## APPENDIX

### PSEUDO-ALGORITHM FOR PATCH GENERATION

The pseudocode in Algorithm 1 clarify how the pattern-matching methodology works to generate patches. This algorithm outlines the logic behind our automated patching strategy to remediate a vulnerable pattern.

**Algorithm 1:** Pattern-matching approach to automatically patch vulnerable code

**Input:** Python code under analysis
**Output:** Securely patched version of the original code

```
Read the input Python code;
Start executing the rule set;
foreach rule in the set do
    // Search for deprecated functions
    if a deprecated function is found then
        Replace the deprecated function with its safe alternative;
    else
        // Search for external input functions
        if an external input is found then
            Extract the variable name storing the input;
            Track its usage in the code;
            Search for validation, escaping, or sanitization applied to the input;
            if no validation or sanitization is detected then
                Insert input validation logic;
                Insert escape or sanitization functions;
                Insert conditional checks (e.g., if statements);
                Insert necessary import statements for the added functions;
// Post-processing
Use VS Code APIs to:;
    Add import statements at the top of the file;
    Adjust indentation to match existing code style;
```

The tool iterates through a predefined set of rules. Firstly, each rule uses regular expressions to search for vulnerability patterns to remediate. As a first step, the rules attempt to identify the presence of potentially dangerous source functions, which fall into two main categories: *deprecated* or *misused functions* (e.g., `yaml.load()` or `app.run(debug=True)`), and *external input functions* (i.e., functions that read input from an external source, such as `input()`, `request.args.get()`, etc.). If a *deprecated* or *misused function* is detected, the rule immediately triggers the patching process by replacing it with a safe alternative (e.g., `yaml.safe_load()`, `app.run(debug=False, use_debugger=False, use_reloader=False)`). In contrast, if an *external input function* is found, the rule extracts the name of the variable receiving the input and analyzes how it is used within the code. Specifically, it checks whether the input is validated, escaped, or sanitized using additional regex-based inspections. If no such protections are found, the rule automatically applies a multi-step patch. This includes inserting appropriate validation logic, escape or sanitization functions, conditional checks (e.g., `if` statements), and the corresponding `import` statements required to support the added secure code. Finally, in a post-processing step, the tool leverages VS Code APIs to ensure that `import` statements are correctly positioned at the top of the file and that indentation matches the original code style.

### LIMITATIONS AND FUTURE WORK

**Language Generalization.** *PatchitPy* currently targets Python, raising questions about its applicability to other programming languages with different syntax and vulnerability patterns. Although the rule-based methodology, grounded in regular expressions and pattern extraction, is inherently adaptable, generalizing to languages like C/C++ introduces challenges. These include more verbose and structurally diverse constructs that complicate rule design. Future work will explore extending *PatchitPy* to additional languages, with expert-driven rule development ensuring accuracy and relevance.

**Rule-based Approach and Edge Cases.** Relying on manually crafted regular expression rules introduces inherent limitations. Rules derived from a finite set of vulnerable examples may be overly generic or overly specific, impacting accuracy on unseen patterns. Furthermore, pattern-matching struggles with complex vulnerabilities requiring deeper semantic understanding or involving unconventional constructs. Examples include vulnerabilities spread across multiple functions/files, deeply nested data flows, or atypical coding styles. Although validation was performed on diverse CWEs and OWASP categories [1], [2], future work will focus on systematically identifying and addressing such failure cases, potentially by integrating hybrid techniques, such as lightweight semantic analysis.

**Adaptation to Evolving Threats.** Another limitation concerns the tool's ability to remain current as new vulnerabilities and secure coding standards emerge. While *PatchitPy*'s modular rule infrastructure facilitates updates, new rules must still be manually crafted by experts. This restricts immediate protection against zero-day exploits or unconventional patterns. Future directions include automating the mining of vulnerability databases (e.g., CWEs) and incorporating expert-in-the-loop processes to continuously expand and refine the rule set, ensuring ongoing relevance in an evolving security landscape.

**Evaluation Scope.** Our evaluation, based on 609 samples generated from 203 natural language prompts using the SecurityEval and LLMSecEval datasets [1], [3], provides a diverse and realistic benchmark. However, the dataset may not fully capture the variability of real-world development scenarios. Although prompts cover a wide range of tasks and vulnerabilities, some corner cases may be underrepresented. Future

evaluations will expand the dataset and validation scenarios to further stress-test the tool under more diverse and challenging conditions.

## REFERENCES


[1] Security & Software Engineering Research Lab at University of Notre Dame, "SecurityEval," https://github.com/s2e-lab/SecurityEval, 2023.

[2] Pearce et al., "Copilot CWE Scenarios Dataset," https://zenodo.org/records/5225651, 2023.

[3] SoftSec Institute, "LLMSecEval," https://github.com/tuhh-softsec/LLMSecEval/, 2023.