# Improving Source Code Security: A Novel Approach Using Spectrum of Prompts and Automated State Machine

**Claudio A. S. Lelis[1], Cesar A. C. Marcondes[1] and Kevin Fealey[2]**

[1]Divisão de Ciência da Computação – Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos – SP – Brazil

[2]AppSecAI, Inc.
Los Altos – California – USA

claudio.lelis@ga.ita.br, cesar.marcondes@gp.ita.br, kevin@appsecure.ai

***Abstract.*** *As software security becomes increasingly vital, automating source code vulnerability remediation is essential for enhancing system reliability. This research presents an integrated framework that combines large language models (LLMs) with a patch-compile-test state machine (PCT-SM) to generate accurate, functional code repairs with minimal human intervention. The solution is organized into three stages: the Editing Plan to identify necessary code edits; the Patch Plan to generate unified patches; and the Verification Plan to rigorously validate repairs through PCT-SM. Moreover, the process is refined by the Spectrum of Prompts (SoP) technique, which iteratively optimizes prompt variations to improve remediation effectiveness. Experimental evaluations indicate that our approach yields higher remediation success rates and more robust testing performance compared to conventional methods, with the SoP component exhibiting prominent repair outcomes.*

## 1. Introduction

Software vulnerabilities pose significant security risks, often stemming from programming errors that can be exploited by attackers. Addressing these issues manually is time-consuming and error-prone, as fixing one bug may inadvertently introduce new defects [Britton et al. 2013, Gu et al. 2010]. Automated Program Repair (APR) has emerged as a promising solution, leveraging automated techniques to generate patches and reduce human effort [Weimer et al. 2009]. However, despite advances in APR, the correctness of generated patches remains a challenge, requiring further verification and validation steps.

Recent breakthroughs in Large Language Models (LLMs) have revolutionized APR by enabling models to generate, refine, and improve source code with impressive accuracy [Zhang et al. 2023b, Zhang et al. 2024b]. While APR traditionally focuses on fixing general software defects, an urgent need has arisen for Automated Vulnerability Repair (AVR), which specifically targets security-related software flaws. Unlike general bug fixes, security vulnerabilities require precise and reliable remediation to prevent exploitation. This transition from APR to AVR introduces new challenges, particularly in ensuring that fixes not only resolve vulnerabilities but also preserve the original functionality of the software.

Despite promising efforts in integrating LLMs into AVR [Fan et al. 2023, Xia and Zhang 2023, Zhang et al. 2023a], several gaps remain. One major challenge is the repair cost associated with LLMs: as inference time increases, developers must wait longer for patches, reducing the time available for validation [Zhang et al. 2024b]. Additionally, security-focused code repair demands context-aware solutions that effectively analyze vulnerability patterns and generate targeted patches while avoiding redundancy and errors. Existing solutions often struggle with these aspects, highlighting the need for improved AVR methodologies.

This research proposes a novel, fully automated approach to mitigating software vulnerabilities using prompt-engineering techniques with general-purpose LLMs. Our methodology is structured into three key stages: the *Editing Plan*, which identifies and outlines necessary code modifications; the *Patch Plan*, which applies corrections to generate candidate patches; and the *Verification Plan*, which employs a state machine to validate fixes and ensure functional correctness. These components work together to streamline the vulnerability remediation process, reducing manual intervention and enhancing efficiency. Through this iterative feedback loop, our method adapts to various code contexts, enhancing the robustness of automated vulnerability repair.

A core contribution of this research is the introduction of a novel **Spectrum of Prompts** technique, which iteratively refines repair prompts based on performance metrics. By dynamically adjusting prompts throughout the editing, patching, and verification stages, this approach enables continuous improvement in the accuracy and effectiveness of vulnerability fixes.

**The main contributions of this research are as follows:**

- A fully automated AVR approach leveraging prompt engineering rather than fine-tuned models, making it adaptable to general-purpose LLMs.
- A structured three-stage framework (Editing Plan, Patch Plan, and Verification Plan) to improve the precision and reliability of generated patches.
- The development of the Spectrum of Prompts technique, enabling iterative prompt refinement for more accurate and effective repairs.

The remainder of this paper is structured as follows: Section 2 reviews relevant background and related work. Section 3 presents an overview of our proposed solution. Section 4 details the Patch-Compile-Test State Machine (PCT-SM), while Section 5 introduces the Spectrum of Prompts (SoP) technique. Section 6 describes the experimental setup and results. Section 7 discusses key findings, limitations, and future directions. Finally, Section 8 concludes the paper.

## 2. Background and Related Works

Large Language Models (LLMs) such as OpenAI's GPT series [Brown et al. 2020] and Codex [Chen et al. 2021] along with prompt engineering techniques, have revolutionized natural language processing by learning statistical patterns from large-scale textual data. These models use attention mechanisms [Vaswani 2017] to generate context-aware representations. Their performance in specialized tasks—such as source code repair—can be enhanced through task-specific prompt engineering, which involves crafting precise instructions that guide the LLM's output without

the need for retraining [Liu et al. 2023, Chen et al. 2023]. Techniques like zero-shot and few-shot prompting [Radford et al. 2019, Brown et al. 2020] provide initial frameworks, while methods such as chain-of-thought prompting [Wei et al. 2022] have recently been adapted to promote step-by-step reasoning in vulnerability remediation.

A key trend in automated vulnerability repair (AVR) methods is the use of reasoning-based prompt engineering for iterative vulnerability repair. For instance, VRpilot [Kulsum et al. 2024] employs chain-of-thought prompts combined with iterative patch validation to refine repairs, achieving improvements over baseline approaches on vulnerabilities in C and Java. Similarly, the VSP framework [Nong et al. 2024] uses vulnerability-semantics-guided prompting to enhance both identification and patching precision, thereby underscoring the effectiveness of structured prompt methods in security-critical scenarios.

