Analysis of Malicious Code Insertion Detection in AI Coding Agent-Generated Code Using SHA-256 Hash Baseline and Static Security Rules

Vandega Arozan Musholine - 18223010
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 18223010@std.stei.itb.ac.id, egaarozan20@gmail.com

Abstract—AI coding agents can accelerate implementation work, but their generated code can still include unsafe constructs associated with malicious code insertion. This paper analyzes malicious code insertion detection in AI coding agent-generated code using a SHA-256 hash baseline and static security rules. The detector reads official SecCodeBench result files directly from a ZIP archive, extracts each generatedCode value as text, computes SHA-256 digests against an empty safe baseline, and applies static rules for dynamic code execution, command execution, unsafe deserialization, server-side request forgery, path traversal, template injection, and credential access. The detector is read-only: it does not execute, import, compile, or redistribute generated code. In an experiment on 4213 code outputs generated from scratch in SecCodeBench version 2.2.0, all outputs differed from the baseline and all matched at least one static rule. The most frequent indicators were dynamic code execution (1137), command execution (1077), server-side request forgery or unvalidated request (677), path traversal (669), unsafe deserialization (576), and template injection (340). The result shows that SHA-256 provides an integrity signal, while static rules provide interpretable malicious code insertion indicators.

Keywords—SHA-256; hash baseline; static security rules; AI coding agent; malicious code insertion; SecCodeBench

I. INTRODUCTION

AI coding agents are now used in ordinary programming workflows to draft functions, patch bugs, and explain code. The security issue is that generated code can look correct while still containing constructs commonly associated with malicious behavior, such as runtime evaluation, shell invocation, unsafe deserialization, or unvalidated network and file access.

This paper addresses a narrow detection question: whether an AI coding agent output differs from a safe baseline and whether the changed output contains static indicators of malicious code insertion. In this paper, malicious code insertion is an operational label for inserted code constructs associated with malicious-capable behavior; it does not claim or infer model intent.

The cryptographic component is a SHA-256 hash baseline. SHA-256 provides a fixed-size digest for a code string and is used only as an integrity signal: it detects whether the generated output is identical to the baseline. The security meaning comes from static security rules applied after the hash comparison.

The implementation reads SecCodeBench official result files directly from a ZIP archive, extracts generatedCode as text, computes digests, applies rule matching, and writes CSV outputs. It never imports, compiles, executes, or stores generated code as runnable source files.

II. RELATED WORK

Recent work on AI-generated code security usually defines a precise evaluation unit, uses a benchmark or repository-scale dataset, separates functional correctness from security posture, and reports quantitative results. SecCodeBench-V2 is the closest benchmark reference because it provides security-focused code generation tasks and official result releases [1], [2]. This paper does not rerun SecCodeBench verifiers, it performs a read-only analysis of the official result JSON files.

The paper also follows the empirical style of recent static-analysis studies on AI-generated code. Prior work has used tools such as SonarQube or CodeQL to show that code can be functionally acceptable while still containing security weaknesses [3]-[6]. This study keeps the claim narrower: it reports hash changes and textual malicious code indicators within the selected SecCodeBench result subset.

III. PROBLEM DEFINITION

The practical problem is that a developer needs a fast first-pass check for AI-produced code before it is copied into a project. A generated function may pass simple tests while still using dangerous primitives such as eval, subprocess, unsafe deserialization, user-controlled URLs, or user-controlled file paths.

The input is a code string produced by an AI coding agent. In the experiment, this string is the generatedCode field from SecCodeBench official model-testcase result JSON files. The baseline is the empty safe string, representing the state before the AI writes code in the code-from-scratch setting.

A generated output is labeled as malicious code insertion only when two conditions hold: the SHA-256 digest differs from the baseline digest, and at least one static security rule is matched. Thus, the hash component records change and the rule component explains the security-relevant indicator.

