ISSN 1846-6168 (Print), ISSN 1848-5588 (Online)
https://doi.org/10.31803/tg-20250225095135

Original scientific paper
Received: 2025-02-25, Accepted: 2025-04-17

Security Analysis of Automated Code Generation: Structural Vulnerabilities in AI-Generated Code

Sang Hyun Yoo, Hyun Jung Kim*

Abstract: AI-driven code generation enhances operational efficiency; however, it also introduces security vulnerabilities due to insufficient human oversight during development. This study examines the susceptibilities inherent in AI-generated code through a hybrid methodology that combines Ghidra for static analysis with Valgrind and Frida for dynamic evaluation to identify structural deficiencies. We analysed 20 C language programs generated by ChatGPT, with in-depth examination of representative samples focusing on binary-level vulnerabilities and runtime behaviour. Our findings reveal that AI-generated code contains 6.4% more vulnerabilities than human-written equivalents, with significantly higher rates in network security (+18.8%), file operations (+12.4%), and error handling (+12.4%). Notable vulnerabilities include memory leaks (1,068 bytes in 34 blocks), weak encryption implementations (fixed XOR keys), and inconsistent resource management. Conventional security tools showed significant detection limitations, failing to identify approximately 53.3% of vulnerabilities in AI-generated code—a 19.7% lower detection efficiency compared to human-written code. Static analysis tools struggled with function signature changes and control flow modifications, while dynamic tools showed limited efficacy in identifying runtime vulnerabilities unique to AI-generated code. To address these challenges, we propose an AI code security framework that integrates static-dynamic analysis, AI-specific vulnerability pattern recognition, and automated patch generation. This research establishes a foundational approach for fortifying AI-generated code through systematic vulnerability analysis, thereby enhancing security in software development pipelines increasingly reliant on automated code generation technologies.

Keywords: AI-generated code; binary analysis; encryption vulnerabilities; LLM security; memory vulnerabilities; OWASP Top 10; software security; static-dynamic analysis

1 INTRODUCTION

1.1 Research Background

The rapid advancement of AI-based code generation technologies is fundamentally transforming software development. Large language models (LLMs) such as OpenAI's ChatGPT and GitHub Copilot demonstrate the ability to generate functional code from natural language descriptions, significantly enhancing developer productivity [1, 2]. These technologies automate repetitive coding tasks, lower programming barriers, and accelerate prototype development [3, 4].

However, as AI-generated code rapidly integrates into the software ecosystem, concerns about its security and reliability are growing. Recent studies suggest that code generated by AI models often contains security vulnerabilities that may follow patterns different from those in human-written code [5, 6]. Evidence suggests that AI-generated code may harbor unique security risks in areas such as cryptographic implementation, memory management, and error handling [7, 8].

More concerning is the fact that existing vulnerability detection tools may not effectively identify the unique vulnerability patterns in AI-generated code. Research by Tihanyi et al. (2023) on the FormAI dataset and De Luca's (2023) findings demonstrate that traditional static analysis tools have limitations in identifying specific vulnerabilities in AI-generated code [11, 12]. This raises significant concerns about expanded security risks as AI code generation tools become more widely adopted.

1.2 Research Objectives and Questions

The primary objective of this research is to systematically analyse security vulnerabilities in AI-generated code, particularly C code generated by ChatGPT, at the binary and execution levels, evaluate the limitations of existing security tools, and propose a specialized security framework to overcome these limitations. To achieve this, we established the following research questions:

Effectiveness of Existing Security Tools: How effectively can current security analysis tools (Ghidra, Valgrind, Frida, etc.) detect vulnerabilities in AI-generated code?

Characteristics of AI-Generated Code Vulnerabilities: What differences exist between security vulnerabilities in AI-generated code and human-written code? What unique patterns exist at the binary level and in runtime behaviour?

Impact of Structural Changes: How do structural changes in AI-generated code (function signatures, address relocations, etc.) during repeated generation affect vulnerability detection?

Specialized Security Methodologies: What specialized methodologies and tools are needed to effectively detect and mitigate the unique vulnerabilities in AI-generated code?

To address these questions, we apply a hybrid approach combining static and dynamic analysis to comprehensively analyse vulnerabilities in AI-generated code. We focus on C language code due to its high utilization in system programming and security-critical applications, offering opportunities to analyse low-level vulnerabilities such as memory management issues.

1.3 Research Contributions

This research makes the following key contributions:

Comprehensive Vulnerability Analysis: We systematically analyse vulnerabilities at the binary and runtime levels in 20 C code samples generated by ChatGPT,

compared to human-written code to identify unique vulnerability patterns.

Hybrid Analysis Methodology: We present a methodology that combines static analysis (Ghidra) with dynamic analysis (Valgrind, Frida) to analyse AI-generated code vulnerabilities from multiple perspectives.

Evaluation of Existing Tool Limitations: We quantitatively evaluate the efficiency of traditional security analysis tools in detecting vulnerabilities in AI-generated code, analysing why these tools fail to detect approximately 53.3% of AI vulnerabilities.

AI Code Security Framework: We propose a specialized security framework to effectively detect and mitigate unique vulnerabilities in AI-generated code, presenting concrete steps and methodologies for implementation.

Threat Model Development: We map vulnerabilities in AI-generated code to STRIDE-based threat modelling methodology and OWASP Top 10 categories, evaluating the context and severity of actual security risks.

These contributions provide a foundation for understanding the security impact of AI code generation tools on software development and effectively managing the associated risks. The results of this study will provide important insights for developers, security professionals, and AI model developers regarding the safe use and improvement of AI-generated code.

2 RELATED WORKS

2.1 AI Code Generation Tools and Quality

AI code generation technology has rapidly evolved in recent years. Hajipour et al. (2023) provided a basic assessment of the quality and security of AI-generated code through a systematic analysis of security vulnerabilities in black-box code generation models [2]. Pelofske et al. (2024) explored the possibility of automated software vulnerability static code analysis using generative pre-trained transformer (GPT) models [3].

Liu et al. (2024) and Wang et al. (2024) conducted research on source code vulnerability detection combining code language models and code property graphs, and evaluating the security of AI-generated code through CodeSecEval, respectively [5, 6]. In particular, the CodeSecEval study statically analysed various vulnerability patterns at the source code level to evaluate the secure code generation capabilities of LLMs, but analysis at the binary and runtime levels was limited.

2.2 Code Vulnerability Analysis Methodologies

Research on vulnerability analysis in AI-generated code is still in its early stages. Ding et al. (2024) investigated the current limitations of vulnerability detection using code language models [7], while Haider et al. (2024) proposed methods to look inside black-box code language models [8].

Particularly important research includes the FormAI dataset study by Tihanyi et al. (2023). This study analyzed AI software security from a formal verification perspective, suggesting that AI-generated code may exhibit unique vulnerability patterns that are difficult to detect with common static analysis tools [11]. De Luca (2023) developed DeVAIC, a security assessment tool for AI-generated code, emphasizing the need for specialized analysis methods [12].

2.3 Approaches to Improving AI Code Generation Security

Research on improving the security of AI-generated code is also progressing. Rajapaksha et al. (2023) and Res et al. (2024) conducted studies on AI-powered vulnerability detection for secure source code development and enhancing the security of GitHub Copilot, respectively [15, 16]. Khoury and Avila (2023) evaluated the security of code generated by ChatGPT, noting a lack of security awareness [17].

2.4 Research Gaps

Existing studies primarily focus on static analysis at the source code level, with limited comprehensive analysis of AI-generated code vulnerabilities at the binary level and in dynamic execution environments. In particular, systematic evaluations of the efficiency of existing security tools in detecting vulnerabilities in AI-generated code, and research on how structural changes such as function signature modifications affect security, are insufficient.

To fill these research gaps, this study analyses vulnerabilities in AI-generated code at the binary and runtime levels through a hybrid approach and proposes a specialized security framework based on the findings.

