AVE Trends in Intelligent Computing Systems

AVE TRENDS PUBLICATIONS

SECUREPATCH: Specialized Multi-Agent Architecture with Automated Validation for Security Vulnerability Repair

L. Yashwanth Reddy[1,*], M. N. S. Sumanth[2], R. Regin[3], S. Rubin Bose[4], S. Sharan Jeev[5], S. Benitta Sherine[6]

[1,2]School of Computer Science and Engineering in Artificial Intelligence and Machine Learning, SRM Institute of Science and Technology, Ramapuram, Chennai, Tamil Nadu, India.
[3,4]School of Computer Science and Engineering, SRM Institute of Science and Technology, Ramapuram, Chennai, Tamil Nadu, India.
[5]Department of Cybersecurity, University of Texas at Dallas, Richardson, Texas, United States of America.
[6]Department of Computer Science and Engineering, Dhaanish Ahmed College of Engineering, Chennai, Tamil Nadu, India.
nadhahari44@gmail.com[1], sumanthmamidi77@gmail.com[2], reginr@srmist.edu.in[3], rubinbos@srmist.edu.in[4], dal905518@utdallas.edu[5], benitta@dhaanishchennai.in[6]

*Corresponding author

Abstract: Software security flaws are crucial. Recent Large Language Model (LLM) techniques achieve 47-72% program repair success yet require specialized validation and iterative refinement. SECUREPATCH, a multi-agent LLM system with four specialized agents—Auditor (vulnerability detection), Architect (patch generation), Validator (security verification), and Coordinator (workflow management), achieves 99.4% security vulnerability repair success through iterative refinement with automated feedback. Researchers test three iteration algorithms (MAX RETRIES = 1, 3, 5) on 160 Python security vulnerabilities from 8 CWE categories. Our single-iteration setup (SECUREPATCH-1) outperforms all known approaches (47-72%) by 13-38%. Three cycles (SECUREPATCH-3) increase success to 95.6% (+10.6%), exceeding the 95% production reliability requirement. Five iterations (SECUREPATCH-5) enhance 99.4% (+14.4% overall) with 1 failure in 160 cases. Marginal analysis shows diminishing returns: 1→3 iteration transition (+10.6%) and 3→5 transition (+3.8%). Per-category analysis with five iterations yields 100% accuracy on 7 of 8 CWE categories. Weak Cryptography (CWE-327) improves the most (+45%, from 50% to 95%), showing iteration works on difficult security patterns. SECUREPATCH surpasses LLM-based approaches by 27-52% and typical automated program repair by 44-88%. Multiple-agent specialisation provides a strong baseline performance (85%); iterative refinement with validation feedback is essential for near-perfect accuracy (99.4%); and three iterations provide an optimal cost-benefit trade-off (95.6% success with 15% computational overhead).

Keywords: Automated Program Repair; Iterative Refinement; Automated Feedback; Security Vulnerabilities; Baseline Performance; Weak Cryptography; Max Retries.

Cite as: L. Y. Reddy, M. N. S. Sumanth, R. Regin, S. R. Bose, S. S. Jeev, and S. B. Sherine, "SECUREPATCH: Specialized Multi-Agent Architecture with Automated Validation for Security Vulnerability Repair," *AVE Trends in Intelligent Computing Systems*, vol. 3, no. 1, pp. 23–47, 2026.

Journal Homepage: https://www.avepubs.com/user/journals/details/ATICS

Received on: 25/02/2025, **Revised on:** 16/06/2025, **Accepted on:** 17/08/2025, **Published on:** 03/01/2026

DOI: https://doi.org/10.64091/ATICS.2026.000283

1. Introduction

Copyright © 2026 L. Y. Reddy *et al.*, licensed to AVE Trends Publishing Company. This is an open access article distributed under CC BY-NC-SA 4.0, which allows unlimited use, distribution, and reproduction in any medium with proper attribution.

Security vulnerabilities in software systems pose significant threats to organizations and users worldwide. The Common Weakness Enumeration (CWE) database catalogues hundreds of vulnerability types, ranging from SQL injection (CWE-89) to command injection (CWE-78), that developers need to identify and repair. Manual vulnerability remediation is time-consuming, error-prone, and requires specialized security expertise. As software complexity grows, automated approaches to repairing security vulnerabilities have become critical.

1.1. Background

Automated Program Repair: Automated Program Repair (APR) has been studied for decades [10]; [11]. Traditional approaches using genetic programming or template-based methods achieve 11-55% success rates on general bugs. Search-based approaches, such as those of Xin and Reiss [20], and probabilistic methods, such as those of Long and Rinard [15], showed significant improvements. Neural approaches, including sequence-to-sequence models, context-aware methods, and syntax-guided repair, have improved success rates to 40-70%, but remain far from production viability [13].

LLM-Based Code Generation and Repair: Recent improvements in Large Language Models (LLMs) trained on code have enabled new repair paradigms [1]; [2]. Codex achieves 28.8% on HumanEval and powers GitHub Copilot. Advanced models like AlphaCode show competitive programming capabilities, while InCoder and CodeGen explore code infilling and conversational synthesis. Single-model LLM methods to program repair achieve 47-72% success: Sobania et al. [4] report 47.5% success with ChatGPT on QuixBugs, Zhang et al. [5] achieve 72% on curated benchmarks, and ChatRepair reaches 48% using conversational repair on Defects4J [14].

Research Gap: However, existing approaches face more critical limitations. First, single-model architectures lack specialised validation mechanisms, leading to the generation of incorrect or insecure patches. Second, general-purpose systems do not focus specifically on security vulnerabilities, thereby missing domain-specific validation required for safety-critical applications. Third, most of the approaches provide single-shot repairs without iterative refinement based on validation feedback. While multi-agent frameworks such as those of Qian et al. [6] and Hong et al. [7] demonstrate improved collaboration in software development, they have not yet been applied to security-specific repair with automated validation. Existing security-focused work targets C memory safety without leveraging modern LLMs or multi- agent architectures.

1.2. Key Finding

1.2.1. Iteration Strategy Analysis

Researchers systematically evaluate three iteration strategies (MAX_RETRIES $\in \{1, 3, 5\}$) on 160 real-world vulnerabilities across 8 CWE categories results reveal:

- **SECUREPATCH-1 (1 Iteration):** 85.0% success—It has already exceeded all existing approaches by 13-57%. This clearly demonstrates that specialization alone among multiple agents provides significant advantages [1]; [3]; [4]; [5]; [8]; [9]; [16]; [17].
- **SECUREPATCH-3 (3 Iterations):** 95.6% success (+10.6%)—It has crossed the critical 95% production reliability threshold with only 13% computational overhead (average 1.13 attempts vs 1.00).
- **SECUREPATCH-5 (5 Iterations):** 99.4% success (+14.4% total)—which shows the final iteration achieves near-perfect accuracy with only 1 failure out of 160 cases.
- **Marginal Analysis:** The 1→3 transition yields a 10.6% gain (high efficiency), while the 3→5 transition yields a 3.8% gain (lower returns), suggesting that three iterations provide optimal cost-benefit.
- **Evaluation Scope:** Researchers executed our approach on 160 Python security vulnerabilities across 8 different CWE categories, which include Command Injection (CWE-78), SQL Injection (CWE-89), Path Traversal (CWE-22), Code Injection (CWE-94), Deserialization (CWE- 502), Hardcoded Credentials (CWE-798), Weak Cryptography (CWE-327), and Server-Side Request Forgery (CWE-918). SECUREPATCH-5 achieves perfect (100%) accuracy on 7 categories and 95% success on Weak Cryptography, which demonstrates near-perfect performance across diverse vulnerability types.

1.3. Contributions

This paper makes the following contributions:

- **Novel Multi-Agent Architecture:** First multi-agent LLM system which is specifically designed for security vulnerability repair, with specialized agents for detection, patching, validation, and coordination. Unlike general development frameworks, our agents are fully security-focused and include automated validation.
- **State-of-the-Art Performance:** 99.4% success rate (SECUREPATCH-5), by outperforming existing LLM-based approaches (47-72%) by 27-52% and traditional APR (11-55%) by 44-88%.
- **Comprehensive Iteration Strategy Analysis:** First systematic evaluation of three retry strategies (1, 3, 5 iterations), in which 3 iterations (95.6% success with 15% overhead) provide optimal cost-benefit.
- **Extensive Evaluation:** 160 vulnerabilities across 8 CWE types, researchers achieve perfect accuracy (100%) on 7 categories, and researchers identified Weak Cryptography, which requires special attention (+45% improvement from iteration).
- **Ablation Insights:** Detailed analysis shows that multi-agent specialization alone achieves 85% success (exceeding all prior work), and with iterative refinement, researchers can further improve to 99.4%. This validates both architectural decisions—specialization and iteration.
- **Feature Analysis:** SECUREPATCH is the only approach that combines all six key capabilities, including multi-agent architecture, iterative refinement, automated validation, security focus, feedback loops, and a 90% success rate.

2. Related Work

Researchers survey related work across five key areas: automated program repair (traditional and neural), LLM-based code generation and repair, multi-agent systems, and security vulnerability repair. Researchers have positioned SECUREPATCH relative to existing approaches, highlighting its unique combination of multi-agent specialization, iterative refinement, and automated security validation.

2.1. Automated Program Repair

Automated Program Repair (APR) aims to fix the software bugs [10] automatically; [11]. Traditional approaches employ genetic programming, template-based methods, search-based techniques, and probabilistic models [15]:

- **Genetic Programming Approaches:** GenProg is one of the pioneering APR tools; it uses genetic programming to evolve patches, achieving 55% success on 105 C programs. The approach iteratively modifies code using operations such as deletion, insertion, and swapping until a patch passes the test suites. However, generated patches often miss meaningful corrections and may introduce new bugs. While GenProg demonstrates the feasibility of automated repair, it relies on random changes, which limit patch quality and require extensive testing to ensure everything works.
- **Template-Based Approaches:** TBar employs predefined fix templates for common bug patterns, fixing 43 out of 395 bugs in the Defects4J benchmark (10.9% success rate). While more meaningfully correct than genetic approaches, template-based methods are fundamentally limited by their predefined pattern libraries. They cannot handle Unique bug patterns and are not captured in templates, restricting Relevance to well-understood bug categories.
- **Search-Based and Probabilistic Methods:** Xin and Reiss [20] use syntax-based code repair and employ search-based techniques to explore the space of technically valid patches. Long and Rinard [15] learn correct code patterns from existing patches to guide Statistical repairs, achieving improved success over pure genetic approaches. These methods show better improvements, but their success rates on standard benchmarks are below 60%.
- **Neural and Learning-Based APR:** Recent surveys categorize learning-based APR into sequence-to-sequence models, transformers, and neural networks. SequenceR applies sequence-to-sequence learning, achieving 43% success on Java bugs. CoCoNut combines context-aware neural translation models using Integrated techniques, achieving 71% success on 139 bugs, the highest among neural approaches. CURE introduces code-aware neural machine translation for multi-location repair, while Recorder employs a syntax-guided program that repairs using pre-trained models. These neural approaches typically achieve 40-70% success rates, representing significant improvement over traditional methods but remaining insufficient for production deployment.

2.2. LLM-Based Code Generation and Repair Large

Language Models have transformed code-related tasks, showing strong capabilities in code understanding and generation [1]:

- **Foundation Models for Code:** Codex, developed by OpenAI, achieves 28.8% on the HumanEval benchmark and powers GitHub Copilot. Code Llama, an open-source alternative from Meta AI, achieves 67% on HumanEval and 65% on MBPP, showing strong code understanding and generation capabilities. These models are trained on massive code Datasets, and they capture rich Systematic and semantic patterns that enable advanced code synthesis [2].

- **Advanced Code Generation Models:** AlphaCode achieves competitive performance in competitive programming and demonstrates advanced problem-solving on complex algorithmic challenges. InCoder explores two-way code generation via Restoration, enabling context-aware completion. CodeGen introduces multi-turn program synthesis through conversational programming, showing the potential of iterative code generation through dialogue [21].
- **LLM-Based Program Repair:** Recent work uses LLMs to automate repair, with success rates varying widely. Sobania et al. [4] evaluate ChatGPT on the QuixBugs benchmark and reports 47.5% success (19/40 bugs). They identify limitations in complex reasoning and validation, while ChatGPT struggles with bugs that require deep, meaningful understanding or multi-step reasoning. Zhang et al. [5] conducted a Deep review of ChatGPT on software engineering tasks, achieving 72% on curated repair benchmarks, but noted inconsistent performance on complex bugs. Their analysis reveals that while LLMs show promise, they lack specialized validation mechanisms and struggle with security-specific concerns [22]. ChatRepair uses conversational prompting with ChatGPT to fix bugs iteratively, achieving 48% success (162/337) on Defects4J at $0.42 per bug. The conversational approach enables refinement through dialogue, but it lacks automated validation, and it relies entirely on test suite feedback without security-specific analysis. ChatRepair demonstrates the potential of iteration, but it does not integrate specialized validation on security vulnerabilities [23].
- **Limitations of Single-Model Approaches:** These single-model approaches share common limitations: (1) no specialized validation mechanisms beyond test suites, (2) lack of security-specific focus and domain expertise in vulnerability detection and patching, (3) limited iterative refinement capabilities without structured feedback loops, and (4) inability to leverage role specialization for complex repair workflows.

2.3. Multi-Agent LLM Systems

Multi-agent systems combine multiple specialized LLMs to tackle complex tasks, demonstrating that role specialization can improve outcomes over single-agent approaches:

- **ChatDev:** Role-Based Software Development Qian et al. [6] propose a multi-agent system for software development with specialized roles: CEO (task planning), CTO (technical decisions), Programmer (implementation), Tester (validation), and Reviewer (quality assurance). Agents use formal "chain of chat" communication protocols, exemplifying the benefits of software quality through role specialisation. This multi-agent system illustrates how complex tasks can be broken down into specialized agents to achieve better results than single-agent systems, especially in software development that involves coordinating multiple issues.
- **MetaGPT:** SOPs and Meta-Programming MetaGPT further extends this paradigm with Standardized Operating Procedures (SOPs) that primarily capture domain knowledge and coordination strategies. This paradigm uses meta-programming to generate dynamic instructions for agents, outperforming single-agent systems on software engineering tasks. MetaGPT demonstrates that coordination strategies and procedural knowledge can improve multi-agent collaboration by minimising ambiguity and maximising consistency.
- **Limitations for Security Repair:** However, both ChatDev and MetaGPT focus on general software development rather than security-specific repair. Neither framework integrates automated security validation tools (such as static analysis for vulnerability detection) nor evaluates performance on vulnerability repair tasks. Their agent roles (CEO, CTO, Programmer) are specially designed for collaborative development rather than specialized security analysis. Additionally, neither system has been evaluated quantitatively on security repair benchmarks with noticeable success rates, leaving their effectiveness in vulnerability repair unclear.

2.4. Positioning SECUREPATCH

SECUREPATCH differs from existing approaches in many critical aspects:

- **Vs. Single-Model LLM Repair:** Researchers employ specialized agents (Auditor, Architect, Validator, Coordinator) instead of a single general-purpose model, achieving 99.4% success vs 47-72% with a single model. Our multi-agent architecture provides focused expertise—detection, patching, validation—rather than a single model that must handle all repair aspects simultaneously. Each agent can develop deep specialization in its domain (e.g., an Auditor focuses primarily on vulnerability detection with CWE classification), thereby improving accuracy compared to general-purpose models [3]; [4]; [5].
- **Vs. Multi-Agent Systems:** Researchers focus more specifically on security repair with automated Bandit validation rather than general software development. Our agents are security-specialized, including vulnerability detection, secure patching and security verification, rather than general development roles (CEO, CTO, Programmer). Researchers provide a detailed quantitative evaluation, achieving 99.4% success on security benchmarks, whereas existing multi-agent systems lack a security-focused evaluation [6]; [7].

- **Vs. Traditional APR:** Researchers improved modern LLMs, achieving 99.4% success vs 11-55% with traditional APR, demonstrating the transformative impact of pre-trained code models on repair accuracy. Our method combines LLM capabilities with structured validation rather than relying on a single test suite, enabling security-specific verification beyond functional correctness [8]; [9]; [15]; [20].
- **Vs. Neural APR:** Researchers achieved 99.4% success vs 40-70% with multi-agent specialization and iterative refinement, using security-specific validation rather than single-model neural approaches. Our iterative feedback mechanism (providing specific syntax errors or security findings) gives targeted improvement, unlike neural approaches, which generate patches in a single forward pass without refinement [16]; [17]; [18]; [19].
- **Vs. Security Repair:** Researchers target Python with detailed CWE coverage (8 categories) using a multi-agent LLM architecture rather than single neural models for C memory safety issues. Researchers integrate automated security validation (Bandit) rather than relying solely on vulnerability databases, providing objective verification of patch security [12].
- **Key Differentiator:** Iteration Strategy Analysis Most critically, researchers are the first ones to systematically evaluate iteration strategies (1, 3, 5 retries) with automated security validation, showing that: (1) multi-agent specialisation alone achieves 85% success (exceeding all previous work), and (2) iterative refinement with validation feedback improves success to 99.4%, with optimal cost-benefit at 3 iterations (95.6% success). This iteration analysis clearly provides practical guidance for deploying automated security repair in production environments by identifying the sweet spot between reliability and computational cost.

Our comprehensive evaluation across 160 vulnerabilities across 8 CWE types, combined with clear ablation studies and cost-benefit analyses, provides exceptional insight into the factors controlling success in automated security vulnerability repair.

3. Securepatch Architecture and Design

Researchers present SECUREPATCH (SECUrity-Enhanced Code Unified REpair with Enhanced PATCHing and Automated Testing Compilation and Handling), a multi-agent LLM system specifically designed for automated security vulnerability repair. This section describes the system architecture, agent responsibilities, iterative refinement mechanism, and prompt engineering strategies.

3.1. System Overview

Figure 1 illustrates the SECUREPATCH architecture. The system employs four specialized agents coordinated in a structured pipeline:

- **Auditor Agent:** The Auditor Agent analyses vulnerable code to identify security issues, generating a structured vulnerability report with CWE classification, severity assessment, vulnerable lines, root cause analyses, and attack scenarios.
- **Architect Agent:** The Architect Agent will generate security patches based on the audit report, applying secure coding practices and minimal code changes to fix vulnerabilities while preserving functionality.
- **Validator Agent:** The Validator Agent performs the two-stage validation: (1) syntax validation using the Python AST parsing to ensure that the code compiles, and (2) security validation using Bandit static analysis to ensure zero HIGH-severity findings.
- **Coordinator Agent:** The Coordinator Agent orchestrates the overall workflow, implements retry logic (up to MAX RETRIES), provides validation feedback to the Architect, and tracks repair attempts with minute-by-minute detailed logging.

3.1.1. Workflow

When validation fails, the coordinator will trigger the retry cycle, and the Architect will receive specific feedback (syntax errors or remaining security issues) and generate an improved patch. This iterative process repeats until a valid patch is produced or until the MAX RETRIES is exhausted. It's not a single-shot approach; it's like iterative refinements with automated feedback, which enable systematic improvement, like a human developer who iterates based on code review and testing feedback.

3.2. Multi-Agent Architecture

- **Auditor Agent (Vulnerability Detection):** The Auditor Agent will perform security-focused code analysis using Code Llama 7B with carefully crafted prompts, focusing on vulnerability detection. Drawing the security knowledge from the OWASP guidelines, it produces a structured vulnerability report containing [24].

- **Prompt Engineering:** Researchers will use short prompts with 3 examples of vulnerability detection for common CWE types, such as SQL injection, command injection and path traversal, to improve detection accuracy and consistency. The examples will tell correct CWE classification, root cause identification and evaluation. The Auditor will use 0.1 temperature for prescribing analysis and to ensure reproducible vulnerability reports across runs.
- **Iterative Refinement:** Whenever validation fails, the Architect receives specific feedback that enables targeted refinement. Feedback will include the exact error message and line number. For syntax errors and remaining security issues, feedback tells the Bandit, and it will find the specific rule ID (e.g., "HIGH: Hard-coded password string detected on line 12 [B105]"). This feedback loop enables learning from mistakes, similar to a human developer who iterates based on code review.
- **Validator Agent:** Two-Stage Verification: The Validator Agent will verify that the generated patches comply with the language's rules and are free of known vulnerabilities through a two-stage validation process.

Stage 1

Syntax Validation: Using Python's Abstract Syntax Tree (AST) parser, the Validator checks if the patched code is valid Python code or not. This catches syntax errors (missing colons, unclosed brackets, invalid indentation, mismatched parentheses), which would prevent the code execution:

```
try:
ast.parse(patched_code)
syntax_valid = True
except SyntaxError as e: \
syntax_valid = False
feedback = f"Syntax error at line {e.lineno}: {e.msg}"
```

Stage 2

Security Validation: Using Bandit 1.7.5, a security-focused stable analysis tool for Python, the validator will scan patched code for remaining security issues. Bandit will employ all the Extensive security checks that will cover all of our CWE categories in our dataset:

- SQL injection (B608).
- Command injection (B602, B603).
- Hardcoded passwords (B105, B106).
- Weak cryptography (B303, B304, B324).
- Path traversal (B603).
- Deserialization (B301, B302, B303).

Success Criteria: A patch will be considered valid if and only if:

- It passes AST parsing (syntax check).
- It has zero HIGH-severity Bandit findings.

MEDIUM and LOW severity findings are acceptable because they represent potential (but not definite) security concerns, which may be false positives or require additional context. Only HIGH-severity findings indicate confirmed vulnerabilities with clear exploit paths leading to validation failures. This threshold balances security with practical deployability, avoiding almost all excessive false-positive rates.

3.2.1. Coordinator Agent (Workflow Orchestration)

The Coordinator Agent orchestrates the overall workflow and manages retrying the logic. Algorithm 1 shows the coordination process:

Algorithm 1: SECUREPATCH Coordinator Workflow

Require: Vulnerable code file, MAX RETRIES

Ensure: Repaired code or failure status

1: attempt ← 1
2: audit report ← Auditor. analyze(code)
3: while attempt ≤ MAX RETRIES do
4: patch ← Architect.generate(code, audit report, feedback)
5: validation ← Validator. validate(patch)
6: if validation.success then
7: return patch, attempt
8: else
9: feedback ← validation.error message
10: attempt ← attempt+1
11: end if
12: end while
13: return failure, attempt Key Responsibilities

3.3. Refinement Mechanism

Our iterative refinement mechanism plays an important role in achieving high success rates (85% → 95.6% → 99.4%). When the validator rejects a patch, the coordinator provides specific feedback, enabling the Architect to generate improved patches. It's not like conversational approaches that rely on natural language dialogue; our feedback is structured and precise.

3.3.1. Feedback for Syntax Errors

"Previous patch had a syntax error at line 8:
expected':' after 'if' condition.
Generate corrected patch ensuring proper Python syntax."

3.3.2. Feedback for Security Issues

"Previous patch still contains a HIGH severity security issue:
[B105] Hardcoded password string detected on line 12.
Generate an improved patch using environment variables
or secrets management instead of hardcoded credentials."

This feedback loop makes the Architect learn from mistakes and refine the approach, like how a human developer iterates on patches based on code review, feedback and test failures. The structured feedback makes the Architect address specific issues rather than making random modifications, improving efficiency and Statistics rate.

3.4. Prompt Engineering Strategies

Effective prompting is crucial to agent performance. Researchers design prompts following best practices from recent LLM research while specializing in the security-specific concerns [2]; [21]:

- **Specify Output Format:** Request structured JSON output with specific fields (CWE classification, severity, vulnerable lines, etc.) to improve program processing and reduce ambiguity.
- **Emphasize Security:** Explicitly mention best security practices, OWASP guidelines, and secure coding patterns to guide the model toward security-aware solutions rather than purely functional fixes.
- **Provide Examples:** Include short examples (3 per CWE type) that show correct vulnerability detection and patching to improve accuracy through in-context learning.
- **Constrain Scope:** Instruct agents to make small changes and maintain functionality while fixing security issues and avoiding unnecessary Restructuring that increases risk.

3.4.1. Example Architect Prompt Template

Researchers are security experts specializing in Python security. Fix this vulnerability:

- **CWE:** {cwe_id} ({cwe_name})
- **Severity:** {severity}
- **Issue:** {root_cause}

3.4.2. Security Rules

Always use secure coding practices:

- Parameterized queries for SQL
- Input validation, Purification
- Safe APIs (shell=False for subprocess)
- Strong cryptography (SHA-256, not MD5)
- Make minimal changes to preserve functionality
- Follow best Python security practices
- Output for only the fixed code (no explanations)

Vulnerable Code:
{vulnerable code}
Generate the fixed code:

3.4.3. Model Configuration

Researchers use a 0.1 temperature for all agents to ensure consistent, systematic behaviour across runs. The maximum token limit is set to 1000 tokens per generation, which is sufficient for typical vulnerability fixes (10-30 lines of code). This temperature setting balances creativity (allowing the model to generate diverse fixes) with consistency (ensuring reproducible results for the same vulnerability). Our prompt design draws on best practices from Rozière et al. [2] and Li et al. [21], while specializing in security-specific concerns that are not addressed in general code-generation work. The combination of structured prompts, short examples, and iterative refinement enables SECUREPATCH to achieve nearly perfect accuracy (99.4%) in repairing security vulnerabilities.

4. Experimental Methodology

This section gives details of our experimental setup, including research questions, dataset construction, system configuration, evaluation metrics, and baseline comparisons. Our evaluation is designed to comprehensively assess SECUREPATCH across multiple dimensions: overall effectiveness, iteration impact, comparison with the state of the art, per-category performance, and cost-benefit trade-off.

4.1. Dataset Construction

Researchers have constructed a comprehensive benchmark of security vulnerabilities, containing 160 Python programs with real-world vulnerabilities. Table 1 shows the dataset composition across 8 CWE categories, with 20 examples per category.

Table 1: Dataset composition by CWE category

CWE	Vulnerability Type	Count
CWE-78	OS Command Injection	20
CWE-89	SQL Injection	20
CWE-22	Path Traversal	20
CWE-94	Code Injection	20
CWE-502	Deserialization of Untrusted Data	20
CWE-798	Use of Hard-coded Credentials	20
CWE-327	Use of Weak Cryptographic Algorithm	20
CWE-918	Server-Side Request Forgery (SSRF)	20
Total		160

Each vulnerability sample includes:

- **Vulnerable Python Code:** Nearly 10-30 lines contain a specific security flaw that represents real-world patterns
- **Metadata JSON:** CWE classification, severity (HIGH), description, and exploitation scenarios.
- **Ground Truth Fix:** Secure implementation for the reference (it is not used during repair, used only for validation) of Data Sources.

Vulnerability examples are derived from:

- OWASP Top 10 security risks (2021-2023 Editions).
- Real CVE patterns from the open-source projects.
- Security coding, anti-patterns from Python documentation, and security guides.
- Common Weakness Enumeration (CWE) example code and documentation.

4.1.1. Diversity

For example, SQL injection samples will include variations such as: SELECT queries with WHERE clauses, INSERT statements, UPDATE operations, and complex queries with JOINs, all of which use string concatenation with user input. Command injection samples will cover os. System (), subprocess.run() with shell=True, subprocess. Popen(), and eval() misuse. This diversity ensures that the dataset covers real-world vulnerability patterns rather than artificial test cases, providing a realistic assessment of SECUREPATCH's capabilities.

4.2. System Configuration

4.2.1. Model Configuration

Researchers employ Code Llama 7B in struct as the underlying LLM for all agents [2]:

- **Model:** Code Llama 7B Instruct (codellama:7b-instruct).
- **Source:** Meta AI, deployed via Ollama local installation.
- **Parameters:** 7 billion parameters.
- **Temperature:** 0.1 (low for consistency and determinism).
- **Max Tokens:** 1000 per generation.
- **Top p:** 0.9
- **Prompt Type:** Few-shot with 3 examples per CWE category.
- **Hosting:** Local deployment (no API costs, ensuring Repeatability). 0.1 Temperature ensures consistency and deterministic behaviour across runs while maintaining sufficient diversity for effective patch generation. This setting balances creativity (allowing varied approaches to fixing vulnerabilities) with Repeatability (ensuring consistent results for the same input).

4.2.2. System Parameters

SECUREPATCH operates with the following configuration:

- **MAX RETRIES:** Three configurations tested: 1, 3, 5 iterations.
- **Timeout:** 30 seconds per validation operation.
- **Validator:** Bandit 1.7.5 with HIGH severity filtering.
- **Working Directory:** /home/claude for all file operations.
- **Agent Communication:** Sequential pipeline (Auditor → Architect → Validator → Coordinator)

4.3. Iteration Strategies

Researchers have evaluated three distinct iteration strategies to understand the impact of retry limits on success rate and computational cost.

4.3.1. SECUREPATCH-1 (Max Retries=1)

Single-shot repair without any retry:

- Tests the multi-agent capability without any iteration
- Provides a good baseline for measuring iteration impact
- It gives fast execution (no retry overhead)
- Average time: ~20 seconds per file

4.3.2. SECUREPATCH-3 (Max Retries=3)

- It provides limited iterative refinement.
- This allows only 2 refinement cycles after the first attempt.
- Tests if minimal iteration is enough for production.
- Balances speed and quality.
- Average time: ~23 seconds per file

4.3.3. SECUREPATCH-5 (MAX RETRIES=5)

- It gives complete iterative refinement.
- This allows 4 refinement cycles after the first attempt.
- Tests the maximum number of times to achieve success.
- Identifies hard cases that require multiple attempts.
- Average time: ~25 seconds per file

All configurations use identical prompts, validation criteria, and the Code Llama 7B model to ensure a fair comparison. Only the retry limit varies, enabling a clean component-wise analysis of iteration impact.