Other works have explored the limitations of general-purpose models in vulnerability repair. [Pearce et al. 2023] demonstrate that while black-box LLMs can handle synthetic or hand-crafted vulnerabilities, they often struggle with real-world cases due to insufficient domain-specific guidance. Complementary studies by [Le et al. 2024] on JavaScript vulnerabilities and [Ahmad et al. 2024] on hardware security further emphasize the necessity of detailed contextual prompts when fixing security bugs.

Recent studies have also proposed advanced prompting strategies. [Liu et al. 2024] introduce tailored prompt templates and iterative refinement techniques that show significant gains in repair effectiveness. The authors in [Zhang et al. 2024a] focus on memory corruption vulnerability repair, revealing that limitations in reasoning can hinder performance in challenging security scenarios.

Despite these significant strides, none of the existing approaches manage to integrate all critical aspects into a single, streamlined process. The reviewed works highlight individual strengths—such as leveraging LLMs, iterative feedback, and vulnerability-specific datasets—but typically focus on isolated phases of remediation or lack robust verification mechanisms.

In contrast, this work introduces a novel, holistic framework that not only identifies necessary code edits (Editing Plan) but also generates patches in a standardized diff format (Patch Plan) and rigorously verifies their correctness through a patch-compile-test state machine (PCT-SM) in the Verification Plan. Additionally, the Spectrum of Prompts (SoP) method leverage prompt variations for automatically optimizes remediation. This methodology streamlines the automated vulnerability remediation process, encompassing confirmation through to validation, designed to minimize human intervention while enhancing the reliability of software systems.

## 3. Solution

This research introduces a comprehensive solution to the challenge of managing source code vulnerabilities through an automated approach that leverages large language models (LLMs) and a state machine framework. The proposed solution is structured into three primary stages: the *Editing Plan*, which involves generating a list of necessary code edits; the *Patch Plan*, where patch files are created in a

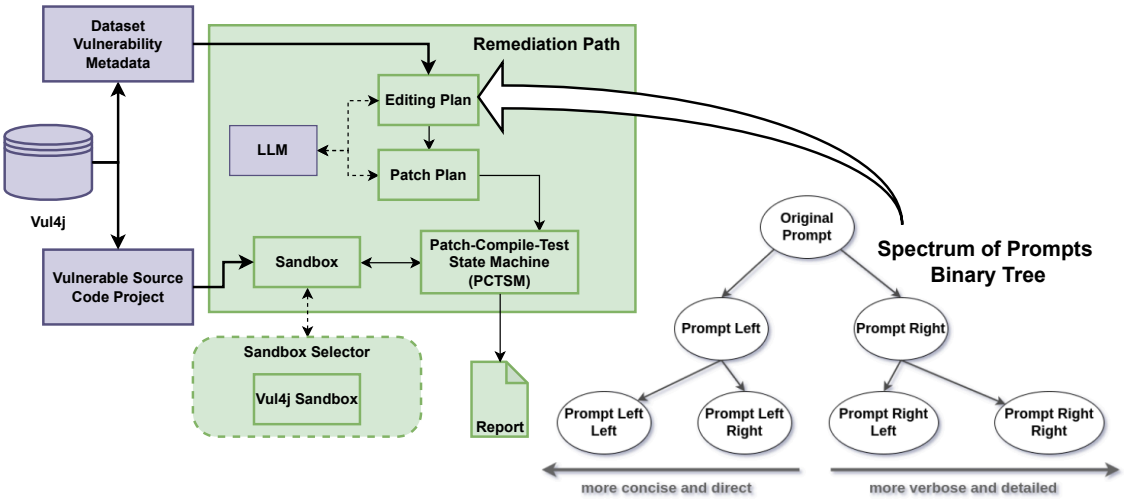



**Figure 1. Proposed Solution Overview**

unified diff format; and the *Verification Plan*, which utilizes a patch-compile-test state machine to ensure the robustness of the applied patches. This process is further refined by the *Spectrum of Prompts*, a technique designed to iterate over prompt variations, thereby enabling continuous improvement of the remediation process.

The proposed solution is illustrated in Figure 1. This diagram provides a high-level overview of how the proposed components and external resources work together to address source code vulnerabilities, employing a color-coding scheme to clearly differentiate between the various elements. The color **green** represents the components including the key stages of the proposed solution—such as the Editing Plan, Patch Plan, and Verification Plan. The Spectrum of Prompts tree is also displayed, where each node represents a distinct prompt variation derived via meta-prompts, using the prompt rephrasing criterion (more details are provided later). In the same diagram (Figure 1), the color **gray** is used to represent external projects and resources that are incorporated into the solution. The calls to an external LLM (which can be OpenAI's models or open sourced models like Llama 3.1) are employed at multiple stages, including the Editing and Patch Plans, to assist in generating and refining the necessary code edits.

The Vul4j [Bui et al. 2022] dataset is a curated collection of vulnerable Java projects. It includes a wide range of vulnerability types, accompanied by corresponding fixes, making it an ideal resource for testing the accuracy and robustness of the proposed solution as applicable to a variety of real-world scenarios. In this research, the Vul4j dataset was employed to evaluate Patch-Compile-Test State Machine (PCT-SM) ability to remediate vulnerabilities and the effectiveness of Spectrum of Prompts (SoP) through the various prompting strategies implemented in the Editing and Patch Plans.