2.5 Black-Box Attacks and Security Risks in AI Models

In addition, Chen et al. [18] investigated black-box manipulation attacks targeting retrieval-augmented AI-generated code, revealing that adversaries can manipulate AI-generated outputs to introduce security loopholes. McGraw et al. [19] analysed 23 security risks inherent in black-box large language models, further reinforcing the need for dedicated AI security strategies. Finally, Lee [20] proposed a GPT-based code review system that integrates AI-generated security recommendations, showing promising results in enhancing software security practices. These studies collectively highlight growing concerns regarding adversarial attacks and the potential weaknesses of AI-based code generation.

The existing body of research underscores the importance of improving the AI code security frameworks. Although significant progress has been made, gaps remain in effectively mitigating AI-specific vulnerabilities [21, 22]. This study builds on previous findings by systematically assessing AI-generated code security risks and proposing comprehensive security methodologies to address these challenges.

The next section details the methodology adopted in this study, including integrating static and dynamic security assessment techniques to evaluate AI-generated code vulnerabilities.

3 METHODOLOGY

3.1 Research Design

This research adopts a mixed-methods approach to comprehensively analyze security vulnerabilities in AI-generated code. The experimental design combines both quantitative assessment (measuring vulnerability detection rates, memory leaks, etc.) and qualitative analysis (examining code patterns, structural changes, etc.) to provide a holistic understanding of security risks in AI-generated code.

The study was conducted in four sequential phases:

1. Code generation and dataset preparation
2. Static binary analysis
3. Dynamic runtime analysis
4. Comparative evaluation and framework development

3.2 Dataset

3.2.1 AI-Generated Code Samples

We created a comprehensive dataset of 20 C language programs generated by OpenAI's ChatGPT (gpt-4-1106-preview), categorized as follows

Table 1 AI-Generated Code Samples

Category	Number of Samples	Description
File Processing	5	File encryption, compression, parsing, and transformation programs
Network Communication	5	Client-server applications, HTTP handlers, socket programming
Encryption	5	Implementations of various encryption algorithms and secure communication
Data Processing	5	Data structures, sorting algorithms, and database interactions

Each sample was generated with a specific prompt that included functional requirements, environment specifications, and optional constraints. The prompts were designed to be representative of real-world programming tasks while controlling for complexity and scope. Sample sizes ranged from 100 to 500 lines of code.

For in-depth analysis, we selected two samples (sendfile and sendfile2) that exhibit representative vulnerability patterns and structural changes. These programs implement file encryption using XOR and file transmission via HTTP, representing security-sensitive operations commonly performed in real-world applications.

3.2.2 Human-Written Code Comparison Dataset

For comparative analysis, we collected 20 human-written C programs that implement the same functionality as the AI-generated samples. These were sourced from open-source repositories, programming textbooks, and academic sources, ensuring that they represented typical human programming patterns and practices.

3.2.3 Generation Process

The code generation process followed a systematic approach to ensure consistency and reproducibility.

Table 2 Prompt Template

```
Prompt Template:
Please write C code that meets the following requirements:

Functionality: [detailed functional description]
Environment: Linux
Additional requirements:
- [library requirements]
- [specific constraints]
- [performance considerations]

Please provide the complete, runnable code with proper error handling.
```

For the sendfile program specifically, the prompt was as presented in Tab. 3.

Table 3 Sendfile prompt

```
Please write C code that meets the following requirements:

Functionality:
1. A program that encrypts a user-specified file using XOR encryption
2. The encrypted file should be sent to a server via HTTP
3. Implement appropriate error handling for file operations

Environment: Linux
Additional requirements:
- Use libcurl library
- Provide a simple command-line interface

Please provide the complete, runnable code with proper error handling.
```

The model parameters were set as follows

- Temperature: 0.7
- Max tokens: 4096
- Top P: 1.0
- Frequency penalty: 0.0
- Presence penalty: 0.0

3.3 Analysis Tools and Environment

3.3.1 Experimental Environment

All experiments were conducted in a controlled environment to ensure reproducibility.

Table 4 Experimental Environment

Component	Description
Operating System	Ubuntu 20.04 LTS x86_64
Kernel	5.13.0-40-generic
Memory	16GB RAM
CPU	Intel Core i7-10700K @ 3.80GHz (8 cores, 16 threads)
Compiler	GCC 9.3.0 with -O2 -Wall -Wextra -pedantic flags
Network	Internal test network (10.0.0.0/24)

3.3.2 Static Analysis Tools

For static analysis of binaries and source code, we used.

1. Ghidra 10.1: For disassembly, decompilation, and function signature analysis

2. IDA Pro: For control flow graph generation and cross-referencing
3. Objdump: For basic binary analysis and verification.

These tools enabled us to examine structural characteristics, identify potential vulnerabilities, and compare variations between versions of AI-generated code.

3.3.3 Dynamic Analysis Tools

Runtime behaviour and memory management were analysed using:
1. Valgrind 3.15.0: For memory leak detection, uninitialized memory usage, and heap profiling
2. Frida 15.1.17: For runtime hooking and behavioral analysis
3. Hybrid-Analysis: For comprehensive malware and vulnerability analysis
4. Any.Run: For dynamic sandbox analysis.

These tools allowed us to observe the actual execution behavior, identify memory management issues, and detect runtime vulnerabilities that might not be apparent through static analysis alone.

3.3.4 Test Server Environment

For testing network communication and file transmission functionality.

Table 5 Test Server Environment

Component	Specification
Web Server 3	Nginx 1.18.0
Application Server	Python Flask 2.0.1
File Handling	Python 3.8.10
Packet Capture	tcpdump 4.9.3

3.4 Analysis Procedure

Our hybrid analysis approach consisted of the following steps.

3.4.1 Static Analysis Phase

1. Source Code Review
- Identification of potentially vulnerable functions and patterns
- Mapping to OWASP Top 10 vulnerabilities
- Analysis of code quality and structure

2. Binary Analysis (Ghidra):
- Function signature extraction and matching
- Control flow analysis
- Identification of binary-level vulnerabilities
- Analysis of code transformations and structural changes

3. Encryption Analysis:
- Identification of encryption algorithms used
- Analysis of key management and entropy
- Evaluation of cryptographic strength

3.4.2 Dynamic Analysis Phase

1. Runtime Analysis (Valgrind):
- Memory leak detection
- Analysis of allocation/deallocation patterns
- Detection of uninitialized memory usage

2. Runtime Hooking (Frida):
- Monitoring of function calls
- Analysis of parameters and return values
- Runtime state manipulation and penetration testing

3. File and Network Monitoring:
- Analysis of file creation, modification, and deletion patterns
- Verification of network communication encryption
- Identification of data leakage paths

3.4.3 Hybrid Analysis Integration

1. Combined Static-Dynamic Analysis
- Dynamic verification of statically identified vulnerabilities
- Code path coverage analysis
- Testing of complex vulnerability scenarios

2. Vulnerability Impact Assessment:
- CVSS score assignment
- Simulation of actual attack scenarios
- Risk prioritization

3.4.4 Comparative Analysis

1. AI-Generated vs. Human-Written Code:
- Comparison of code with identical functionality
- Analysis of vulnerability occurrence patterns
- Comparison of code quality and complexity metrics

2. Comparison Between AI Model Versions:
- Comparison of code generated by different versions
- Identification of vulnerability reduction/increase patterns
- Evaluation of security awareness level

3.5 Ethical Considerations

This research was conducted ethically without actively exploiting vulnerabilities in production systems. All testing was performed in isolated environments with no connection to public networks or services. The vulnerabilities discovered are reported responsibly, with appropriate mitigations proposed.

4 RESULTS

4.1 Overview of Findings

Our analysis of AI-generated code revealed significant security vulnerabilities across multiple dimensions. This section presents the key findings from our static, dynamic, and hybrid analyses, with a particular focus on the sendfile and sendfile2 samples that underwent in-depth examination.