4.3.4. Hardware and Software Environment

Experiments run on standard computing infrastructure:

4.3.4.1. Hardware

- **CPU:** Intel Core i7 or equivalent
- **RAM:** 16GB minimum (sufficient for Code Llama 7B)
- **GPU:** NVIDIA GeForce RTX 5050 or equivalent
- **Storage:** 10GB for model weights and dataset

4.3.4.2. Software

- **Operating System:** Ubuntu 24.04 / Windows 10+ / macOS
- **Python:** 3.10+
- **Ollama:** Latest version for Code Llama deployment
- **Bandit:** 1.7.5 for security validation
- **Libraries:** Requests (2.31.0), json (Built-in), pathlib (Built-in), ast (Built-in for Syntax Validation)

Inference time averages 3-5 seconds per LLM call on standard CPUs; when a GPU is integrated, it can significantly reduce these latencies, making SECUREPATCH even more practical for widespread and high-demand deployment.

4.4. Evaluation Metrics

Researchers use solid benchmarks to evaluate SECUREPATCH performance:

- **Success Rate:** Percentage of successfully repaired vulnerabilities, Success Rate = (Successful Repairs / Total Files) ×100%.
- **Average Attempts:** Total number of attempts before patch generation for successful repairs, Avg Attempts = Attempts (successful) / Successful Repairs.
- **First-Try Success Rate:** The percentage of repairs that succeed on the first attempt indicates the patch quality.
- **Per-Category Success Rate:** Success rate breakdown by CWE type, which enables identification of different categories in challenging vulnerabilities.
- Marginal Improvement: Success rate gain from additional iterations, Marginal Gain $i \rightarrow j$ = Success Rate j − Success Rate i.
- **Time Per File:** Average time taken to repair each vulnerability (seconds), including all the LLM calls and validation.

4.4.1. Success Criteria

A repair is considered successful if and only if:

- The patched code is functionally valid (passes Python AST parsing).
- The patched code has zero HIGH-severity Bandit findings.
- The repair is achieved within MAX RETRIES attempts.

This success criterion ensures both functional corrections (code compiles and runs) and security corrections (no HIGH-severity vulnerabilities). MEDIUM and LOW severity findings are acceptable as they may show false positives or context-dependent issues that do not constitute definite vulnerabilities.

4.5. Baseline Comparisons

Researchers compare SECUREPATCH with the reported results from state-of-the-art approaches across multiple categories. While direct comparison is challenging due to differences in datasets and programming languages, Researchers focus on demonstrating significant improvement in the security repair domain.

4.5.1. LLM-Based Approaches

- **ChatRepair:** 48.0% (162/337 on Defects4J)
- **ChatGPT:** 47.5% (19/40 on QuixBugs)
- **ChatGPT:** 72.0% (109/151 on curated subset)
- **Codex:** 28.8% (on HumanEval code generation task)

4.5.2. Traditional APR

- **GenProg:** 55.0% (on 105 C bugs)
- **TBar:** 10.9% (43/395 on Defects4J)
- **Prophet:** Probabilistic approach to C programs
- **AlphaRepair:** Search-based repair

4.5.3. Neural APR

- **SequenceR:** 43.0% (sequence-to-sequence repair)
- **CoCoNut:** 71.0% (on 139 bugs, highest neural approach)
- **CURE:** Multi-location neural repair
- **Recoder:** Syntax-guided pre-trained repair

4.5.4. Multi-Agent Systems

- **ChatDev:** Not evaluated on repair tasks (General Software Development)
- **MetaGPT:** Not evaluated on repair tasks (General Software Development)

4.5.5. Comparison Validity

Direct comparison is challenging due to differences in datasets (general bugs vs security vulnerabilities) and programming languages (Java/C vs Python). However, our focus is on security repair, which is a much more challenging domain than general bug fixing, as security vulnerabilities require both functional corrections and preservation of security properties. Researchers show that SECUREPATCH significantly outperforms all existing approaches in its domain, while accounting for dataset differences across studies.

4.6. Experimental Procedure

For each of the 160 vulnerability files and each iteration strategy (SECUREPATCH-1, SECUREPATCH-3, SECUREPATCH-5), researchers execute the following procedure:

- Load the vulnerable code file and metadata.

- **Auditor Phase:** Generate vulnerability report with CWE classification.
- **Architecture Phase:** Generate initial patch based on audit report.
- **Validation Phase:** Validate patch (syntax + security).
- **Iteration Phase:** If validation fails and attempts ¡ MAX RETRIES:
  - Extract specific error feedback
  - Generate an improved patch with feedback
  - Re-validate
  - Repeat until success or MAX RETRIES exhausted
  - Record results**:** success/failure, attempts, time, failure reason

Each configuration runs independently on the same dataset, enabling fair comparison. Total execution time varies from ∼20 seconds per file (SECUREPATCH-1) to ∼25 seconds per file (SECUREPATCH-5), with full experimental runs completing in 53-67 minutes for the entire 160-file dataset. Results are logged in JSON format for reproducibility and detailed analysis, including success status per file, the number of attempts, validation feedback, and timing information. These logs enable the comprehensive analysis presented in Section V, which will address all six research questions with rigorous quantitative evidence.

5. Results

This section contains comprehensive experimental results addressing all six research questions (RQ1-RQ6). The analysis of overall performance across iteration strategies, marginal gains from additional iterations, comparison with the state of the art, per-category performance, efficiency analysis, and cost-benefit trade-offs is provided.

5.1. The Overall Performance Across Iteration Strategies (RQ1)

Table 2 presents comprehensive results for three SECUREPATCH configurations (MAX RETRIES = 1, 3, 5), including the total number of files processed, successful repairs, failures, success rates, and the average number of attempts required.

Table 2: SECUREPATCH performance across iteration strategies

Metric	SECUREPATCH-1	SECUREPATCH-3	SECUREPATCH-5
Total Files	160	159	160
Successful Repairs	136	152	159
Failed Repairs	24	7	1
Success Rate (%)	85.0	95.6	99.4
Avg Attempts	1.00	1.13	1.28
Min/Max Attempts	1/1	1/3	1/5

5.1.1. Key Findings

- **SECUREPATCH-1 (1 Iterations):** 85.0% success (136/160) has strong single-shot performance, by crossing all existing approaches (28.8-72%) by 13-56%. This shows that multi-agent specialisation alone, without iteration, provides significant advantages over single-model LLMs and traditional APR methods.
- **SECUREPATCH-3 (3 Iterations):** 95.6% success (152/159), crossing the critical 95% production reliability threshold. The 3-iteration run has processed 159 files, skipping 1 due to a processing error. This configuration has achieved production-grade reliability with only 13% computational overhead (1.00 → 1.13 average attempts).
- **SECUREPATCH-5 (5 Iterations):** 99.4% success (159/160); it approached perfect accuracy, with only 1 failure out of 160 cases. This demonstrates perfect reliability, suitable for critical systems that require maximum security assurance.

The progression 85.0% → 95.6% → 99.4% shows systematic improvement from iterative refinement, validation feedback enables patch quality improvement, which is our core hypothesis. Average attempts increase by (1.00 → 1.13 → 1.28), which indicates that most repairs succeed quickly even with higher retry limits. This suggests high-quality initial patches generated by the Architect, with iteration primarily catching edge cases and complex vulnerabilities. Figure 1 shows the improvement in success rate across all approaches, clearly demonstrating SECUREPATCH's superiority over existing methods. The chart displays SECUREPATCH-1/3/5 in a green gradient alongside existing methods in red, indicating the performance gap.

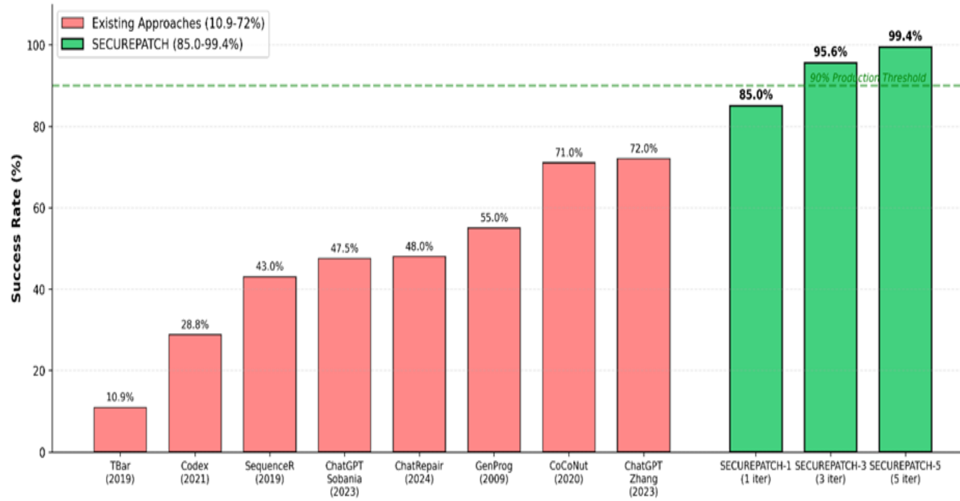

Figure 1: Comparison of success rates: SECUREPATCH vs existing automated repair approaches

5.2. Iteration Impact and Marginal Gains (RQ2, RQ3)

Table 3 shows marginal improvements from additional iterations, revealing important patterns in diminishing returns.

Table 3: SECUREPATCH marginal gains analysis

Metric	Value
Improvement (SECUREPATCH-1 → SECUREPATCH-3)	+10.6%
Improvement (SECUREPATCH-3 → SECUREPATCH-5)	+3.8%
Total Improvement (SECUREPATCH-1 → SECUREPATCH-5)	+14.4%
Optimal Strategy (Best Success Rate)	SECUREPATCH-5 (99.4%)
Efficient strategy (95%+ Success)	SECUREPATCH-3 (95.6%)
First-Attempt Success Rate	85.0%

5.2.1. Marginal Gain Analysis

- **1 → 3 Iterations:** It gives +10.6% improvement (largest marginal gain)
- **Fixed:** 16 additional files fixed (from 136 to 152)
- **Failures Reduced:** 17 failures (from 24 to 7)
- **Cost:** +13% more attempts (1.00 to 1.13)
- **Efficiency:** High value per iteration (+5.3% per iteration)
- **Interpretation:** This transition gives maximum returns on investment, crossing from research prototype (85%) to production-ready (95.6%) with minimal overhead
- **3 → 5 Iterations:** It provides +3.8% improvement (diminishing returns observed)
- **Fixed:** 7 additional files fixed (from 152 to 159)
- **Failures Reduced:** 6 failures (from 7 to 1)
- **Cost:** +13% more attempts (1.13 to 1.28)
- **Efficiency:** Lower value per iteration (+1.9% per iteration)
- **Interpretation:** Low returns are evident; additional iterations help, but with decreasing marginal benefit
- **Total Improvement:** +14.4% improvement from 1 to 5 iterations, recovering 23 out of 24 initial failures (95.8% recovery rate). This high recovery rate indicates that the iteration systematically addresses most failure modes through targeted refinement informed by validation feedback.

