针对Python中CWE预测的大语言模型与静态应用安全测试工具的自适应层次化评估

Muntasir Adnan开放源代码研究所 科学与技术学院 堪培拉大学堪培拉，澳大利亚
A
dnan.adnan@canberra.edu.au *
通讯作者

Carlos C.N.Kuhn开放源代码研究所 科学与技术学院 堪培拉大学堪培拉，澳大利亚
C
arl
os.NoschangKuhn@canberra.edu.au

arXiv:2601.01320v1 [cs.SE] 4 Jan 2026

摘要—大语言模型已成为软件开发不可或缺的一部分，但其经常生成易受攻击的代码。现有代码漏洞检测基准采用二元分类，缺乏迭代纠错系统所需的CWE级别的特异性，因而难以提供可执行的反馈。我们提出ALPHA（基于层次评估惩罚的自适应学习），这是首个函数级Python基准，使用具备层级感知能力的CWE特定惩罚，同时评估大语言模型和静态应用安全测试工具。ALPHA区分过度泛化、过度具体化和横向错误，反映出它们在诊断效用方面的实际差异。通过评估七个大语言模型和两个静态应用安全测试工具，我们发现大语言模型的表现显著优于静态应用安全测试工具，但在检测到漏洞时，静态应用安全测试工具展现出更高的精确率。更为关键的是，不同模型之间的预测一致性差异极大（协议率为8.26%—81.87%），这对反馈驱动系统具有重要影响。我们进一步提出未来工作的方向：将ALPHA惩罚纳入监督微调，这有望在经过实证验证后，为具备层级感知能力的漏洞检测提供有原则的方法。

索引词——**漏洞检测、大语言模型、静态分析、CWE 分类、层次化评估**

I. 引言

大语言模型（LLM）从根本上改变了软件开发实践。如今，借助 GitHub Copilot、Amazon CodeWhisperer 和 ChatGPT 等工具，代码生成能力已融入主流开发工作流。这一范式转变大幅提升了开发人员的生产力，支持快速原型开发并加快开发周期 [1]。然而，近期研究表明，大语言模型经常生成含有可利用漏洞的代码；根据任务复杂度和模型的不同，40% 至 60% 的生成代码片段中存在安全缺陷 [2], [3]。研究界主要通过两种途径应对这一漏洞挑战：提示工程和监督微调。

本研究由与澳大利亚首都领地政府、未来就业基金—开放源代码研究所（OpenSI）签订的协议（R01553）以及与开放源代码研究所签订的NetApp技术联盟协议（R01657）资助。此外，本研究还获得澳大利亚政府通过教育部国家产业博士项目（项目编号 36337）提供的支持。本文所表达的观点仅代表作者，不一定代表澳大利亚政府或教育部的观点。

提示工程技术已展现出潜力；在 GPT-4o 等模型中，面向安全的提示可将漏洞生成率降低多达 61% [4]。自反思方法则要求模型检测其先前生成的代码中的漏洞并进行修复，在收到专门提示时，可实现 41.9% 至 68.7% 的漏洞修复率 [4],[5]。然而，这些技术在实际应用中存在显著局限。它们不仅需要针对特定漏洞的指令才能取得实质性改进，更令人担忧的是，其表现极不稳定。即使提示修改幅度很小，结果也会出现显著差异；无论在单项研究内部还是不同研究之间，结果都缺乏一致性 [6], [7]。

微调方法已显示出效果提升 [8]–[10], ，但存在若干关键局限，包括灾难性遗忘 [11], 、泛化能力不足 [12] ，以及随着漏洞态势演变而频繁重新训练所需的高昂计算成本 [11], [13]。近期证据表明，采用改进训练方案的当代小语言模型（SLM）能够达到甚至超过较早期、更大型模型的性能 [14],，这或许意味着针对漏洞的专门微调已无必要。

鉴于这些局限，迭代反馈循环已成为一种颇具前景的替代方案，并已展现出提升代码生成中功能正确性的成效 [15]–[17]。这类系统会对生成的代码进行迭代分析，识别具体问题，并提示模型加以修正。然而，一个关键问题仍未得到解答：漏洞检测应由什么工具提供反馈？目前有两种候选方案：传统的静态分析安全测试（SAST）工具，以及基于大语言模型的漏洞检测。当前实践主要采用静态应用安全测试工具来生成反馈 [12], [18],，但这种选择似乎更多出于惯例，而非基于证据。近期的比较研究表明，大语言模型在漏洞检测方面可能优于静态应用安全测试工具 [19], [20], ，但研究结果仍然有限、各有不同，且取决于具体情境。

A. 研究空白与贡献

现有的 SAST 与大语言模型的比较存在一个根本性的方法论局限：大多数采用二元分类（存在漏洞/不存在漏洞）

[20], [21], ，这无法提供迭代修正所需的充分具体性。有效的反馈机制要求精准识别漏洞，具体包括 CWE 类型、位置和缺陷性质 [5]。如果缺乏这种细粒度信息，模型就无法获得可执行且针对特定漏洞的指导，而提示工程研究表明，这类指导对于显著提升安全性必不可少 [4]。

尽管已有一些研究比较静态应用安全测试工具与大语言模型 [20], [21]，但尚无研究提供基准测试，以指导 Python 函数级迭代反馈循环的设计。这种具体性至关重要；函数级分析既符合大语言模型的上下文窗口限制，也能支持细粒度调试 [16]。鉴于 Python 在开发工作流中无处不在，且大语言模型最常用于生成 Python 代码，因此 Python 是最关键的应用场景。本研究直接填补了这一空白。

我们提出首个全面的基准测试，在多个 Python 漏洞数据集上，以函数级 CWE 粒度比较多种大语言模型（包括现代小语言模型）与静态应用安全测试工具。我们的贡献如下：

- 首个函数级 Python 漏洞检测基准测试——**基于层次评估惩罚的自适应学习（ALPHA）**，用于评估大语言模型和静态应用安全测试工具对特定 CWE 的识别能力，并为反馈机制的选择提供实证依据。
- 分析各大语言模型在多次运行中的检测一致性，评估其在迭代纠错系统中的可靠性，因为一致的反馈至关重要。

II. 相关工作

早期提升代码安全性的工作主要聚焦于微调大语言模型，使其生成安全代码 [9], [10]。研究采用了漏洞令牌分类 [9] 和对抗训练 [10]，报告准确率提升了 30–32%。然而，微调后的模型往往泛化能力较差且容易过拟合 [12]；已有案例记录了“逆转诅咒”现象，即微调后的模型会失去知识对称性 [13]。灾难性遗忘 [11] 会导致先前学会的任务能力突然丧失，而微调所需的高昂计算成本 [22], [23]，使得为跟踪新出现的漏洞而频繁重新训练在经济上难以实现。值得注意的是，一项针对漏洞增强提示框架 DLAP [24] 的研究发现，无论在漏洞检测准确率还是成本效益方面，提示工程都优于微调，因此我们着手研究基于提示的方法。

提示工程提供了一种计算效率更高的替代方案，其中基于反思的方法展现出尤为可观的前景。近期一项研究 [4] 报告称，使用 GPT 模型并采用递归批评与改进（RCI），性能提升了 61%；该方法让模型反复批评并完善自身输出。然而，基于提示的方法表现出令人担忧的不稳定性。基于角色的人设提示在不同研究中得出的结果差异很大。Kong 和 Zhao [6] 报告准确率提升了 60%，Bruni 等人 [4] 报告使用 GPT 模型后性能提升了 43%，而 Zheng 等人 [7]发现，在四种大语言模型系列中，人设提示的效果可能是随机的，也可能产生负面影响。这种敏感性甚至延伸至措辞变化 [4]，削弱了方法的可靠性。

鉴于微调不切实际且提示工程不稳定，迭代反馈循环已成为最有前景的方法，在 Python 代码生成方面展现出 15–35% 的提升 [15]–[17]。反馈循环将提示工程的高效性与结构化的外部验证和反馈相结合。这引出了一个关键问题：应使用哪种工具为漏洞修复生成反馈？值得注意的是，现有用于代码漏洞检测的迭代反馈循环实现，无一例外都依赖静态应用安全测试工具，并主要使用大语言模型进行结果解读和代码修复 [12], [18], [25], [26]。此外，静态应用安全测试工具经常被用作评估数据集的真实值 [12], [26]，由此形成方法论循环论证，使得这些工具是否适用于漏洞检测这一问题始终未得到检验。因此，反馈循环中对静态应用安全测试工具的偏好似乎源于经验做法，而非系统性验证。

静态应用安全测试工具与大语言模型在漏洞检测方面的直接比较，结果不一且取决于具体情境。针对较小代码库中的 C 程序，Castle [19]报告称大语言模型表现更优（GPT-3.5-mini：977/1250，最佳静态工具：661/1250）；而 Gong 等人 [27] 发现，GPT-4 的检测率达到 76%，误报率仅为 3.1%，但仍建议使用静态工具而非大语言模型。以 Python 为重点的研究也呈现出类似的多样化结果：Li 等人 [28] 使用 DeepSeek R1 达到了 67% 的准确率；Zhou 等人 [20] 报告称，在仓库级分析中，大语言模型得分高达 100，而静态应用程序安全测试的得分为 44.4；而 PyVul [21] 发现，在提交级漏洞检测中，CodeQL 的准确率仅为 10.8%，相比之下，大语言模型在函数级检测中的准确率为 49.5%–58.5%。

大多数研究采用二元分类（有漏洞/无漏洞）[20], [21], [28]，而非在 CWE 层面进行识别；评估对象是仓库或提交，而非函数；更关键的是，没有研究评估多次运行中的检测稳定性。对于迭代反馈系统而言，这些局限是致命的：二元分类无法指导针对性修复，仓库级分析超出大语言模型上下文窗口的限制，而不稳定的反馈会削弱迭代改进效果。

目前尚无研究直接比较静态应用程序安全测试和大语言模型作为反馈工具，在 CWE 级粒度下检测函数级 Python 漏洞的能力——而这正是实用迭代反馈系统的具体要求。鉴于 Python 在大语言模型代码生成中的主导地位，以及必须进行函数级分析的上下文窗口限制，确定哪种工具能够提供更可靠、更具可操作性的反馈，仍是一个悬而未决的问题，并具有直接的实践意义。

III. 方法

A. 数据集

我们在两个特征各异的 Python 漏洞数据集上进行评估。数据集的选择受以下条件限制：

三个要求：函数级粒度、明确的 CWE 真实标签，以及人工标注标签而非基于 SAST 的真实标签。SecurityEval [29] 包含 121个易受攻击函数，分布于 CWE 层级结构的不同层级：73 个基础弱点、26 个类别弱点、16 个变体弱点和 3 个支柱弱点。SVEN [10] 包含 342 个 Python样本（从原始的 720 个多语言数据集中筛选而来），所有样本均标注为基础弱点。抽象层级分布上的这一差异十分显著：SecurityEval 的层级多样性带来了更具挑战性的检测任务，且具体性要求各不相同；而 SVEN 统一的基础层级标签则提供了更精确的预测目标。两个数据集均提供适用于评估层级感知检测的 CWE真实标签。

*B. CWE*图构建

CWE 提供了软件弱点的层级分类体系。我们根据 MITRE CWE 数据库构建了一个有向图 $G = (V, E)$，其中 -

- 节点（V）：具体的 CWE 弱点类型包括：支柱弱点、类弱点、基础弱点、变体弱点、复合弱点和链式弱点。
- 边（E）：CWE 层级结构中的父子关系。

我们排除了两种 CWE 类型——视图和类别，它们在 CWE 数据库中充当组织性构造。这总共移除了 49 个节点和 480 条边，同时没有损失任何泛化能力。视图呈现关于弱点的不同视角（例如，按开发阶段或受众划分），而类别则根据共同特征对弱点进行分组（例如，身份验证错误）。与弱点类型（支柱、类、基础、变体）不同，这些构造并不描述具体、可检测的漏洞，而是作为层级分类体系中的导航辅助工具。最终的图包含 944 个节点和 1153 条边，而完整图（含所有 CWE）包含 993 个节点和 1633 条边。

C. ALPHA：自适应惩罚指标

对于给定代码样本，其真实CWE为 c_{true}，预测的 CWE为 c_{pred}，我们将惩罚分数定义为 P:

$$P(c_{\text{pred}}, c_{\text{true}}) = d(c_{\text{pred}}, c_{\text{true}}) \times \alpha(c_{\text{pred}}, c_{\text{true}})$$

其中，距离分量 $d(c_{pred}, c_{true})$ 是图中节点之间的最短路径长度 G，使用 Dijkstra算法 [30]在无向表示中计算得出。它反映了预测结果在弱点分类体系中与 c_{true} 相差多“远”。方向乘数 $\alpha(c_{pred}, c_{true})$ 编码了错误的语义类型——

$$\alpha(c_{\text{pred}}, c_{\text{true}}) = \begin{cases} \alpha_{\text{up}} & \text{if } c_{\text{pred}} \text{ is an ancestor of } c_{\text{true}} \\ & (generalising) \\ \alpha_{\text{down}} & \text{if } c_{\text{pred}} \text{ is a descendant of } c_{\text{true}} \\ & (over-specifying) \\ \alpha_{\text{lateral}} & \text{otherwise (lateral error)} \end{cases}$$

方向判定使用有向图：如果存在从 c_{pred} 到 c_{pred} 的有向路径，则 c_{pred} 是 c_{true} 的祖先节点；反之则为后代节点；否则为横向关系。虽然最短路径距离能够反映误差幅度，但它将距离相同的所有误差视为同等严重。这无法体现实际效用中的重要不对称性。我们区分三种误差类型：

- **泛化错误**（$\alpha_{up} = 2.0$）：当预测结果沿层级结构向上移动时，关键诊断信息会丢失。例如，当真实弱点为 CWE- 89（SQL注入，基础类）时，预测为 CWE-707（不当中和，支柱）只能提供极少的可执行指导。我们赋予其最高的固定惩罚 2.0，以体现这种效用的严重损失。

- **横向错误**（$\alpha_{lateral} = 1.8$）：不同分支中的预测结果表明存在概念性误解，但如果弱点共享父类或属于相关领域，仍可能保留一定的诊断价值。我们赋予其适中的固定惩罚 1.8，低于泛化错误但高于过度具体化，反映其在弱点大类别上的部分正确性。

- **过度具体化错误**（α_{down}，**自适应**）：当预测沿层级结构向下移动时，错误会引入缺乏依据的具体性，同时仍保留正确的整体理解。关键在于，错误的严重性取决于真实值后代层级结构中可供具体化的“空间”。我们定义：

$$\alpha_{down}(c_{true}) = \alpha_{max} - (\alpha_{max} - \alpha_{min}) \times \frac{\delta(c_{true})}{\Delta(type(c_{true}))}$$

其中，$\delta(c_{true})$ 是从 c_{true} 到家族子树（深度）中任意后代叶节点的最大距离，$\Delta(type(c_{true}))$ 表示该类型所有 CWE 中的最大 δ。这里，$\alpha_{max} = \alpha_{lateral} = 1.8$ 和 $\alpha_{min} = 1.1$。

采用这种自适应方法的理由如下：

- 叶节点（$\delta = 0$）：当 c_{true} 是没有子节点的叶节点时，预测得更具体在语义上等同于横向错误。该预测对并不存在的具体性作出了缺乏依据的分类学假设。因此，$\alpha_{max} = \alpha_{lateral} = 1.8$。
- 深层子树（δ 较高）：当 c_{true} 是拥有许多后代节点的类或支柱时，存在很大的“具体化空间”。过度具体化属于轻微错误：预测在概念上是正确的，但假定了某种具体实例化。因此，$\alpha_{min} = 1.1$（超过 1.0，以确保惩罚有效）。

过度具体化惩罚取决于真实值上下文；相同的距离在 c_{true} 允许进一步细化与否的情况下，其严重性不同。与过度具体化不同，无论真实值的特征如何，泛化始终会丢失信息。其严重性始终较高。因此，固定的高惩罚能够恰当地体现这种一致的损害。参数值（$\alpha_{up} = 2.0$、$\alpha_{\text{lateral}} = 1.8$、$\alpha_{\text{min}} = 1.1$）维持惩罚层级 $\alpha_{\text{up}} > \alpha_{\text{lateral}} > \alpha_{\text{down}}$，同时确保距离和方向都能对最终得分产生有意义的影响。

D. 边缘情况处理

图外预测中，$c_{pred} \notin V$ 表明c_{pred} 可能是一个 类别、视图或无效的 CWE。对于此类情况，我们赋予最大惩罚乘数 $\alpha_{oog} = 2.5$，并根据图结构 $d_{oog} = \lceil \bar{d} \rceil$ 设定距离，其中 $\bar{d}$是 G中所有连通的 CWE 节点对之间的均最短路径距离：

$$\bar{d} = \frac{1}{|C|} \sum_{(u,v) \in C} d_G(u, v)$$

其中，C 是 G中所有 CWE 节点连通对的集合，¯的取值为 $|C| = 445$，096，由此得到 $d = 6.26$，进而得出 $d_{oog} = \lceil 6.26 \rceil = 7$。这一选择使图外错误的严重性高于典型的图内错误，确保对幻觉式或过于模糊的预测施加高于平均图内错误的基础惩罚。结合 $\alpha_{oog} = 2.5$后，惩罚为 $P = 7 \times 2.5 = 17.5$，对超出漏洞层级结构的预测施以重罚。

如果图中 c_{pred} 和 c_{true} 之间不存在路径（位于不同的连通分量中），我们将施加相同的图外惩罚（d_{oog}，α_{oog}），这表示弱点类型完全无关。

当某个 CWE 通过不同的关系路径拥有多个父节点时，我们使用最短路径计算距离。方向取决于指定方向上是否存在任何有向路径。

E. 评估协议

对于每个测试样本 i，其真实值为 c^i_{true}，预测值为 c^i_{pred}：

- 计算单个惩罚

$$P_i = P(c^i_{pred}, c^i_{true})$$

- 在整个数据集上进行汇总 -

$$ALPHA = \frac{1}{N} \sum_{i=1}^{N} P_i$$

平均惩罚越低，表示性能越好。

IV. 实验结果

所有大语言模型实验均在 NVIDIA A100-PCIE-40GB 图形处理器上进行了 3 次。我们评估了七种开源模型：Qwen2.5-Coder-32B [31]，Qwen2.5-Coder- 7B [31]，Mistral-7B [32],Llama3.1-8B [33]，Phi4-14B [34]，DeepSeek-Coder-6.7B [35]，和Devstral-24B [36]。完整的提示策略和模板详见附录 A。

静态应用程序安全测试实验仅进行了一次，因为这些工具会生成确定性输出，这与随机性大语言模型推理不同。我们评估了两种静态分析工具：CodeQL [37](v2.23.0) 和 Semgrep [38](v1.136.0)。对于 CodeQL，我们使用了 Python security-extended 和 security-experimental 查询套件，两者共通过 101 条内置查询覆盖 123 个 CWE [21]。CodeQL 通过将源代码转换为可查询数据库，同时保留程序语义（包括数据和控制流）来执行语义分析。对于 Semgrep，我们采用了四个规则集：python（通用 Python 规则）、安全审计、OWASP十大安全风险和 Bandit（39 条模式匹配规则，覆盖 17 个 CWE） [20]。两种工具均以结构化格式输出结果（CodeQL 使用 SARIF，Semgrep 使用 JSON），以便自动解析。

表 I SecurityEval 和 SVEN Python 漏洞检测的 ALPHA 分数。分数越低表示性能越好。大语言模型结果显示了 3 次运行的均值 ± 标准差。

模型	SVEN	SecurityEval
Qwen2.5-Coder-32B	574.7 ± 42.4	617.8 ± 18.5
Mistral-7B	923.3 ± 77.0	1156.9 ± 32.0
Qwen2.5-Coder-7B	753.2 ± 39.5	921.5 ± 15.1
Llama3.1-8B	813.5 ± 51.0	856.5 ± 42.7
Phi4-14B	646.4 ± 47.4	797.4 ± 13.0
DeepSeek-Coder-6.7B	2557.0 ± 194.3	1228.5 ± 60.9
Devstral-24B	695.1 ± 71.2	578.5 ± 27.4
静态应用安全测试工具		
CodeQL（置信度）	5981.9	1261.0
CodeQL（任意匹配）	5981.9	1212.9
Semgrep（置信度）	3663.5	1307.6
Semgrep（任意匹配）	3562.7	1307.6

所有实验均使用 Ollama 的默认参数（温度=0.8），以反映真实的部署条件。这会引入随机性，我们通过第四节B部分的一致性分析对其进行了明确测量

A. 贡献1

表 I 展示了七种大语言模型和两种静态应用安全测试工具在 SVEN 和 SecurityEval 数据集上的 ALPHA 分数，分数越低表示 CWE 特定检测性能越好。对于大语言模型，我们报告三次独立运行的均值和标准差，以考虑随机波动。模型排名在不同数据集上表现出很强的一致性（大语言模型的斯皮尔曼 $\rho = 0.86, p = 0.014$；纳入静态应用安全测试工具时为 $\rho = 0.92, p$ <0.001），这表明无论具体漏洞样本如何，ALPHA 都能提供稳定的评估。Qwen2.5-Coder-32B 和 Devstral-24B 在两个数据集上均取得最低惩罚分，而 DeepSeek-Coder-6.7B 的惩罚分始终显著偏高。

静态应用安全测试工具的 ALPHA 分数显著高于大多数大语言模型，这主要是由于检测覆盖率较低。对于超过 50% 的样本，CodeQL 和 Semgrep 均未作出预测。在作出预测时，CodeQL 的精确率更高（72.4% 的预测包含真实CWE），而 Semgrep 为 50.0%，凸显了覆盖率与精确率之间的权衡。

对于静态应用安全测试工具，我们报告两种 ALPHA 分数，分别对应不同的 CWE 选择策略，因为这类工具通常会针对每个样本报告多个潜在漏洞。置信度策略选择置信度评级最高（或最先报告）的 CWE；任意匹配策略则在真实值出现在输出的任何位置时给予认可。这两种分数差异很小，表明当静态应用安全测试工具报告多个 CWE 时，真实值

很少出现在其中。它们的错误源于漏检，而非优先级排序不当。

B. 贡献 *2*

对于迭代纠错系统而言，稳定的反馈至关重要。如果模型对相同代码给出不同的漏洞评估，就无法提供可靠的反馈信号。我们在相同条件下对每个大语言模型在两个数据集上分别运行三次，以衡量各次运行结果的一致程度，从而评估检测一致性。表 II 和表 III 展示了两项一致性指标：完全一致 （三次运行均给出相同的 CWE 预测）和多数一致 （至少两次运行结果一致）。

不同模型的一致性差异显著。完全一致率从 81.87%（Qwen2.5-Coder-32B 在 SVEN 上）到仅 8.26%（ DeepSeek-Coder-6.7B 在 SecurityEval 上）不等，这对反馈机制的选择有重要影响。一致性较低的模型会在反馈中引入噪声，因此无论平均准确率如何，都不适用于迭代反馈循环。所有模型在 SecurityEval 上的一致性均低于在 SVEN 上的表现。这反映了 SecurityEval 的层次多样性（第 III-A 节详述的支柱、类、基础级和变体弱点）与 SVEN 统一的基础级标签之间的差异。对于较小的模型，这种影响尤为明显。Qwen2.5-Coder-7B 在两个数据集之间的完全一致率从 71.64% 降至 21.49%。模型排名在不同数据集间仍高度稳定（斯皮尔曼 $\rho = 0.96, p <$0。001），表明一致性是模型的内在属性。Qwen2.5-Coder-32B 和 Devstral-24B 始终排名最可靠，而 DeepSeek-Coder-6.7B 和 Llama3.1-8B 的变异性最高。完全一致与多数一致之间的差距，在 SVEN 上平均为 24.1 个百分点，在 SecurityEval 上平均为 36.1 个百分点，这表明模型虽然常能达成两次运行结果一致，但要实现完全一致仍颇具挑战。

V. 讨论

据我们所知，ALPHA 是首个面向 Python 的基准，使用基于分类体系的惩罚，在 CWE 层级对静态应用安全测试工具和大语言模型的性能特征进行了实证区分。尽管大语言模型取得了明显更高的 ALPHA 分数，静态应用安全测试工具却呈现出互补的特征：覆盖率较低，但在产生检测结果时具有相对较高的精确率。

表 II SVEN 数据集三次运行中的大语言模型预测

模型	完全一致（%）	多数一致率（%）
deepseek-coder:6.7b	23.98	72.51
devstral:24b	73.39	91.81
llama3.1:8b	50.88	75.73
mistral:7b	63.74	86.84
phi4:14b	66.96	91.23
qwen2.5-coder:32b	81.87	92.98
qwen2.5-coder:7b	71.64	89.77

表 III SecurityEval 数据集三轮大语言模型预测

模型	完全一致率（%）	多数一致率（%）
deepseek-coder:6.7b	8.26	49.59
devstral:24b	52.07	85.95
llama3.1:8b	16.53	55.37
mistral:7b	20.66	56.20
phi4:14b	34.71	71.90
qwen2.5-coder:32b	61.98	86.78
qwen2.5-coder:7b	21.49	62.81

同时报告 ALPHA 和普通准确率揭示了这一差异，并促成了一种混合方法：在可用时优先采用静态应用安全测试工具的输出，而让大语言模型在迭代循环中充当备用或细化机制。

观察到的标准差较大，尤其是 DeepSeek-Coder（在 SVEN 上的 ± 194.3 ），这既反映了真实的模型随机性，也与温度为 =0.8 的采样设置有关。因此，应将一致性视为与原始性能同等重要的首要评估标准。

值得注意的是，所有模型在 Securi- tyEval 上的一致性都低于在 SVEN 上的表现（表 II 和表 III）。第 III-A 节详述的数据集特征对此给出了具体解释，也验证了 ALPHA 的设计选择：当真实值涵盖多个抽象层级时，相较于过度具体化，对过度泛化施加更重的惩罚；此时，模型必须应对不同的具体性要求。

由于样本量（121 对 342）和难度分布不同，不宜直接比较不同数据集上的 ALPHA 分数。不过，模型排名保持稳定（斯皮尔曼 $\rho = 0.86$），这表明即使绝对分数有所不同，相对性能仍与数据集无关。

温度设置（0.8）引入了有意的随机性，以反映大语言模型实际部署中的情况；在此场景下，由于确定性解码（ temperature=0）会降低输出质量，通常并不实用。因此，我们的一致性分析是在现实条件下测量方差，而不是在人为限制的设置下进行测量。

ALPHA 的层次结构天然适合作为训练信号。不同于二元分类损失函数，ALPHA 提供连续的、方向感知的惩罚，可引导模型形成层次感知表示。我们设想通过双头架构，利用 ALPHA 惩罚增强监督微调，不过这一方法仍有待实证验证，是未来的重要研究方向（见第 VI 节）。

ALPHA 惩罚参数（$\alpha_{up} = 2.0$，$\alpha_{lateral}$ =1。8，$\alpha_{min} = 1.1$）在构造时即被定义为确保层次排序，$\alpha_{up} > \alpha_{lateral} > \alpha_{down}$ 。其他参数化方式可能产生不同的绝对分数，同时保持相对排名不变。

为促进可复现性，所有代码、数据和实验配置将在发表后发布。

VI. 未来工作

ALPHA 的一个关键设计目标是能够直接作为改进模型的训练信号。不同于二元准确率指标，ALPHA 提供连续的、方向感知惩罚，用于编码 CWE 预测之间的语义关系。我们提出使用 ALPHA 惩罚增强监督微调，以引导模型形成具备层级结构感知能力的表示。

我们提出一种扩展的 Transformer 架构，其中分类头与生成式语言建模头并行运行。给定输入序列 x =(x_1，…，x_n)，Transformer 会生成隐藏状态 $\mathbf{H}$ = $[h_1, ..., h_n] \in \mathbb{R}^{n \times d_{hidden}}$ 其中 d_{hidden} 是大语言模型的隐藏维度。分类头通过均值池化 $\bar{h} = \frac{1}{n}\sum_{i=1}^{n} h_i$ 聚合$\mathbf{H}$，并通过单个线性层或多层感知机（MLP）将其投影到 CWE 标签空间：$z_{cls} = \mathbf{W}_{cls} \times \bar{h} + \mathbf{b}_{cls} \in \mathbb{R}^{|V|}$其中$\mathbf{W}_{cls}$ 和$\mathbf{b}_{cls}$ 分别是权重矩阵和偏置向量，V表示所有 CWE 的集合；由此得到 CWE 类别上的概率分布$p_\theta(v|x) = softmax(z_{cls})$。

训练目标将文本生成的标准交叉熵 $\mathcal{L}_{CE}(\theta)$ 与预期惩罚损失相结合，后者直接纳入 ALPHA 的层次结构：

$$\mathcal{L}_{ALPHA}(\theta) = \sum_{v \in V} p_\theta(v|x) \cdot \hat{P}(v, c_{true})$$

其中，$\hat{P}(v, c_{true}) = \frac{P(v, c_{true})}{P_{max}}$ 是经归一化的 ALPHA 惩罚，其中 $P_{max} = d_{max} \times \alpha_{oog}$，$d_{max}$ 是 G的直径。组合损失为-

$$\mathcal{L}_{total} = \mathcal{L}_{CE}(\theta) + \lambda \cdot \mathcal{L}_{ALPHA}(\theta)$$

该公式完全可微，两个目标的梯度都通过共享的 transformer 主干网络进行传递。预期的惩罚公式根据每种潜在误分类的概率和层级严重性对其进行加权，而 ALPHA 的方向感知惩罚自然地塑造了$\hat{}$损失景观。归一化确保 $P \in [0, 1]$，从而提供数值稳定性，并支持超参数 λ 调优。

这种非对称惩罚结构从理论上将导致：

1) **具体性偏好**：当在某个通用 CWE 及其具体子类之间存在不确定性时，模型会倾向于预测更具体的变体。2) **层级结构一致性**：仍处于 CWE 层级结构正确分支内的预测，其所受惩罚低于跨越到无关分支的预测，从而鼓励语义一致的错误。3) **置信度校准**：连续的惩罚信号使模型能够学习细致的漏洞关系，而不是将所有误分类一视同仁。

未来的工作将通过在更大规模的数据集上对该框架进行实证验证，并衡量其在已部署系统中的下游安全性改进。

此外，未来的工作将着重于在端到端迭代修复流水线中部署表现最佳的大语言模型和混合式 SAST–大语言模型系统，以直接衡量修复成效。我们还将研究将 ALPHA 惩罚用作训练损失函数的做法，并更系统地探讨提示工程和解码约束。这些方向共同旨在将 ALPHA 的评估洞见转化为实用且可部署的漏洞修复系统。我们专注于开源大语言模型，正是为了实现这一目标。第 VI 节提出的 ALPHA 引导训练框架要求模型权重可访问，以便进行微调——而这只有开源模型才能做到。通过仅评估后续可利用 ALPHA 惩罚进行训练的模型，我们建立了从基准测试评估到针对性改进的完整路径。此外，开源模型便于在迭代反馈循环中以较低成本部署，无需支付逐次查询的 API 成本，从而确保我们的研究成果能够转化为经济可行、可用于生产环境的漏洞检测系统。

VII. 局限性

我们明确要求大语言模型仅生成单个CWE预测，以便为迭代纠正提供清晰且可执行的信号；在反馈循环设置中，这一点至关重要，因为多标签输出会削弱纠正信号。尽管大多数模型都遵守了这一要求，但仍有少数模型生成了多个CWE。人工检查表明，这些情况反映的是迭代式细化，而非独立检测；因此，我们将最终报告的CWE选作模型隐含的最佳估计。

这一限制造成了与静态应用安全测试工具之间的不对称，因为后者天然会生成多个预测，且无法配置为单输出模式。对大语言模型多标签输出的初步分析显示，其误报率显著更高，因此不适合反馈系统。然而，这种不对称并未实质性地使静态应用安全测试工具获益：基于置信度的选择策略与任意匹配策略之间的差异极小（表I），这表明额外的预测很少包含真实CWE。因此，这两种方法都是在最具实际意义的运行模式下对工具进行评估。

我们的评估聚焦于使用两个数据集（SecurityEval：121 个样本，SVEN：342 个样本）进行 Python 函数级漏洞检测。研究结果未必适用于其他语言或粒度（例如仓库级）。

双头架构（第 VI 节）为将 ALPHA 纳入监督微调提供了理论上合理的途径，但缺乏实证验证。其实际有效性，包括收敛行为、计算开销以及下游性能提升，仍需通过实验评估。

VIII. 结论

本文介绍了 ALPHA，这是一种能够感知分类体系的基准测试，用于评估 Python 代码中的 CWE 级漏洞识别能力。借助 CWE 图的层次结构和方向敏感惩罚，ALPHA 不仅能衡量是否检测到漏洞，还能根据可操作的具体程度，衡量预测与真实值的接近程度。这使评估能够更准确地反映

现实世界效用，而非仅依赖扁平的准确率指标。我们的实证结果表明，在 ALPHA 的评估下，大语言模型的表现显著优于传统静态应用安全测试工具，尤其是在识别细粒度 CWE 变体方面。同时，静态应用安全测试工具在检出漏洞时表现出较高的精确率，这凸显了两者之间互补的权衡，也说明了混合系统设计的必要性。我们进一步证明，要求大语言模型仅输出单个 CWE 预测，并分析多次运行结果的一致性，是反馈驱动修复流程中的关键考量。除评估之外，ALPHA 还为未来的优化提供了有原则的基础。该基准揭示了明确的信号，可用于指导迭代式漏洞纠正中的损失函数和奖励设计，同时也展现了模型不稳定性和分类体系依赖性等实际局限。总而言之，本研究确立了 ALPHA 的双重价值：既是诊断基准测试，也是迈向稳健混合型漏洞修复系统的基石，这类系统能够整合大语言模型与静态分析的优势。

参考文献

[1] M. A. Haque，“大语言模型：软件工程师的游戏规则改变者？”*BenchCouncil Transactionson Benchmarks, Standards and Evaluations*，第 5 卷，第 1 期，第 100204 页，2025年。 [在线]。链接：https://www.sciencedirect.com/science/article/pii/S2772485925000171

[2] H. Pearce、B. Ahmad、B. Tan、B. