Computers, Materials & Continua

Tech Science Press

https://doi.org/10.32604/cmc.2026.084639

ARTICLE

RAVE-Code: A Risk-Aware Verification Engine for AI-Generated Code Security Using Composite Risk Scoring and CWE-Conditioned Model Checking

Maher Alharby[1,*] **and Ali Alssaiari**[2,3]

[1]Department of Cybersecurity, College of Computer Science and Engineering, Taibah University, Madinah, Saudi Arabia
[2]Computer Science Department, College of Science and Arts at Sharoura, Najran University, Sharurah, Saudi Arabia
[3]Scientific and Engineering Research Center, Najran University, Najran, Saudi Arabia

*Corresponding Author: Maher Alharby. Email: mharby@taibahu.edu.sa

Received: 27 April 2026; Accepted: 12 June 2026; Published: 13 August 2026

ABSTRACT: Large Language Models (LLMs) are increasingly being used to generate source code. However, a substantial proportion of their output contains security vulnerabilities. Existing defenses typically apply uniform analysis to all code fragments, irrespective of their risk profiles. This study presents RAVE-Code, a three-layer framework that calibrates the verification effort based on the risk associated with each detected weakness. The Detection layer employs Bandit for pattern-based static analysis, annotating findings with their respective Common Weakness Enumeration (CWE) classes. The Risk Scoring layer calculates a composite risk score for each weakness instance by integrating the Common Vulnerability Scoring System (CVSS) severity, the prevalence of CWEs in AI-generated code, and actual exploitability data. The Verification layer directs each code fragment to an appropriate depth of Bounded Model Checking (BMC) using the Efficient SMT-Based Bounded Model Checker (ESBMC), with solver parameters customized for each CWE class. The framework is evaluated on three public benchmarks. On SecurityEval, Bandit identifies 28.5% of vulnerabilities with zero false positives, establishing a baseline for static analysis coverage. On FormAI, RAVE-Code achieves a detection rate of 86.0% (F_1 = 0.909), compared to 73.8% (F_1 = 0.834) for uniform verification, and uniquely detects 41 vulnerabilities that uniform analysis misses. Because FormAI's labels are based on ESBMC, these results show how consistent two ESBMC configurations rather than agreement with an independent standard. On the Juliet Test Suite, which provides structurally independent ground truth, RAVE-Code achieves a detection rate of 57.3% (F_1 = 0.614) compared to 40.8% (F_1 = 0.479), detecting a total of 104 vulnerabilities. Applying all flags uniformly results in a detection rate 18.8 percentage points lower than the proposed per-CWE tailoring. This indicates that detection gains arise from routing rather than flag selection.

KEYWORDS: AI-generated code security; bounded model checking; vulnerability detection; risk prioritisation; static analysis; CWE; DevSecOps; formal verification

1 Introduction

Large Language Model (LLM)-based code generation has become essential to modern software engineering. Tools such as GitHub Copilot, Amazon CodeWhisperer, and OpenAI ChatGPT allow developers to produce code from natural-language prompts with minimal manual effort. However, these productivity gains are accompanied by significant security risks. Pearce et al. found that approximately 40% of Copilot-generated programs were vulnerable [1]. Siddiq and Santos reported that 74% of Copilot completions on the SecurityEval benchmark were insecure [2], and Fu et al. identified weaknesses in 29.5% of Python snippets generated by Copilot from live GitHub repositories [3]. At the application level, Tóth et al. detected

exploitable vulnerabilities in 26% of GPT-4 generated web applications [4]. These findings indicate that the problem is related to the training data rather than the failures of the model. LLMs learn from public code repositories that contain large amounts of insecure patterns [5], and the problem is exacerbated when adversaries intentionally poison training corpora to inject specific vulnerability classes [6,7].

Current defenses fall into two broad categories, each with significant limitations. Lightweight static analyzers, such as Bandit, scan for suspicious Abstract Syntax Tree (AST) patterns, but fail to detect multi-step and context-dependent vulnerabilities, resulting in false negatives [8]. Formal verification tools, including the Efficient SMT-Based Bounded Model Checker (ESBMC) [9] and CBMC [10], provide bounded safety guarantees through model checking. However, their computational cost makes uniform deployment impractical for large codebases. Importantly, neither approach accounts for risk stratification. All vulnerabilities receive the same level of scrutiny regardless of their severity or likelihood of exploitation.

Resource allocation is therefore the main challenge. Applying uniform analysis to all code fragments results in inefficient resource allocation because it prioritizes low-risk constructs and neglects higher-risk components. A more effective strategy would allocate the verification depth based on assessed risk and adjust the solver parameters for specific weakness classes. Currently, no system has formally defined or empirically evaluated a risk-adaptive approach for AI-generated code.

This paper introduces the Risk-Aware Verification Engine for Code (RAVE-Code), a three-layer verification pipeline designed to address this gap. The first layer (*Detection*) applies Bandit to list weakness instances and annotate each with its Common Weakness Enumeration (CWE) class. The second layer (*Risk Scoring*) computes a composite Risk Priority Score (RPS) per instance from three signals: the Common Vulnerability Scoring System (CVSS) v3.x severity, CWE prevalence in AI-generated code, and Cybersecurity and Infrastructure Security Agency (CISA) Known Exploited Vulnerabilities (KEV) exploitability data. The third layer (*Verification*) routes each instance to one of three tiers: a static-analysis report for low-risk items, standard Bounded Model Checking (BMC) for medium-risk items, or CWE-conditioned BMC with solver parameters tuned to the specific weakness class for high-risk items.

The principal contributions of this paper are as follows:

1. We introduce RAVE-Code as, to the best of our knowledge, the first system to combine composite risk scoring with CWE-conditioned adaptive verification depth for AI-generated code security.
2. We define the RPS as a composite metric that incorporates CVSS severity, CWE prevalence in AI-generated code, and actual exploitability data.
3. We introduce a CWE-conditioned verification strategy that incorporates weakness-specific ESBMC solver parameters, thereby facilitating the detection of vulnerability classes that uniform configurations often overlook.
4. We establish a design principle demonstrating that risk-tiered verification achieves an expected detection rate at least as high as any uniform policy under a bounded resource constraint.
5. We evaluate RAVE-Code on three public benchmarks (SecurityEval, FormAI, and Juliet). We demonstrate improvements of 12.2 and 16.5 percentage points in the detection rate on the FormAI and Juliet benchmarks, respectively. An ablation study confirms that CWE-conditioned routing is the primary contributor to these gains.

The remainder of this paper is organized as follows. Section 2 presents the background foundations of RAVE-Code. Section 3 reviews related work. Section 4 defines the threat model, problem statement, and three-layer architecture of RAVE-Code. Section 5 details the RPS computation and CWE-conditioned verification configurations. Section 6 explains the experimental setup. Section 7 reports the results. Section 8 discusses the findings, limitations, and directions for future work. Section 9 concludes the paper.

2 Background

This section presents the technical foundations of RAVE-Code, including the security characteristics of LM-based code generation, the vulnerability classification and scoring systems employed in RPS computation, and the BMC tools supporting the verification layer.

2.1 LLM-Based Code Generation and Security

LLM-based code generators establish statistical associations between prompts and completions using extensive training corpora. These models do not explicitly reason about security properties, so the code they generate simply mirrors the security patterns present in their training data [1,2,5]. Two primary mechanisms exacerbate this inherent insecurity. First, adversaries can inject insecure code into public repositories, biasing the model toward specific CWE classes [6]. Second, particular prompt phrasings can cause the model to generate vulnerable code [11]. Consequently, AI-generated code requires systematic post-generation verification rather than reliance on the model's implicit security properties.

2.2 CWE Taxonomy, CVSS, and EPSS

The CWE catalog offers a hierarchical taxonomy of software weakness types, each corresponding to a distinct class of repeatable coding errors. The CVSS v3.x [12] quantifies vulnerability severity on a scale from 0 to 10, based on six intrinsic characteristics, and categorizes results into qualitative bands (Low, Medium, High, Critical). RAVE-Code computes the severity component $\sigma(w)$ as the median CVSS Base Score across all National Vulnerability Database (NVD) CVEs mapped to the given CWE class.