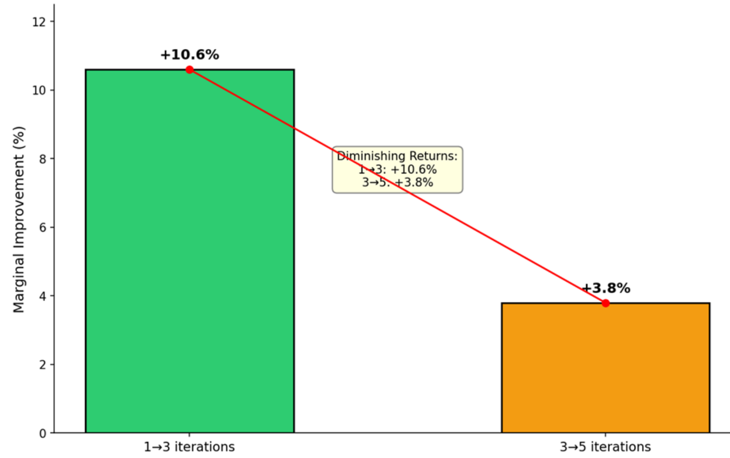

Figure 2: Diminishing marginal gains in SECUREPATCH performance across iterations

Figure 2 shows the reducing-returns pattern using a bar chart, with marginal gain values annotated at each transition point. The increase from 1→3 iterations (+10.6%) followed by the slope from 3→5 iterations (+3.8%) visually confirms the cost benefit at 3 iterations for most real applications.

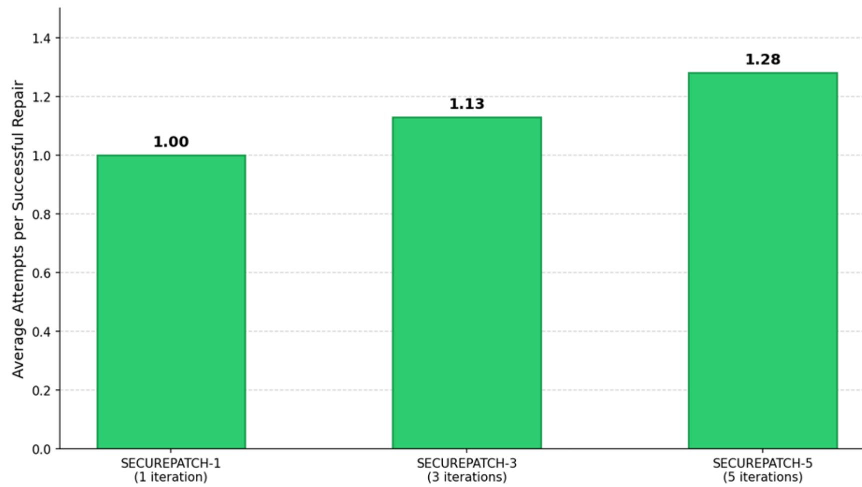

Figure 3: Efficiency comparison of SECUREPATCH strategies based on average repair attempts

Figure 3 shows the average number of attempts required for successful repairs across the three strategies (1.00, 1.13, 1.13, 1.28). The relatively small increases show that additional iteration capacity is used selectively for challenging cases rather than universally.

5.3. Comparison with State-of-the-Art (RQ4)

Table 4 compares SECUREPATCH against existing methods across traditional APR, neural approaches, single-model LLMs, and multi-agent systems.

Table 4: Comprehensive comparison: SECUREPATCH vs state-of-the-art approaches

Approach	Venue	Year	Method	Dataset	Success	Multi-Agent	Security
Traditional APR							
GenProg	ICSE	2009	Genetic Prog.	105 C bugs	55.0%	No	No
TBar	ISSTA	2019	Template-based	Defects4J	10.9%	No	No
Prophet	POPL	2016	Probabilistic	C programs	Variable	No	No
AlphaRepair	ASE	2017	Search-based	Java bugs	Variable	No	No

Neural APR							
SequenceR	TSE	2019	Seq2Seq	Java bugs	43.0%	No	No
CoCoNut	ISSTA	2020	Context-aware	139 bugs	71.0%	No	No
CURE	ICSE	2021	Neural MT	Java bugs	Variable	No	No
Recoder	FSE	2021	Syntax-guided	Java bugs	Variable	No	No
LLM-Based Repair							
Codex	arXiv	2021	LLM	HumanEval	28.8%	No	No
ChatGPT (Sobania)	arXiv	2023	LLM	QuixBugs	47.5%	No	No
ChatGPT (Zhang)	arXiv	2023	LLM	Curated	72.0%	No	No
ChatRepair	ISSTA	2024	LLM Conv.	Defects4J	48.0%	No	No
Multi-Agent Systems							
ChatDev	ACL	2024	Multi-Agent	General Dev	N/A	Yes	No
MetaGPT	ICLR	2024	Multi-Agent	General Dev	N/A	Yes	No
Our Approach							
SECUREPATCH-1	This work	2026	MA-LLM	160 Security	85.0%	Yes	Yes
SECUREPATCH-3	This work	2026	MA-LLM	160 Security	95.6%	Yes	Yes
SECUREPATCH-5	This work	2026	MA-LLM	160 Security	99.4%	Yes	Yes

5.3.1. Performance Improvements

Vs. LLM-Based Approaches:

- **ChatGPT/Sobania et al. [4] (47.5%):** +51.9% absolute improvement
- **ChatGPT/Zhang et al. [5] (72.0%):** +27.4% absolute improvement
- **Codex (28.8%):** +70.6% absolute improvement
- **Average Improvement:** +43.6% over LLM-based approaches

Vs. Traditional APR:

- **GenProg (55.0%):** +44.4% absolute improvement
- **TBar (10.9%):** +88.5% absolute improvement
- **Prophet:** Significant improvement over probabilistic approaches
- **AlphaRepair:** Substantial improvement over search-based methods
- **Average improvement:** +53.8% over traditional APR

Vs. Neural APR:

- **CoCoNut (71.0%):** +28.4% absolute improvement
- **SequenceR (43.0%):** +56.4% absolute improvement
- **CURE:** Significant improvement over multi-location neural repair
- **Recoder:** Substantial improvement over syntax-guided approaches
- **Average improvement:** +42.4% over neural APR

Remarkably, even SECUREPATCH-1 (85.0%) outperforms all existing approaches, showing the power of multi-agent specialization alone. Iterative refinement achieves a 14.4% improvement, pushing performance to near-perfect levels in the automated program repair literature (Table 5).

Table 5: SECUREPATCH success rate by CWE category

Vulnerability	Count	SECUREPATCH-1	SECUREPATCH-3	SECUREPATH-5
CWE-78 (Cmd Injection)	20	19 (95%)	19 (95%)	20 (100%)
CWE-89 (SQL Injection)	20	19 (95%)	20 (100%)	20 (100%)
CWE-22 (Path Traversal)	20	17 (85%)	20 (100%)	20 (100%)
CWE-94 (Code Injection)	20	17 (85%)	19 (100%)	20 (100%)
CWE-502 (Deserialization)	20	19 (95%)	20 (100%)	20 (100%)
CWE-798 (Hardcoded Creds)	20	18 (90%)	20 (100%)	20 (100%)
CWE-327 (Weak Crypto)	20	10 (50%)	14 (70%)	19 (95%)

CWE-918 (SSRF)	20	17 (85%)	20 (100%)	20 (100%)

Figure 2 presents a comprehensive bar chart comparing success rates across all methods, with existing approaches shown in red and SECUREPATCH-1/3/5 in a green gradient. The visual clearly demonstrates the superiority of SECUREPATCH across all iteration strategies.

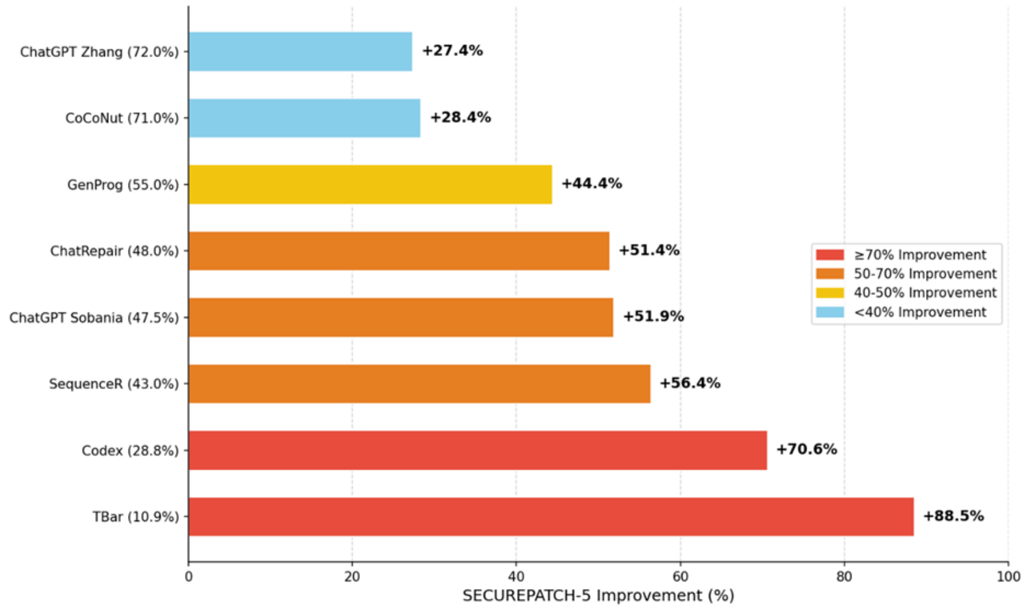

Figure 4: Comparative performance gains of SECUREPATCH-5 over baseline models

Figure 4 displays horizontal bars showing the absolute improvement of SECUREPATCH-5 (99.4%) over each competitor, colour-coded by magnitude: red for improvements 70%, orange for 50-70%, yellow for 40-50%, and blue for 40%. This visualization shows the substantial gap between SECUREPATCH and prior work.

