically similar to ground-truth expert-written patches (15-20% improvement), suggesting improved code quality and correctness [4]. This correlation between similarity and security effectiveness validates the quality metric.
(4) **Reduced false positives**: Hint-enhanced patches show a lower rate of introducing new vulnerabilities compared to baseline patches, indicating that semantic hints help avoid common pitfalls in automated patching.
(5) **Category-specific effectiveness**: The approach is particularly effective for input validation and memory safety vulnerabilities, while showing more modest improvements for complex cryptographic issues.

7 Discussion

7.1 Impact of Semantic Hints

The incorporation of semantic hints provides several advantages that explain the observed performance improvements:

Contextual Awareness: Hints enable the patch generation model to understand not just what to fix, but why the fix is necessary, leading to more appropriate solutions. Unlike direct patch generation where the model must infer both the problem and solution, hints provide explicit vulnerability context that guides the model toward correct fixes. This is particularly important for security vulnerabilities, where subtle distinctions between safe and unsafe patterns may not be obvious from code structure alone.

Domain Knowledge Transfer: CodeAstra-7B's cybersecurity-specific training provides domain expertise that general-purpose code models may lack [3]. Security vulnerabilities often involve patterns and edge cases that require specialized knowledge. By leveraging a model trained specifically for security analysis, we inject this domain expertise into the patch generation process without requiring extensive security training data for the patch generation model itself. This separation of concerns—analysis vs. generation—allows each model to excel in its specialized role.

Reduced Ambiguity: Hints clarify the nature of vulnerabilities, helping the model avoid introducing new issues or missing edge cases. Security patches must be precise; incorrect fixes can leave residual vulnerabilities or introduce new security flaws. Semantic hints reduce the ambiguity inherent in raw code by explicitly identifying vulnerability types and suggesting appropriate fix strategies, reducing the likelihood of incorrect or incomplete patches.

Improved Generalization: The hint-guided approach shows consistent improvements across diverse CWE categories, suggesting that the method generalizes well. Rather than requiring category-specific models or techniques, semantic hinting provides a unified approach that adapts to different vulnerability types through the hint generation stage.

7.2 Analysis of Failure Cases

While semantic hinting improves overall performance, analysis of failure cases reveals important insights:

Complex Multi-Step Vulnerabilities: Some vulnerabilities require multi-step fixes involving multiple code locations or complex control flow changes. In these cases, semantic hints may identify the vulnerability correctly but the patch generation model may struggle to synthesize comprehensive fixes. Future work could explore hierarchical hinting strategies that break complex fixes into multiple guided steps.

Cryptographic and Protocol-Level Issues: Vulnerabilities involving cryptographic operations or network protocols showed more modest improvements. These categories often require domain-specific knowledge that may not be fully captured in general security hints. Specialized hint generation strategies or domain-specific models may be needed for these cases.

Edge Case Handling: Some generated patches correctly address the primary vulnerability but may not handle all edge cases as comprehensively as expert-written patches. For example, the divide-by-zero patch uses exact comparison rather than epsilon-based comparison for floating-point values. While functionally correct, this represents a difference in defensive coding practices rather than a security flaw.

7.3 Implications for Practice

Our results have several implications for real-world security patch generation:

Two-Stage Architecture Benefits: The separation of vulnerability analysis (hint generation) and code generation (patch creation) provides flexibility. Organizations can update or replace either component independently—for example, incorporating new security analysis tools or switching to different code generation models without retraining the entire system.

Scalability Considerations: The approach scales effectively because hint generation can be parallelized and cached. Once hints are generated for a codebase, they can be reused across multiple patch generation attempts or shared across similar vulnerability patterns. This makes the approach practical for large-scale codebases.

Human-in-the-Loop Integration: Semantic hints provide a natural interface for human review and intervention. Security experts can review, refine, or override generated hints before patch generation, ensuring that critical vulnerabilities receive appropriate attention. The hints also serve as documentation of the analysis process, improving traceability and auditability.

Model Selection Flexibility: The approach is not tied to specific models. As new, more capable code generation or security analysis models emerge, they can be integrated into the pipeline without architectural changes. This future-proofing is important given the rapid pace of LLM development.

7.4 Limitations

Several limitations should be acknowledged:

Evaluation Scope: Our evaluation is limited to the Juliet Test Suite, which, while comprehensive, may not fully represent real-world vulnerability patterns. Real-world vulnerabilities often occur in larger codebases with complex dependencies and context that may not be captured in isolated test cases. Future evaluation on real-world vulnerability datasets would strengthen the claims.

Model Dependencies: The approach relies on the quality of both the hint generation and patch generation models. Poor-quality hints will mislead patch generation, while limitations in patch generation will prevent even perfect hints from producing good results.

Error propagation through the pipeline is a concern, though our results suggest the approach is robust to typical model imperfections.

Floating-Point and Numerical Handling: As seen in the divide-by-zero example, our generated patches may use exact comparisons rather than epsilon-based approaches, which could be refined. Similarly, integer overflow handling may not always match expert-written defensive coding practices. While these differences may not represent security vulnerabilities, they reflect differences in coding style and defensive practices.

Context Limitations: The approach processes functions in isolation, which may miss vulnerabilities or fixes that require broader program context. Some security fixes require changes across multiple functions or files, which our current approach doesn't address. Integration with whole-program analysis could address this limitation.