## 4. PCT-SM: Patch-Compile-Test State Machine

We introduce PCT-SM (Patch-Compile-Test State Machine), an automated source code remediation method enhanced by the Spectrum of Prompts (SoP, discussed later). PCT-SM employs a sequential three-plan approach: Editing, Patch, and

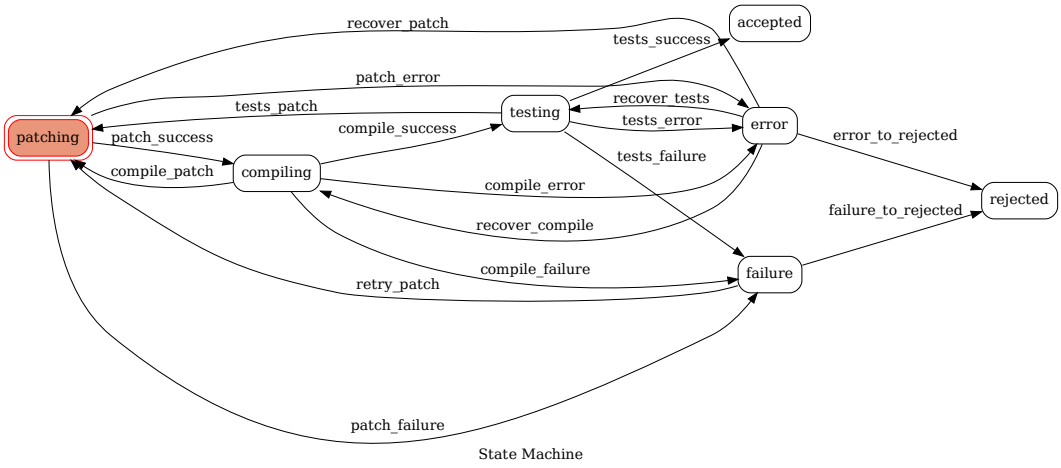



**Figure 2. State Machine Diagram of the Verification Plan**

Verification, implemented as the Patch-Compile-Test State Machine (PCT-SM).

### 4.1. Structure and Operation of PCT-SM

The **Editing Plan** generates a pseudo-code action plan targeting only vulnerable code regions, using the full source with annotated line numbers for accuracy and avoiding redundant code. The subsequent **Patch Plan** converts this plan into a directly applicable unified diff format patch using precise line references, the starting state of the FSM. This separation of concerns between edit and patch is robust [1]. The **Verification Plan** automates patch validation using the Patch-Compile-Test (PCT) state machine (Fig. 2) as a funneling process. The FSM orchestrates patch application, compilation, and testing through distinct states: **Patching** (apply patch), **Compiling** (compile patched code), **Testing** (run vulnerability checks), **Error** (handle unexpected issues), **Failure** (manage patch/compile/test failures), **Accepted** (successful remediation), and **Rejected** (irrecoverable failure or error limit reached). Compilation and testing occur in isolated, language-specific (i.e. java) Sandboxes for security and environment consistency (tools, libs and dependencies).

### 4.2. Behavior and Rules of PCT-SM

PCT-SM funnels a strict successful trajectory Patch → Compile → Test sequence with built-in retry mechanisms. Key rules govern transitions:

- **Success:** Successful completion of a stage transitions to the next (Patching → Compiling → Testing → Accepted). The consecutive attempts counter for the completed stage is reset upon successful transition.
- **Failure Handling:** Failure at any stage (Patching, Compiling, Testing) returns the machine to the Patching state for a new attempt. Total and consecutive failure attempts per stage are tracked.

[1]The task of unified patch generation can be difficult for smaller LLM models, according to benchmarks conducted by Aider `https://aider.chat/2023/11/06/benchmarks-1106.html`

- **Error Handling:** Errors trigger retries within **same** state. A counter (*total_error_tries*) tracks these; exceeding a threshold leads to Rejected state.
- **Stop Criteria:** The process terminates in the Accepted state upon full success, or the Rejected state if error thresholds or maximum retry limits for failures are reached.

### 4.3. Performance Metrics and Output

PCT-SM outputs key performance metrics: a final *remediation_accepted* status (1 for success, 0 for rejection) and counters for total attempts per stage (*total_patch_tries*, *total_compile_tries*, *total_test_tries*) and total errors (*total_error_tries*). These metrics evaluate the remediation process's effectiveness, remediation journey, and resilience. Furthermore, ablation insights can be obtained by inspecting the error areas.

## 5. SoP: Spectrum of Prompts

The PCT-SM is a robust tool validation, but still there is the problem of generating effective patches for software vulnerabilities via LLMs that often require refinement beyond the initial attempt. Our Spectrum of Prompts (SoP) method addresses this by systematically exploring controlled variations of the initial prompt, thereby generating a diverse set of potential patches. This broadens the solution space and increases resilience against failures where a single patch might prove insufficient due to subtle code interactions or test case edge scenarios. The objective of the SoP is twofold: to broaden the solution space and to reduce search itself. By generating multiple prompt variations at each level, it explores a wide range of potential patches to effectively address vulnerabilities. This variation helps to manage failures due to unforeseen dependencies, test cases, or logic gaps, ensuring that the system is not limited to a single approach. In addition, it will prune ineffective prompts.

### 5.1. Optimization Basis: Branch and Bound

SoP employs Branch and Bound (B&B) [Lawler and Wood 1966, Morrison et al. 2016], an optimization heuristic search algorithm, to efficiently search the space of prompt variations. Applied to our problem of vulnerability, SoP applies B&B to find the prompt variation that maximizes an objective function $f(x)$ reflecting successful remediation in PCT-SM (Sect. 4) through a combination of success and a valuation of intermediate errors in PCT-SM. Explores prompt variations (branching) while pruning unpromising options based on estimated heuristic performance (bounding).