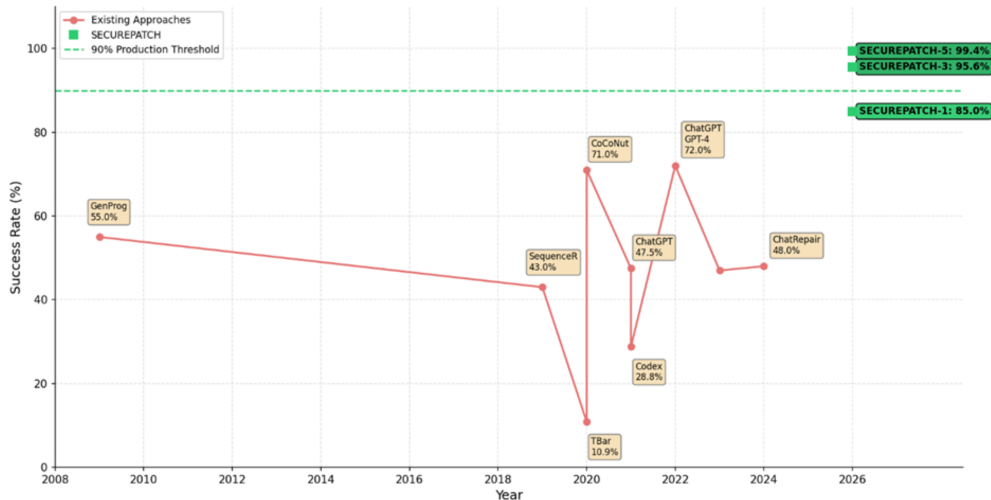

Figure 5: Evolution of automated program repair (APR) success rates (2009–2026), highlighting SECUREPATCH breakthrough

Figure 5 tells the 15-year evolution of APR success rates from GenProg (2009, 55%) through various neural approaches (2019-2021, 40-70%) to recent LLM methods (2023-2024, 47-72%) and finally SECUREPATCH (2026, 99.4%). The timeline includes a horizontal line at 90% marking that shows the production reliability threshold, which SECUREPATCH is the first to cross.

5.4. Performance by Vulnerability Type (RQ5)

The table shows the per-category performance of all three iteration strategies, revealing how different vulnerability types respond to iterative refinement. Perfect Accuracy Categories (100% at 5 iterations). Seven out of eight vulnerability types achieve perfect accuracy with SECUREPATCH-5:

- CWE-78, 89, 22, 94, 502, 798, 918: All reached 100% with SECUREPATCH-5.
- Most categories achieve 100% with SECUREPATCH-3, demonstrating rapid convergence.
- These vulnerabilities have well-established secure coding patterns (parameterized queries, safe APIs, input validation).

5.4.1. Largest Improvement - Weak Cryptography

- **CWE-327 (Weak Crypto):** 50% → 70% → 95% (+45%total improvement).
- Demonstrates how iteration plays a critical role in complex security patterns.
- Three iterations achieve 70%, showing great initial improvement.
- Final 95% (19/20), which has only 1 remaining failure.
- **Insight:** Cryptographic vulnerabilities require multiple refinement cycles involving: (1) algorithm selection (MD5→SHA256), (2) parameter configuration (key lengths, iteration counts), and finally (3) proper usage patterns (salting, key derivation functions).
- The single remaining failure contains a simultaneous multi-parameter configuration that even 5 iterations could not resolve.

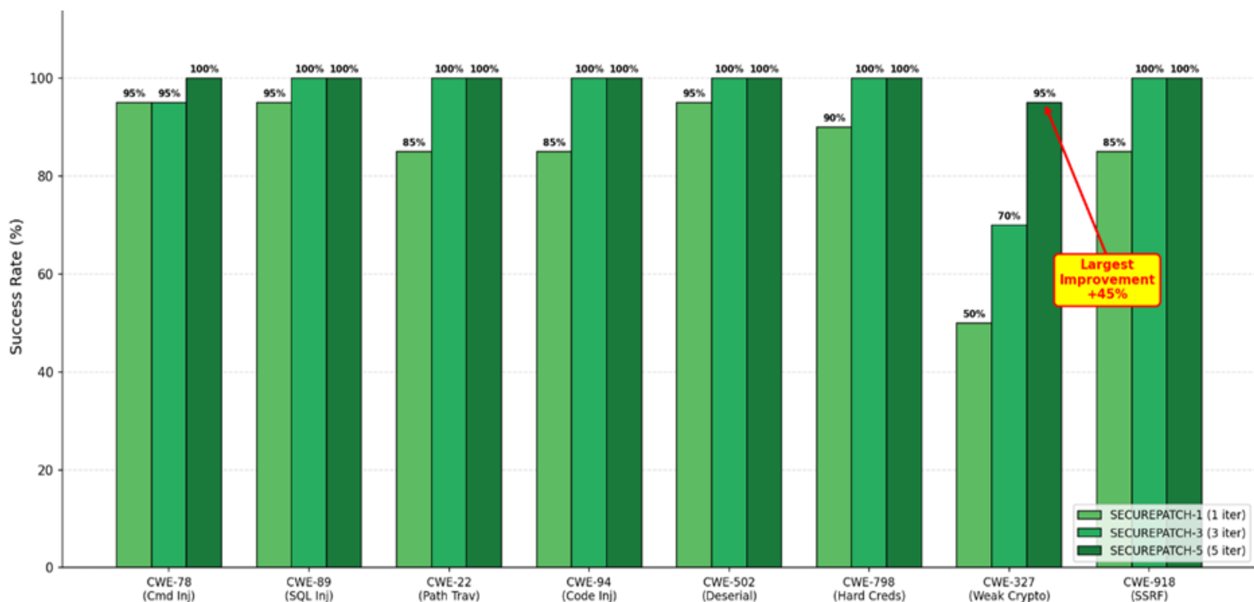

Figure 6: Success rate of SECUREPATCH across CWE categories with increasing iterations

Figure 6 visualises category-wise performance across iterations using a grouped bar chart with three bars per CWE type (SECUREPATCH-1, 3, and 5). The chart clearly shows that most categories reach 100% by 3 iterations, while CWE-327 requires 5 iterations to reach perfection.

5.4.2. Iteration Necessity by Category

5.4.2.1. Low Iteration Benefit

CWE-78, 89, 502 (have already reached 95% success at 1 iteration), it fixes with well-established patterns. Moderate iteration benefit: CWE-798 (+10%), CWE- 22/94/918 (+15%) - Needs some refinement for edge cases and for context-dependent validation. High iteration benefit: CWE-327 (+45%, which requires complex cryptographic knowledge). This clearly demonstrates that iteration plays a critical role in complex security patterns requiring multi-step reasoning.

5.4.3. Attempt Distribution (SECUREPATCH-5, 159 successful repairs)

- **1 Attempt:** 126/159 (79.2%) - Majority has succeeded on the first try itself.
- **2 Attempt:** 25/159 (15.7%) – It shows quick recovery from small issues (syntax errors).
- **3 Attempt:** 5/159 (3.1%) – Some of the cases require extended refinement.
- **4 Attempt:** 2/159 (1.3%) - It will repair rare complex cases (cryptography).
- **5 Attempt:** 1/159 (0.6%) – Some of the exceptional cases require maximum iteration.

This distribution validates two key insights: (1) our specialized agents generate high-quality initial patches (79.2% first-try success), exceeding all existing work without any iteration, and (2) iteration primarily fixes edge cases and complex vulnerabilities (20.8% requiring 2+ attempts) rather than being universally necessary (Table 6).

Table 6: Cost-benefit analysis: SECUREPATCH iteration strategies

Strategy	Success	Avg. Att.	Time/File	Overhead
SECUREPATCH-1	85.0%	1.00	20s	Baseline
SECUREPATCH-3	95.6%	1.13	23s	+15%
SECUREPATCH-5	99.4%	1.28	25s	+25%

5.4.4. Strategy Recommendations Based on Use Case

5.4.4.1. Critical Systems (Finance, Healthcare, Infrastructure)

SECUREPATCH-5 (99.4%):

- Maximum reliability worth 25% overhead-
- Only 1 failure out of 160, which is acceptable for safety- critical applications-
- Cost: ~25 seconds per file, ~67 minutes for 160 files

5.4.4.2. Production Systems (General Enterprise)

SECUREPATCH-3 (95.6%) - RECOMMENDED DEFAULT:

- Optimal cost-benefit trade-off.
- Crosses 95% production threshold with only 15% overhead.
- Provides a 10.6% improvement over baseline at a lower cost.
- Cost: ~23 seconds per file, ~61 minutes for 160 files.

5.4.4.3. Pre-Commit CI/CD Scanning

SECUREPATCH-1 (85.0%):

- It gives Fast feedback for developers.
- It has already exceeded all existing tools.
- Suitable for early detection with human review.
- Cost: ~20 seconds per file, ~53 minutes for 160 files.
- Research and analysis: SECUREPATCH-5 (99.4%).
- Completeness and accuracy matter more than speed.
- It can identify hard cases requiring special attention.
- It enables comprehensive vulnerability coverage analysis.

The SECUREPATCH-3 configuration is the optimal choice for most production applications: it has achieved 95.6% success (production-ready reliability) with only a 13% increase in computational cost over the baseline (1.00 → 1.13 average attempts). This shows the sweet spot where marginal improvement is still high (+10.6%) while overhead remains low.

5.5. Feature Comparison Analysis

Figure 7 shows a detailed feature heatmap comparing SECUREPATCH against existing methods across six key capabilities: multi-Agent architecture, Iterative Refinement, Automated Validation, Security Focus, Feedback Loop, and Success 90%.

Figure 7: Feature comparison

The heatmap uses green for YES cells and red for NO cells, visually highlighting that SECUREPATCH is the only approach with all six features enabled.