False Positives in Security Scanning: Flawfinder and similar static analysis tools may produce false positives, which could affect our vulnerability reduction metrics. However, since both baseline and hint-enhanced patches are evaluated using the same tool, the relative comparison remains valid even if absolute metrics include some noise.

7.5 Future Work

Potential directions for future research include:

Expanded Evaluation: Expanding evaluation to larger, more diverse vulnerability datasets, including real-world vulnerabilities from security advisories and bug reports. This would validate the approach's effectiveness in production settings and reveal limitations not apparent in synthetic test cases.

Multi-Hint Strategies: Investigating multi-hint strategies for complex vulnerabilities that require multiple perspectives or analysis techniques. Different hint types (e.g., vulnerability identification, fix pattern suggestions, edge case warnings) could be combined to provide richer guidance.

Security-Specific Fine-Tuning: Exploring fine-tuning patch generation models specifically for security contexts, potentially creating a specialized security patch generation model rather than relying on general-purpose code models with hints.

Hint Quality Validation: Developing automated hint quality validation mechanisms that can assess whether hints are accurate and actionable before patch generation, potentially reducing error propagation through the pipeline.

Incremental and Interactive Patching: Extending the approach to support incremental patching where multiple rounds of hint generation and patch refinement occur, with human or automated feedback guiding improvements. Interactive interfaces could allow developers to guide hint refinement based on domain knowledge.

Cross-Language Generalization: Evaluating whether the approach generalizes to other programming languages beyond C/C++, potentially revealing language-specific adaptations needed for effective semantic hinting.

8 Conclusion

This paper presents a novel approach to automated security patch generation that leverages semantic hints from specialized cybersecurity models to guide general-purpose code generation models. Our evaluation on 92 C/C++ vulnerability cases from the Juliet Test Suite [1] demonstrates that semantic hinting significantly improves both security vulnerability removal (40-50% overall reduction, 60-70% for high-severity issues) and code quality (15-20% improvement in semantic similarity to expert-written patches) while maintaining functional correctness (above 85% compilation and execution success rates).

The key contribution of this work is the introduction of semantic hinting as a mechanism for incorporating domain-specific expertise into automated code generation. By separating vulnerability analysis (performed by a specialized model) from patch generation (performed by a general-purpose model), we create a flexible, modular architecture that leverages the strengths of both specialized and general-purpose models. This two-stage approach enables organizations to integrate security expertise without requiring full retraining of large code generation models.

Our experimental results provide strong evidence for the effectiveness of semantic hinting across diverse vulnerability categories. The consistent improvements observed across input validation, memory safety, and other CWE categories suggest that the approach generalizes well beyond specific vulnerability types. The correlation between semantic similarity scores and security effectiveness indicates that hints successfully guide models toward expert-quality solutions.

The results indicate that incorporating domain-specific knowledge through semantic hints is a promising direction for improving automated program repair in security-critical contexts. As LLMs continue to evolve, the integration of specialized domain knowledge through techniques like semantic hinting may prove essential for achieving production-ready automated vulnerability remediation. The modular architecture enables continuous improvement as both security analysis and code generation models advance, making the approach future-proof and adaptable to new capabilities.

Looking forward, semantic hinting could be extended beyond security patching to other domains requiring specialized knowledge, such as performance optimization, accessibility compliance, or regulatory adherence. The general principle of using specialized analysis models to guide general-purpose generation models may prove broadly applicable across software engineering challenges. As automated code generation becomes increasingly prevalent, techniques that incorporate domain expertise will be essential for ensuring quality and correctness in specialized contexts.

9 Acknowledgments

We thank the developers of the Juliet Test Suite for providing comprehensive vulnerability test cases, and the open-source community for CodeAstra, CodeLlama, and related tools.

References

[1] NIST, "Juliet Test Suite for C/C++," National Institute of Standards and Technology, 2011.
[2] Rozière, B., et al. "Code Llama: Open Foundation Models for Code," Meta AI, 2023.

[3] "CodeAstra-7B: Cybersecurity-Focused Code Analysis Model," Hugging Face Model Hub, 2024.
[4] Feng, Z., et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," EMNLP, 2020.
[5] Wheeler, D. A., "Flawfinder: A Source Code Security Analyzer," 2011.
[6] Weimer, W., et al. "Automatically Finding Patches Using Genetic Programming," ICSE, 2009.
[7] Nguyen, H. D. T., et al. "SemFix: Program Repair via Semantic Analysis," ICSE, 2013.
[8] Gupta, R., et al. "DeepFix: Fixing Common C Language Errors by Deep Learning," AAAI, 2017.
[9] Guo, D., et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow," ICLR, 2021.
[10] Wang, Y., et al. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," EMNLP, 2021.
[11] Chen, M., et al. "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.
[12] Monperrus, M. "Automatic Software Repair: A Survey," IEEE Transactions on Software Engineering, 2018.
[13] Gao, Q., et al. "Fixing Security Vulnerabilities with Code Generation," ESEM, 2020.
[14] Hu, E. J., et al. "LoRA: Low-Rank Adaptation of Large Language Models," ICLR, 2022.
[15] Lewis, P., et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, 2020.