A. Mathematical and Theoretical Basis Used

Only two theoretical elements are used in the implementation: SHA-256 for code-string change detection and static rule matching for malicious-code-indicator labeling. Let b be the baseline string and o be an AI-generated code string. The scanner computes:

$$h_b = SHA256(b), \; h_o = SHA256(o) \quad (1)$$

$$HashChanged(o) = \begin{cases} 1, & h_b \neq h_o \\ 0, & h_b = h_o \end{cases} \quad (2)$$

For static rules, each rule r_i has a pattern set P_i. The matched malicious-code-indicator set is:

$$R(o) = \{r_i \mid P_i \; appears \; in \; o\} \quad (3)$$

The final detector is defined as:

$$M(o) = \begin{cases} 1, & HashChanged(o) = 1 \wedge) \\ 0, & \text{lainnya} \end{cases} \quad (4)$$

Equation (4) is the exact decision rule used by the scanner. It does not prove exploitability or malicious intent; it records that the output is new relative to the baseline and contains at least one malicious code indicator.

The summarizer uses the same decision variables to compute aggregate results. Let D be the analyzed dataset. The total number of detected malicious-code-insertion records, the count for each finding label, and the functional-security mismatch count are computed as:

$$N = |D|, \quad F = \sum_{o \in D} M(o) \quad (5)$$

$$C(r_i) = \sum_{o \in D} 1[r_i \in R(o)] \quad (6)$$

$$G = \sum_{o \in D} 1[FunctionalSuccess(o) \wedge SecurityFailure(o)] \quad (7)$$

Equations (5)-(7) correspond to the reported CSV summaries: total analyzed records, total detected records, finding distribution, and records that pass functional tests while failing security tests. No cryptographic encryption, digital signature, MAC, or key management model is used in this work.

From a cryptographic perspective, SHA-256 is used only for deterministic change recording. The paper does not rely on secrecy, authentication, or non-repudiation properties. From a program-analysis perspective, the static rules are used only as lexical indicators of malicious-code-related constructs, they do not replace manual review or dynamic validation.

TABLE I. Operational Definition Used in This Paper

Element	Operational meaning	Role in detection
Baseline	Empty safe code string	Reference state before AI output
Generated code	generatedCode text from SecCodeBench result JSON	Analyzed AI coding agent output
SHA-256 digest	Fixed digest of baseline and generated code	Integrity comparison signal
Static security rules	Text patterns for malicious code indicators	Security classification signal
Malicious code insertion	hash_changed is true and at least one rule is matched	Final detection label

IV. DESIGN OF THE DETECTION PIPELINE

The pipeline combines a cryptographic integrity check with lightweight static analysis. SHA-256 is applied to the baseline and generated code string, the digest comparison records whether the output changed. Static rules are then applied to the generated code string to explain the type of malicious code indicator. Each static rule is a transparent textual rule for a construct commonly associated with security risk.

Examples include eval or exec for dynamic code execution, subprocess or shell invocation for command execution, pickle or unsafe YAML loading for deserialization, and unvalidated request or file-path use for SSRF and path traversal. The pipeline is read-only. It opens the SecCodeBench ZIP archive, parses JSON, extracts generatedCode as text, computes SHA-256, scans rules, and writes CSV rows. Generated code remains input data and is never executed.

Fig. 1 summarizes the detector: SHA-256 records the change signal, while static rules provide the malicious-code-indicator label.

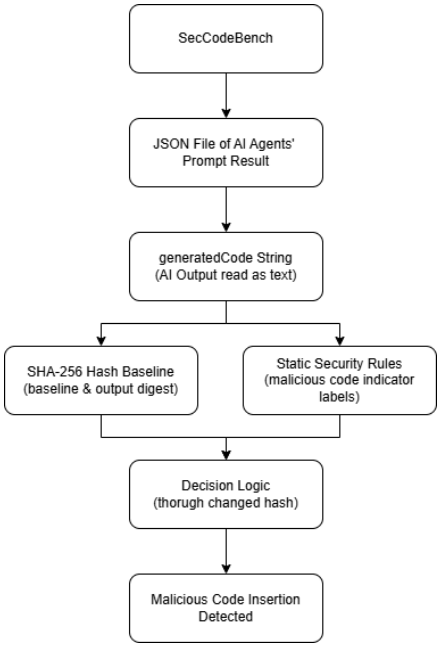


Fig. 1. Proposed pipeline for analyzing malicious code insertion detection

.