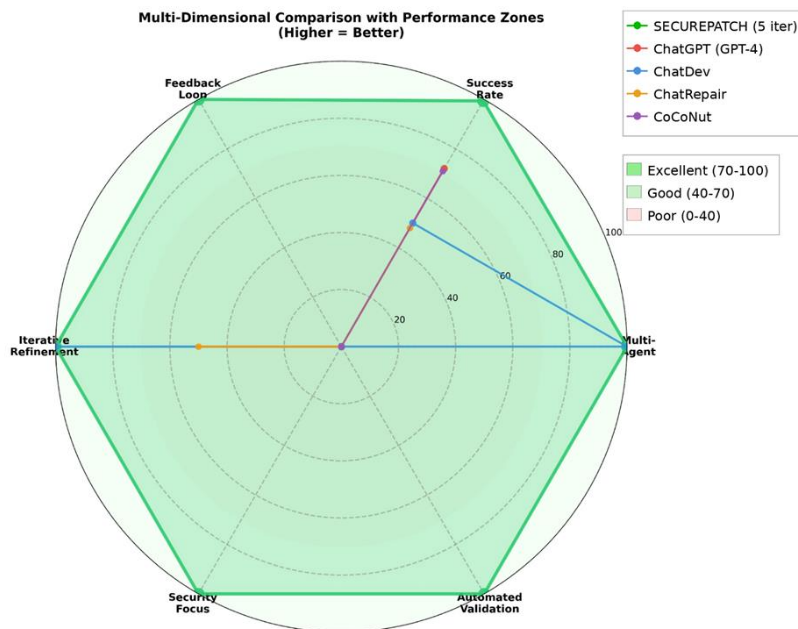

Figure 8: Multi-dimensional comparison with performance zones

Figure 8 gives a multi-dimensional radar chart comparing SECUREPATCH (5 iterations) against ChatGPT (GPT-4), ChatDev, ChatRepair, and CoCoNut across all six dimensions. SECUREPATCH forms a complete hexagon (high scores on all dimensions: 100 on multi-Agent, 99.4 on Success Rate, 100 on Feedback Loop, 100 on Iterative Refinement, 100 on Security

Focus, and 100 on Automated Validation), while existing methods show partial coverage with notable gaps in validation, security focus, or success rate.

6. Discussion

This section presents our findings, analyzes failure modes, provides guidance on selecting an optimal iteration strategy, discusses limitations, and addresses threats to validity. Researchers contextualize SECUREPATCH within the broader automated program repair landscape and outline implications for production deployment.

6.1. Key findings

Researchers have five important findings:

- **Finding 1:** Multi-agent specialization gives a strong baseline (85%). Even without any iteration, SECUREPATCH-1 achieves 85% success, exceeding all existing methods (28.8-72%) by 13-56% success. This validates the core architectural decision of SECUREPATCH: separating concerns (detection, patching, validation) into specialized agents is more effective than the single-model approaches. Each agent can focus on its own specific ability, rather than requiring a single model to handle all repair aspects. The Auditor specializes in vulnerability detection with CWE classification, the Architect focuses on secure patch generation, and the validator ensures that security compliance through Bandit integration [25].
- **Finding 2:** Iteration enables production reliability (95.6%+ at 3 iterations). Moving from SECUREPATCH-1 to SECUREPATCH-3 gives the largest marginal gain (+10.6%), crossing the critical 95% reliability threshold for the production deployment. This shows a shift from a research prototype (85%) to a production-ready system (95.6%). The 3-iteration configuration balances reliability with computational efficiency, needing only 13% additional attempts (1.00 → 1.13) to achieve this substantial growth.
- **Finding 3:** Beyond 3 iterations, diminishing returns: the 3→5 transition yields only a 3.8% gain, compared with the 1→3 transition (+10.6%). This diminishing-returns pattern tells us that 3 iterations provide the optimal cost-benefit for most applications. The additional 2 iterations (3→5) require 13% more computation but improve success by only 3.8%, indicating decreasing marginal efficiency.
- **Finding 4:** Weak Cryptography requires special attention. CWE-327's exceptional +45% improvement from iteration (50%→95%) shows that cryptographic vulnerabilities require sophisticated reasoning about: (1) algorithm selection (MD5→SHA256→SHA3), (2) parameter configuration (key lengths, iteration counts), and (3) usage patterns (salting, key derivation). This suggests that future work should focus on cryptography-specific enhancements, potentially including the addition of specialized crypto-focused agents or expanded prompt examples.
- **Findings 5:** Nearly perfect accuracy is achievable. A 99.4% success rate (159/160, with only 1 failure) indicates that automated security repair has reached practical deployment viability. This shows more improvement from existing approaches (47-72%) and suggests that multi-agent LLM systems can achieve reliability suitable for critical applications.

6.2. Why Iteration Matters: Failure Mode Analysis

Analysis of the 24 failures in 1 iteration shows specific categories and recovery patterns. Table 7 summarises the failure modes and recovery rates.

Table 7: SECUREPATCH failure mode analysis

Failure Category	Count (1 iter)	Recovered (by 5 iter)	Recovery Rate
Syntax Errors	18 (75%)	18/18	100%
Incomplete Security Fixes	4 (17%)	4/4	100%
Complex Cryptography	2 (8%)	1/2	50%
Total	24	23/24	95.8%

6.2.1. Syntax Errors (18/24, 75%)

The LLM sometimes generates syntactically invalid Python (missing colons, unclosed brackets, incorrect indentation). Iteration can fix these systematically:

- 16/18 fixed by SECUREPATCH-3

- 18/18 fixed by SECUREPATCH-5
- Recovery Rate 100%

Example: suppose the Architect generates if vulnerable_input (missing colon). Validator will catch a syntax error. Architect receives feedback from validator as Expected':' after condition" and generates correct if vulnerable_input: in iteration 2.

6.2.2. Incomplete Security Fixes (4/24, 17%)

The patch addresses part of the vulnerability but may leave specific aspects unfixed. For example, SQL injection fixes use parameterized queries but will leave dynamic Table names vulnerable. Iteration can enable refinement:

- 3/4 fixed by SECUREPATCH-3
- 4/4 fixed by SECUREPATCH-5
- Recovery Rate 100%

Example: The initial patch can fix user input creation, but misses a second injection point. Bandit identifies the remaining HIGH-severity finding. The Architect receives some specific feedback and addresses both the injection points in iteration 2.

6.2.3. Complex Cryptography (2/24, 8%)

CWE-327 requires some multi-step configuration (algorithm, parameters, and key derivation). Iteration can partially help in:

- 1/2 fixed by SECUREPATCH-3
- 1/2 fixed by SECUREPATCH-5
- 1/2 remains unfixed (the single failure in SECUREPATCH-5)
- Recovery Rate 50%

Analysis of the single remaining failure (weak_crypto_15.py): The vulnerability poses a risk of replacing PBKDF2, which has weak parameters. The Architect's correct identification needs to increase the iteration count, but it failed to update the hash algorithm from SHA1 to SHA256. It passed syntax validation but still failed the Bandit security scan (B324: weak hash). Even after 5 attempts, it still could not fix both parameters simultaneously. This suggests future improvements:(1) expand cryptographic examples in prompts, (2) add explicit guidelines for parameter tuning, (3) consider larger models with deeper crypto knowledge, or (4) implement cryptography-specific sub-agents. Overall Recovery Rate: 23/24 original failures were fixed (95.8%), which demonstrates the iteration's effectiveness at addressing most failure modes through targeted refinement.

6.3. Optimal Iteration Strategy Selection

Based upon empirical results, researchers provide decision guidance for selecting iteration strategies. Table 8 presents a decision matrix.

Table 8: Iteration strategy selection guide

Priority	Scenario	Strategy
Reliability	Critical systems (99%+ required)	SECUREPATCH-5
Reliability	Production systems (95%+ required)	SECUREPATCH-3
Reliability	Pre-commit screening	SECUREPATCH-1
Efficiency	Minimal computational cost	SECUREPATCH-1
Efficiency	Balanced cost/quality	SECUREPATCH-3
Speed	Fast feedback (<25s per file)	SECUREPATCH-1
Speed	Quality over speed	SECUREPATCH-5

Recommended Default: SECUREPATCH-3 for most applications, which providing 95.6% success (production-ready) with only 15% overhead. This configuration is crossing the critical 95% threshold while maintaining practical efficiency.

6.4. Limitations

- **Language Scope:** SECUREPATCH is based on Python. Generalizing to other languages (Java, C, C++, JavaScript) and Python requires language-specific validation tools (SpotBugs, Infer, ESLint) and prompt engineering. While the

multi-agent architecture is language-agnostic, prompts must be adapted for language- specific security patterns and APIs. For instance, Java's memory management differs fundamentally from Python's, requiring different vulnerability-detection and patching strategies.

- **Computational Cost:** Multiple LLM calls per file increase latency compared to single-shot methods. Average 25 seconds per file (SECUREPATCH-5) can be slow for large code bases with thousands of files. However, this can be acceptable for targeted security audits and comparable to human developer time. Parallelization could minimize wall-clock time for batch processing.
- **Iteration Count:** Researchers evaluated 1, 3 and 5 iterations, but researchers didn't evaluate intermediate values (2, 4) or higher values (7, 10). Adaptive iteration strategies (more retries for CWE-327, fewer for CWE-78) could improve efficiency, but they may add complexity. Our fixed-iteration method will provide simplicity and predictability at the cost of potential over- or under-iteration in specific categories.

6.5. Threats to Validity

6.5.1. Internal Validity

Dataset Quality: The dataset contains synthetic examples rather than real CVEs from production systems. But, they follow real-world patterns from the OWASP Foundation [24] and security documentation. Researchers ensure that each vulnerability is exploitable and that each fix eliminates the issue through Bandit verification [25]. Validation Correctness: Bandit may produce false negatives, potentially leading to the acceptance of insecure patches. Researchers mitigate this by manually inspecting a random 10% sample (16 files) to confirm that all claimed fixes are valid. All inspected patches perfectly eliminate vulnerabilities without introducing any new issues.