A key limitation of CVSS is its focus on measuring potential impact rather than the likelihood of exploitation. The Exploit Prediction Scoring System (EPSS) [13] addresses this gap by applying machine learning techniques to CVE metadata. At a threshold of 0.088, EPSS achieves equivalent exploit coverage to CVSS scores of 7.0 or higher, while flagging only 7.3% of CVEs. RAVE-Code incorporates this insight by deriving the exploitability weight $\varepsilon(w)$ from CISA KEV catalog membership, rather than directly from EPSS scores. This approach is adopted because the KEV entries indicate confirmed active exploitation rather than predicted exploitation probability.

2.3 Bounded Model Checking and ESBMC

BMC formulates the question of whether any execution of a program P of length less than or equal to k violates the property ϕ as a Satisfiability Modulo Theories (SMT) formula [10]. If the formula is satisfiable, a concrete counterexample trace is produced; if unsatisfiable, no violation exists within the bound k. ESBMC [9] offers robust C/C++ frontends, supports incremental BMC, and integrates with Z3 [14]. For RAVE-Code, ESBMC provides per-run configuration flags that determine which verification condition classes (VCCs) are generated. These include `-overflow-check` for integer arithmetic, `-no-bounds-check` to disable array bounds checking, and `-force-malloc-success` to suppress NULL returns from memory allocation. As experimentally shown in Section 7, the selection of appropriate flags for each CWE class is the main mechanism by which the RAVE-Code exceeds the uniform BMC.

3 Related Work

The relevant literature for RAVE-Code spans three primary domains: the security of AI-generated code, vulnerability risk scoring, and software bounded model checking. This section reviews each domain and identifies the specific research gap addressed by RAVE-Code.

3.1 Security of AI-Generated Code

Pearce et al. [1] established an empirical baseline, reporting that approximately 40% of Copilot-generated programs exhibited vulnerabilities in 89 CWE scenarios. Majdinasab et al. [15] confirmed the persistence of these vulnerability patterns in subsequent Copilot versions. Fu et al. [3] extended this analysis to live GitHub repositories. Tóth et al. [4] found that 26% of GPT-4-generated PHP web applications contained exploitable vulnerabilities. He and Vechev [16] investigated the misuse of cryptographic API in LLM-generated code, while Khoury et al. [17] reported that only 5 of 21 responses to ChatGPT-generated code were vulnerability-free. Sandoval et al. [18] conducted a controlled user study and found no statistically significant differences in security between AI-assisted and unassisted code. Collectively, these findings indicate that developers do not mitigate LLM-introduced vulnerabilities during code review.

For vulnerability detection, DeVAIC [8] is a machine learning-based detector trained on AI-generated Python weakness patterns, employing CWE-specific regular expressions in 35 types of weakness and achieving high accuracy in its evaluation corpus. However, DeVAIC has three limitations. First, it is only compatible with Python code and cannot be directly applied to C. Second, it relies on labeled training data and cannot handle new CWE classes without retraining. Third, it provides only classification results and lacks the formal correctness guarantees offered by model checking. SecureQwen [19] utilizes a fine-tuned LLM for vulnerability detection. Cotroneo et al. [6] and Improta [7] provide empirical evidence that training-time data poisoning constitutes a viable attack vector. However, none of these approaches integrates composite risk scoring with formal verification to allow adaptive allocation of verification depth.

3.2 LLM-Generated Code Datasets

Several benchmark datasets have been developed to assess the security of LLM-generated code. SecurityEval [2] contains 130 manually labeled Python samples that span 75 CWE types, generated by Copilot from structured prompts. LLMSecEval [11] provides 150 security-related prompts in 18 CWEs, supporting evaluation across multiple weakness categories. The FormAI dataset [20] is substantially larger, comprising 112,000 GPT-3.5-generated C programs, each formally verified using ESBMC. Because FormAI offers ground-truth labels derived from formal verification rather than manual annotation, it is particularly well-suited for evaluating formal analysis tools. Consequently, FormAI is adopted in this paper as the primary AI-generated C evaluation dataset.

3.3 Vulnerability Risk Scoring

CVSS v3.x [12] is the standard for assessing the severity of vulnerability, although it primarily measures the potential impact rather than the probability of exploitation. Jacobs et al. [13] demonstrated that EPSS achieves a similar exploit coverage while significantly reducing the remediation burden, underscoring the value of exploitation-likelihood signals. Previous research on risk scoring for AI-generated code has considered CWE severity and generation frequency, but has not incorporated formal verification or practical exploitability signals. RAVE-Code advances this area by integrating severity, prevalence, and exploitability signals. In addition, it operationalizes the resulting composite score to guide the allocation of verification resource at the per-instance level.

3.4 Bounded Model Checking for Software

CBMC [10] is the standard implementation of BMC for C and C++ and has served as the foundation for subsequent verification tools. Schrammel et al. [21] formalized incremental BMC for embedded software, and Günther and Weissenbacher [22] established the robustness of this incremental approach. Wu et al. [23] introduced a specification-guided unwinding strategy for incremental BMC. Hsu et al. [24] formalized loop

completeness conditions that determine when a bounded proof can be extended to an unbounded one. RAVE-Code applies the loop completeness criterion of Hsu et al. for its bounded soundness check. To the best of our knowledge, no prior work has applied CWE-conditioned verification parameter configuration to AI-generated code.

4 RAVE-Code: Threat Model and Architecture

This section outlines the operational context of RAVE-Code, specifies the adversarial roles it addresses, formulates the underlying optimization problem, and introduces the three-layer architecture designed to solve it.

4.1 System and Attacker Model

The system considered is an AI-assisted code generation pipeline. A developer provides a natural-language prompt that an LLM processes to produce source code that can be integrated into a deployed application. RAVE-Code functions as a verification step between code generation and deployment. It receives the generated code as input, analyzes it for security weaknesses, and outputs a safety certificate or counterexample trace for each identified weakness. The following definitions formalize this pipeline.

Definition 1 (AI Code Generation Pipeline): $\mathcal{P} = \langle \mathcal{M}, \mathcal{D}, p_{\mathrm{NL}}, \mathcal{C} \rangle$ *where* $\mathcal{M}$ *is an LLM trained on corpus* $\mathcal{D}$, p_{NL} *is a natural-language prompt, and* $\mathcal{C} = \mathcal{M}(p_{\mathrm{NL}})$ *is the generated artifact.*

Definition 2 (Vulnerability Instance): $v = (c, w)$ *where* $c \subseteq \mathcal{C}$ *is a code fragment and* $w \in \mathcal{W}_{\mathrm{CWE}}$ *is its weakness category. The complete vulnerability set is* $V(\mathcal{C}) = \{(c_i, w_i)\}_{i=1}^{n}$.

The threat model addresses two types of adversaries. Attacker A1 (Training-Time Poisoning) injects crafted insecure code snippets into public repositories that make up the training corpus $\mathcal{D}$, thus biasing the model $\mathcal{M}$ to generate code containing specific CWE classes [6]. Research indicates that poisoning as little as 3% of training data can significantly increase vulnerability rates in generated code [6]. The Attacker A2 (Runtime Exploitation) targets the deployed application by exploiting vulnerabilities that persist after development. A2 interacts with the application through its public interface and does not require access to the model or its training data. The defender has access to the generated code $\mathcal{C}$, public vulnerability databases (NVD and CISA KEV), and the ESBMC verification tool, but is limited by a finite total verification cost B.

Fig. 1 depicts the pipeline and the point at which RAVE-Code intercepts the code. The flow from left-to-right represents the generation process: a prompt p_{NL} is provided to the LLM $\mathcal{M}$, which generates code $\mathcal{C}$ for deployment. Attacker A1 influences the pipeline by poisoning the training data, whereas Attacker A2 exploits vulnerabilities in the deployment stage. RAVE-Code intercepts the generated code prior to deployment, applying a three-layer analysis to assess whether each fragment is bounded-safe before integration into the deployed application.

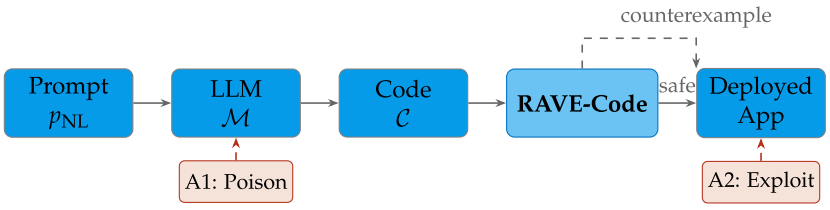

Figure 1: RAVE-Code threat model. A prompt is processed by the LLM, generating code for deployment. RAVE-Code intercepts this code for risk-based verification: safe code is deployed, while vulnerable code yields a counterexample trace. Attacker A1 poisons training data; Attacker A2 exploits deployed code.