#### 5.1.1. Objective Function

The objective function $f(x)$ heuristically quantifies the success of a prompt variation based on all PCT-SM output metrics. It rewards successful remediation (*remediation_accepted*=1) with a constant $c_{default}$ and incorporates a weighted sum of the counts of attempts (tries) and errors from PCT-SM states:

$$f(x) = c_{default} \cdot remediation\_accepted + w_1 \cdot total\_patch\_tries + w_2 \cdot total\_compile\_tries + w_3 \cdot total\_test\_tries - w_1 \cdot error\_tries \quad (1)$$

Weights $w_1, w_2, w_3$ are derived from the PCT-SM 'max_tries' limit ($n$), prioritizing progress through later stages (lower weight for patching, higher for testing) and

penalizing errors using $w_1$. For a max retries of $n = 6$, $w_3 = \frac{1}{3n}$, $w_2 = \frac{1}{3n(n-1)}$, $w_1 = \frac{1}{3(n(n-1)^2+n-1)}$. This design favors resilient prompt variations that successfully navigate the Patch-Compile-Test sequence journey.

**Original Root LLM Prompt**

```
Considering the following Java {{
    compliance_level}} source code `
    original_code.java`:
```
{{source_code}}
```
A vulnerability was detected in this
    Java code file.
Vulnerability Name: {{vulnerability_name
    }}
Vulnerability Description: {{
    vulnerability_description}}

Location in Code, this (these) are the
    exact vulnerable location(s) in the
    source code: {{
    vulnerability_location}}

Generate an action edit plan in pseudo-
    code format to repair only the
    vulnerability code:
- Do not copy redundant parts of the
    code in the response.
- Be specific and detail-oriented.
- The output should be a list of EDITs
    required to fix the vulnerability.
```

**Prompt Left (More Concise)**

```
Given the Java {{compliance_level}}
    source code `original_code.java`:
```
{{source_code}}
```
A vulnerability was detected: {{
    vulnerability_name}} ({{
    vulnerability_description}}) at
    location(s) {{vulnerability_location
    }}.

Create a list of pseudo-code EDITs to
    repair the vulnerability. Be
    specific and only address the
    vulnerable code.
```

**Prompt Right (More Verbose)**

```
You are provided with a Java source code
     file, `original_code.java`, that
    adheres to the {{compliance_level}}
    standards. The content of this file
    is as follows:

```
{{source_code}}
```

Upon conducting a thorough examination
    of the provided Java code, a
    significant security vulnerability
    was identified. The details of this
    vulnerability are as follows:

Vulnerability Name: {{vulnerability_name
    }}
Vulnerability Description: {{
    vulnerability_description}}

Further analysis has pinpointed the
    exact location(s) within the source
    code where the vulnerability is
    present:

Location in Code: {{
    vulnerability_location}}

Your task is to create a comprehensive
    action edit plan in pseudo-code
    format, specifically designed to
    rectify the identified vulnerability
     in the Java source code. It is
    essential to note that your response
     should focus exclusively on
    addressing the vulnerable code
    segments and must not include any
    redundant or unaffected parts of the
     code.

When developing your edit plan, please
    adhere to the following guidelines
    ...
```

**Table 1.** **Prompt Variations Examples from SoP.**

### 5.1.2. SoP Optimization Process

The B&B process adapted for SoP follows these steps:

1. **Initialization:** Start with the original prompt (root node) and compute its initial score $f(x^*)$.
2. **Branching:** Generate child nodes by applying prompt variations (e.g., rephrasing with a more verbose or more succinct versions).
3. **Bounding:** Estimate the potential success $B(x')$ of each new variation $x'$.

4. **Pruning:** Discard variations where $B(x')$ is lower than current best $f(x^*)$.
5. **Iteration:** Repeat branching, bounding, and pruning, updating $f(x^*)$ when better variations are validated via PCT-SM, until termination.

### 5.2. Application: Variation and Termination

Our implementation primarily used prompt rephrasing (e.g., altering instruction wording, like, changing "Provide a step-by-step plan to edit the vulnerable code" to "Detail the steps necessary to modify the vulnerable section of the code" or "Give a sequential plan for correcting the vulnerability", while preserving intent as the variation criterion, as in SoP tree conceptually illustrated in Fig. 1. Our SoP framework also supports other criteria like reordering template elements, which can alter the sequence of key elements in the prompt template.

The optimization terminates under two conditions: **(i) Convergence:** Improve $f(x^*)$ between iterations falls below threshold; and **(ii) Resource Limit:** A predefined maximum number of nodes (`max_nodes`) is explored. These ensure an adaptive yet computationally bounded search for the optimal prompt variation.

### 5.3. Operational Integration and Reproducibility of SoP

The Spectrum of Prompts component is designed to act as a fallback optimization loop within a real-world AVR tool. When enabled at configuration time—before starting the remediation process—SoP activates only if the initial prompt (root node) fails to produce a successful remediation.

In this mode, the PCT-SM executes an initial attempt using the default root prompt. If remediation fails, the SoP controller evaluates the quality score $f(x)$ and, if justified, triggers branching according to the Branch-and-Bound (B&B) heuristic.

Each SoP run explores up to seven nodes per case, forming a binary tree. The root node represents the original prompt. Its left and right children are generated by meta-prompting to produce concise and verbose rephrasings, respectively. Further branching is allowed only when improvements in $f(x)$ suggest better candidates exist. A detailed example of prompt variation is shown in Table 1, presenting the original prompt or root node (**blue**), and the two self generated child nodes, versions of more verbose and more concise prompts that allow the exploration of prompts variations.

All LLM queries are executed deterministically (`temperature=0`) to ensure reproducibility. Prompt variations are derived from a fixed template via reordering or rewording elements. The system logs all prompt generations, PCT-SM states, and $f(x)$ evaluations. This bounded and feedback-driven strategy enables controlled exploration of the prompt space while ensuring the process remains efficient. A pseudocode for SoP is available in [Lelis 2025].