TABLE II. Static Security Rule Taxonomy

Finding label	Typical pattern family	Security interpretation
dynamic_code_execution	runtime evaluation constructs	Possible code injection surface
command_execution	shell or subprocess invocation	Possible command injection surface
unsafe_deserialization	unsafe object deserialization	Possible object injection or arbitrary object loading
ssrf_or_unvalidated_request	network request from input-controlled URL	Possible server-side request forgery surface
path_traversal_risk	file path usage from input	Possible unauthorized file access surface
template_injection_risk	string-rendered templates	Possible server-side template injection surface
credential_or_secret_access	token, secret, or environment access	Possible credential exposure surface

V. IMPLEMENTATION

The implementation consists of a Python scanner, a summarizer, and a preview utility. The scanner writes the main CSV, the summarizer produces grouped result tables, and the preview utility is used only for manual inspection of one generatedCode value. The scanner uses Python zipfile so the SecCodeBench repository does not need to be extracted. This avoids Windows path issues and prevents generated code from becoming runnable source files in the project directory.

For each record, the scanner writes source path, model identifier, test case identifier, baseline digest, generated-code digest, hash_changed, finding labels, generated-code length, and preserved SecCodeBench metadata such as functional and security test status.

TABLE III. Scanner Output Fields Used in the Experiment

Field group	Representative fields	Purpose
Source metadata	source_json, model_id, test_case_id, severity, round_id	Traceability without copying generated code
Cryptographic comparison	baseline_sha256, generated_sha256, hash_changed	Records integrity difference from baseline
Static detection	findings, is_flagged	Explains the malicious code indicator detected
Benchmark context	functional_test, security_test, score	Preserves original SecCodeBench result context
Size indicator	generated_code_length	Supports sanity checking and descriptive analysis

The core procedure is summarized below. The important safety point is that generatedCode is treated as text data.

```
for each official result JSON in version 2.2.0:
    parse JSON as text data
    read generatedCode from the
code-from-scratch output
    baseline_digest = SHA256(empty_string)
    generated_digest = SHA256(generatedCode)
    hash_changed = baseline_digest !=
generated_digest
    findings =
matched_static_rules(generatedCode)
    write CSV row
```

A. Read-Only Implementation Checks

Before the experiment was accepted, the implementation was checked against read-only constraints. The scanner opens the benchmark archive with zipfile, reads JSON bytes, decodes them as text, and extracts generatedCode as a string value. The generated code is never passed to import, eval, exec, subprocess, node, python, Docker, or a network verifier.

The only executed program is the scanner itself. The potentially unsafe code remains passive input data, similar to a string inside a log file. This boundary matters because several selected test cases intentionally contain patterns related to code injection, command execution, SSRF, path traversal, deserialization, and template injection. Executing those samples would change the project from static analysis into dynamic testing, which is outside the scope of this work.

The scanner output was also designed to reduce unnecessary redistribution of generated code. The main CSV stores source JSON path, SHA-256 digests, labels, lengths, and benchmark metadata. It does not store the full generatedCode body. When manual inspection is needed, the preview utility displays a limited snippet from one selected JSON file without writing it as a runnable source file.

These checks support the reproducibility and safety claims of the paper. Reproducibility comes from deterministic hashing and fixed rule definitions, safety comes from treating generated code as text. Therefore, the experiment can be repeated locally without executing untrusted code or republishing the full benchmark output corpus.

B. Mapping from Theory to Program Output

Equations (1)-(4) appear directly in the CSV fields. baseline_sha256 corresponds to h_b, generated_sha256 corresponds to h_o, hash_changed corresponds to (2), findings corresponds to R(o), and is_flagged corresponds to M(o). Equations (5)-(7) appear in the summary files, especially summary_by_finding.csv and the overall detection totals.

This mapping is intentionally simple. The method does not infer whether a model is malicious, does not prove reachability of a vulnerability, and does not estimate exploit probability. It records a smaller and auditable fact: a new AI-generated code string differs from the baseline and contains one or more malicious code indicators.

