SecMutBench: Evaluating LLM-Generated Security Tests via Mutation-Based Vulnerability Detection

Anonymous Author(s)

Abstract

Existing LLM security benchmarks evaluate code *generation* quality, leaving an open question: can LLMs generate tests that *detect* vulnerabilities? We address this with two technical contributions. First, we propose the **Security Mutation Score (SMS)**, a metric that classifies mutant kills into semantic, functional, incidental, and crash categories using operator-aware heuristics, distinguishing genuine security awareness from coincidental detection. We further define **Effective SMS** (EffSMS = SMS × Secure-Pass Rate) to account for test validity. Second, we design **25 security-specific mutation operators** spanning 30 CWE categories that transform secure Python code into realistic vulnerable variants, extending prior security mutation frameworks to Python and introducing 22 new operators. Evaluating eight LLMs and two static analysis baselines on 339 programs and 1,869 mutants reveals three findings: (i) traditional mutation scores overstate LLM security testing capability by 2.2× on average; (ii) the best LLM achieves only 19.7% EffSMS vs. 47.6% for expert-written tests—a 2.4× gap raw metrics obscure; and (iii) functional kills, not crashes, dominate non-semantic failures (15–36%), showing LLMs detect behavioral side-effects rather than security properties. Static analysis and mutation testing provide complementary coverage across syntactic vs. logic-flaw CWEs. Code and data are publicly available.

CCS Concepts

• **Software and its engineering**;

Keywords

security benchmark, mutation testing, large language models, vulnerability detection, code security, CWE, LLM evaluation, software testing, security testing

ACM Reference Format:

Anonymous Author(s). 2026. SecMutBench: Evaluating LLM-Generated Security Tests via Mutation-Based Vulnerability Detection. In *Proceedings of 3rd ACM International Conference on AI-Powered Software (AIware '26).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software vulnerabilities remain one of the most costly challenges in modern computing. The global cost of cybercrime reached $9.22 trillion in 2024 and is projected to exceed $10.5 trillion in 2025 [6], with the average cost of a single data breach climbing to $4.88 million globally and $10.22 million in the United States alone [15]. Security testing is fundamental to identifying these vulnerabilities before deployment, yet manual security test creation is expensive, time-consuming, and difficult to scale.

At the same time, AI-powered coding assistants have seen explosive adoption. According to Stack Overflow's 2025 Developer Survey [33], 84% of developers now use AI tools in their workflows, with 65% using them at least weekly and 51% of professional developers relying on them daily. Code generation has become AI's first breakout use case, growing into a $1.9 billion ecosystem [20] that includes AI IDEs (Cursor [1], Windsurf [2]), application builders (Lovable, Bolt), and enterprise coding agents (Claude Code, GitHub Copilot [24]).

This rapid adoption introduces a critical question for software security: *as LLMs increasingly generate code, can they also generate effective security tests to catch vulnerabilities in that code?* The stakes are high—if AI-generated code is deployed with inadequate security testing, the same tools that accelerate development may simultaneously accelerate vulnerability introduction.

Several benchmarks have emerged to evaluate LLM security capabilities, but they focus almost exclusively on secure code *generation*. CyberSecEval [3] evaluates whether LLMs produce vulnerable code completions across multiple languages, using static analysis rules for automated detection. SecurityEval [32] provides 130 prompt-based Python coding challenges mapped to 75 CWE types, while SecCodePLT [22] extends this with ground-truth secure/insecure code pairs and automated evaluation pipelines. CWEval [25] offers 119 expert-verified tasks across 31 CWE types with CodeQL-based detection, enabling outcome-driven evaluation of both functionality and security. More recently, CFCEval [5] introduced the Element-Level Relevance Metric (ELRM) to assess code fix quality across four dimensions, addressing dataset bias through the MLVBench benchmark. Pearce et al. [24] assessed the security of GitHub Copilot's code contributions, finding that approximately 40% of generated programs contained vulnerabilities. Tony et al. [34] created natural language prompts for security evaluations across multiple CWE categories.

A critical distinction separates all these benchmarks from SecMutBench: they focus on code *generation* quality—evaluating whether LLM-produced code is secure. SecMutBench instead evaluates test *generation* quality—measuring whether LLM-produced tests can detect vulnerabilities. These are complementary but fundamentally different capabilities. A model that generates secure code may still be unable to articulate *what makes alternative implementations vulnerable* through targeted test assertions. Furthermore, a model may avoid generating vulnerable code simply because it has

AIware '26, Montreal, Canada

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://cursor.sh
[2] https://codeium.com/windsurf

learned to produce safe patterns, without possessing any deeper understanding of the vulnerability mechanism.

Our key contributions are:

(1) **SecMutBench benchmark**: 339 Python programs across 30 CWE categories with 25 mutation operators producing 1,869 pre-generated mutants from curated data sources (SecCodePLT, CyberSecEval, SecurityEval, CWEval, handcrafted templates, and LLM-generated variations), accompanied by an executable evaluation framework with 11 CWE-specific mock environments, a `SafeOS` sandbox blocking dangerous system calls, and a multi-stage contamination prevention pipeline enabling automated, reproducible evaluation.

(2) **Security Mutation Score (SMS) and kill taxonomy**: A metric and operator-aware classification framework that decomposes mutant kills into semantic, functional, incidental, and crash categories to distinguish genuine security awareness from coincidental detection. We formalize the *inflation ratio* $\rho = \text{MS}/\text{SMS}$ and the *Effective Security Mutation Score* $\text{EffSMS} = \text{SMS} \times \text{SPR}$, which together reveal that traditional mutation scores overstate LLM security testing capability by an average of 2.2× and that the best LLM delivers only 19.7% EffSMS vs. 47.6% for expert-written tests.