## 6. Experiments and Analysis

Our experiments and analysis evaluates our proposed methodology, which combines the Edit Plan and Patch Plan as a structured remediation strategy, the PCT-SM as an automated validation framework, and finally, SoP as a novel prompt-space exploration technique, by addressing three research questions (RQ1–RQ3).

The experiment aimed to evaluate several aspects: to compare the Edit Plan remediation method with the Baseline remediation method, both applied to real-world Java vulnerabilities through a subset of the Vul4j dataset. The goal is to assess their ability to generate secure, non-vulnerable code.

To ensure the experiment was conducted on realistic, reproducible vulnerabilities, we utilized the Vul4j dataset – a well-known benchmark of 79 reproducible Java vulnerabilities. Each vulnerability in Vul4j is linked to a corresponding patch and Proof of Vulnerability (PoV) test case. From the 79 vulnerabilities, we focused on 35 cases that involved vulnerabilities impacting a single Java file. This simplification ensures that the remediation is concentrated on isolated issues, making it easier to observe the models' capabilities. We randomly selected 10 vulnerabilities from these 35 cases to create a manageable yet representative experimental subset. Despite the small sample size, these vulnerabilities originate from complex, industrial-scale projects with extensive codebases, lengthy compilation processes, and rigorous testing of functionality and vulnerability, ensuring that they represent realistic challenges rather than trivial benchmark focusing on simple vulnerability errors.

The Baseline method involves directly prompting the LLM to generate a non-vulnerable version of the given vulnerable code, relying on vulnerability information (such as name, description, and location, if available). This approach does not provide any additional guidance or structure but relies on the model's inherent understanding of code and security principles to remediate the vulnerability. The LLM generates the remediation in a single prompt, and the result is tested using the PoV test case provided by Vul4j, with PCT-SM generating the performance metrics.

We applied the analysis through 3 LLMs: *GPT-3.5-turbo*, *GPT-4-0125-preview* and *llama 3.1*. We used the *llama3-405b-instruct-maas* model provided by Google Cloud Vertex AI. This model is based on Meta's LLaMA 3.1 architecture with 405 billion parameters and is optimized for instruction-following tasks. It was accessed via the Vertex AI Generative Models API and integrated into our experimental pipeline using the *litellm* Python client.

The temperature was set to 0, in all cases, to ensure reproducibility. The ability of each method to remediate the selected vulnerabilities was measured and compared based on configurations generated by PCT-SM. Varying the models aim to assess the robustness and the generalization capability of the methods.

In addition to the baseline evaluation, our methodology leverages the Spectrum of Prompts (SoP) approach. SoP explores the effect of structured prompt variations on the performance of LLMs in complex tasks such as vulnerability remediation. A binary branching strategy was used to generate prompt variations—where the left branch produces a concise, focused version, and the right branch produces a verbose, detailed version. By dynamically updating the best admissible configuration based on the performance measure $f(x)$, SoP guides prompt refinement throughout the remediation process.

During the experimentation process, an unexpected phenomenon was observed, termed **Meta-Prompt Contamination (MPC)**. MPC refers to the unintended inclusion of meta-level instructions (e.g., "Here's a rephrased version of the original

prompt") within generated prompt variations. Such contaminations likely stem from the prompt-generation system incorporating its instructional context, which blurs the boundary between meta-instructions and the actual task content. This analysis further explores the impact of MPC on the outcome of the PCT-SM framework.

### 6.1. Goal–Question–Metric (GQM) Setup

The primary goal of our evaluation is to assess whether the integration of a structured remediation process (Edit Plan) with the PCT-SM validation framework and the SoP prompt exploration technique improves the performance of LLMs in addressing real-world software vulnerabilities.

**RQ1: Does the Edit Plan remediation method, supported by PCT-SM, outperform a baseline approach in addressing real-world Java vulnerabilities using LLMs?**

RQ1 evaluates if the Edit Plan remediation method, supported by PCT-SM, outperforms a Baseline approach. We hypothesize that the Edit Plan method will yield a higher remediation success rate compared to the Baseline, and its iterative patch refinement will be reflected in an increased number of test attempts.

**RQ2: Can the Spectrum of Prompts (SoP) method, which systematically rephrases prompts along concise and verbose branches, improve the performance of LLMs in the context of software vulnerability remediation?**

RQ2 assess the impact of the Spectrum of Prompts (SoP) method on improving vulnerability remediation performance. We assume that structured prompt variations, generated via a binary branching rephrasing strategy (concise vs. verbose), can lead to improved results compared to the original prompt alone.

**RQ3: Does the presence of Meta-Prompt Contamination in dynamically generated prompt variations negatively affect the outcome of vulnerability remediation in the PCT-SM framework?**

RQ3 evaluates the effect of Meta-Prompt Contamination (MPC) on the performance of prompt variations within the PCT-SM process. We hypothesize that prompt variations affected by MPC will produce inferior results in terms of PCT-SM configuration success compared to clean, uncontaminated variations.

### 6.2. Experimental Design

The experiment was designed to rigorously assess each component of our proposed methodology:

**Dataset and Selection:** From the Vul4j dataset comprising 79 reproducible Java vulnerabilities, 35 single-file cases were identified, from which 10 representative vulnerabilities were randomly selected. This subset provided a controlled yet realistic environment for evaluation.

**Baseline and Edit Plan Methods:** For each of the 10 selected vulnerabilities, the Baseline method was applied by directly prompting the LLM (either GPT-3.5-turbo or GPT-4-0125-preview) to generate a non-vulnerable version of the code. In contrast, the Edit Plan method involved structured pseudo-code action plan plus the Patch Plan, and validation using the PCT-SM loop.