VI. EXPERIMENTAL SETUP

The experiment uses SecCodeBench version 2.2.0 official model-testcase result files. This single version is used to avoid mixing result formats, model coverage, and test-case definitions across benchmark releases. The analyzed subset contains 4213 outputs in which the AI was asked to create code from scratch based on a prompt rather than repair an existing vulnerable program. This setting fits the baseline model because the initial state is an empty safe string. The subset covers 36 model identifiers and 13 security-focused test cases in Python and Node.js, including code injection, command injection, SSRF, path traversal, unsafe deserialization, and server-side template injection.

TABLE IV. Dataset Summary

Metric	Value
Benchmark source	SecCodeBench official result JSON files
Benchmark version	2.2.0
Task mode	Code generated from scratch
Generated outputs analyzed	4213
Model identifiers	36
Security-focused test cases	13
Baseline	Empty safe code string
Primary output file	seccodebench_generated_scan.csv

VII. RESULTS

The scanner produced 4213 analyzed records. All records have a SHA-256 digest different from the empty baseline, which is expected because each analyzed sample is a code-from-scratch output. All 4213 records were also flagged by at least one static rule. This does not mean that every output is a complete exploit. It means that the selected security-focused subset contains at least one textual malicious code indicator per analyzed output according to the implemented rules. The preserved SecCodeBench metadata show that 4050 records are functionally successful, while 3057 records fail security tests. In 2894 records, the output passes the functional test but fails the security test, which supports the need for a separate security-oriented check.

TABLE V. Hash and Rule Detection Summary

Indicator	Count
Generated outputs analyzed	4213
Hash different from baseline	4213
Flagged by at least one static rule	4213
Functionally successful according to source metadata	4050
Security test failed according to source metadata	3057
Functional success but security failure	2894
Rows with multiple finding labels	274
Median generated-code length in characters	1100

TABLE VI. Distribution of Static Rule Findings

Finding label	Count
dynamic_code_execution	1137
command_execution	1077
ssrf_or_unvalidated_request	677
path_traversal_risk	669
unsafe_deserialization	576
template_injection_risk	340
credential_or_secret_access	11

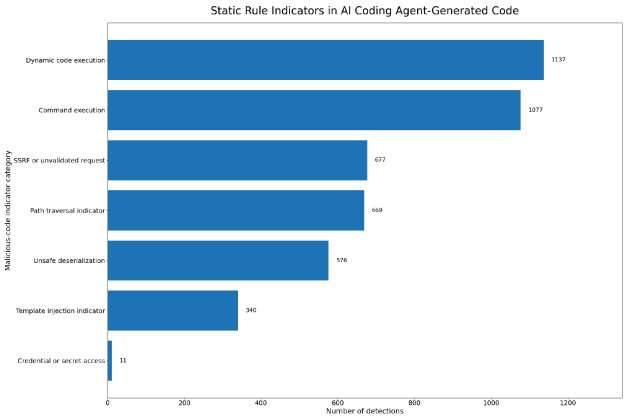


Fig. 3. Distribution of static rule findings across the analyzed AI coding agent-generated code outputs.

Table VI and Fig. 3 show that dynamic_code_execution and command_execution are the largest categories. This is consistent with the selected code-injection and command-injection tasks, and it shows that the scanner can attach an interpretable malicious-code-indicator label to changed code. Fig. 3 visualizes the finding distribution from the experimental CSV summary.

SSRF, path traversal, unsafe deserialization, and template injection findings show that the observed indicators are not limited to execution primitives. The rule taxonomy therefore gives more detail than a single malicious or clean label.

TABLE VII. Distribution by Security-Focused Test Case

Test case identifier	Records
CodeInjectionJavaxScript	360
CommandInjectionChildprocess	360
CommandInjectionSubprocessRun	360
SSRFRequests	359
CommandInjectionSubprocessPopen	357
PathTraversalFsRead	357
CodeInjectionEval	348
DeserializationPickle	343
SSTIFlaskRenderTemplateString	340
SSRFUrllib	318
PathTraversalFsWrite	283
DeserializationPyYAML	233
CommandInjectionShell	195

VIII. CASE STUDY INTERPRETATION

A CSV row is interpreted through three signals: whether the code changed relative to the baseline, which malicious code indicator was matched, and what functional or security outcome is preserved from SecCodeBench metadata. This makes each row auditable without executing the generated code. The scanner stores hashes, labels, and source JSON paths rather than full generated code. This keeps the artifact focused on analysis and avoids spreading unsafe snippets in the paper.