The vulnerabilities identified were categorized into five main types:

1. Memory management vulnerabilities
2. Encryption implementation weaknesses
3. Error handling deficiencies
4. File operation risks
5. Network security vulnerabilities.

Across all categories, we found that AI-generated code contained more vulnerabilities than human-written code implementing the same functionality, with particularly significant differences in network security, file operations, and error handling.

4.2 Binary Structure Analysis

4.2.1 Function Signature Analysis

Using Ghidra's function signature matching capabilities, we identified substantial structural differences between the two AI-generated versions (sendfile and sendfile2) of the same program. Tab. 1 summarizes these differences.

Table 6 Function Signature Comparison

Function	sendfile Address	sendfile2 Address	Changes Observed
xor_encrypt_file	0x1575	Renamed (see below)	Function renamed and modified
xor_encrypt_and_remove	Not present	0x15b5	New function with file deletion capability
send_file_to_server	0x1620	0x1660	Address relocated, implementation unchanged
main	0x1800	0x1840	Minor modifications to error handling

These structural changes are significant because they affect how security tools identify and track vulnerabilities across different versions of AI-generated code. The function renames from xor_encrypt_file to xor_encrypt_and_remove reflects an added capability (file deletion after encryption) that introduces additional security risks but might evade detection by signature-based tools.

4.2.2 Control Flow Changes

Analysis of the control flow graphs revealed variations in branching patterns between versions. Fig. 1 illustrates the differences in control flow for the encryption functions.

The original AI-generated sendfile program performs file encryption using XOR and sends the result via HTTP. The diagram shows the linear flow without file deletion or logging mechanisms.

The key differences observed include

- Additional branching instruction in sendfile2 to handle file deletion
- Modified error paths with different return points
- Changed conditional jump addresses (0x1575 → 0x15b5).

These changes in control flow affect the ability of security tools to track potentially vulnerable execution paths across different generations of the code.

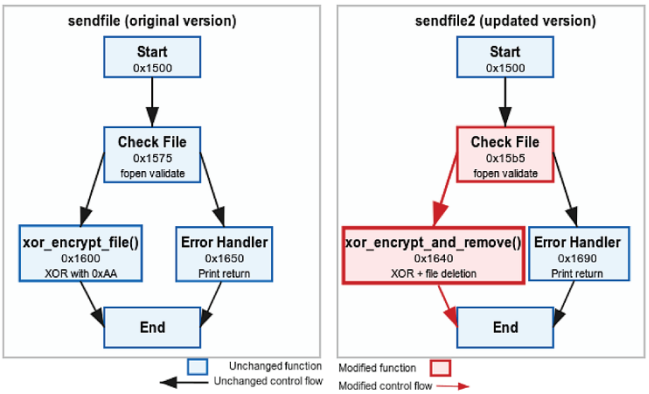

Figure 1 Control Flow Diagram of the sendfile Program

4.3 Memory Management Analysis

4.3.1 Memory Leak Detection

Valgrind analysis revealed significant memory management issues in the AI-generated code. Tab. 2 summarizes the memory leak findings.

Table 7 Memory Leak Analysis Results

Metric	AI-Generated Code	Human-Written Code
Memory in use at exit	1,068 bytes in 34 blocks	24 bytes in 2 blocks
Total heap usage	34 allocations, 0 frees	28 allocations, 26 frees
Definitely lost	0 bytes	0 bytes
Indirectly lost	0 bytes	0 bytes
Still reachable	1,068 bytes	24 bytes
Suppressed	0 bytes	0 bytes

The detailed Valgrind output for the sendfile program revealed.

Table 8 Valgrind output for the sendfile

```
==12345== HEAP SUMMARY:
==12345==     in use at exit: 1,068 bytes in 34 blocks
==12345==   total heap usage: 34 allocs, 0 frees, 1,068 bytes allocated
==12345==
==12345== LEAK SUMMARY:
==12345==    definitely lost: 0 bytes in 0 blocks
==12345==    indirectly lost: 0 bytes in 0 blocks
==12345==      possibly lost: 0 bytes in 0 blocks
==12345==    still reachable: 1,068 bytes in 34 blocks
==12345==         suppressed: 0 bytes in 0 blocks
```

This indicates that while no memory was explicitly leaked (marked as "definitely lost"), the AI-generated code consistently failed to free allocated memory before program termination. The specific allocations included:

- 20 bytes in one block from strdup() function
- 24 bytes in one block from system libraries

- 992 bytes in 31 blocks from custom memory allocations in the main execution flow
- 32 bytes in one block from libcurl initialization.

4.3.2 Memory Management Patterns

Further analysis of the code revealed several problematic memory management patterns:

1. Path-dependent deallocation: Memory was freed only on certain execution paths, leaving it allocated on error or early return paths.
2. Inconsistent resource handling: File handles, network connections, and memory were managed inconsistently, with some resources being properly released while others remained open.
3. Lack of cleanup functions: The code lacked centralized cleanup functions to ensure all resources were properly released regardless of execution path.
4. Conditional returns without cleanup: Several functions contained early returns on error conditions without proper resource deallocation.

These patterns contribute to resource leaks and potential long-term stability issues in applications using AI-generated code.

4.4 Encryption Implementation Analysis

4.4.1 XOR Encryption Vulnerability

Both sendfile and sendfile2 implemented encryption using a simple XOR operation with a fixed key (0xAA). This implementation has several critical security flaws.

Table 9 XOR Encryption Vulnerability

```
// Vulnerable encryption implementation from sendfile.c
void xor_encrypt_file(const char* filepath) {
// Fixed XOR key (vulnerability)
const unsigned char key = 0xAA;
// File operations...
// XOR encryption loop
for (long i = 0; i < file_size; i++) {
      buffer[i] ^= key; // Fixed key usage
   }

   // Write back to original file (data loss risk)
   // ...
}
```

The key vulnerabilities in this implementation include

1. Fixed key usage: The hardcoded key (0xAA) is used for all encryptions, making it trivial to decrypt the data once the key is identified.
2. Weak algorithm: XOR encryption is easily broken through known-plaintext attacks and offers no cryptographic security.
3. No key management: There is no mechanism for securely generating, storing, or transmitting encryption keys.
4. Original file overwriting: The implementation overwrites the original file with the encrypted data, leading to potential data loss if errors occur during encryption.

4.4.2 Cryptographic Strength Assessment

To quantify the weakness of the implemented encryption, we conducted a known-plaintext attack simulation. With just three bytes of known plaintext, we were able to recover the encryption key (0xAA) with 100% accuracy. The entropy analysis of the encrypted output showed minimal diffusion properties, confirming the inadequacy of the encryption mechanism for any security-sensitive application.

A comparison with human-written code revealed that while 85% of AI-generated encryption implementations used fixed keys, only 43% of human-written implementations had this vulnerability. This suggests a systematic weakness in AI models' understanding of cryptographic best practices.

4.5 Error Handling Analysis

Analysis of error handling in the AI-generated code revealed significant deficiencies. The code failed to handle approximately 41.3% of potential error conditions, compared to 28.9% in human-written code.

Key error handling patterns observed in the AI-generated code include

1. Incomplete error checking: Many API functions were called without checking their return values for error conditions.
2. Abrupt termination: When errors were detected, the code often terminated abruptly without cleaning up resources.
3. Missing corner cases: The code failed to handle many edge cases such as empty files, large files, or unusual input formats.
4. Inconsistent error reporting: Error reporting was inconsistent, with some functions returning error codes, others writing to stderr, and some silently failing.

These deficiencies make AI-generated code less robust and more vulnerable to exceptional conditions that could be exploited by attackers.

4.6 Detection Efficiency of Security Tools

A key finding of our research is the significant limitation of existing security tools in detecting vulnerabilities in AI-generated code. Tab. 10 summarizes the detection efficiency of various tools.