(3) **Security-specific mutation operator set**: 25 mutation operators that transform secure Python code into realistic vulnerable variants across 30 CWE categories, extending the framework of Loise et al. [18] from Java to Python and introducing 22 new operators targeting CWEs not previously addressed (SSRF, IDOR, CSRF, insecure deserialization, and 18 others). Each operator is grounded in CWE descriptions and validated against real CVE patterns, producing security faults that model how vulnerabilities actually appear in practice.

(4) **Empirical characterization of LLM security test quality**: A large-scale evaluation of eight LLMs and two static analysis baselines revealing three previously unreported findings: (i) functional kills—not crashes—are the dominant non-semantic kill category (15–36% of kills), indicating LLMs detect behavioral side-effects rather than security properties; (ii) test validity (SPR 7–47%) compounds with SMS inflation to produce up to 4.7× overstatement of capability; and (iii) static analysis and LLM-generated tests provide structurally complementary coverage across syntactic vs. logic-flaw CWEs.

2 Related Work

LLM security benchmarks. Several benchmarks evaluate LLM security capabilities, but all focus on code *generation* rather than test generation. CyberSecEval [3] assesses insecure code suggestions via static analysis; SecurityEval [32] provides 130 prompt-based challenges across 75 CWEs; SecCodePLT [22] offers ground-truth secure/insecure pairs; CWEval [25] provides 119 expert-verified tasks with CodeQL detection; and CFCEval [5] evaluates vulnerability-fixing capability. Pearce et al. [24] found 40% of Copilot-generated code contained vulnerabilities; Tony et al. [34] created NL prompts for security evaluation. SecMutBench differs fundamentally: it evaluates whether LLM-generated *tests* detect vulnerabilities, not whether LLM-generated *code* avoids them.

Table 1: Dataset composition by source.

Source	n	Description
SecMutBench Originals	75	Hand-crafted + curated from SecCodePLT, CyberSecEval, SecurityEval
CWEval [25]	3	Expert-verified pairs
SecurityEval [32]	3	Prompt-based challenges
LLM-Generated Variations	258	Structurally diverse variants
Total	**339**	**30 CWEs, 1,869 mutants**

Security mutation testing. Mutation testing [9] evaluates test adequacy by injecting faults and measuring detection. Security-specific mutation has a two-decade history [16, 23]: Du and Mathur [12] introduced security fault injection; Wimmel and Jürjens [36] pioneered specification-based security mutation; Shahriar and Zulkernine developed MUSIC [29], MUTEC [31], and MUFORMAT [30] for SQL injection, XSS, and format strings; Mouelhi et al. [21] showed functional tests fail to achieve high security mutation scores. Loise et al. [18] defined 15 Java operators for security-aware mutation; we extend three to Python and add 22 new operators. Recent work includes MASC [1] for crypto API misuse, μSE [2] for Android security analyzers, and Görz et al. [13] for mutation-based fuzzer evaluation.SMS and EffSMS extend this line of work as oracle-oriented adequacy metrics [14, 38], shifting evaluation focus from fault injection to security oracle quality.

LLM test generation. LLMs show promise for test generation [10, 17, 27, 37], with MuTAP [7] using mutation as a feedback mechanism for improvement. Concurrent work on TAM-Eval [4], AGONETEST [19], and MUTGEN [35] validates mutation signals for LLM evaluation but applies traditional operators without security-specific kill classification. SecMutBench is the first to combine all five: test evaluation, dynamic execution, kill classification, security-specific mutation, and contamination prevention. Unlike μBERT's [8] learned mutants, our operators are grounded in CWE semantics, ensuring each mutant models a known vulnerability class—a property critical for the SMS/EffSMS distinction between security-aware and coincidental kills.

3 SecMutBench Framework

3.1 Dataset Construction

SecMutBench consolidates samples from multiple sources to ensure diversity in vulnerability patterns, coding styles, and complexity levels. Table 1 summarizes the dataset composition.

Each sample contains a secure implementation, an insecure variant with a known vulnerability, CWE classification, reference security tests, and pre-generated mutants. Samples span three difficulty levels: easy (136), medium (101), and hard (102), assigned based on code complexity and vulnerability subtlety.

Contamination prevention. Adapted samples from public sources undergo a four-stage pipeline: temporal CVE filtering (cutoff 2024), CWE-specific structural perturbation, AST-based identifier renaming, and 5-gram Jaccard contamination audit (threshold

Table 2: Security mutation operators (top 15 by mutant count).

Operator	Target CWEs	n
RVALID	CWE-20, 79, 89	151
WEAKCRYPTO	CWE-327, 328	128
INPUTVAL	CWE-20	119
DESERIAL	CWE-502, 94	106
RMAUTH	CWE-287, 306	105
EVALINJECT	CWE-94, 95	98
SSRF	CWE-918	97
RENCRYPT	CWE-319, 326	90
PATHCONCAT	CWE-22, 73	87
MISSINGAUTH	CWE-862, 863	83
CSRF_REMOVE	CWE-352	82
HARDCODE	CWE-798	81
PSQLI	CWE-89	80
OPENREDIRECT	CWE-601	63
LOGINJECT	CWE-117	62

n = total mutants produced by operator. Additional operators (NOCERTVALID, WEAKPERM, XXE, LDAPINJECT, WEAKKEY, WEAKRANDOM, FILEUPLOAD, INFOEXPOSE, REGEXDOS, IDOR) produce 20–60 mutants each.

0.3). Novel templates and LLM-generated variations (82% of dataset) are exempt. All perturbations are validated via `compile()`.

LLM-generated variations. We generated 258 structural variations of original samples using LLMs, each preserving the CWE category and vulnerability mechanism while introducing diverse coding patterns. All variations pass the same validation pipeline (compilation, vulnerability pattern presence, valid mutant generation).

CWE selection. Our 30 CWEs were selected from OWASP Top 10 and CWE Top 25, filtered to Python-relevant vulnerabilities with recent CVEs, retaining only those with implementable mutation operators. The set covers injection, authentication, access control, cryptography, input validation, deserialization, SSRF, CSRF, and 12 additional categories.

3.2 Security Mutation Operators

SecMutBench defines 25 mutation operators that transform secure code into vulnerable variants. Unlike traditional mutation operators that create arbitrary syntactic changes [9], our operators model specific vulnerability patterns mapped to CWE categories. Table 2 presents the operator taxonomy.

Our operators extend the framework of Loise et al. [18] (15 Java operators) to Python, adapting three of their operators (PSQLI, RHTTPO, XXE) and introducing 22 new ones covering command injection, deserialization, SSRF, IDOR, CSRF, and 17 additional CWE categories.

Each operator implements `applies_to(code)` and `mutate(code)` using pattern-based source transformations rather than AST node replacement, enabling security-specific mutations that tools like MutPy [11] cannot express.

Mutants are pre-generated during dataset construction and stored in each sample's JSON record. This ensures deterministic evaluation—the same mutants are used across all evaluation runs—and eliminates variation from operator application order.

Secure (original)

```
def get_user(user_id)
    :
  query = "SELECT␣*␣
      FROM␣users"
        "␣WHERE␣id␣=
            ␣%s"
  return db.execute(
      query, (user_id
          ,))
```

Mutant (PSQLI)

```
def get_user(user_id)
    :
  query = f"SELECT␣*␣
      FROM␣"
   f"users␣WHERE␣id={
       user_id}"
  return db.execute(
      query)
```

Kill classifications for this mutant:

- **Semantic:** `assert db.last_params is not None, "Query should use parameterized execution"`
- **Functional:** `pytest.raises(ValueError)` — expected exception no longer raised
- **Incidental:** `assert len(result) == 5` — output count changes
- **Crash:** `TypeError: not all arguments converted during string formatting`

Figure 1: The PSQLI operator transforms a parameterized SQL query (secure, left) into string concatenation (vulnerable, CWE-89, right). Below: illustrative kill classifications showing how the same mutant can be killed via different mechanisms.

Mutant validity. Security operators are structurally less susceptible to equivalent mutants than traditional operators because they transform behavioral security properties [16, 23]. All 1,869 mutants (100%) pass `compile()` validation and basic execution. The dataset averages 5.51 mutants per program (range 4–9), with each operator targeting a specific vulnerability pattern. Per-operator executability rates are available in the replication package (https://github.com/Mars-2030/secmutbench).

3.3 Kill Classification and SMS Metric

The central insight of SecMutBench is that not all mutant kills are equal. When a test kills a mutant (passes on secure code, fails on mutant), the kill type reveals whether the test demonstrates genuine security awareness.

Formally, let P denote a secure program and $M = \{m_1, m_2, \ldots, m_k\}$ the set of mutants generated by applying security mutation operators $O = \{o_1, \ldots, o_{25}\}$. For a test suite T generated by an LLM, we define the kill predicate:

$$\text{kill}(t, m) = \begin{cases} 1 & \text{if } t \text{ passes on } P \text{ and fails on } m \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

For each killed mutant, we classify the failure into one of five categories:

- **Semantic**: Assertion error with operator-specific security keywords (e.g., "parameterized" for PSQLI). The test explicitly checks for the vulnerability.
- **Functional**: Behavioral-contract assertion (expected exception, return type) catches a control-flow change without security framing.
- **Incidental**: Data-value assertion (`assert len(result)==64`) catches a value change as a side-effect of the mutation.

- **Crash**: Runtime error (TypeError, NameError, ImportError) before any assertion is evaluated.
- **Other**: Residual non-assertion, non-crash exceptions. Excluded from primary metrics.

The five categories are mutually exclusive, assigned in priority order: crash → semantic → functional → incidental → other.
Let K_s, K_f, K_i, K_c, and K_o denote the sets of semantic, functional, incidental, crash, and other kills respectively, where $K_s \cup K_i \cup K_f \cup K_c \cup K_o \subseteq M$ and the sets are mutually disjoint. We define the **Security Mutation Score** as:

$$\text{SMS} = \frac{|K_s|}{|M|} \tag{2}$$

In contrast, the traditional Mutation Score (MS) counts all kills:

$$\text{MS} = \frac{|K_s| + |K_i| + |K_f| + |K_c| + |K_o|}{|M|} \tag{3}$$

The **inflation ratio** ρ captures the overstatement:

$$\rho = \frac{\text{MS}}{\text{SMS}} \tag{4}$$

When $\rho \gg 1$, the gap is attributable to crash, functional, incidental, and other kills that do not represent genuine security detection.

Because SMS is computed only over samples where the generated test suite passes on secure code, it does not penalize models for producing invalid tests. To capture practical end-to-end effectiveness, we define the **Effective Security Mutation Score**:

$$\text{EffSMS} = \text{SMS} \times \text{SPR} \tag{5}$$

where SPR (Secure-Pass Rate) is the proportion of samples for which the generated test suite passes on the secure code. EffSMS is the primary comparability metric: it reflects how much security-aware detection a model delivers *across the entire benchmark*, accounting for both test validity and security awareness. We additionally report SMS and SPR separately for diagnostic decomposition.

Vulnerability Detection (VD). As a ground-truth sanity check, we compute a binary Vulnerability Detection metric that verifies whether generated tests can distinguish secure from insecure code:

$$\text{VD}(p) = \begin{cases} 1 & \text{if } T_p \text{ passes on } P_p^{\text{sec}} \text{ and fails on } P_p^{\text{insec}} \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

and report the aggregate $\text{VD} = |\{p : \text{VD}(p) = 1\}|/|P|$. Note that VD is *not* a mutation-based metric—it tests against the original insecure code, not mutants. Mutant-based vulnerability detection is captured by MS and SMS.

Operator-aware classification. A key refinement in SecMutBench is that kill classification uses operator-specific keyword lists rather than a flat global list. For example, "path" triggers semantic classification only for PATHCONCAT mutants, not for PSQLI mutants. Each of the 25 operators has a tailored set of security-relevant terms (e.g., WEAKCRYPTO: {md5, sha1, weak, bcrypt, salt}; DESERIAL: {pickle, yaml, safe_load, SafeLoader}). This prevents false inflation of SMS scores from coincidental keyword matches. Complete operator specifications, keyword lists, and LLM-as-Judge evaluation prompts are available in the replication package at https://github.com/Mars-2030/secmutbench.

Table 3: Subject models and baselines.

Model	Version / ID	Size	Prov.	Access
GPT-oss-120B	`gpt-oss-120b`	120B	OpenAI	API
GPT-5.2	`gpt-5.2-2025-12-11`	–	OpenAI	API
GPT-5-mini	`gpt-5-mini-2025-08-07`	–	OpenAI	API
Qwen3-Coder	`qwen3-coder:30b`	30B	Alibaba	Ollama
Qwen2.5-Coder	`qwen2.5-coder:14b`	14B	Alibaba	Ollama
Kimi-K2.5	`kimi-k2.5:cloud`	–	Moonshot	Ollama
GLM-5	`glm-5:cloud`	–	Zhipu AI	Ollama
DeepSeek-V2	`deepseek-coder-v2:latest`	236B MoE	DeepSeek	Ollama
Bandit	1.9.2	–	Static	CLI
Semgrep	`p/security-audit`	–	Static	CLI

3.4 Evaluation Pipeline

The pipeline executes LLM-generated tests in isolated subprocesses with 11 CWE-specific mock environments injected via `sys.modules` and builtins patching. Mocks serve dual purposes: *safety* (blocking real system calls via a `SafeOS` wrapper) and *observability* (tracking security-relevant operations, e.g., `db.last_params`, `subprocess.last_shell`). For each sample: (1) tests run against secure code (should pass), (2) tests run against each mutant (should fail), (3) kills are classified via operator-aware heuristics, and (4) SMS/EffSMS are computed.

4 Experimental Setup

4.1 Research Questions

RQ1: *How effectively do LLMs generate security-aware tests?* We compare overall mutation scores (MS) against Security Mutation Scores (SMS) to quantify the inflation introduced by functional, incidental, and crash kills.

RQ2: *What types of kills do LLM-generated tests produce?* We analyze the kill classification breakdown to understand why traditional metrics overstate LLM security testing capability.

RQ3: *How does security test effectiveness vary across vulnerability types?* We examine per-CWE and per-operator effectiveness to identify systematic blind spots, including comparison with static analysis baselines.

RQ4: *How sensitive is SMS to prompt design and how do LLMs compare to human-written tests?* We ablate prompt detail levels (no hint, CWE ID only, full context) and compare LLM-generated tests against expert-written reference tests to establish upper bounds.

4.2 Subject Models

We evaluate eight LLMs spanning open-weight and proprietary models, plus two static analysis baselines. Table 3 summarizes the models. All LLMs are evaluated in a zero-shot setting with temperature 0 (greedy decoding) and maximum 2048 output tokens.

Open-weight models are served locally via Ollama providing an OpenAI-compatible API. Proprietary models are accessed via their respective cloud APIs. All kill classifications are validated using an LLM-as-Judge (GPT-5.4) to reclassify heuristic labels.

4.3 Prompting Strategy

Each model receives a zero-shot prompt containing the secure code, CWE identifier/name, and mock environment documentation

with detection patterns (e.g., `db.last_params is not None` for SQL injection). We evaluate three prompt variants: (a) *full context* (default, with mock docs and assertion guidance), (b) *CWE-ID only* ("Write tests to detect CWE-89"), and (c) *no hint* ("Write tests for this function"). The complete prompt template is available in the replication package.

4.4 Baselines

We include three baselines: (1) **Bandit** [26], a widely-used Python static analysis tool, run on insecure code to measure detection rate per CWE; (2) **Semgrep** [28] with the `p/security-audit` ruleset, providing a second static analysis comparison with broader rule coverage; and (3) **Reference tests**, the expert-written security tests included in each SecMutBench sample, establishing an upper-bound baseline for what hand-crafted security tests achieve on our mutants.

Bandit detects 77 of 339 samples (22.7% overall detection rate). On the 11 static-analyzable CWEs (147 samples), Bandit achieves 71.4% detection; on the remaining 19 dynamic-only CWEs (192 samples), Bandit achieves 0% detection. Semgrep with `p/python` rules detects 37 samples (10.9%). Reference tests achieve MS of 57.7% and SMS of 47.6% (889 semantic kills out of 1,869 mutants), establishing the expert upper bound.

4.5 Execution Environment

All experiments use Python 3.11.5 on macOS with 5-second test timeouts. Each model evaluates all 339 samples across three prompt variants in a single run with temperature 0 (greedy decoding), producing deterministic outputs (seed 42). Secure-pass rates range from 7.1% (DeepSeek) to 46.6% (Qwen3); tests failing on secure code are excluded from mutation evaluation.

5 Results

5.1 RQ1: Overall Effectiveness

Table 4 presents the overall results. Effective SMS (EffSMS = SMS × SPR) is the primary comparability metric, capturing end-to-end security testing effectiveness across the entire benchmark. SMS and SPR are reported separately for diagnostic decomposition.

As a ground-truth cross-check, the aggregate Vulnerability Detection rate (VD — pass on secure, fail on insecure original) is 91.4% across all models, confirming that our mutants effectively distinguish secure from insecure code before mutation-level analysis.

The headline result is the gap between expert-written and LLM-generated tests. Reference tests achieve 47.6% EffSMS; the best LLM (Qwen3-Coder 30B) achieves 19.7%—a 2.4× gap that raw MS completely obscures (reference MS of 57.7% is *lower* than six of eight LLMs). The ranking by EffSMS differs strikingly from the ranking by MS: GPT-5.2, which ranks second on MS (88.7%), drops to seventh on EffSMS (5.1%) because only 13.3% of its tests are valid. Conversely, Qwen3-Coder ranks sixth on MS but first on EffSMS, because its 46.6% SPR compensates for moderate SMS.

Among valid test suites, the inflation ratio (MS/SMS) ranges from 1.43 (Qwen2.5-Coder) to 4.74 (DeepSeek-Coder-V2), with a mean of 2.2×. Qwen2.5-Coder achieves the highest SMS (58.6%)—exceeding reference tests (47.6%)—but its 16.8% SPR yields only 9.8% EffSMS, demonstrating that raw SMS without test validity is misleading.

Table 4: Overall results (OpenAI-judged, full-context prompt). EffSMS = SMS × SPR is the primary metric.

Model	MS	SPR	EffSMS	SMS	MS/SMS
Qwen3-Coder 30B	75.2	46.6	**19.7**	42.3	1.78
Kimi-K2.5	84.2	42.8	19.6	45.7	1.84
GPT-oss-120B	90.0	36.0	15.6	43.2	2.08
GPT-5-mini	84.2	29.2	11.6	39.8	2.12
Qwen2.5-Coder 14B	83.6	16.8	9.8	58.6	1.43
GLM-5	40.0	33.0	9.0	27.4	1.46
GPT-5.2	88.7	13.3	5.1	38.3	2.32
DeepSeek-Coder-V2	51.7	7.1	0.8	10.9	4.74
Reference Tests	57.7	100.0	**47.6**	47.6	1.21
Bandit	22.7	–	–	–	–

Models sorted by EffSMS (descending). **MS** = Mutation Score (all kills). **SPR** = Secure-Pass Rate (% of tests passing on secure code). **EffSMS** = Effective SMS (SMS × SPR). **SMS** = Security Mutation Score (semantic kills only, among valid tests). All values are percentages.

Table 5: Kill classification breakdown (full-context prompt, counts).

Model	Mut.	Sem.	Func.	Incid.	Crash	Other
GPT-oss-120B	672	290	194	22	54	45
GPT-5.2	266	102	87	15	12	20
GPT-5-mini	538	214	155	18	22	44
Kimi-K2.5	795	363	171	23	66	46
Qwen2.5 14B	324	190	37	26	14	4
Qwen3 30B	839	355	100	53	69	54
DeepSeek-V2	147	16	37	0	19	4
GLM-5	610	167	41	2	26	8
Reference	1869	889	0	60	73	56

Mut. = mutants evaluated (from valid-test samples). **Sem.** = semantic kills. **Func.** = functional kills. **Incid.** = incidental. Reference tests evaluate all 1,869 mutants (100% SPR).

Finding 1a (generation quality): Only 7–47% of LLM-generated test suites pass on secure code. Accounting for this, the best EffSMS is 19.7% (Qwen3-Coder), compared to 47.6% for expert-written reference tests—a 2.4× gap that raw mutation scores completely obscure.

Finding 1b (when tests are valid): Among valid test suites, traditional MS overstates security testing capability by 2.2× on average (range 1.4–4.7×). The best model when tests are valid is Qwen2.5-Coder (58.6% SMS), but its 16.8% SPR makes its practical EffSMS only 9.8%.

5.2 RQ2: Kill Classification Analysis

Table 5 presents the kill classification breakdown for each model's full-context prompt variant. Kill counts reflect only the mutants from samples where the model's tests passed on secure code.

Functional kills—not crashes—dominate non-semantic kills for LLM-generated tests (15–36% of all kills), while reference tests produce zero functional kills. For GPT-oss-120B, functional kills (194) exceed crash kills (54). Qwen2.5-Coder has the lowest non-semantic proportion (29.9%).

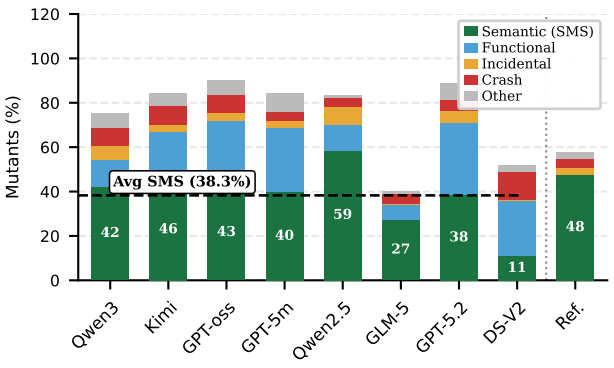

Figure 2: Kill classification breakdown by model. Green = semantic kills (SMS); full bar = MS; gap illustrates inflation ratio. Dashed line = avg SMS across LLMs.

Example. A Kimi-K2.5 test asserts `not hashlib.weak_algorithm_used` after calling a hashing function. The WEAKCRYPTO mutant substitutes `sha256` with `md5`, causing the mock to set `weak_algorithm_used = True`—a semantic kill via mock observability. By contrast, a functional kill on the same mutant would catch it via `assert len(result)==64` (output length change), and a crash kill via `NameError` (removed computation line).

Classification audit. Rather than relying solely on heuristic keyword matching, we validate all kill classifications using an LLM-as-Judge (GPT-5.4). The judge receives each kill's error message, operator, CWE, and test code, and independently classifies it into semantic, functional, incidental, crash, or other. All results reported in this paper use the judge-validated classifications. We note that GPT-5.4 belongs to the same provider family as three evaluated models (GPT-5.2, GPT-5-mini, GPT-oss-120B); potential provider bias is acknowledged as a limitation in Section 7. Human–judge agreement was not separately measured; the degree of reclassification from heuristic to judge labels is quantified in the replication package. We additionally performed a human validation study on a stratified sample of 120 kills to assess heuristic–annotator agreement, yielding Cohen's $\kappa = 0.502$ (moderate agreement).

Table 6

shows per-operator agreement, revealing a systematic pattern rather than random noise: *syntactic* operators (PSQLI, WEAKPERM, LOGINJECT, DESERIAL) achieve 89–100% agreement, while *behavioral* operators (IDOR, OPENREDIRECT, MISSINGAUTH, SSRF) achieve 0–17% agreement. This split directly reflects whether the kill's security intent is visible in the error message (syntactic operators produce assertion failures with security-relevant terms) or only in the test logic (behavioral operators produce `DID NOT RAISE` failures with no security-specific content). The disagreement is therefore *structurally predictable* from operator type, not from annotator subjectivity.

Finding 2: Functional kills—not crash kills—are the dominant non-semantic category for LLM-generated tests, accounting for 15–36% of all kills. Reference tests produce zero functional kills. This reveals that LLMs often detect mutations through behavioral side-effects rather than security-targeted assertions.

Table 6: Per-operator annotator–heuristic agreement on kill classification. Syntactic operators (top) show near-perfect agreement; behavioral operators (bottom) show systematic divergence driven by the `DID NOT RAISE` classification gap.

Operator	Type	n	Annot. sem.	Agreement
PSQLI	Syntactic	7	100%	100%
WEAKPERM	Syntactic	9	100%	100%
LOGINJECT	Syntactic	3	100%	100%
DESERIAL	Syntactic	18	17%	89%
WEAKRANDOM	Syntactic	4	75%	75%
RVALID	Behavioral	11	27%	64%
PATHCONCAT	Behavioral	3	33%	67%
SSRF	Behavioral	13	46%	54%
EVALINJECT	Behavioral	8	13%	50%
XXE	Behavioral	4	50%	50%
INPUTVAL	Behavioral	15	53%	47%
MISSINGAUTH	Behavioral	6	100%	17%
OPENREDIRECT	Behavioral	4	100%	0%
IDOR	Behavioral	7	100%	0%

Annot. sem.: proportion of kills the annotator labeled semantic. **Agreement**: proportion where annotator and heuristic agree. For behavioral operators where annotator labels semantic but heuristic labels functional (e.g., SSRF, IDOR, OPENREDIRECT), the heuristic is conservative: it misses semantic kills whose security intent is expressed in the test logic rather than the error message. The LLM-as-Judge reclassification corrects this conservative bias for the reported results.

Table 7: Per-CWE effectiveness (selected CWEs, aggregated across models).

CWE	Category	n	Mut.	Bandit	
CWE-89	SQL Injection	16	80	71.4%	
CWE-22	Path Traversal	15	87	71.4%	
CWE-94	Code Injection	15	102	71.4%	
CWE-327	Weak Crypto	12	71	71.4%	
CWE-798	Hardcoded Creds	13	81	71.4%	Top group:
CWE-306	Missing Auth	16	97	0%	
CWE-352	CSRF	14	82	0%	
CWE-862	Missing Authz	10	62	0%	
CWE-639	IDOR	5	20	0%	
CWE-918	SSRF	15	97	0%	

static-analyzable CWEs (Bandit achieves 71.4% *aggregate* detection across all 11 pattern-based CWEs; per-CWE rates vary and are in the replication package). Bottom group: logic-flaw CWEs (Bandit achieves 0%). **n** = samples, **Mut.** = total mutants. Full 30-CWE breakdown available in the replication package (https://github.com/Mars-2030/secmutbench).

5.3 RQ3: Vulnerability-Specific Analysis

Table 7 presents per-CWE effectiveness aggregated across models, alongside Bandit detection rates. CWEs are grouped by whether they are amenable to static analysis (pattern-based) or require dynamic behavioral testing (logic-flaw).

Bandit achieves 71.4% detection on 11 pattern-based CWEs but 0% on 19 logic-flaw CWEs. CWE-89 achieves 100% SMS across all models; CWE-611 (XXE) and CWE-95 are universal blind spots at 0% SMS (Figure 3).

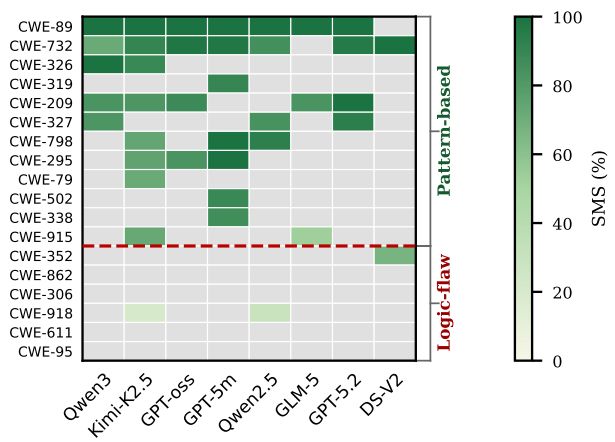

Figure 3: Per-CWE Security Mutation Score across models. Pattern-based CWEs (above dashed line) vs. logic-flaw CWEs (below) reveal distinct detection profiles. CWE-89 achieves 100% SMS across all models; CWE-611 and CWE-95 are universal blind spots at 0%.

Table 8: Prompt ablation: SMS (%) by prompt variant.

Model	No hint	CWE ID	Full context
GPT-oss-120B	0.0	52.3	43.2
GPT-5.2	31.2	54.5	38.3
GPT-5-mini	20.7	38.1	39.8
Kimi-K2.5	30.1	57.5	45.7
Qwen2.5-Coder 14B	12.9	14.9	58.6
Qwen3-Coder 30B	35.5	42.5	42.3
DeepSeek-Coder-V2	7.1	7.5	10.9
GLM-5	25.6	34.3	27.4

Finding 3: Static analysis (Bandit) achieves 71.4% detection on 11 pattern-based CWEs but 0% on 19 logic-flaw CWEs. LLM-generated tests provide complementary coverage on logic-flaw vulnerabilities (authentication, authorization, CSRF, SSRF) where static analysis is structurally blind.

5.4 RQ4: Prompt Sensitivity

Table 8 compares SMS across three prompt variants. No-hint prompts consistently degrade SMS (average delta of 24.8 pp vs. best variant), with GPT-oss dropping to 0%. Explicit vulnerability guidance is essential for security-aware test generation.

Finding 4: No-hint prompts severely degrade SMS (0–35.5%). Expert reference tests achieve 47.6% EffSMS; the best LLM reaches only 19.7% EffSMS.

6 Discussion

The 2.4× EffSMS gap between expert-written tests (47.6%) and the best LLM (19.7%) compounds from two sources: most generated tests fail on secure code (SPR 7–47%), and among valid tests, functional kills and crashes inflate MS by 2.2× over SMS. A model reporting 90% MS may deliver only 5% EffSMS (GPT-5.2: 88.7% → 5.1%). This extends Mouelhi et al.'s [21] finding that functional tests fail to achieve high security mutation scores: LLMs proficient at generating functional tests may nevertheless lack genuine security reasoning.

Static analysis complementarity. Bandit achieves 71.4% detection on 11 syntactic CWEs but 0% on 19 logic-flaw CWEs. Running Bandit on mutants rather than insecure originals would not change this: syntactic operators (PSQLI, WEAKCRYPTO) introduce the same patterns Bandit detects in originals, while behavioral operators (MISSINGAUTH, IDOR, CSRF_REMOVE) produce *missing logic* undetectable by any static rule. LLMs provide unique value on logic-flaw CWEs (e.g., CWE-306: Kimi 75% SMS, CWE-732: GPT-5-mini 97%).

Prompt guidance and ecological validity. The full-context prompt (with mock documentation and assertion hints) represents an optimistic upper bound on SMS. Practitioners without mock documentation would likely observe SMS closer to the CWE-ID-only variant (Table 8), where the average SMS drops by 8–15 percentage points. The no-hint condition (0–35.5% SMS) approximates the realistic developer setting where vulnerability context is unavailable.

SMS robustness. SMS could theoretically be gamed by inserting security keywords without genuine understanding. Three structural mitigations prevent this: (1) operator-aware keywords prevent cross-CWE gaming; (2) the kill predicate requires tests to pass on secure code *and* fail on the mutant; (3) mock state tracking supports future observability-based classification. We recommend that evaluations decompose kills by type, use pre-generated mutants for reproducibility, and review LLM-generated tests for security-specific assertions rather than trusting pass/fail behavior alone.

7 Threats to Validity

Internal validity. Kill classification relies on operator-aware keyword heuristics and an LLM-as-Judge (GPT-5.4), both of which may misclassify kills where security terms appear coincidentally or where genuine assertions use non-standard terminology. The LLM-as-Judge belongs to the same provider family as three evaluated models, introducing potential provider bias. Human–heuristic agreement ($\kappa = 0.502$; Table 6) is moderate, and human–judge agreement was not separately measured. Our crash-first priority order may conservatively depress SMS for tests where a semantic assertion would have fired absent the crash; SMS estimates should be treated as lower bounds for crash-heavy test suites.

External validity. SecMutBench covers 30 CWEs in Python only; temperature-0 single-run evaluation produces deterministic, reproducible results but does not capture variance across stochastic decoding settings or model versions. Future work should report confidence intervals across multiple runs. Our contamination pipeline mitigates training-data overlap but cannot guarantee zero contamination for all evaluated models.

Construct validity. Compilation and executability checks confirm syntactic validity but do not rule out semantic equivalence. Security operators are structurally less susceptible to equivalent mutants than traditional operators because they transform behavioral properties; however, validation-removal operators on unreachable paths may produce equivalent mutants, and this risk is not formally quantified. The mock environment introduces an ecological validity gap: mock-dependent SMS values are conservative lower bounds,

most clearly for CWE-611 (XXE), which achieves 0% SMS due to API mismatch between MockXMLParser and real parsers. Models that detect vulnerabilities via real API behavior may be undercounted.

8 Conclusion

We presented SecMutBench, a mutation-based benchmark with 25 operators, 339 programs, 30 CWEs, and 1,869 mutants for evaluating LLM-generated security tests. The best LLM achieves only 19.7% EffSMS compared to 47.6% for expert-written tests—a 2.4× gap compounding from low test validity (SPR 7–47%) and MS/SMS inflation (2.2×). Static analysis and mutation testing provide complementary coverage: Bandit excels on 11 syntactic CWEs (71.4%) while LLMs cover 19 logic-flaw CWEs where Bandit achieves 0%. Future work includes multi-language support, observability-based kill classification via mock state tracking, and iterative refinement with surviving mutants [7, 35]. The code and replication package are at https://github.com/Mars-2030/secmutbench; the dataset is at https://huggingface.co/datasets/Mars203020/secmutbench.

References

[1] Amit Seal Ami, Syed Yusuf Ahmed, Radowan Mahmud Redoy, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2023. MASC: a tool for mutation-based evaluation of static crypto-API misuse detectors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2162–2166.

[2] Amit Seal Ami, Kaushal Kafle, Adwait Nadkarni, Denys Poshyvanyk, and Kevin Moran. 2021. μse: Mutation-based evaluation of security-focused static analysis tools for android. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 53–56.

[3] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, et al. 2023. Purple llama cyberseceval: A secure coding benchmark for language models. *arXiv preprint arXiv:2312.04724* (2023).

[4] Elena Bruches, Vadim Alperovich, Dari Baturova, Roman Derunets, Daniil Grebenkin, Georgy Mkrtchyan, Oleg Sedukhin, Mikhail Klementev, Ivan Bondarenko, Nikolay Bushkov, et al. 2026. TAM-Eval: Evaluating LLMs for Automated Unit Test Maintenance. *arXiv preprint arXiv:2601.18241* (2026).

[5] Cheng Cheng and Jinqiu Yang. 2025. CFCEval: Evaluating Security Aspects in Code Generated by Large Language Models. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*. IEEE, 01–10.

[6] Cybersecurity Ventures. 2025. Cybercrime to Cost the World $10.5 Trillion Annually by 2025. https://cybersecurityventures.com/.

[7] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. 2024. Effective test generation using pre-trained large language models and mutation testing. *Information and Software Technology* 171 (2024), 107468.

[8] Renzo Degiovanni and Mike Papadakis. 2022. μBert: Mutation Testing using Pre-Trained Language Models. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 160–169.

[9] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. 2006. Hints on test data selection: Help for the practicing programmer. *Computer* 11, 4 (2006), 34–41.

[10] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[11] Anna Derezinska and Konrad Halas. 2014. Experimental evaluation of mutation testing approaches to python programs. In *2014 IEEE seventh international conference on software testing, verification and validation workshops*. IEEE, 156–164.

[12] Wenliang Du and Aditya P Mathur. 2002. Testing for software vulnerability using environment perturbation. *Quality and Reliability Engineering International* 18, 3 (2002), 261–272.

[13] Philipp Görz, Björn Mathis, Keno Hassler, Emre Güler, Thorsten Holz, Andreas Zeller, and Rahul Gopinath. 2023. Systematic assessment of fuzzers using mutation analysis. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4535–4552.

[14] Soneya Binta Hossain and Matthew B. Dwyer. 2022. A Brief Survey on Oracle-based Test Adequacy Metrics. *arXiv preprint arXiv:2212.06118* (2022).

[15] IBM Security. 2024. Cost of a Data Breach Report 2024. https://www.ibm.com/reports/data-breach.

[16] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[17] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. 2023. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 919–931.

[18] Thomas Loise, Xavier Devroey, Gilles Perrouin, Mike Papadakis, and Patrick Heymans. 2017. Towards security-aware mutation testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 97–102.

[19] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, and Michelantonio Trizio. 2024. AgoneTest: Automated creation and assessment of Unit tests leveraging Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2440–2441.

[20] Menlo Ventures. 2025. 2025 Mid-Year LLM Market Update: Foundation Model Landscape and Economics. https://menlovc.com/perspective/2025-mid-year-llm-market-update/.

[21] Tejeddine Mouelhi, Yves Le Traon, and Benoit Baudry. 2009. Transforming and selecting functional test cases for security policy testing. In *2009 International Conference on Software Testing Verification and Validation*. IEEE, 171–180.

[22] Muhammed Abeed Nabith. 2025. SECCODEPLT: A Unified Evaluation Platform for Code GenAI Security Risks. In *International Conference on Emerging Trends and Technologies on Intelligent Systems*. Springer, 92–104.

[23] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in computers*. Vol. 112. Elsevier, 275–378.

[24] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Commun. ACM* 68, 2 (2025), 96–105.

[25] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. IEEE, 33–40.

[26] PyCQA. 2024. Bandit: A Tool Designed to Find Common Security Issues in Python Code. https://bandit.readthedocs.io/.

[27] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105.

[28] Semgrep Inc. 2024. Semgrep: Lightweight Static Analysis for Many Languages. https://semgrep.dev/.

[29] Hossain Shahriar and Mohammad Zulkernine. 2008. MUSIC: Mutation-based SQL injection vulnerability checking. In *2008 The Eighth International Conference on Quality Software*. IEEE, 77–86.

[30] Hossain Shahriar and Mohammad Zulkernine. 2008. Mutation-based testing of format string bugs. In *2008 11th IEEE High Assurance Systems Engineering Symposium*. IEEE, 229–238.

[31] Hossain Shahriar and Mohammad Zulkernine. 2009. Mutec: Mutation-based testing of cross site scripting. In *2009 ICSE Workshop on Software Engineering for Secure Systems*. IEEE, 47–53.

[32] Mohammed Latif Siddiq and Joanna CS Santos. 2022. Securityeval dataset: mining vulnerability examples to evaluate machine learning-based code generation techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. 29–33.

[33] Stack Overflow. 2025. 2025 Developer Survey. https://survey.stackoverflow.co/2025/.

[34] Catherine Tony, Markus Mutas, Nicolás E Díaz Ferreyra, and Riccardo Scandariato. 2023. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 588–592.

[35] Guancheng Wang, Qinghua Xu, Lionel C Briand, and Kui Liu. 2025. Mutation-Guided Unit Test Generation with a Large Language Model. *arXiv preprint arXiv:2506.02954* (2025).

[36] Guido Wimmel and Jan Jürjens. 2002. Specification-based test generation for security-critical systems using mutations. In *International Conference on Formal Engineering Methods*. Springer, 471–482.

[37] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726.

[38] Peng Zhang, Yang Wang, Xutong Liu, Yibiao Yang, Yanhui Li, Lin Chen, Ziyuan Wang, Chang-ai Sun, and Yuming Zhou. 2022. Test suite effectiveness metric evaluation: what do we know and what should we do? *arXiv preprint arXiv:2204.09165* (2022).

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009