4.2 Problem Statement

The primary challenge addressed by RAVE-Code is the allocation of finite computational resources across code fragments with varying risk levels. Since not all weaknesses exhibit the same severity or likelihood of exploitation, uniform verification is suboptimal. The following definitions formalize the tier assignment and the associated optimization objective.

Definition 3 (Verification Tier):

$$T(v) = \begin{cases} L & \text{RPS}(v) < \theta_L, \\ M & \theta_L \leq \text{RPS}(v) < \theta_H, \\ H & \text{RPS}(v) \geq \theta_H, \end{cases} \tag{1}$$

with $\theta_L = 0.4$, $\theta_H = 0.7$. *The RPS is defined formally in Section 5.*

Each tier corresponds to a different level of verification effort. The lower tier fragments receive only a static-analysis report from Layer 1. Medium-tier fragments are analyzed with standard BMC using a fixed ESBMC configuration. High-tier fragments receive CWE-conditioned BMC, in which ESBMC solver parameters are tailored to the specific weakness class (detailed in Section 5).

Definition 4 (CWE-Conditioned Property): *For the CWE class w,* ϕ_w *is a safety property on the symbolic state space of c, falsified by any state showing the vulnerability pattern of w. In RAVE-Code,* ϕ_w *is instantiated as a combination of ESBMC flags specific to w.*

Core Problem. Given a set of code fragments $\{c_1, \ldots, c_m\}$ and a finite total verification cost B, find a tier assignment $T : \{c_i\} \to \{L, M, H\}$ that maximizes the expected number of detected vulnerabilities while keeping the total cost within B.

Design Principle (Risk-Tiered Dominance). Let $p_H > p_M > p_L > 0$ and $C_H > C_M > C_L > 0$ be the detection rates per-tier and the expected costs. Under a binding cost constraint B, a risk-tier policy achieves at least as high an expected true-positive count as any uniform single-tier policy, with strict improvement when the portfolio spans multiple risk levels.

Informal justification. Reassigning a Low-RPS fragment from the tier H to the tier L releases $\Delta = C_H - C_L > 0$, allowing additional High-RPS verification at expected gain $p_H - p_L > 0$. This follows from standard resource-allocation arguments.

4.3 Three-Layer Architecture

RAVE-Code addresses the above-mentioned problem through a three-layer pipeline that intercepts AI-generated code prior to deployment and applies risk-stratified verification. The architecture is illustrated in Fig. 2.

The rationale for this layered design is the separation of concerns. The Detection Layer operates quickly, but with limited precision, identifying candidate weaknesses through pattern matching without confirming exploitability. The Risk Scoring Layer integrates multiple risk signals to determine the appropriate verification effort for each candidate. The Verification Layer delivers formal guarantees by employing bounded model checking to either confirm vulnerabilities with concrete counterexamples or certify bounded safety. Each layer utilizes distinct resources and generates output that informs subsequent stages. The following paragraphs provide a high-level overview of each layer, while Section 5 presents the formal definitions and configurations.

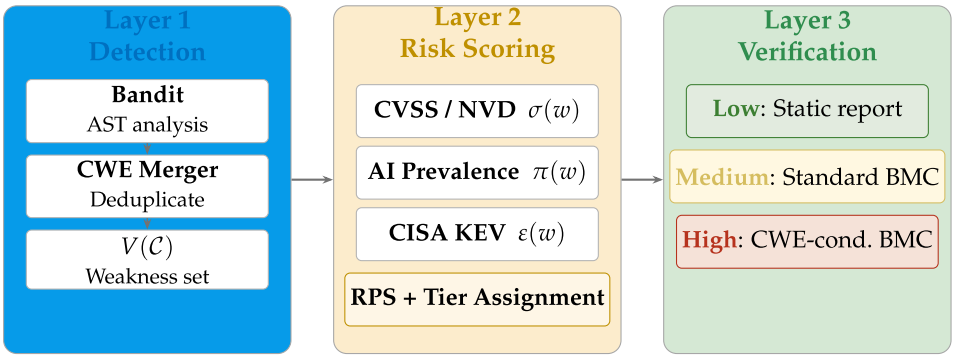

Figure 2: RAVE-Code's three-layer architecture: Layer 1 detects and annotates weaknesses. Layer 2 computes a composite RPS and assigns a verification tier. Layer 3 routes instances to static analysis (Low), standard BMC (Medium), or tailored BMC (High).

Layer 1: Detection. Bandit performs AST-based pattern matching on the input code and annotates each finding with its corresponding CWE class. The CWE Instance Merger deduplicates findings that reference the same source location, resulting in a consolidated vulnerability set $V(\mathcal{C})$. Bandit is selected for its zero-configuration deployability, integrated CWE annotations compatible with the risk-scoring layer, and high precision (zero false positives on the SecurityEval benchmark). The detection layer is designed for extensibility, allowing integration of additional analysis engines as future components without modifying downstream layers.

Layer 2: Risk Scoring. Each detected weakness instance is assigned a composite RPS that integrates signals from the NVD, published AI-code benchmarks, and the CISA KEV catalog. The RPS determines the verification tier (Low, Medium, or High) for each instance. This scoring mechanism ensures that verification resources are prioritized for weaknesses that are severe, prevalent in AI-generated code, and actively exploited. The formal definition of RPS and the tier thresholds are detailed in Section 5.

Layer 3: Verification. The verification layer employs distinct analysis strategies for each tier. the Lower tier instances receive only the static-analysis report from Layer 1, as their risk profile does not warrant formal verification. Medium-tier instances are analyzed with ESBMC using a standard configuration. High-tier instances undergo CWE-conditioned bounded model checking, where ESBMC solver parameters are tailored to the specific weakness class. This tailored approach is essential, as different CWE classes require distinct ESBMC configurations for effective detection. Applying a uniform configuration can result in missed vulnerabilities. The specific configurations for each CWE class and their justifications are provided in Section 5.

Proposition 1: (Bounded Soundness): *The following guarantee applies to Layer 3 of the RAVE-Code framework. For fragment c verified at bound k^*, if ESBMC returns VERIFICATION SUCCESSFUL and the loop completeness criterion of [24] holds, then no execution of c of length $\leq k^*$ violates ϕ_w.*

Remark 1: *Fragments that do not reach completeness within the timeout τ are reported as "bounded-safe to depth k^*" and flagged for human review.*

5 Risk Scoring and CWE-Conditioned Verification

This section provides a formal definition of the Risk Priority Score (RPS) and describes the CWE-conditioned verification configurations derived from it. These topics are presented jointly because the per-CWE ESBMC parameters are determined by the tier assignments (Definition 3) established through the RPS.

5.1 Risk Priority Score

The RPS incorporates three components: severity, which quantifies potential impact using CVSS; prevalence, which measures the frequency of each CWE in AI-generated code; and exploitability, which is informed by evidence of active exploitation from the CISA KEV catalog. The dependence on CVSS alone is inadequate, as it assesses only the potential impact and not the likelihood of exploitation. Furthermore, CVSS does not account for the domain distribution of weakness types in AI-generated code [13].

Definition 5 (Risk Priority Score):

$$\mathrm{RPS}(v) = \alpha\,\sigma(w) + \beta\,\pi(w) + \gamma\,\varepsilon(w), \quad \alpha + \beta + \gamma = 1, \tag{2}$$

where $\sigma(w) = \overline{\mathrm{CVSS}}(w)/10$ *is the normalized median NVD CVSS v3.x Base Score;* $\pi(w)$ *is the normalized prevalence of CWE in AI-generated code [2,11]; and* $\varepsilon(w)$ *is the exploitability weight, defined as 1.0 if* w *has* ≥ 5 *CISA KEV entries, 0.6 for 1–4 entries, 0.3 if in CWE Top-25 but not KEV, and 0.1 otherwise. The default weights are* $\alpha = 0.4$, $\beta = 0.35$, $\gamma = 0.25$. *Table 1 lists RPS values for the evaluated CWE classes.*

Table 1: RPS component values. $\alpha = 0.4$, $\beta = 0.35$, $\gamma = 0.25$. H ≥ 0.70; M $\in [0.40, 0.70)$; L < 0.40.

CWE	Weakness	$\bar{s}$	ε	Tier
CWE-078	OS Command Injection	9.0	1.00	L†
CWE-122	Heap Buffer Overflow	9.8	1.00	M*
CWE-476	NULL Pointer Deref	7.5	0.60	M
CWE-190	Integer Overflow	7.8	1.00	M
CWE-191	Integer Underflow	7.8	0.60	M
CWE-590	Free Not on Heap	7.5	0.30	L*
CWE-690	NULL Deref from Return	7.5	0.60	M
CWE-789	Uncontrolled Mem. Alloc	7.5	0.60	M
CWE-377	Insecure Temporary File	5.5	0.30	M
CWE-400	Resource Exhaustion	7.5	0.60	M
CWE-681	Numeric Conversion	6.5	0.30	L*
CWE-835	Infinite Loop	5.9	0.30	L*
CWE-197	Numeric Truncation	5.5	0.10	L
CWE-459	Incomplete Cleanup	4.3	0.10	L

Note: *Tier shifts to H under severity-heavy or exploit-heavy weights (see sensitivity analysis). †It is set from High to Low tier because BMC cannot analyze past `execl/system` calls.

Algorithm 1 processes each detected weakness as follows: (1) queries the NVD REST API for the median CVSS v3.x base score (cached offline), (2) computes normalized severity σ, prevalence weight π, and exploitability weight ε, (3) calculates the composite score RPS $= \alpha\sigma + \beta\pi + \gamma\varepsilon$, and (4–5) assigns a verification tier by comparing RPS to thresholds $\theta_L = 0.4$ and $\theta_H = 0.7$. This requires one NVD lookup and three table accesses per instance, resulting in negligible overhead relative to BMC verification.

Algorithm 1: Risk priority score computation

Require: Instance *(c, w)*; weights (α, β, γ)
Ensure: RPS $\in [0,1]$; tier $T \in \{L, M, H\}$

(Continued)

Algorithm 1 (continued)

$\bar{s} \leftarrow$ NVD_API(w) ▷ Median CVSS v3.x; cached offline
$\sigma \leftarrow \bar{s}/10$; $\pi \leftarrow$PrevalenceTable (w); $\varepsilon \leftarrow$ KEV_Lookup (w)
RPS $\leftarrow \alpha\sigma + \beta\pi + \gamma\varepsilon$
if RPS $\geq \theta_H$ **then return** (RPS, H)
else if RPS $\geq \theta_L$ **then return** (RPS, M)
else return (RPS, L)
end if

5.2 RPS Weight Sensitivity

The weights $\alpha = 0.4$, $\beta = 0.35$, and $\gamma = 0.25$ reflect the intended importance of each component. CVSS Base Score receives the most weight. Prevalence is given a moderate weight. The binary KEV exploitability indicator receives the least weight. The weights sum to 1 and satisfy $\alpha > \beta > \gamma > 0$. Sensitivity analysis (Table 2) demonstrates that the framework is robust to reasonable changes in these weights. To evaluate robustness, RPS is tested with four weight settings: Baseline ($\alpha = 0.40$, $\beta = 0.35$, $\gamma = 0.25$), Equal (0.33, 0.33, 0.33), Severity-heavy (0.60, 0.20, 0.20), and Exploit-heavy (0.20, 0.20, 0.60). Future work will automate weight selection using labeled CVE data.

Analysis is conducted at both the tier assignment and detection performance levels, leveraging existing ESBMC results without rerunning verification. Table 2 reports the detection metrics for both FormAI and Juliet benchmarks.

Table 2: RPS weight sensitivity: DR, Precision, F_1 across Four Weight Combinations. FormAI is invariant. Juliet varies at most 8.6 percentage points in DR. All variation is upward from the conservative Baseline.

Weights (α, β, γ)	FormAI (GPT-3.5 C)			Juliet (Synthetic C)		
	DR%	P%	F_1	DR%	P%	F_1
Baseline (0.40, 0.35, 0.25)	86.0	96.3	0.909	43.3	58.1	0.496
Equal (0.33, 0.33, 0.33)	86.0	96.3	0.909	43.3	58.1	0.496
Severity-heavy (0.60, 0.20, 0.20)	86.0	96.3	0.909	43.3	66.0	0.523
Exploit-heavy (0.20, 0.20, 0.60)	86.0	96.3	0.909	51.9	62.7	0.568

Note: Juliet values use tier-routing mode (High → RAVE-Code, others → Uniform) to isolate the weight effect. Full RAVE-Code pipeline on Juliet: DR = 57.3%, $F_1 = 0.614$.

At the tier assignment level, 7 of the 14 CWEs evaluated retain identical tier assignments across all four weight sets. The remaining 7 tier shifts occur only under the two extreme configurations (severity-heavy or exploit-heavy), and all shifts are upward (Low to Medium or Medium to High), indicating that affected CWEs receive more extensive verification under alternative weights. In both the Baseline and Equal weight sets, only CWE-122 advances to a higher verification tier under the RPS calculation. Despite its high RPS, CWE-078 is manually assigned to the Low tier (Table 3) because BMC cannot analyze behavior beyond the execl system call. These results demonstrate that the tier assignments are robust to minor weight variations.

Table 3: CWE-conditioned ESBMC configurations. All use Z3 solver and `no-unwinding-assertions`. Timeout $\tau = 120$ s. ✓ = enabled; – = disabled/absent.

CWE	Tier	k	oflow-chk	no-bnd-chk	frc-malloc	Rationale
CWE-122	H	25	–	✓	✓	High unwind exposes overflow; eliminates FP on safe variant
CWE-476	H	15	–	✓	–	NULL returns must be reachable
CWE-690	H	15	–	–	–	Same rationale as CWE-476
CWE-078	L	10	–	–	✓	`Excel/system` lack bodies, so BMC cannot analyze them
CWE-190	M	15	✓	–	✓	Overflow check required for integer VCCs
CWE-191	M	15	✓	–	✓	Same as CWE-190
CWE-400	M	15	–	–	✓	Memory-leak-check enabled
CWE-197, 681	M/L	10	✓	–	✓	Overflow check for numeric conversion
Others	M	10	–	✓	✓	Standard configuration

At the detection performance level, FormAI results remain unchanged across all four weight sets, with a Detection Rate (DR) of 86.0% and a F_1 score of 0.909 for each combination. Juliet's results are stable under Baseline and Equal weights (DR of 43.3% in both cases), with modest increases observed under extreme configurations (up to 51.9% DR under Exploit-heavy). All observed variation is upward. The Baseline weights are the most conservative, and no weight set results in reduced performance.

5.3 CWE-Conditioned Verification

An empirical finding of this study is that different CWE classes require fundamentally different ESBMC configurations. The assignment of tier generated by the RPS (Definition 5, Eq. (2)) determines the configuration applied to each code fragment. This subsection details the per-CWE setups and their underlying rationale.

The Integer overflow and underflow (CWE-190, CWE-191) require enabling the `–overflow-check` flag. Without this flag, ESBMC does not generate integer arithmetic verification condition classes (VCCs) and cannot detect these weaknesses, regardless of unwind depth. For NULL pointer dereferences from return values (CWE-690), the `–force-malloc-success` flag must be disabled. When enabled, this flag symbolically suppresses NULL returns from `malloc`, rendering the vulnerability unreachable. CWE-476 (general NULL dereferences) also requires disabling this flag. OS command injection (CWE-078) relies on

external calls that BMC cannot analyze, making intensive verification inefficient. Thus, CWE-078 is assigned to the Low tier as we will discuss in Section 7.

Buffer overflows (CWE-122) require higher unwind bounds ($k = 25$ compared to $k = 10$) to expose memory-copy overflows within loops. The uniform application of these configurations to all CWE classes can lead to performance regressions for classes that do not require them, as demonstrated by the ablation study in Section 7. Table 3 presents the per-CWE configurations. Algorithm 2 presents the High-tier procedure.

Algorithm 2: High-tier CWE-conditioned verification

Require: Fragment c; CWE w; config table (Table 3); timeout τ
Ensure: UNSAFE(c, w)+CE, BNDSAFE(c, w, k^*), or TIMEOUT

```
flags ← ConfigTable(w)                                   ▷ Table 3
P ← InstrumentFunction(c,w)
r ← Esbmc(P, flags, τ)
if r =VERIFICATION FAILED then
    return Unsafe, Getcounterexample()
else if r =VERIFICATION SUCCESSFUL then
    return BndSafe(c, w, k*)
else
    return Timeout
end if
```

6 Experimental Setup

This section presents the research questions, datasets, baseline methods, implementation details, and assessment criteria. The experimental design follows the practices established by Wohlin et al. [25].

The evaluation addresses four research questions. RQ1 measures how much of the vulnerability surface Bandit detects in AI-generated Python code, evaluating the static analysis layer used alone. RQ2 assesses whether a CWE-conditioned BMC identifies more vulnerabilities than a uniform BMC in AI-generated C code. RQ3 determines which CWE classes benefit the most from conditioning and which remain undetectable by BMC. RQ4 evaluates whether observed improvements result from CWE-conditioned routing or from a uniform configuration using RAVE-Code's flags, which may yield comparable results.

6.1 Datasets

This study uses three publicly available benchmarks. SecurityEval [2] consists of 130 Python samples spanning 75 CWE types, generated by Copilot from docstring prompts and labeled by security experts. All samples are vulnerable (ground truth = 1), making it the baseline for the evaluation of static analysis (RQ1). ESBMC is not applied to this dataset because its Python frontend requires pre-built module stubs, which are unavailable for SecurityEval's diverse third-party imports.

FormAI [20] contains 112,000 C programs generated by GPT-3.5-Turbo and labeled using ESBMC. A sample is VULNERABLE if ESBMC detects a security violation, and NOT VULNERABLE otherwise. From this dataset, 580 samples were selected, balanced across five CWE classes (CWE-122, CWE-190, CWE-191, CWE-476, CWE-590) and safe samples. After excluding entries without valid ESBMC verdicts, 436 remain. FormAI is the primary evaluation dataset for AI-generated C code (RQ2, RQ3). Because ground-truth labels and evaluation are based on the same tool, the results can overestimate the true detection ability of ESBMC, as we will discuss in Section 8.

The Juliet Test Suite (C/C++) [26] is a synthetic benchmark developed by NIST/NSA, containing 734 C entries spanning 14 CWE types. The ground truth is defined structurally: `bad()` functions are labeled vulnerable (ground truth = 1) and `good()` functions are labeled safe (ground truth = 0). After removing entries without valid ESBMC verdicts, 618 remain. Since Juliet's labels are independent of ESBMC, it offers the strongest check on RAVE-Code's detection results. Juliet is used for the per-CWE analysis (RQ3) and the ablation study (RQ4). Table 4 summarizes all three datasets.

Table 4: Evaluation datasets.

Dataset	N	Vulnerable	Safe	CWEs	Language	Ground Truth
SecurityEval [2]	130	130	0	75	Python	Expert labels
FormAI [20]	436	214	222	5	C (GPT-3.5)	ESBMC formal
Juliet [26]	618	309	309	14	C (synthetic)	Structural

6.2 Baselines and Ablation Configurations

The Bandit-only baseline applies Bandit v1.7.5 to the SecurityEval dataset. The uniform ESBMC baseline applies ESBMC 8.1.0 with a fixed configuration to all C samples without CWE-specific tuning. It uses the Z3 solver, an unwind bound of $k = 10$, bounds checking disabled, malloc forced to succeed, unwinding assertions suppressed, and a 60 s timeout.

Three ablation configurations are used to isolate the contribution of CWE-conditioned routing. Table 5 summarizes the flag settings. Ablation A (Uniform-High) applies a high unwind bound ($k = 25$) and overflow detection uniformly across all CWEs, matching RAVE-Code's High tier. Ablation B (Uniform-Mid) adds overflow detection to the standard uniform configuration at $k = 10$. Ablation C (Uniform-Strict) is identical to B but also enables NULL-return detection by removing the forced-malloc-success flag. All ablation configurations turn off unwinding assertions to avoid false alarms caused by reaching the unwind limit.

Table 5: Ablation configuration summary. All configurations use the Z3 solver. ✓ = flag enabled; – = flag absent.

Configuration	*k*	Overflow-Check	No-Bounds-Check	Force-Malloc-Success
Uniform ESBMC	10	–	✓	✓
Ablation A (High)	25	✓	✓	✓
Ablation B (Mid)	10	✓	✓	✓
Ablation C (Strict)	10	✓	✓	–

6.3 Implementation and Assessment Criteria

All experiments were conducted on Ubuntu 22.04 (Docker Desktop, WSL2, 5.8 GB RAM) using Bandit 1.7.5 and ESBMC 8.1.0, compiled from source with Z3 4.8.12. NVD severity data were obtained via the REST API v2.0. KEV exploitability data were obtained from the CISA JSON feed.

Four standard metrics are reported. A *True Positive* (TP) is a vulnerable fragment correctly identified as vulnerable. A *False Positive* (FP) is a safe fragment incorrectly flagged as vulnerable. A *False Negative* (FN) is a vulnerable fragment that the tool cannot detect. A *True Negative* (TN) is a safe fragment correctly classified as safe. Four derived metrics are also used. *Detection Rate* (DR) is $\mathrm{TP}/(\mathrm{TP} + \mathrm{FN})$, the proportion of true vulnerabilities correctly identified. *False Negative Rate* (FNR) is $1 - \mathrm{DR}$. *Precision* (P) is $\mathrm{TP}/(\mathrm{TP} + \mathrm{FP})$, measuring the reliability of positive predictions. The F_1 score is the harmonic mean of DR and Precision.

Pairwise comparisons are assessed using McNemar's test at $p < 0.05$. SecurityEval contains only vulnerable samples (TN = 0), so a precision of 100% on that dataset reflects zero false positives and is not independently informative. Samples that exceed the 60s ESBMC timeout receive a VERIFICATION UNKNOWN verdict. These are counted as false negatives for vulnerable samples and true negatives for safe ones. This conservative approach is applied consistently to both the uniform-ESBMC baseline and RAVE-Code.

7 Results

Results are reported across all three datasets, organized by research question. On SecurityEval, Bandit detects 28.5% of Python vulnerabilities with zero false positives, establishing a static analysis baseline (RQ1). On FormAI, RAVE-Code achieves DR = 86.0% and $F_1 = 0.909$, compared to DR = 73.8% and $F_1 = 0.834$ for uniform ESBMC, a gain of 12.2 percentage points (McNemar $p < 0.001$). RAVE-Code also detects 41 vulnerabilities that uniform ESBMC overlooks entirely (RQ2). CWE-190 and CWE-476 benefit the most from conditioning, with gains of 32 and 33 percentage points, respectively. CWE-078 remains undetectable by BMC, and CWE-122 drops by 13 percentage points due to timeouts on larger programs (RQ3). The ablation study confirms that CWE-conditioned routing drives the improvement. On Juliet, the best uniform ablation reaches only 38.5% DR, against RAVE-Code's 57.3%. Ablation A performs worse than the standard uniform baseline (RQ4). Per-tier runtime measurements show that the High tier completes in an average of 0.50 s without timeouts, whereas the Low tier (dominated by CWE-078) is the most time-consuming. This confirms the decision to place CWE-078 in the Low tier (RQ5).

7.1 RQ1: Static-Analysis Baseline (SecurityEval, Python)

The SecurityEval results assess the detection capability of the static analysis layer on its own. Among 130 Python samples, Bandit identified 37 (28.5%) with zero false positives (TP = 37, FP = 0, FN = 93). Detected samples correspond to cases where Bandit's AST patterns directly match the weakness signature. The 93 false negatives involve multi-step patterns, such as SQL injection spanning function boundaries or insecure randomness in cryptographic contexts where insecurity depends on the calling context. DeVAIC [8], trained specifically on AI-generated Python weakness patterns, achieves higher detection rates on its own evaluation corpus than general-purpose static analyzers. This difference highlights the advantage of specialized training over general-purpose pattern matching.

ESBMC was also applied to SecurityEval, but did not produce valid verdicts because its Python frontend requires pre-built module stubs for third-party libraries. Of the 130 samples, 85 failed due to missing library support (such as PyYAML, Flask, and cryptography imports), 25 produced other parsing errors, 16 returned misleading VERIFICATION SUCCESSFUL verdicts because the missing imports prevented ESBMC from reaching the vulnerable code paths, and 4 produced no output. These results are excluded from the detection metrics.

7.2 RQ2: FormAI (GPT-3.5-Generated C)

Fig. 3 reports the FormAI results. RAVE-Code achieves a DR of 86.0% and $F_1 = 0.909$, compared to 73.8% and $F_1 = 0.834$ for uniform ESBMC. The 41 vulnerabilities detected only by RAVE-Code comprise 14 for CWE-476, 20 for CWE-190, and 7 for CWE-191, all missed by uniform ESBMC. Both methods achieve zero false positives after correcting for unwinding assertion failures. False FAILED verdicts caused by reaching the unwind limit are eliminated by adding `--no-unwinding-assertions`.

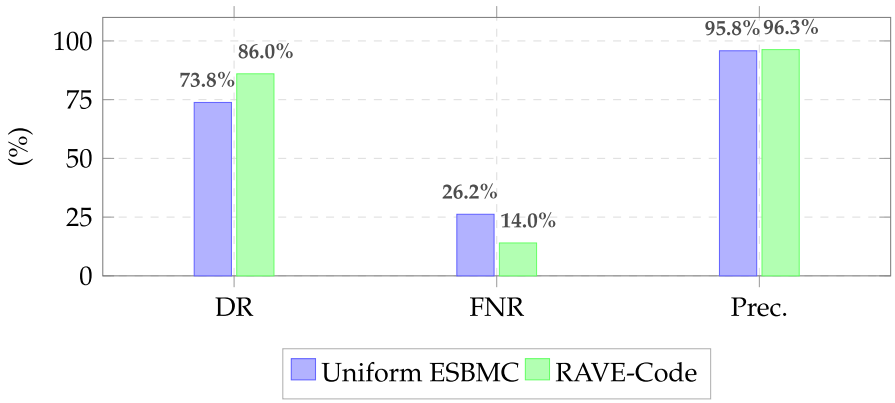

Figure 3: Aggregate metrics on FormAI (GPT-3.5-generated C, 436 samples). DR = Detection Rate; FNR = False Negative Rate; Prec. = Precision.

7.3 RQ3: Per-CWE Analysis

Table 6 presents the per-CWE results for FormAI. CWE-190 and CWE-476 exhibit the largest improvements (+32.1 and +33.4 percentage points, respectively). These gains are driven by configurations unavailable in uniform ESBMC, namely the `-overflow-check` flag for CWE-190/191 and the removal of `-force-malloc-success` for CWE-476. CWE-122 shows a regression of 13.2 percentage points because the higher unwind bound ($k = 25$) increases per-sample verification time, causing more timeouts on larger programs generated by GPT-3.5.

Table 6: Per-CWE results on FormAI (GPT-3.5-Generated C).

CWE	N	Uniform DR	RAVE DR	ΔDR
CWE-122 (Heap Buffer Overflow)	38	71.1	57.9	−13.2
CWE-190 (Integer Overflow)	56	64.3	96.4	+32.1
CWE-191 (Integer Underflow)	57	86.0	87.7	+1.7
CWE-476 (NULL Pointer Deref.)	36	58.3	91.7	+33.4
CWE-590 (Free Not on Heap)	27	92.6	92.6	0.0
Overall	**214**	**73.8**	**86.0**	**+12.2**

Table 7 presents the per-CWE results for Juliet across all 14 CWE types. The results mirror those of FormAI: CWE-190 and CWE-191 each gain 100 percentage points (from 0% to 100%) through the use of `-overflow-check`, CWE-197 gains 36.3 percentage points, and CWE-122 and CWE-690 improve precision (from 50% to 100%) through targeted parameter selection.

Table 7: Per-CWE detection rate on Juliet (Synthetic C, 618 samples).

CWE	Tier	N	Uniform DR	RAVE DR	ΔDR
CWE-122 (Heap Buffer Overflow)	H	52	100.0	100.0	0.0
CWE-690 (NULL Deref. from Return)	H	50	92.0	92.0	0.0
CWE-078 (OS Command Injection)	L	50	0.0	0.0	0.0
CWE-789 (Uncontrolled Mem. Alloc.)	H	50	0.0	0.0	0.0

(Continued)

Table 7 (continued)

CWE	Tier	N	Uniform DR	RAVE DR	ΔDR
CWE-190 (Integer Overflow)	M	50	0.0	100.0	+100.0
CWE-191 (Integer Underflow)	M	50	0.0	100.0	+100.0
CWE-590 (Free Not on Heap)	M	48	100.0	100.0	0.0
CWE-681 (Numeric Conversion)	M	52	100.0	100.0	0.0
CWE-835 (Infinite Loop)	M	12	100.0	100.0	0.0
CWE-377 (Insecure Temp. File)	M	52	42.3	42.3	0.0
CWE-400 (Resource Exhaustion)	M	50	32.0	20.0	−12.0
CWE-197 (Numeric Truncation)	L	22	18.2	54.5	+36.3
CWE-459 (Incomplete Cleanup)	L	30	0.0	0.0	0.0
Overall		**618**	**40.8**	**57.3**	**+16.5**

Note: RAVE-Code: CWE-190 (50), CWE-191 (50), CWE-197 (4) = 104 total. Precision gains: CWE-122 (50% → 100%), CWE-690 (50% → 100%)

7.4 RQ4: Ablation Study

Table 8 addresses whether the observed improvement results from CWE-conditioned routing or if any uniform configuration using RAVE-Code's flags would yield similar outcomes. The results indicate a clear distinction.

Table 8: Ablation study on Juliet (618 Samples). All methods use ESBMC. The only difference is how the flags are selected.

Method	DR (%)	FNR (%)	P (%)	F_1
RAVE-Code (CWE-conditioned)	57.3	42.7	66.0	0.614
Uniform ESBMC (standard)	40.8	59.2	58.1	0.479
Ablation A: k = 25, overflow, uniform	30.7	69.3	78.5	0.442
Ablation B: k = 10, overflow, uniform	38.5	61.5	57.2	0.460
Ablation C: k = 10, overflow, no-suppress	38.5	61.5	57.2	0.460

Note: RAVE-Code exceeds best ablation (B/C) by 18.8 percentage points. McNemar $p < 0.001$.

RAVE-Code achieves a detection rate of 57.3%, outperforming the best uniform ablation (38.5%) by 18.8 percentage points. Ablation A, which applies k = 25 and overflow uniformly, performs worse than the standard uniform baseline (30.7% compared to 40.8%). These findings indicate that applying configurations tailored to specific CWE classes across all CWEs results in performance regressions. The routing mechanism is essential because it ensures that each configuration is matched appropriately to the CWE class.

7.5 RQ5: Runtime Cost per Tier

To assess how RAVE-Code allocates verification resources, we measured per-sample wall-clock time using a single verification pass for each Juliet source file (387 files; 339 within the 13 tier-assigned CWEs). The ESBMC timeout was set to 60 s per sample. Table 9 reports mean, median, and timeout counts by tier and CWE.

Three observations follow from these measurements. First, the High tier is the most efficient. Despite using larger unwind bounds and conditioned flags, all 90 High-tier samples completed without timeout, with a mean of 0.50 s. The CWE-conditioned configurations terminate quickly on the targeted patterns because they direct the solver to the relevant verification conditions rather than exploring unrelated paths. Second, the Medium tier shows moderate cost dominated by CWE-400 (mean 29.70 s), which performs resource-exhaustion analysis that requires deeper exploration of allocation paths. The remaining Medium-tier CWEs complete in under 0.5 s on average. Third, the Low tier is the most expensive (mean 25.23 s, 41.3% timeout rate), driven almost entirely by CWE-078 (mean 48.12 s, median at the 60 s timeout). This supports the decision to assign CWE-078 to the Low tier, since intensive verification of OS command injection is ineffective. Overall, these results demonstrate that verification is most efficient where BMC is effective, while unnecessary computation is avoided for cases where BMC has fundamental limitations. Since all samples use a 60 s timeout, the reported times accurately represent the bounded effort required by each tier.

Table 9: Per-tier and per-CWE wall-clock runtime on Juliet (339 samples). Each source file experiences a single verification. Mean and median values include timeouts, which are capped at 60 s.

Group	Tier	*N*	Mean (s)	Median (s)	Timeouts
Per tier					
High	H	90	0.50	0.54	0 (0.0%)
Medium	M	186	5.17	0.47	12 (6.5%)
Low	L	63	25.23	0.54	26 (41.3%)
Per CWE					
CWE-122 (Heap Buffer Overflow)	H	30	0.48	0.53	–
CWE-690 (NULL Deref. from Return)	H	30	0.48	0.51	–
CWE-789 (Uncontrolled Mem. Alloc.)	H	30	0.55	0.60	–
CWE-190 (Integer Overflow)	M	30	0.42	0.44	–
CWE-191 (Integer Underflow)	M	30	0.42	0.44	–
CWE-377 (Insecure Temp. File)	M	30	0.49	0.48	–
CWE-400 (Resource Exhaustion)	M	30	29.70	26.13	–
CWE-590 (Free Not on Heap)	M	30	0.45	0.47	–
CWE-681 (Numeric Conversion)	M	30	0.49	0.54	–
CWE-835 (Infinite Loop)	M	6	0.45	0.44	–
CWE-078 (OS Command Injection)	L	30	48.12	60.05	–
CWE-197 (Numeric Truncation)	L	15	9.23	0.45	–
CWE-459 (Incomplete Cleanup)	L	18	0.42	0.47	–

8 Discussion

Across all three datasets, the combination of risk stratification and CWE-conditioned verification consistently outperforms the uniform ESBMC approach. Neither component, when applied independently, achieves comparable results.

Risk stratification provides two main advantages. First, it directs formal verification efforts toward high-risk code fragments and applies static analysis to low-risk fragments, thereby optimizing resource allocation. Second, it enables configuration on a per-class basis. Identifying the CWE class of each fragment allows the selection of appropriate flags, such as `-overflow-check` for integer weaknesses, removal

of `-force-malloc-success` for NULL-dereference classes, and increased unwind bounds for buffer-overflow classes. In the absence of Layer 2's tier assignment, Layer 3 defaults to a uniform configuration. The ablation study demonstrates that applying all RAVE-Code flags uniformly results in an 18.8 percentage-point decrease in performance, primarily due to regressions in CWE classes that do not require those flags.

The FormAI results ($F_1 = 0.909$) are particularly informative because FormAI includes programs generated by GPT-3.5, which represents the type of code RAVE-Code is designed to protect. Of the 41 vulnerabilities detected exclusively by RAVE-Code, 20 are integer overflows (CWE-190), and 14 are NULL dereferences (CWE-476), both categories that uniform configurations did not identify. However, since FormAI's ground-truth labels were generated by ESBMC, the reported detection rate of 86.0% reflects consistency between two ESBMC runs rather than validation against an independent oracle. The Juliet results mitigate this concern, as its ground truth is independent of ESBMC.

The observed precision improvements on Juliet CWE-122 and CWE-690 are of practical significance. Developers generally place greater trust in formal verification results than in static analysis warnings. Therefore, an incorrect VERIFICATION FAILED verdict on safe code can significantly undermine confidence in the verification pipeline. Removing 49 such false positives on Juliet (26 for CWE-122 and 23 for CWE-690), while maintaining the same number of true positives, demonstrates an additional benefit of CWE-conditioned configuration. This approach not only enhances vulnerability detection, but also reduces false alerts on safe code.

The bandit was selected for Layer 1 on the basis of three practical considerations. First, it does not require a trained model or a GPU. Second, its CWE annotations integrate seamlessly into the RPS computation. Third, it produces zero false positives on SecurityEval. Although purpose-trained detectors such as DeVAIC [8] achieve higher detection rates on AI-generated Python code, substituting such a detector in Layer 1 would improve Python coverage without impacting the subsequent layers.

8.1 Limitations and Future Work

The evaluation has several limitations. Formal verification was demonstrated only on C code, as ESBMC's Python frontend lacks support for SecurityEval's third-party imports. FormAI's ground-truth labels were generated by ESBMC itself, so the detection results reflect consistency between two ESBMC runs rather than validation against an independent oracle. The FormAI evaluation covers only five of the available CWE classes. Two CWE-specific regressions were observed. For CWE-122, the $k = 25$ unwind bound caused timeouts on larger programs generated by GPT-3.5, resulting in a 13.2 percentage point regression and showing that the configurations in Table 3 are dataset-specific. For CWE-400, the `-memory-leak-check` flag alters the analysis state in ways that hide resource-exhaustion patterns. The Detection Layer was evaluated using Bandit alone, which cannot track data flow across function boundaries, so multi-step vulnerabilities such as SQL injection are systematically missed. The Juliet Test Suite uses isolated synthetic test cases that differ structurally from real AI-generated code. Of the 734 Juliet entries, 116 were excluded due to ESBMC parser errors. These exclusions are systematic and unrelated to RAVE-Code. All 36 samples for CWE-667 (Improper Locking) were excluded because ESBMC does not support the necessary threading primitives. The remaining 80 excluded samples are distributed among the other evaluated CWE classes. Because these exclusions occur before any tier assignment or verification, they affect both Uniform-ESBMC and RAVE-Code equally and do not bias the results. The reduction from 734 to 618 samples is due to ESBMC's parser limitations and does not reflect any limitation of the proposed approach. Runtime data are based on 339 samples, with a single verification run per source file. A full runtime analysis across all tiers and the entire evaluation set would offer a more complete assessment of how verification effort is distributed.

Several directions could further enhance our contribution. Adding an inter-procedural taint analysis tool such as CodeQL or Infer would extend coverage to data-flow CWE classes that Bandit and BMC cannot handle. We aim to extend our approach by implementing adaptive unwind-bound selection using the loop-completeness criterion proposed by Hsu et al. [24]. This strategy would replace the current manually tuned per-CWE boundaries with adaptive values, effectively addressing the CWE-122 regression issue. Additionally, we plan to replace the existing heuristic assignments by automating the learning of RPS weights (α, β, γ) from labeled CVE data. Our evaluation will also include JavaScript, TypeScript, and various large language models such as GPT-4, Code Llama, and CodeWhisperer [20]. This expansion aims to determine whether our results generalize across different programming languages and model architectures. Furthermore, we intend to integrate incremental proof certificates [27], which would provide verifiable safety assurance and support compliance with IEC 62443 and NIST SP 800-218 workflows. Lastly, we will also extend RAVE-Code to identify vulnerabilities in AI-generated smart contracts [28].

9 Conclusions

This paper has presented RAVE-Code, a three-layer framework that combines static analysis, composite risk scoring, and CWE-conditioned bounded model checking for AI-generated code security. On FormAI, RAVE-Code achieves a detection rate of 86.0% (F_1 = 0.909), a gain of 12.2 percentage points over uniform ESBMC, and detects 41 vulnerabilities that uniform ESBMC overlooks entirely. Because FormAI's ground-truth labels were generated by ESBMC, its results show consistency between two ESBMC runs. On the Juliet Test Suite, which provides an independent ground truth, RAVE-Code achieves 57.3% (F_1 = 0.614), a gain of 16.5 percentage points over uniform ESBMC. It also detects 104 vulnerabilities that uniform ESBMC completely overlooks. The ablation study indicates that the CWE-conditioned routing is the primary driver of this improvement. The best uniform configuration using all of RAVE-Code's flags achieves only 38.5% on Juliet, which is 18.8 percentage points lower than the routed approach.

The core empirical finding is that applying the right verification configuration to the right weakness class, guided by a composite risk score, produces substantially better detection than any uniform strategy. Applying RAVE-Code's flags uniformly across all CWE classes results in lower performance than the standard baseline, as cross-CWE interference outweighs per-CWE improvements. Risk stratification mitigates this interference by ensuring that each configuration is applied exclusively to its intended class.

Acknowledgement: The authors are thankful to the Deanship of Graduate Studies and Scientific Research at Najran University for funding this work under the Consortium Funding Program grant code (NU/CPL/SERC/14/1918-1).

Funding Statement: This research was funded by the Deanship of Graduate Studies and Scientific Research at Najran University under the Consortium Funding Program (grant code NU/CPL/SERC/14/1918-1).

Author Contributions: Conceptualization, Maher Alharby and Ali Alssaiari; methodology, Maher Alharby; software, Maher Alharby; validation, Maher Alharby and Ali Alssaiari; formal analysis, Maher Alharby; investigation, Maher Alharby and Ali Alssaiari; writing original draft, Maher Alharby; review and editing, Ali Alssaiari. All authors reviewed and approved the final version of the manuscript.

Availability of Data and Materials: Source code and results are available from the corresponding author upon request.

Ethics Approval: This research did not involve any studies with human participants or animals performed by the authors.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Pearce H, Ahmad B, Tan B, Dolan-Gavitt B, Karri R. Asleep at the keyboard? Assessing the security of GitHub copilot's code contributions. Commun ACM. 2025;68(2):96–105. doi:10.1145/3610721.
2. Siddiq ML, Santos JCS. 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 (MSR4P&S); 2022 Nov 18; Singapore. New York, NY, USA: ACM. p. 29–33. doi:10.1145/3549035.3561184.
3. Fu Y, Liang P, Tahir A, Li Z, Shahin M, Yu J, et al. Security weaknesses of copilot-generated code in GitHub projects: an empirical study. ACM Trans Softw Eng Methodol. 2025;34(8):1–34. doi:10.1145/3716848.
4. Tóth R, Bisztray T, Erdődi L. LLMs in web development: evaluating LLM-generated PHP code unveiling vulnerabilities and limitations. In: Proceedings of the Computer Safety, Reliability, and Security. SAFECOMP 2024 Workshops; 2024 Sep 17; Florence, Italy. Cham, Switzerland: Springer; 2024. p. 425–37. doi:10.1007/978-3-031-68738-9_34.
5. Negri-Ribalta C, Geraud-Stewart R, Sergeeva A, Lenzini G. A systematic literature review on the impact of AI models on the security of code generation. Front Big Data. 2024;7:1386720. doi:10.3389/fdata.2024.1386720.
6. Cotroneo D, Improta C, Liguori P, Natella R. Vulnerabilities in AI code generators: exploring targeted data poisoning attacks. In: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC); 2024 Apr 15–16; Lisbon, Portugal. New York, NY, USA: ACM; 2024. p. 280–92. doi:10.1145/3643916.3644416.
7. Improta C. Poisoning programs by un-repairing code: security concerns of AI-generated code. In: Proceedings of the 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW); 2023 Oct 9–12; Florence, Italy. Piscataway, NJ, USA: IEEE; 2023. p. 128–31. doi:10.1109/ISSREW60843.2023.00060.
8. Cotroneo D, De Luca R, Liguori P. DeVAIC: a tool for security assessment of AI-generated code. Inf Softw Technol. 2025;177(1):107572. doi:10.1016/j.infsof.2024.107572.
9. Gadelha MR, Menezes RS, Cordeiro LC. ESBMC 6.1: automated test case generation using bounded model checking. Int J Softw Tools Technol Transf. 2021;23(6):857–61. doi:10.1007/s10009-020-00571-2.
10. Cordeiro L, Fischer B, Marques-Silva J. SMT-based bounded model checking for embedded ANSI-C software. IEEE Trans Softw Eng. 2012;38(4):957–74. doi:10.1109/TSE.2011.59.
11. Tony C, Mutas M, Ferreyra NED, Scandariato R. LLMSecEval: a dataset of natural language prompts for security evaluations. In: Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR); 2023 May 15–16; Melbourne, Australia. Piscataway, NJ, USA: IEEE; 2023. p. 588–92. doi:10.1109/MSR59073.2023.00084.
12. National Institute of Standards and Technology. NVD vulnerability metrics—CVSS [Internet]. Gaithersburg, MD, USA: NIST; 2024 [cited 2026 Jan 15]. Available from: https://nvd.nist.gov/vuln-metrics/cvss.
13. Jacobs J, Romanosky S, Adjerid I, Baker W. Exploit prediction scoring system (EPSS). Digit Threat Res Pract. 2021;2(3):1–17. doi:10.1145/3436242.
14. de Moura L, Bjørner N. Z3: an efficient SMT solver. In: Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS); 2008 Mar 29–Apr 6; Budapest, Hungary. Berlin/Heidelberg, Germany: Springer; 2008. p. 337–40. doi:10.1007/978-3-540-78800-3_24.
15. Majdinasab V, Bishop MJ, Rasheed S, Moradidakhel A, Tahir A, Khomh F. Assessing the security of GitHub copilot's generated code—A targeted replication study. In: Proceedings of the 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER); 2024 Mar 12–15; Rovaniemi, Finland. Piscataway, NJ, USA: IEEE; 2024. p. 435–44. doi:10.1109/SANER60148.2024.00051.
16. He J, Vechev M. Large language models for code: security hardening and adversarial testing. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS); 2023 Nov 26–30; Copenhagen, Denmark. New York, NY, USA: ACM; 2023. p. 1865–79. doi:10.1145/3576915.3623175.
17. Khoury R, Avila AR, Brunelle J, Camara BM. How secure is code generated by ChatGPT? In: Proceedings of the 2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC); 2023 Oct 1–4; Honolulu, HI, USA. Piscataway, NJ, USA: IEEE; 2023. p. 2445–51. doi:10.1109/SMC53992.2023.10394237.

18. Sandoval G, Pearce H, Nys T, Karri R, Jana S, Dolan-Gavitt B. Lost at C: a user study on the security implications of large language model code assistants. In: Proceedings of the 32nd USENIX Security Symposium; 2023 Aug 9–11; Anaheim, CA, USA. Berkeley, CA, USA: USENIX Association; 2023. p. 2205–22.
19. Mechri A, Ferrag MA, Debbah M. SecureQwen: leveraging LLMs for vulnerability detection in Python codebases. Comput Secur. 2025;148(3):104151. doi:10.1016/j.cose.2024.104151.
20. Tihanyi N, Bisztray T, Jain R, Ferrag MA, Cordeiro LC, Mavroeidis V. The FormAI dataset: generative AI in software security through the lens of formal verification. In: Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE); 2023 Dec 8; San Francisco, CA, USA. New York, NY, USA: ACM; 2023. p. 33–43. doi:10.1145/3617555.3617874.
21. Schrammel P, Kroening D, Brain M, Martins R, Teige T, Bienmüller T. Incremental bounded model checking for embedded software. Form Asp Comput. 2017;29(5):911–31. doi:10.1007/s00165-017-0419-1.
22. Günther H, Weissenbacher G. Incremental bounded software model checking. In: Proceedings of the 2014 International SPIN Symposium on Model Checking of Software; 2014 Jul 21–23; San Jose, CA, USA. New York, NY, USA: ACM; 2014. p. 40–7. doi:10.1145/2632362.2632374.
23. Wu Y, Deng Q, Zhang W. A strategy to unwind loops in incremental bounded model checking for software. In: Proceedings of the 2023 3rd Guangdong-Hong Kong-Macao Greater Bay Area Artificial Intelligence and Big Data Forum (AIBDF); 2023 Sep 22–24; Guangzhou, China. New York, NY, USA: ACM; 2023. p. 382–7. doi:10.1145/3660395.3660460.
24. Hsu TH, Sánchez C, Sheinvald S, Bonakdarpour B. Efficient loop conditions for bounded model checking hyperproperties. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS); 2023 Apr 22–27; Paris, France. Paris, France. Cham, Switzerland: Springer; 2023. p. 66–84. doi:10.1007/978-3-031-30823-9_4.
25. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. Experimentation in software engineering. Berlin/Heidelberg, Germany: Springer; 2012, 10.1007/978-3-642-29044-2.
26. Boland T, Black PE. Juliet 1.1 C/C++ and Java test suite. Computer. 2012;45(10):88–90. doi:10.1109/MC.2012.345.
27. Fazekas K, Pollitt F, Fleury M, Biere A. Incremental proofs for bounded model checking. In: Proceedings of the 27th Workshop on Methods and Description Languages for Modelling and Verification of Circuits and Systems (MBMV); 2024 Mar 13–14; Kaiserslautern, Germany. Berlin, Germany: VDE Verlag; 2024. p. 133–43.
28. Alrehaili A, Alharby MW. Exploring machine learning techniques for the detection and multi-label classification of smart contract vulnerabilities. J Syst Sci Syst Eng. 2026;39(1):233. doi:10.1007/s11518-025-5720-6.