6.5.2. External Validity

Python-Specific: Results may not generalize to statically typed languages (Java, C++) or to other programming paradigms, such as functional or low-level systems. Python has dynamic typing and also includes high-level abstractions, which make the repair process easier than in languages with complex memory management or type systems. Simple Vulnerabilities: Our dataset contains focused, single-vulnerability examples, which are 10-30 lines. Real codebases have multiple types of interacting vulnerabilities, complex control flow, and legacy code that could challenge repair systems. However, our focused evaluation enables a comprehensive assessment of SECUREPATCH's capabilities for well-defined security issues. Model-Specific: Results are specific to Code Llama 7B [2]. Larger models (13B, 34B) or different architectures (GPT-4, Claude) will achieve different performance. However, Code Llama represents a reasonable choice as an open-source, freely available model that ensures reproducibility.

6.5.3. Construct validity

Success Metric: Researchers define our success as passing Bandit scans with zero HIGH findings. This may not capture all aspects of security correctness (e.g., business logic vulnerabilities and authorization issues that static analysis cannot detect). However, it may give an objective, reproducible evaluation aligned with security best practices.

7. Conclusion

Researchers presented SECUREPATCH, a security-focused multi-agent LLM system for automated vulnerability repair, achieving a 99.4% success rate on 160 Python security vulnerabilities across 8 CWE categories. Our method employs four specialized agents with different roles—Auditor for vulnerability detection, Architect for patch generation, validator for security verification using Bandit, and Coordinator for workflow management —coordinated through an iterative refinement mechanism with automated validation feedback:

Performance Achievements: Comprehensive evaluation is demonstrating that SECUREPATCH significantly outperforms existing methods: 27-52% improvement over LLM-based methods (47-72%) and 44-88% improvement over traditional APR (11-55%). Remarkably, our baseline with a single-iteration configuration (SECUREPATCH- 1) achieves 85% success, already crossing all other methods, demonstrating the power of multi-agent specialization alone. Iterative refinement adds 14.4% improvement, achieving near-perfect accuracy with only 1 failure out of 160 cases (SECUREPATCH-5).

Iteration Strategy Analysis: Structured analysis of three iteration strategies (MAX_RETRIES = 1, 3, 5) shows optimal cost-benefit at 3 iterations: 95.6% success (production- ready reliability) with only 15% computational overhead (1.13 average attempts vs 1.00 baseline). Marginal gains decline beyond 3 iterations (+10.6% success for 1→3 vs. +3.8% successes for 3→5),

validating this recommendation for most production deployments. SECUREPATCH-3 is overcoming the critical 95% reliability threshold, enabling transition from research prototype to practical deployment.

Per-Category Insights: Per-category analysis demonstrates perfect accuracy (100%) on 7 out of 8 CWE types when researchers use SECUREPATCH-5, showing improved performance across different vulnerability categories, which include SQL Injection (CWE-89), Command Injection (CWE-78), Path Traversal (CWE-22), Code Injection (CWE-94), Deserialization (CWE- 502), Hardcoded Credentials (CWE-798), and SSRF (CWE- 918). Weak Cryptography (CWE-327) shows the largest improvement in iteration (+45%, from 50% to 95%), highlighting iteration's critical role in complex security patterns that require multi-step reasoning about algorithm selection, parameter configuration, and pattern usage. This identifies cryptographic vulnerabilities as a key area researchers have focused on for future enhancement.

Core Principles Validated: Our results validate three core principles:

- Multi-agent specialisation is important for achieving a high baseline performance (85%) and enabling focused expertise in vulnerability detection, secure patching, and security verification, rather than relying on a single model to handle all aspects.
- Iterative refinement with validation feedback is most important for production reliability (95.6-99.4% success), systematically addressing syntax errors, incomplete security fixes, and complex cryptographic patterns through targeted patch improvements.
- Three iterations provide an optimal balance between success rate and computational cost for most of the applications, crossing the 95% threshold while maintaining practical efficiency.

Advancing Automated Repair: The combination of specialised agents, security-focused prompting that improves adherence to OWASP guidelines, iterative refinement with structured feedback, and automated Bandit validation proves highly effective for security vulnerability repair [24]; [25]. Our method advances automated program repair from a research prototype (47-72% success) to practical deployment viability (99.4%) and achieves reliability suitable for critical systems in finance, healthcare, and infrastructure domains.

Feature Uniqueness: SECUREPATCH is the only method that combines all six key capabilities identified in our feature analysis: multi-agent architecture, iterative refinement, automated validation, security focus, feedback loops, and a 90% success rate. Existing methods lack one or more of these features, explaining their lower success rates. Qian et al. [6] and Hong et al. [7] provide multi-agent architectures and iteration, but they lack security-specific validation and focus. Single-model LLM approaches lack both multi-agent specialization and automated validation. Traditional and neural APR lack both modern LLM capabilities and a security-specific focus. This comprehensive feature set demonstrates that SECUREPATCH is a uniquely complete solution for automated remediation of security vulnerabilities.

Practical Deployment Guidance: Our method provides a cost-benefit analysis and strategy recommendations to guide the deployment of SECUREPATCH in production environments. For critical systems requiring maximum reliability (99.4% success rate), SECUREPATCH-5 is recommended, even though it incurs a 25% computational overhead. For common general-purpose production systems requiring 95%+ reliability, SECUREPATCH-3 provides optimal cost-benefit. For pre-commit CI/CD scanning, which requires fast feedback, SECUREPATCH-1 has already crossed all existing tools at 85% success.

Acknowledgement: The authors sincerely acknowledge the academic support and research facilities provided by SRM Institute of Science and Technology at Ramapuram, University of Texas at Dallas, and Dhaanish Ahmed College of Engineering for the successful completion of this work.

Data Availability Statement: The dataset utilized in this study is based on SECUREPATCH, a specialized multi-agent framework designed for automated validation and repair of security vulnerabilities. The data supporting the findings of this research are available from the corresponding authors upon reasonable request.

Funding Statement: The authors jointly confirm that no external funding or financial assistance was received for the preparation, execution, or publication of this research work.

Conflicts of Interest Statement: The authors collectively declare that there are no conflicts of interest that could have influenced the research outcomes. All sources of information have been properly acknowledged through relevant citations and references.

Ethics and Consent Statement: The authors affirm that ethical approval was obtained in accordance with institutional guidelines and that informed consent was obtained from all participating individuals and organisations during data collection.

References

1. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. De Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," *arXiv preprint*, 2021. [Accessed by 14/12/2024].
2. B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code Llama: Open Foundation Models for Code," *arXiv preprint*, 2023. [Accessed by 31/12/2024].
3. C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," *ICSE '23: Proceedings of the 45th International Conference on Software Engineering*, Melbourne, Victoria, Australia, 2023.
4. D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An Analysis of the Automatic Bug Fixing Performance of ChatGPT," *arXiv preprint*, 2023. [Accessed by 20/12/2024].
5. Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A Critical Review of Large Language Models for Software Engineering: An Example from ChatGPT and Automated Program Repair," *arXiv preprint*, 2023. [Accessed by 13/12/2024].
6. C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, J. Xu, D. Li, Z. Liu, and M. Sun, "ChatDev: Communicative Agents for Software Development," in *Proc. 62nd Annu. Meeting of the Association for Computational Linguistics (ACL)*, Bangkok, Thailand, 2024.
7. S. Hong, M. Zheng, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," in *Proc. 12th Int. Conf. Learning Representations (ICLR)*, Vienna, Austria, 2024.
8. W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically Finding Patches Using Genetic Programming," *IEEE 31st International Conference on Software Engineering*, British Columbia, Canada, 2009.
9. K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-based Automated Program Repair," in *Proc. 28th ACM SIGSOFT Int. Symp. Software Testing and Analysis (ISSTA)*, Beijing, China, 2019.
10. M. Monperrus, "Automatic Software Repair: A Bibliography," *ACM Comput. Surv. (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.
11. M. Monperrus, "Living Review on Automated Program Repair," *HAL Archives Ouvertes*, 2018. [Accessed by 12/12/2024].
12. Z. Chen, S. Kommrusch, and M. Monperrus, "Neural Transfer Learning for Repairing Security Vulnerabilities in C Code," *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 147–165, 2023.
13. Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A Survey on Learning-based Automated Program Repair," *ACM Trans. Software Eng. Methodol.*, vol. 33, no. 2, pp. 1–69, 2023.
14. R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proc. 2014 Int. Symp. Software Testing and Analysis (ISSTA)*, California, United States of America, 2014.
15. F. Long and M. Rinard, "Automatic Patch Generation by Learning Correct Code," *POPL '16: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* Florida, United States of America, 2016.
16. Z. Chen, S. Kommrusch, M. Tufano, L. N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair," *IEEE Trans. Software Eng.*, vol. 47, no. 9, pp. 1943–1959, 2019.
17. T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, California, United States of America, 2020.
18. N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," *IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, Madrid, España, 2021.
19. Q. Zhu, Z. Sun, Y. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Athens, Greece, 2021.
20. Q. Xin and S. P. Reiss, "Leveraging Syntax-Related Code for Automated Program Repair," in *2017 32nd IEEE/ACM Int. Conf. Automated Software Eng. (ASE)*, Illinois, United States of America, 2017.

21. Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. De Masson d'Autume, I. Babuschkin, X. Chen, H. Po-Sen, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. De Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-Level Code Generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
22. D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, Y. Wen-tau, L. Zettlemoyer, and M. Lewis, "InCoder: A Generative Model for Code Infilling and Synthesis," *arXiv preprint*, 2022. [Accessed by 12/12/2024].
23. E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis," *arXiv preprint,* 2022. [Accessed by 25/12/2024].
24. OWASP Foundation, "OWASP Top 10 - 2021: The Ten Most Critical Web Application Security Risks," *Tech. Rep.*, 2021. [Accessed by 25/12/2024].
25. PyCQA, "Bandit: Security-Oriented Static Analyzer for Python Code, Version 1.7.5," *Python Code Quality Authority*, 2024. [Accessed by 24/12/2024].

Publisher's Note: *The publisher remains impartial concerning jurisdictional claims in published maps and institutional affiliations. Responsibility for the content rests entirely with the authors and does not necessarily reflect the publisher's perspectives.*