**SoP Setup:** The Spectrum of Prompts (SoP) approach was implemented using the llama 3.1 model to explore whether structured prompt variations could enhance remediation performance. A binary branching strategy was employed:

- **Left Branch:** Variations were designed to be concise and focused.
- **Right Branch:** Variations were designed to be verbose and detailed.

A tree structure was created with a maximum of 7 nodes—including the original prompt as the root—and dependencies between the parent and child prompts were maintained according to the PCT-SM configuration evaluation.

**MPC Evaluation:** To evaluate Meta-Prompt Contamination (MPC), a second set of six prompt variations was generated starting from `prompt_left_left` (where MPC was observed). These variations were compared against the non-contaminated prompt variations using the same 10 vulnerability cases.

**Metrics Collection:** For every experimental run, the following metrics were collected: Patch attempts (patch tries), Compile attempts (compile tries), Test attempts (test tries) and Remediation success rate (remediation accepted indicator). These metrics were aggregated using average (avg) and standard deviation (std) and compared across methods and models as shown in the results.

### 6.3. Experiments and Results

#### 6.3.1. Evaluation of the Edit Plan Remediation Method (RQ1)

**Hypothesis:** We hypothesize that the Edit Plan remediation method will yield a higher remediation success rate compared to the Baseline method and that the iterative PCT-SM loop will result in a higher number of test attempts, indicative of robust patch exploration.

**Experiment:** For each of the 10 selected vulnerabilities, both the Baseline and Edit Plan methods were executed using GPT-3.5-turbo and GPT-4-0125-preview. The PCT-SM framework was used to track the progression of patching, compiling, and testing attempts.

**Table 2. Comparison between Baseline and Edit Plan methods on GPT-3.5 and GPT-4 models**

| Method | Model | Patch tries (avg ± std) | Compile tries (avg ± std) | Test tries (avg ± std) | Remediation Rate |
|---|---|---|---|---|---|
| Baseline | GPT-3.5-turbo | 8.0 ± 1.054 | 1.7 ± 0.823 | 0.3 ± 0.675 | 0.00% |
| Baseline | GPT-4-0125-preview | 10.0 ± 6.182 | 2.1 ± 2.807 | 1.2 ± 2.098 | 0.00% |
| Edit Plan | GPT-3.5-turbo | 6.0 ± 2.000 | 0.5 ± 0.707 | 0.2 ± 0.422 | 10.00% |
| Edit Plan | GPT-4-0125-preview | 8.8 ± 3.795 | 1.6 ± 2.221 | 0.7 ± 0.949 | 10.00% |

**Results:** Table 2 presents the aggregated performance metrics. Across both GPT models, the Edit Plan method achieved a remediation success rate of 10%,

while the Baseline approach achieved none. Notably, Edit Plan exhibited a higher number of test attempts on average—particularly with GPT-4—suggesting increased resilience through deeper exploration.

**Answer to RQ1:** Yes, the Edit Plan remediation method outperformed the Baseline approach in terms of successful remediation rate and testing effort across both LLMs.

### 6.3.2. Evaluation of the SoP Method (RQ2)

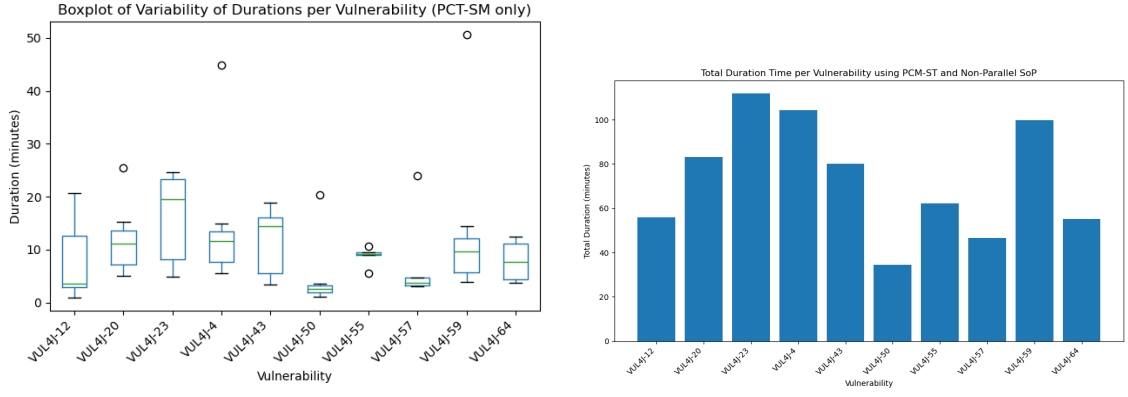
**Hypothesis:** We hypothesize that structured prompt variations generated through a binary branching strategy (concise and verbose) will lead to improved outcomes in vulnerability remediation when compared with the original prompt.

**Experiment:** The experiment employed the llama 3.1 model to evaluate the SoP approach. The original prompt served as the baseline, and SoP was applied to generate prompt variations as detailed in a binary branching tree. Six prompt variations were generated, maintaining parental dependencies, and the best prompt was identified via the PCT-SM's objective function $f(x)$. Remediation was then tested over the same 10 cases from Vul4j.

**Table 3. Methods Comparison on patching, compiling, testing, and remediation rate.**

| Method | Patch tries (avg ± std) | Compile tries (avg ± std) | Test tries (avg ± std) | Remediation Rate |
|---|---|---|---|---|
| Original Prompt | 10.0 ± 5.033 | 3.7 ± 3.020 | 0.3 ± 0.675 | 10.00% |
| SoP | 10.2 ± 7.239 | 5.5 ± 2.593 | 1.4 ± 1.776 | 40.00% |
| SoP with MPC | 10.4 ± 6.720 | 5.4 ± 2.875 | 1.4 ± 1.838 | 30.00% |

**Results and Performance Analysis:** The Table 3 compares the performance metrics for the original prompt, SoP, and SoP with MPC. The results indicated that SoP led to an improvement in remediation performance in 9 of 10 cases, with an increase of 3 successfully remediated cases compared to the original prompt. The structured prompt variations resulted in enhanced testing and verification outcomes.




**(a) Time per FSM pass.** **(b) Cumulative FSM duration.**

**Figure 3. Non-Functional Performance Analysis of the SoP Method.**

In addition, we measured runtime of the experiments, using an off-the-shelf laptop. The results shown in Figure 3 provides an overview of the non-functional performance of the Spectrum of Prompts (SoP) method by presenting two key perspectives. The left subfigure displays the time (in minutes) consumed during each pass of the finite state machine (FSM), effectively illustrating the cost per iteration, while the right subfigure aggregates these durations to reveal the cumulative overhead associated with the patch-compile-test sequence over SoP. This quantitative analysis is particularly salient given that VUL4J is a realistic dataset featuring actual Java vulnerabilities, where the compilation process in some projects is notably long, in the order of 10s of minutes. To gain further insights, we collected timestamp measurements for every complete trajectory in the FSM. The variability observed in the boxplot can be attributed to the probabilistic nature of the models in generating effective patches—with compilation and testing times generally dominating, though mostly remaining within acceptable limits aside from a few outliers.

Moreover, the structured, iterative prompt variations of SoP not only improve remediation outcomes (as reflected in the success rate improvements) but also enhance overall efficiency by reducing excessive runtime accumulation Figure 3(b). Although we did not parallelize the execution of compilation and tests (which would allow each level of the tree to run concurrently), the inherent bound and branching cuts of the SoP method ensure that fix times remain manageable. This balance guarantees that full vulnerability repairs using open source models are practical, reinforcing the potential of the SoP framework for scalable, automated vulnerability remediation.

**Answer to RQ2:** Yes, the Spectrum of Prompts method improved the performance of vulnerability remediation, validating the hypothesis that structured prompt variations offer better results compared to the original prompt.

#### 6.3.3. Evaluation of MPC (RQ3)

**Hypothesis:** We assume that prompt variations affected by Meta-Prompt Contamination will yield inferior results in PCT-SM verification, as the contamination degrades prompt clarity and contributes to task drift.

**Experiment:** A second set of six prompt variations was generated using `prompt_left_left`—a configuration where MPC was observed. These MPC-affected variations were compared with those from the non-contaminated SoP experiment across the same 10 vulnerability cases. The evaluation criteria were consistent with earlier experiments (patch, compile, and test attempts, along with remediation success rate).

**Results:** The comparative analysis revealed that in 3 cases the configuration remained unchanged, 3 cases showed improvements, and 4 cases experienced a decline in performance metrics. In terms of accepted remediations, there was a decrease of one successful remediation; however, when compared to the baseline (no SoP), SoP with MPC still demonstrated overall improvement.

**Answer to RQ3:** The preliminary empirical results indicate that while MPC leads to a slight decrease in the total number of successful remediations, the method remains effective. Thus, although MPC negatively affects performance metrics to a degree, SoP with MPC still outperforms the original, uncontaminated baseline.

This holistic evaluation validates the primary goal of leveraging structured remediation, automated verification, and prompt exploration to improve source code security.

## 7. Discussion and Limitations

The SoP method relies on a Branch & Bound (B&B) search strategy guided by a scoring function $f(x)$, designed to estimate the promise of remediation candidates in a binary tree (seven nodes per vulnerability). A fundamental assumption is that the node selected as *best* should exhibit the highest $f(x)$ value among all candidates, including those pruned or not created.

To verify this, we conducted a post-evaluation analysis of all *not-created* nodes by retrospectively computing their $f(x)$ values. This allowed us to fully reconstruct the decision space and validate the pruning logic. We found two potential inconsistencies (VUL4J-20 and VUL4J-23) where a *not-created* node had a higher $f(x)$ than the selected *best* node.

Despite these deviations, the method's empirical effectiveness remains uncompromised: no unexplored or pruned node with higher $f(x)$ led to an accepted remediation. One accepted fix was pruned, but its $f(x)$ was correctly lower than the selected node. In four cases, the chosen *best* node also produced the accepted remediation, reflecting strong alignment between heuristic estimates and successful outcomes.

These findings illuminate a critical insight: while heuristic pruning introduces a theoretical risk of excluding high-potential paths, no empirically superior remediations were lost in our evaluation.

### 7.1. Case Study: Tree Exploration for VUL4J-12

Figure 4 shows the SoP tree for VUL4J-12. Node labels indicate prompt variants. Boxes show PCT-SM configuration, $f(x)$ score, and outcome. The Left child led to a valid remediation and had the highest $f(x)$, becoming the selected best. The deeper node `Left_Left` also produced a correct fix, but with a lower $f(x)$, and was pruned accordingly.

This case illustrates how SoP handles multiple valid fixes: only the most promising is retained. Despite pruning a successful node, the framework correctly prioritized the better option. This validates the pruning logic as both safe and efficient. More examples like this are available in [Lelis 2025].