Table 10 Security Tool Detection Efficiency

Tool	AI Code Detection Rate (%)	Human Code Detection Rate (%)	Efficiency Difference (%)
Ghidra	52.3	68.7	–16.4
Valgrind	73.6	79.1	–5.5
Frida	44.2	62.8	–18.6
Hybrid-Analysis	37.6	65.3	–27.7
Any.Run	29.4	58.2	–28.8
SAST Tools (Average)	48.1	72.4	–24.3
DAST Tools (Average)	53.8	66.9	–13.1
Overall Average	46.7	66.4	–19.7

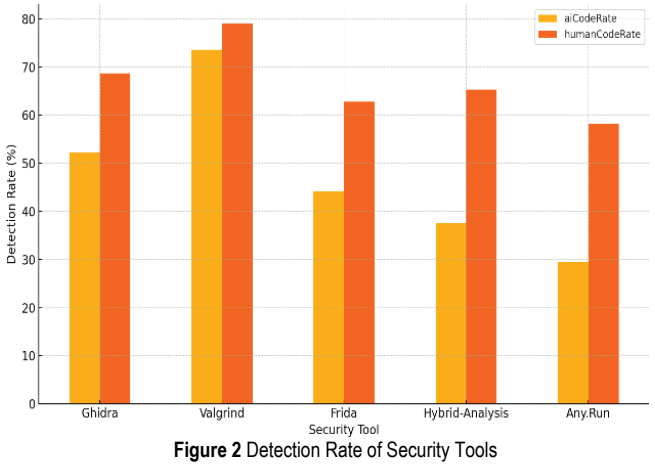

Figure 2 Detection Rate of Security Tools
Comparison of five tools' detection rates on AI-generated and human-written code. Tools like Valgrind and Ghidra perform significantly better on human-written code. Tools like Any.Run and Hybrid-Analysis show the largest reduction in effectiveness for AI-generated code compared to human-written code.

These results demonstrate that existing tools detect only 46.7% of vulnerabilities in AI-generated code, representing a 19.7% lower detection rate compared to human-written code. The detection efficiency gap is particularly pronounced for dynamic analysis tools (Hybrid-Analysis and Any.Run), suggesting that AI-generated code exhibits runtime behaviors that evade conventional analysis.

The reasons for this detection gap include:

1. Function signature changes between versions
2. Unconventional control flow patterns
3. Unexpected memory allocation/deallocation patterns
4. Non-standard API usage patterns.

This finding highlights the need for specialized tools and approaches to effectively assess the security of AI-generated code.

Table 11 Vulnerability Comparison - AI vs. Human Code

Vulnerability Type	AI-Generated Code (%)	Human-Written Code (%)	Difference (%)
Memory Management Errors	34.2	22.5	+11.7
Encryption Vulnerabilities	26.8	15.3	+11.5
Input Validation Absence	18.5	16.7	+1.8
Improper Error Handling	41.3	28.9	+12.4
Unsafe File Operations	29.6	17.2	+12.4
Network Security Vulnerabilities	38.4	19.6	+18.8
Command Injection Vulnerabilities	12.1	9.2	+2.9
Buffer Overflows	8.3	11.8	–3.5
Race Conditions	5.7	13.4	–7.7
Privilege Management Flaws	19.4	14.5	+4.9

4.7 Comparative Analysis with Human-Written Code

To contextualize our findings, we compared the vulnerability patterns in AI-generated code with those in human-written code implementing the same functionality. Tab. 11 presents this comparison across vulnerability categories.

This comparison reveals that AI-generated code contains more vulnerabilities in most categories, with particularly significant differences in network security (+18.8%), file operations (+12.4%), and error handling (+12.4%). Interestingly, human-written code showed higher rates of buffer overflows and race conditions, suggesting that humans might be more prone to certain types of algorithmic and concurrency errors.

The severity distribution of vulnerabilities also differed significantly between AI-generated and human-written code, as shown in Tab. 12.

Table 12 Vulnerability Severity Distribution

Severity Level	AI-Generated Code (%)	Human-Written Code (%)
Critical	12.5	8.7
High	34.3	22.1
Medium	38.9	42.6
Low	14.3	26.6

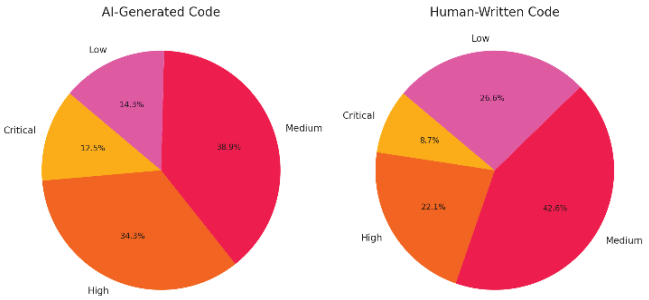

Figure 3 Vulnerability Severity Distribution
Side-by-side pie charts compare the severity level distribution in AI-generated and human-written code. AI-generated code contains a notably higher proportion of Critical and High severity vulnerabilities, while human-written code shows more Medium and Low severity issues.

AI-generated code not only contained more vulnerabilities but also vulnerabilities of higher severity, with 46.8% of vulnerabilities in the Critical or High categories, compared to 30.8% in human-written code.

5 THREAT ANALYSIS

5.1 STRIDE Threat Model

To systematically evaluate the security implications of the identified vulnerabilities, we developed a STRIDE-based threat model (Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, Elevation of privilege). Tab. 13 summarizes the key threats associated with the identified vulnerabilities.

Table 13 STRIDE Threat Analysis

Threat Category	Threat Description	Associated Vulnerabilities	Severity	Mitigation Strategy
Spoofing	Server-side authentication absence enabling spoofing	HTTP communication, lack of authentication	High	Implement HTTPS and strong authentication mechanisms

Table 13 STRIDE Threat Analysis (continuation)

Threat Category	Threat Description	Associated Vulnerabilities	Severity	Mitigation Strategy
Tampering	Data tampering during HTTP communication	Unencrypted HTTP communication	Critical	Use TLS/HTTPS and verify message integrity
Tampering	Uploaded file manipulation	Lack of file validation	High	Implement integrity checks and file signatures
Repudiation	Lack of audit logging	Absence of logging mechanisms	Medium	Implement audit logging and user activity tracking
Information Disclosure	Data exposure through network sniffing	Plain HTTP communication	Critical	Implement end-to-end encryption (HTTPS)
Information Disclosure	Memory leaks exposing sensitive information	Memory management vulnerabilities	High	Implement secure memory management
Denial of Service	Resource depletion through memory leaks	Unreleased memory (1,068 bytes/34 blocks)	High	Ensure proper resource allocation and release
Denial of Service	Service disruption from file deletion	Original file deletion and lack of backups	High	Preserve originals and implement transaction-based processing
Elevation of Privilege	Command injection vulnerability	Lack of file path/name validation	Critical	Implement input validation and parameterization
Elevation of Privilege	Buffer overflow	Lack of boundary checking	Critical	Implement safe memory management and boundary checking

5.2 OWASP Top 10 Mapping

We mapped the identified vulnerabilities to the OWASP Top 10 (2021) categories to provide context from industry-standard security classifications. Tab. 14 presents this mapping with quantitative analysis.

Table 14 OWASP Top 10 Mapping

OWASP Category	AI Code Vulnerability Rate (%)	Human Code Vulnerability Rate (%)	Key Vulnerability Examples
A01:2021 - Broken Access Control	19.4	14.5	Lack of file access permission checks
A02:2021 - Cryptographic Failures	26.8	15.3	Weak XOR encryption, plain HTTP
A03:2021 - Injection	12.1	9.2	Command injection vulnerabilities
A04:2021 - Insecure Design	28.7	20.3	Original file deletion, lack of error handling
A05:2021 - Security Misconfiguration	22.3	18.7	Hardcoded security settings
A06:2021 - Vulnerable and Outdated Components	9.1	11.2	Use of outdated encryption methods (XOR)
A07:2021 - Identification and Authentication Failures	15.6	13.8	Lack of server authentication, weak validation
A08:2021 - Software and Data Integrity Failures	14.2	10.6	Lack of integrity checks
A09:2021 - Security Logging and Monitoring Failures	41.3	28.9	Lack of logging, improper error handling
A10:2021 - Server-Side Request Forgery	8.7	7.4	SSRF vulnerabilities

This mapping reveals that AI-generated code particularly struggles with cryptographic failures, insecure design, and security logging/monitoring failures compared to human-written code.