For a code-injection task, `hash_changed = true` and `finding = dynamic_code_execution` means that the AI output introduced a runtime-evaluation pattern relative to the baseline. The safe interpretation is not model intent, but the presence of a security-sensitive construct that requires review. For a command-injection task, `finding = command_execution` means the output contains a shell or subprocess pattern. A reviewer should then check argument handling, shell usage, allowlists, and input validation before accepting the code.

TABLE VIII. Representative Interpretation of Finding Labels

Finding label	Typical review question	Safer engineering direction
dynamic_code_execution	Can user input reach runtime evaluation?	Use structured parsing or an allowlist interpreter
command_execution	Can user input alter an OS command?	Avoid shell execution and validate arguments
ssrf_or_unvalidated_request	Can input control the requested URL?	Restrict destination hosts and block internal ranges
path_traversal_risk	Can input escape the intended directory?	Normalize paths and enforce a fixed root directory
unsafe_deserialization	Can untrusted bytes create objects?	Use safe formats or restricted loaders
template_injection_risk	Can input become executable template syntax?	Use escaped variables and avoid string-rendered templates

IX. DISCUSSION

The findings are best read as a targeted security analysis, not as a general ranking of model safety. The selected test cases are security-focused, so high detection counts are expected. The contribution is the detection record: hash-based change plus an interpretable malicious-code-indicator label.
The two components answer different questions. The SHA-256 baseline answers whether the output changed. The static rules answer what malicious code indicator appears in the changed output. Combining them makes the label more informative than either component alone.

The preserved SecCodeBench metadata show that functional success and security success can diverge. This supports the use of security checks even when generated code passes ordinary functional tests. The method is conservative because it does not reproduce SecCodeBench dynamic validation. It provides a safe first-pass inspection layer that can run locally without Docker verifiers, network activity, or executing generated code.

X. SAFETY AND LICENSE CONSIDERATIONS

The implementation does not execute untrusted code. It reads generatedCode as text, computes SHA-256 over that text, applies static rules, and writes analysis rows. It does not import generated code, call eval or exec on it, run subprocesses from it, or access verification URLs. The allowed operations are limited to safe static-processing activities:

- Reading official JSON files as UTF-8 text
- Extracting the `generatedCode` field as an input string
- Computing the SHA-256 digest of the baseline and AI-generated output
- Matching the output against predefined static security rules
- Writing the resulting hash values, findings, labels, and summaries into CSV files

Several operations are explicitly avoided to prevent unsafe execution. Therefore, the experiment remains a static analysis process, where the generated code is inspected for malicious code insertion patterns without being run. The study also avoids redistributing SecCodeBench itself. This is consistent with using the Apache-licensed benchmark as a cited research artifact rather than republishing the full dataset.

XI. LIMITATIONS

The scanner uses static textual rules, not semantic program analysis. It may miss obfuscated variants or data-flow-dependent vulnerabilities, and it may flag a security-sensitive API that is used safely. Therefore, the output should be treated as a review signal rather than a final proof of exploitability. The empty baseline is suitable for code-from-scratch outputs, but not for repair studies where a vulnerable template already exists. For repair studies, the baseline should be the original file or a known-good version.

The analysis is limited to SecCodeBench version 2.2.0 and selected Python and Node.js security-focused test cases. This improves consistency but limits generalization to other languages and benchmark versions. The work does not measure model intent. Malicious code insertion means a changed output plus a matched malicious code indicator, not proof that the AI coding agent was intentionally malicious.

XII. CONCLUSION

This paper presented a technical report on malicious code insertion detection in AI coding agent-generated code using a SHA-256 hash baseline and static security rules. The contribution is a detection design, a read-only implementation, and an experiment on 4213 SecCodeBench version 2.2.0 code outputs generated from scratch. The result shows that SHA-256 can record code-string change, while static rules can explain the malicious code indicator present in the changed output. The method is simple, reproducible, and safe to run because generated code is treated only as text data.

APPENDIX A. STATIC RULE PATTERNS USED

The implementation uses only the static patterns listed in Table IX. The list is intentionally small, transparent, and auditable. It is not a complete malicious-code detector, it is a first-pass review filter paired with SHA-256 change recording. The exact pattern list is included to clarify the boundary of the experiment. Results should be interpreted only within these implemented rules. A record without a matched pattern is not proven safe, and a matched pattern is not automatically proven exploitable.

TABLE IX. Static Rule Patterns Used in the Implementation

Finding label	Patterns searched	Purpose
dynamic_code_execution	eval, exec	Detect runtime code evaluation
command_execution	os.system, subprocess, shell=True, child_process	Detect operating-system command execution
unsafe_deserialization	pickle.load, pickle.loads, yaml.load	Detect unsafe object reconstruction
ssrf_or_unvalidated_request	requests.get/post, urllib, axios, fetch	Detect unvalidated network request patterns
path_traversal_risk	path.join, open, readFile, writeFile with user path context	Detect potentially unsafe file-path use
template_injection_risk	render_template_string, template rendering from string	Detect server-side template injection patterns
credential_or_secret_access	secret, token, API key, password, environment access	Detect credential-related access patterns

ACKNOWLEDGMENT

The author would like to express sincere gratitude to Prof. Dr. Ir. Rinaldi, M.T., as the lecturer of the II4021 Cryptography course, for the knowledge, guidance, and academic direction provided throughout the course, both in cryptography and in broader areas of scientific thinking. The author also acknowledges the II4021 Cryptography course for providing the paper template and assignment framework used in this study. SecCodeBench is gratefully acknowledged as the public benchmark dataset used as the basis for local static analysis in this research.

REFERENCES

[1] L. Chen, J. Zhao, L. Cui, T. Su, X. Pan, Z. Li, Y. Wu, Q. Cao, Q. Cai, J. Zhang, Y. Ni, J. He, Z. Zhang, C. Ge, X. Lu, Z. Gao, Y. Cui, W. Chen, Y. Peng, S. Wang, Q. Li, Y. Huang, Y. Liu, T. Zhou, T. Y. Zhuo, J. Lin, and C. Zhang, "SecCodeBench-V2 Technical Report," arXiv:2602.15485, 2026.

[2] Alibaba Group, "SecCodeBench: a benchmark suite focusing on evaluating the security of code generated by large language models," GitHub repository, 2026. [Online]. Available: https://github.com/alibaba/sec-code-bench

[3] S. G. Morkonda, M. Selim, and H. Assal, "Security of LLM-generated Code: A Comparative Analysis," arXiv:2605.23091, 2026.

[4] A. Sabra, O. Schmitt, and J. Tyler, "Assessing the Quality and Security of AI-Generated Code: A Quantitative Analysis," arXiv:2508.14727, 2025.

[5] M. Schreiber and P. Tippe, "Security Vulnerabilities in AI-Generated Code: A Large-Scale Analysis of Public GitHub Repositories," arXiv:2510.26103, 2025.

[6] Y. Wang, Z. Zhang, C. Wang, X. Xu, M. Liu, Y. Wang, J. Chen, and Z. Zheng, "RealSec-bench: A Benchmark for Evaluating Secure Code Generation in Real-World Repositories," arXiv:2601.22706, 2026.

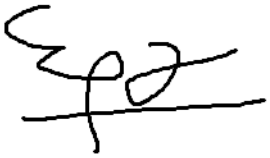
[7] National Institute of Standards and Technology, "Secure Hash Standard (SHS)," FIPS PUB 180-4, Aug. 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4

[8] Apache Software Foundation, “Apache License, Version 2.0,” Jan. 2004. [Online]. Available: https://www.apache.org/licenses/LICENSE-2.0

STATEMENT

I hereby declare that this paper is my own original work. It is not an adaptation, translation, or plagiarism of any other person's work.

Bandung, 19 June 2026



Vandega Arozan Musholine - 18223010