### 7.2. Comparison with Traditional Approaches

Unlike general APR tools focused on syntactic fixes and unit test pass rates, AVR requires both (i) elimination of provable vulnerabilities—often validated by PoV

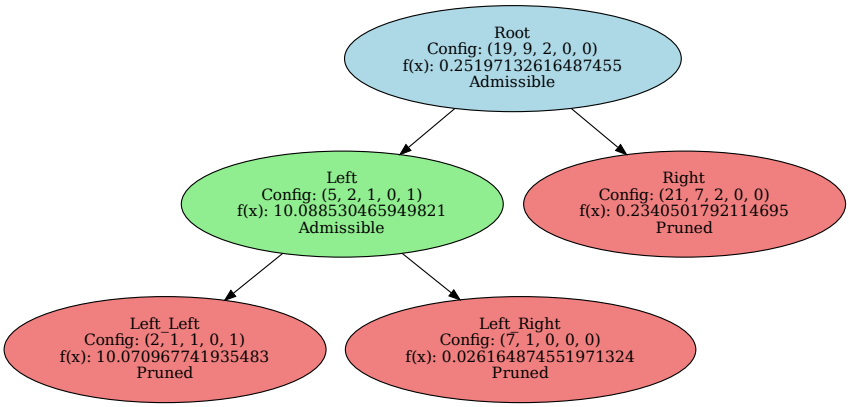



**Figure 4. SoP tree for VUL4J-12.**

tests—and (ii) preservation of full program behavior through functional testing. Our framework directly addresses these dual requirements. It evaluates patches via the PCT-SM state machine, which enforces a complete patch-compile-test cycle. This enables meaningful comparisons rooted in real-world remediation goals.

While traditional APR methods are effective for general bug fixing, they often lack the security validation required for AVR. In contrast, our system integrates structured repair (Edit Plan), prompt optimization (SoP), and rigorous validation, aligning with secure software engineering needs. Future comparisons will incorporate datasets like VJBench [Wu et al. 2023], which extend Vul4j with additional AVR-focused cases.

## 8. Closing Remarks

This paper presents an integrated framework for automated vulnerability remediation that combines a structured Edit Plan, the Patch-Compile-Test State Machine (PCTSM), and the Spectrum of Prompts (SoP) technique. Through evaluation on the Vul4J dataset, we demonstrate that:

- the Edit Plan outperforms baseline approaches in remediation success.
- structured prompt variation via SoP improves the effectiveness of LLM-guided patching.
- the method remains robust even under prompt contamination possibility.

Although our findings are encouraging, certain limitations should be noted: (i) Vul4J was chosen for its integrated PoV tests. In settings lacking such tests, our method could be extended to generate or suggest them; (ii) Multi-file scenarios are not yet supported. Extending to them requires solving prompt scope and output integration challenges; and (iii) While tested on Java, the PCT-SM pipeline can be adapted to other languages. For interpreted environments (e.g., Python), compilation steps may be skipped or replaced.

## Acknowledgment

This work was supported by the Financiadora de Estudos e Projetos (FINEP), under Contract 01.22.0615.02, and by AppSecAI, Inc.

## References

Ahmad, B., Thakur, S., Tan, B., Karri, R., and Pearce, H. (2024). On hardware security bug code fixes by prompting large language models. *IEEE Transactions on Information Forensics and Security*.

Britton, T., Jeng, L., Carver, G., Cheak, P., and Katzenellenbogen, T. (2013). Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Bui, Q.-C., Scandariato, R., and Ferreyra, N. E. D. (2022). Vul4j: A dataset of reproducible java vulnerabilities geared towards the study of program repair techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 464–468.

Chen, B., Zhang, Z., Langrené, N., and Zhu, S. (2023). Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., and Tan, S. H. (2023). Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE.

Gu, Z., Barr, E. T., Hamilton, D. J., and Su, Z. (2010). Has the bug really been fixed? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 55–64.

Kulsum, U., Zhu, H., Xu, B., and d'Amorim, M. (2024). A case study of llm for automated vulnerability repair: Assessing impact of reasoning and patch validation feedback. In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, pages 103–111.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719.

Le, T. K., Alimadadi, S., and Ko, S. Y. (2024). A study of vulnerability repair in javascript programs with large language models. In *Companion Proceedings of the ACM on Web Conference 2024*, pages 666–669.

Lelis, C. A. S. (2025). Sop experiments results data. `https://github.com/ClaudioLelis/SoP-experiments-results-data`. Accessed: 2025-07-29.

Liu, P., Wang, H., Zheng, C., and Zhang, Y. (2024). Prompt fix: Vulnerability automatic repair technology based on prompt engineering. In *2024 International Conference on Computing, Networking and Communications (ICNC)*, pages 116–120. IEEE.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Morrison, D. R., Jacobson, S. H., Sauppe, J. J., and Sewell, E. C. (2016). Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102.

Nong, Y., Aldeen, M., Cheng, L., Hu, H., Chen, F., and Cai, H. (2024). Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. *arXiv preprint arXiv:2402.17230*.

Pearce, H., Tan, B., Ahmad, B., Karri, R., and Dolan-Gavitt, B. (2023). Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Vaswani, A. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE.

Wu, Y., Jiang, N., Pham, H. V., Lutellier, T., Davis, J., Tan, L., Babkin, P., and Shah, S. (2023). How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1282–1294.

Xia, C. S. and Zhang, L. (2023). Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*.

Zhang, L., Zou, Q., Singhal, A., Sun, X., and Liu, P. (2024a). Evaluating large language models for real-world vulnerability repair in c/c++ code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics*, pages 49–58.

Zhang, Q., Fang, C., Xie, Y., Ma, Y., Sun, W., and Chen, Y. Y. Z. (2024b). A systematic literature review on large language models for automated program repair. *arXiv preprint arXiv:2405.01466*.

Zhang, Q., Zhang, T., Zhai, J., Fang, C., Yu, B., Sun, W., and Chen, Z. (2023a). A critical review of large language model on software engineering: An example from chatgpt and automated program repair. *arXiv preprint arXiv:2310.08879*.

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., and Wang, R. (2023b). A survey on language models for code. *arXiv preprint arXiv:2311.07989*.