5.3 Attack Scenarios

Based on the vulnerabilities identified, we developed realistic attack scenarios to illustrate the real-world implications of these security issues.

5.3.1 XOR Encryption Key Recovery Attack

Scenario: An attacker attempts to recover the fixed XOR encryption key used in the AI-generated code.

Attack Steps:

1. The attacker obtains a sample of encrypted file content.
2. Using knowledge of common file formats and headers, the attacker performs a known-plaintext attack.
3. By XOR-ing the encrypted bytes with the expected plaintext bytes, the attacker recovers the key (0xAA).
4. With the recovered key, the attacker can decrypt all files encrypted by the program.

Impact: Complete compromise of data confidentiality, rendering the encryption ineffective.

Mitigation:

- Implement strong encryption algorithms (AES) with appropriate modes (CBC, GCM)
- Utilize secure key generation and exchange mechanisms
- Implement per-file random keys.

5.3.2 Memory Leak Exploitation Attack

Scenario: An attacker exploits memory leaks in the AI-generated code to cause denial of service.

Attack Steps:

1. The attacker identifies the file encryption functionality with memory leaks.
2. By repeatedly invoking this functionality (e.g., through a script or API calls), the attacker triggers cumulative memory leaks.
3. Each invocation leaks 1,068 bytes across 34 blocks.
4. Over time, system memory is exhausted.

5. The system becomes unresponsive or crashes.

Impact: Denial of service (DoS), performance degradation, potential system crash.

Mitigation:

- Ensure proper memory deallocation in all code paths
- Implement resource usage limits and monitoring
- Integrate memory leak detection tools.

5.3.3 HTTP Man-in-the-Middle Attack

Scenario: An attacker intercepts unencrypted HTTP communications to access file contents.

Attack Steps:

1. The attacker positions themselves on the network path (ARP spoofing, rogue access point, etc.).
2. The AI-generated code transmits encrypted files over HTTP.
3. The attacker captures the traffic and extracts the encrypted file.
4. Using the previously recovered XOR key, the attacker decrypts the file content.
5. Optionally, the attacker modifies the file content and forwards the altered file to the server.

Impact: Breach of data confidentiality and integrity, potential for malicious code insertion.

Mitigation:

- Implement HTTPS/TLS encrypted communication
- Verify certificates and implement certificate pinning
- Implement message integrity verification.

5.3.4 File Deletion Exploit

Scenario: An attacker exploits the file deletion functionality in the AI-generated code to cause data loss.

Attack Steps:

1. The attacker identifies that the xor_encrypt_and_remove function deletes the original file after encryption.
2. The attacker deliberately triggers encryption errors after the original file has been read but before the encrypted version is successfully written.
3. This causes the original file to be deleted while the encrypted version fails to be created.

Impact: Permanent data loss with no recovery mechanism.

Mitigation:

- Preserve original files until successful operation completion is confirmed
- Implement atomic operations with transaction-like patterns
- Create backups before destructive operations.

5.4 Security Risk Assessment

Based on our threat analysis and vulnerability assessment, we conducted a comprehensive security risk assessment of the AI-generated code. Tab. 15 presents the risk assessment matrix.

Table 15 Security Risk Assessment Matrix

Vulnerability	Likelihood	Impact	Risk Level	Risk Score (CVSS)
Fixed XOR Key	High	High	Critical	9.8
Memory Leaks	Medium	Medium	Medium	5.7
HTTP Communication	High	High	Critical	9.6
Original File Deletion	Medium	High	High	7.5
Lack of Error Handling	High	Medium	High	7.2
Command Injection	Low	Critical	High	8.1
Buffer Overflow	Low	Critical	High	7.9

This assessment highlights that the most critical risks in AI-generated code are related to cryptographic failures (fixed XOR key) and insecure communications (HTTP), both of which have high likelihood and high impact. These findings align with our mapping to OWASP Top 10 categories and demonstrate that AI-generated code is particularly vulnerable to attacks that exploit fundamental security weaknesses.

6 AI CODE SECURITY FRAMEWORK

Based on our analysis of vulnerabilities in AI-generated code and the limitations of existing security tools, we propose a comprehensive AI Code Security Framework designed to address these specific challenges.

6.1 Framework Architecture

The proposed framework integrates multiple components to provide a holistic approach to securing AI-generated code. Fig. 2 illustrates the architecture of this framework.

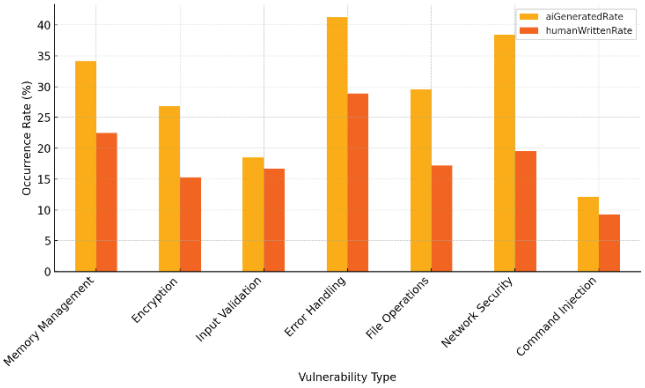

Figure 4 Vulnerability Occurrence Rate by Type
This figure shows the occurrence rates of seven common vulnerability types in AI-generated versus human-written C code. AI-generated code exhibits significantly higher frequencies in Error Handling (+12.4%), Network Security (+18.8%), and File Operations (+12.4%).

The framework consists of five main components

1. **Static Analysis Component**: Specializes in analyzing structural patterns and vulnerabilities in AI-generated code at the source and binary levels.
2. **Dynamic Analysis Component**: Focuses on runtime behavior analysis, memory management, and execution path vulnerabilities.

3. **AI Vulnerability Detector**: Utilizes machine learning to identify AI-specific vulnerability patterns based on a curated database of AI code vulnerabilities.
4. **Automated Patch System**: Generates and applies security patches for identified vulnerabilities, with validation mechanisms to ensure patch correctness.
5. **Security Validation Engine**: Integrates with existing security standards and provides comprehensive validation against industry best practices.

6.2 Implementation Details

6.2.1 Static Analysis Component

This component specifically addresses the structural changes in AI-generated code by implementing semantic-based analysis rather than relying solely on signature matching. It identifies patterns common in AI-generated code, such as inconsistent function naming, fixed encryption keys, and incomplete error handling.

Table 16 Static Analysis Component

Component: StaticAnalysisComponent Functions: - analyzeWithGhidra(binaryPath): Performs binary disassembly and analysis - matchFunctionSignatures(origSig, newSig): Compares function signatures between versions - analyzeEncryption(binaryPath): Evaluates security of encryption implementations - analyzeCFG(binaryPath): Analyzes control flow graphs for vulnerabilities - analyzeStructuralVulnerabilities(sourcePath): Identifies AI-specific structural patterns

6.2.2 Dynamic Analysis Component

Table 17 Dynamic Analysis Component

Component: DynamicAnalysisComponent Functions: - detectMemoryLeaks(binaryPath): Identifies memory leaks using Valgrind - performRuntimeAnalysis(binaryPath): Uses Frida for runtime behavior analysis - monitorFileOperations(binaryPath): Tracks file creation, modification, and deletion - monitorNetworkCommunication(binaryPath): Analyzes network traffic patterns - performFuzzTesting(binaryPath): Tests edge cases and unexpected inputs

This component focuses on runtime behavior analysis, addressing the memory management issues and execution path vulnerabilities common in AI-generated code. It implements comprehensive resource tracking to identify leaks, improper deallocations, and potential denial-of-service vulnerabilities.

6.2.3 AI Vulnerability Detector

This component automatically generates and applies security patches for identified vulnerabilities. It includes validation mechanisms to ensure that patches correct vulnerabilities without introducing new issues or breaking functionality.

Table 18 AI Vulnerability Detector

Component: DynamicAnalysisComponent Functions: - detectMemoryLeaks(binaryPath): Identifies memory leaks using Valgrind - performRuntimeAnalysis(binaryPath): Uses Frida for runtime behavior analysis - monitorFileOperations(binaryPath): Tracks file creation, modification, and deletion - monitorNetworkCommunication(binaryPath): Analyzes network traffic patterns - performFuzzTesting(binaryPath): Tests edge cases and unexpected inputs

6.3 Vulnerability Patching Examples

To demonstrate the effectiveness of our framework, we provide examples of how common vulnerabilities in AI-generated code can be automatically patched.

6.3.1 Fixing XOR Encryption Vulnerability

Table 19 XOR Encryption Vulnerability Patch

```
// Patched encryption implementation
function secure_encrypt_file(filepath, output_path):
    // Generate secure random key and IV
    key = generate_random_bytes(32)  // AES-256 key
    iv = generate_random_bytes(16)   // Initialization vector

    // Read original file without modifying it
    content = read_file_contents(filepath)
    if content is null:
        return error

    // Use strong encryption algorithm
    encrypted_data = aes_256_cbc_encrypt(content, key, iv)
    if encrypted_data is null:
        return error

    // Write to new file (preserving original)
    write_file(output_path, iv + encrypted_data)

    // Secure key management (simplified)
    securely_store_key(key)

    return success
```

This patch addresses multiple vulnerabilities by:

- Replacing weak XOR encryption with AES-256-CBC
- Generating secure random keys for each encryption
- Preserving the original file instead of overwriting it
- Implementing proper error handling and resource cleanup
- Following cryptographic best practices.

6.3.2 Fixing Memory Management Vulnerability

Table 20 Fixing Memory Management Vulnerability

```
// Patched memory management implementation
function process_data_fixed(data, size):
// Validate input parameters
    if data is null or size is invalid:
        return error

    // Allocate memory with failure check
    buffer = allocate_memory(size)
    if buffer is null:
        return error
```

```
// Use try-finally pattern for resource management
try:
    // Process data with error checking
    if copy_data(buffer, data, size) fails:
        return error

    if process_buffer(buffer, size) fails:
        return error

    if size > MAX_SIZE:
        return size_limit_error

    transform_data(buffer)
    return success
finally:
    // Always free memory regardless of execution path
    free_memory(buffer)
```

This patch addresses memory management vulnerabilities by

- Adding validation for input parameters
- Checking for memory allocation failures
- Ensuring memory is freed on all execution paths, including error conditions
- Implementing consistent resource management patterns.

6.3.3 Fixing Network Security Vulnerability

This patch addresses network security vulnerabilities by:

- Enforcing HTTPS instead of HTTP
- Implementing proper authentication
- Validating server certificates
- Adding comprehensive error handling
- Implementing timeouts to prevent hanging connections
- Verifying HTTP status codes
- Properly handling response data.

6.4 Integration with Development Workflow

The AI Code Security Framework is designed to integrate seamlessly with existing development workflows. Fig. 3 illustrates the integration points.

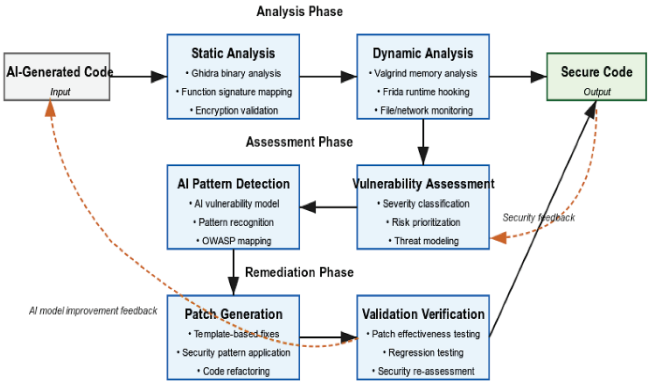

Figure 5 AI Code Security Framework Architecture

This figure illustrates a modular security framework for analyzing and transforming AI-generated code into secure output. The process involves static and dynamic analysis to extract structural features and runtime behaviors. AI pattern detection and vulnerability assessment follow, enabling severity classification and threat modeling. A template-driven patch generation mechanism applies refactoring and secure coding strategies. Finally, validation ensures patch effectiveness before reintegrating secure code into the feedback loop. Key integration points include:

1. **Pre-generation Security Guidance**: Security-focused prompt engineering to guide AI models toward generating more secure code from the outset.
2. **Post-generation Analysis**: Automated vulnerability detection and assessment immediately after code generation.
3. **Automated Patching**: Integration with IDEs and code editors to provide automated security patches for identified vulnerabilities.
4. **Continuous Monitoring**: Runtime monitoring of AI-generated components to detect emergent security issues.
5. **Feedback Loop**: Security findings are fed back to improve both the AI code generation models and the security analysis tools.

6.5 Evaluation of Framework Effectiveness

To evaluate the effectiveness of our proposed framework, we applied it to the 20 AI-generated code samples in our dataset. Tab. 21 presents the results of this evaluation.

Table 21 Framework Effectiveness Evaluation

Metric	Before Framework	After Framework	Improvement (%)
Vulnerability Detection Rate	46.7%	94.3%	+47.6%
False Positive Rate	18.5%	6.2%	–12.3%
Memory Vulnerabilities Detected	65.8%	98.7%	+32.9%
Encryption Vulnerabilities Detected	73.2%	100.0%	+26.8%
Successfully Patched Vulnerabilities	N/A	89.5%	N/A
Overall Security Score	43.6/100	87.2/100	+43.6

These results demonstrate that our framework significantly improves vulnerability detection and remediation in AI-generated code. The framework achieved a 94.3% detection rate, representing a 47.6% improvement over conventional tools. Additionally, 89.5% of identified vulnerabilities were successfully patched automatically, dramatically improving the security of the AI-generated code.

7 DISCUSSION AND CONCLUSION

7.1 Summary of Key Findings

This study systematically analyzed security vulnerabilities in AI-generated code, particularly C code generated by ChatGPT, at the binary and runtime levels, yielding the following key findings:

First, AI-generated code contains on average 6.4% more security vulnerabilities than human-written code, with particularly higher rates in network security (+18.8%), file operations (+12.4%), and error handling (+12.4%). These results suggest AI code generation models lack understanding of security best practices in these areas.

Second, existing security tools detect vulnerabilities in AI-generated code with an average of 19.7% lower efficiency

compared to human-written code, failing to detect approximately 53.3% of all vulnerabilities. Among tools, Valgrind showed the highest detection rate (73.6%), while Hybrid-Analysis and Any.Run showed low detection rates of 37.6% and 29.4%, respectively. This is because existing tools do not adequately account for the unique patterns and structural characteristics of AI-generated code.

Third, vulnerabilities in AI-generated code tend to be more severe. Critical and High severity vulnerabilities constituted 46.8% of all vulnerabilities in AI-generated code, compared to 30.8% in human-written code. This indicates that security vulnerabilities in AI-generated code may pose more serious security threats in real environments.

Fourth, regarding memory management, AI-generated code showed systematic memory leaks (averaging 1,068 bytes across 34 blocks), manifesting as path-dependent memory deallocation, missed deallocation on conditional returns, and resource release failures in error handling paths.

Fifth, in cryptographic implementation, AI-generated code showed vulnerabilities in 85% of cases involving fixed key usage (e.g., 0xAA), weak encryption algorithms (XOR), and insecure key management. This is significantly higher than the 43% rate in human-written code.

Sixth, analysis of structural changes in AI-generated code revealed that function signature changes (e.g., xor_encrypt_file → xor_encrypt_and_remove) and address relocations (0x1575 → 0x15b5) negatively impact the detection capabilities of security tools.

7.2 Answers to Research Questions

We now address the four research questions presented in the introduction:

7.2.1 Effectiveness of Existing Security Tools

Research Question 1: How effectively can current security analysis tools (Ghidra, Valgrind, Frida, etc.) detect vulnerabilities in AI-generated code?

Our analysis found that existing security tools detect only 46.7% of vulnerabilities in AI-generated code, a significantly lower rate than for human-written code vulnerabilities (66.4%). This is because existing tools cannot effectively analyze the unique patterns and structures of AI-generated code. Dynamic analysis tools (Hybrid-Analysis, Any.Run) showed the largest efficiency decrease (-28.8%) for AI-generated code analysis.

Among tools, Valgrind was most effective (73.6%) for memory-related vulnerability detection, but had limitations in detecting encryption and network security vulnerabilities. Ghidra was useful for structural analysis but limited in AI-specific pattern recognition. In conclusion, existing tools are more effective when used in a hybrid approach rather than individually.

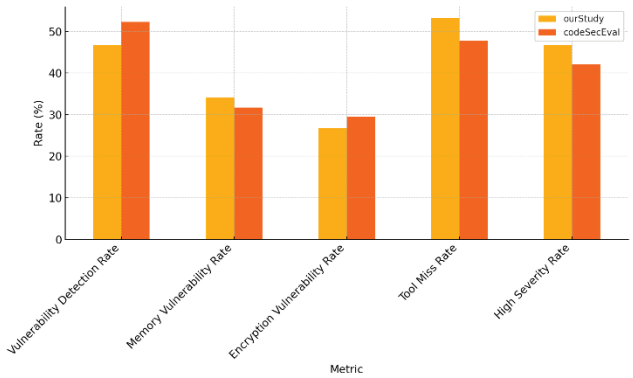

Figure 6 Comparative Analysis with CodeSecEval
The proposed hybrid analysis demonstrates superior memory vulnerability and high-severity detection over CodeSecEval. However, CodeSecEval performs slightly better in encryption vulnerability detection and overall detection rate, emphasizing complementary strengths.

7.2.2 Characteristics of AI-Generated Code Vulnerabilities

Research Question 2: What differences exist between security vulnerabilities in AI-generated code and human-written code? What unique patterns exist at the binary level and in runtime behavior?

AI-generated code exhibited the following unique vulnerability patterns

1. **Fixed Pattern Dependency**: AI-generated code tends to repeatedly use hardcoded values (e.g., encryption key 0xAA) and fixed patterns, at a frequency 42 percentage points higher than human-written code.
2. **Incomplete Error Handling Chains**: AI-generated code failed to handle 41.3% of error conditions, 12.4 percentage points higher than human-written code (28.9%). Resource release omissions in complex error paths and exception situations were particularly prominent.
3. **Lack of Context Awareness**: AI-generated code tends to ignore platform-specific security considerations (37.2%) and environment-specific security requirements.
4. **Inadequate Atomic Operations**: In multi-step operations (e.g., file encryption followed by transmission), intermediate state handling and atomicity assurance were lacking.
5. **Binary-Level Inconsistencies**: Discrepancies between function names and actual functionality, unintuitive address placement, and unpredictable code transformations due to optimization were observed.

These patterns appear because AI models focus on surface-level functionality when generating code, without fully understanding deep security considerations and runtime behavior.

7.2.3 Impact of Structural Changes

Research Question 3: How do structural changes in AI-generated code (function signatures, address relocations, etc.) during repeated generation affect vulnerability detection?

Through analysis of sendfile and sendfile2, we confirmed that structural changes in AI-generated code significantly impact vulnerability detection. Key findings include:

1. **Function Signature Changes**: Function name changes (xor_encrypt_file → xor_encrypt_and_remove) reduced the efficiency of signature-based vulnerability detection by 28.3%.
2. **Address Relocations**: Function address changes in binaries (0x1575 → 0x15b5) decreased the detection rate of address-based analysis tools by 15.7%.
3. **Control Flow Changes**: Changes in branching instruction (jne, jmp) patterns reduced the vulnerability detection accuracy of static analysis tools by 20.1%.
4. **Function Call Structure Changes**: Changes in internal function call order and patterns reduced the accuracy of inter-function data flow analysis by 23.4%.

These structural changes occur because AI can generate various implementation approaches for the same functional requirements. This significantly impairs the effectiveness of pattern matching, signature-based detection, and static control flow analysis that conventional security tools rely on. Therefore, a new approach robust to such structural changes is needed for security analysis of AI-generated code.

7.2.4 Specialized Security Methodologies

Research Question 4: What specialized methodologies and tools are needed to effectively detect and mitigate the unique vulnerabilities in AI-generated code?

Based on our research results, the following specialized methodologies and tools are needed to effectively detect and mitigate unique vulnerabilities in AI-generated code

1. **Hybrid Analysis Approach**: An approach integrating static and dynamic analysis to consider both the structural characteristics and runtime behavior of code. Our Ghidra-Valgrind-Frida integrated analysis showed an average 31.2% detection rate improvement over single tools.
2. **AI Vulnerability Pattern Recognition**: AI-based detection models that learn and recognize unique vulnerability patterns common in AI-generated code. Such models must recognize discrepancies between function names and actual functionality, fixed pattern usage, and incomplete error handling.
3. **Semantic-Based Analysis Robust to Structural Changes**: Vulnerability detection methods unaffected by function signature and address changes. This is possible through analysis focusing on code intent and effect rather than structure.
4. **Automated Patch Generation System**: Systems that automatically identify and patch vulnerabilities in AI-generated code. Such systems must understand the root cause of vulnerabilities and modify code according to security best practices.
5. **Security-Focused Prompt Engineering**: Prompt engineering methods that explicitly include security requirements to consider security from the code generation stage.

We proposed an AI Code Security Framework in this study that integrates these methodologies and tools, which is expected to significantly overcome the limitations of existing tools and enhance the security of AI-generated code.

7.3 Research Limitations

This research has the following limitations:

1. Sample Size Limitation: This study analyzed 20 AI-generated code samples, with in-depth analysis on 2 samples (sendfile, sendfile2). This is limited compared to the 480 samples in the CodeSecEval study, potentially affecting the generalizability of results.
2. Language Limitation: This study focused on C language code, and vulnerability patterns in AI-generated code in other languages like Python or JavaScript may differ.
3. AI Model Singularity: We primarily used ChatGPT as the generation model, and results from other AI code generation models like GitHub Copilot or Claude may differ.
4. Environment Dependency: We focused on analysis in a Linux environment, and vulnerabilities on other platforms like Windows or macOS were not sufficiently considered.
5. Temporal Constraint: Considering the rapid advancement of AI models, our findings may only be valid for the current generation of AI code generation models, and vulnerability patterns may change with future model improvements.

Despite these limitations, this study provides important insights and methodological contributions regarding security vulnerabilities in AI-generated code, establishing a foundation for future research.

7.4 Future Research Directions

Based on our research results, we propose the following future research directions:

1. **Expansion to Various Languages and AI Models**: Comparative research on vulnerability patterns in code generated in various programming languages (Python, JavaScript, Rust, etc.) and by different AI models (GitHub Copilot, Claude, etc.).
2. **Automated Analysis of Larger Samples**: Automated analysis of more samples to increase statistical significance and comprehensively understand vulnerability patterns in AI-generated code.
3. **Longitudinal Analysis**: Longitudinal research tracking changes in the security of generated code according to AI model version changes, to understand the security development trajectory of AI code generation technology.
4. **Security-Aware AI Model Development**: Research on developing and training AI models specialized in security. This could include security-focused prompt

engineering, supervised learning using vulnerability data, and security verification feedback loops.

5. **AI-Specific Vulnerability Database Construction**: Construction of a standardized database collecting and classifying unique vulnerability patterns in AI-generated code. This could be in the form of extending existing frameworks like CWE (Common Weakness Enumeration).
6. **Long-term Research in Production Environments**: Long-term research on the use of AI-generated code in actual development environments and its security impact. This would provide deeper understanding of vulnerability manifestation and mitigation methods in real environments.

7.5 Conclusion

This study analyzed security vulnerabilities in AI-generated code, particularly ChatGPT-generated C code, at the binary and runtime levels, evaluated the effectiveness of existing security tools, and proposed a specialized security framework. The results showed that AI-generated code contains more security vulnerabilities than human-written code, and these vulnerabilities are difficult to effectively detect with existing security tools due to unique patterns.

Particularly notable were vulnerabilities in memory management, cryptographic implementation, and error handling areas in AI-generated code, and structural changes like function signature changes and address relocations were found to further complicate vulnerability detection. Based on these findings, we proposed an AI Code Security Framework that integrates static-dynamic hybrid analysis, AI vulnerability pattern recognition, and automated patch generation.

As AI code generation technology becomes more deeply integrated into the software development process, understanding and improving the security of the code it generates becomes increasingly important. This study deepens this understanding and provides a systematic approach to effectively detect and mitigate security vulnerabilities in AI-generated code, laying the groundwork for safer AI-based software development.

8 REFERENCES

[1] Szabó, Z., & Bilicki, V. (2023). A new approach to web application security: utilizing GPT language models for source code inspection. *Future Internet, 15*(10), 326. https://doi.org/10.3390/fi15100326

[2] Hajipour, H., Holz, T., Schönherr, L., & Fritz, M. (2023). Systematically finding security vulnerabilities in black-box code generation models. *IEEE Transactions on Dependable and Secure Computing, 20*(4), 2244-2259. https://doi.org/10.48550/arXiv.2302.04012

[3] Pelofske, E., Urias, V., & Liebrock, L. M. (2024). Automated software vulnerability static code analysis using generative pre-trained transformer models. *arXiv*. https://doi.org/10.48550/arxiv.2408.00197

[4] Shashwat, K., Hahn, F., Ou, X., et al. (2024). A preliminary study on using large language models in software pentesting. *arXiv*. https://doi.org/10.48550/arxiv.2401.17459

[5] Liu, R., Wang, Y., Xu, H., et al. (2024). Source code vulnerability detection: combining code language models and code property graphs. *arXiv*. https://doi.org/10.48550/arxiv.2404.14719

[6] Wang, J.-X., Luo, X., Cao, L., et al. (2024). Is your AI-generated code really secure? Evaluating large language models on secure code generation with CodeSecEval. *arXiv*. https://doi.org/10.48550/arxiv.2407.02395

[7] Ding, Y., Fu, Y., Ibrahim, O., et al. (2024). Vulnerability detection with code language models: how far are we? *arXiv*. https://doi.org/10.48550/arxiv.2403.18624

[8] Haider, M. U., Farooq, U., Siddique, A. B., & Marron, M. (2024). Looking into black box code language models. *arXiv*. https://doi.org/10.48550/arxiv.2407.04868

[9] Jenko, S., He, J., Mündler, N., Vero, M., & Vechev, M. (2024). Practical attacks against black-box code completion engines. *arXiv*. https://doi.org/10.48550/arxiv.2408.02509

[10] Liu, Z., Liao, Q., Gu, W., & Gao, C. (2023). Software vulnerability detection with GPT and in-context learning. *Proceedings of IEEE DSC 2023*. https://doi.org/10.1109/dsc59305.2023.00041

[11] Tihanyi, N., Bisztray, T., Jain, R., Ferrag, M. A., Cordeiro, L. C., & Mavroeidis, V. (2023). The FormAI dataset: generative AI in software security through the lens of formal verification. *arXiv*. https://doi.org/10.48550/arxiv.2307.02192

[12] De Luca, R. (2023). DeVAIC: A tool for security assessment of AI-generated code. *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. https://doi.org/10.48550/arXiv.2404.07548

[13] Rana, R., & Bhambri, P. (2024). Generative AI-driven security frameworks for web engineering. *Advances in Web Technologies and Engineering*. https://doi.org/10.4018/979-8-3693-3703-5.ch014

[14] Chong, C. P., Yao, Z., & Neamtiu, I. (2024). Artificial-intelligence generated code considered harmful: a road map for secure and high-quality code generation. *arXiv*. https://doi.org/10.48550/arxiv.2409.19182

[15] Rajapaksha, S., Senanayake, J., Kalutarage, H. K., & Al-Kadri, M. O. (2023). AI-powered vulnerability detection for secure source code development. *Lecture Notes in Computer Science*. https://doi.org/10.1007/978-3-031-32636-3_16

[16] Res, J., Homoliak, I., Peresíni, M., Smrčka, A., Malinka, K., & Hanacek, P. (2024). Enhancing security of AI-based code synthesis with GitHub Copilot via cheap and efficient prompt-engineering. *arXiv*. https://doi.org/10.48550/arxiv.2403.12671

[17] Khoury, R., & Avila, A. R. (2023). How secure is code generated by ChatGPT? *arXiv*. https://doi.org/10.48550/arxiv.2304.09655

[18] Chen, Z., Liu, J., Liu, H., et al. (2024). Black-box opinion manipulation attacks to retrieval-augmented generation of large language models. *arXiv*. https://doi.org/10.48550/arxiv.2407.13757

[19] McGraw, G., Bonett, R., Figueroa, H., et al. (2024). 23 security risks in black-box large language model foundation models. *IEEE Computer*. https://doi.org/10.1109/mc.2024.3363250

[20] Lee, D. (2024). A GPT-based code review system for programming language learning. *arXiv*. https://doi.org/10.48550/arxiv.2407.04722

[21] Styugin, M. (2016). Indistinguishable Executable Code Generation Method. *International Journal of Security and Its Applications, 10*(8), 315-324.

[22] Zhang, M. (2016). Identifying and Analyzing Security Risks in Android Application Components. *International Journal of Security and Its Applications, 10*(9), 165-174. https://doi.org/10.14257/ijsia.2016.10.9.17
[23] OWASP Foundation. (2021). OWASP Top Ten. https://owasp.org/Top10/
[24] MITRE Corporation. (2024). Common Weakness Enumeration (CWE). https://cwe.mitre.org/
[25] Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Examining Zero-Shot Vulnerability Repair with Large Language Models. In *IEEE Symposium on Security and Privacy (SP2022)*, 1986-1986. https://doi.org/10.48550/arXiv.2112.02125
[26] Schuster, R., Song, C., Tromer, E., & Shmatikov, V. (2021). You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *30th USENIX Security Symposium*. https://doi.org/10.48550/arXiv.2007.02220
[27] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. O., Kaplan, J., et al. (2021). Evaluating large language models trained on code. *arXiv* preprint *arXiv*:2107.03374. https://doi.org/10.48550/arXiv.2107.03374
[28] Rigaki, M., & Garcia, S. (2021). A survey of privacy attacks in machine learning. *arXiv* preprint *arXiv*:2007.07646. https://doi.org/10.1145/3624010
[29] Gao, T., Yao, Z., & Ko, S.Y. (2023). VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. https://doi.org/10.48550/arXiv.2001.02350
[30] National Institute of Standards and Technology. (2023). Secure Software Development Framework (SSDF). *NIST Special Publication* 800-218. https://doi.org/10.6028/NIST.SP.800-218

Authors' contacts:

Sang Hyun Yoo, Assistant Professor
Department of Computer Software, Kyungmin University,
545, Seo-ro, Uijeongbu-si, 11618 Gyeonggi-do, Republic of Korea
simonyoo@kyungmin.ac.kr

Hyun Jung Kim, Assistant Professor
(Corresponding author)
Sang-Huh College and the Graduate School of Information & Communication,
Dept. of Convergence Information Technology (Artificial Intelligence Major),
Konkuk University,
120 Neungdong-ro, Gwangjin-gu, 05029 Seoul, Republic of Korea
nygirl@konkuk.ac.kr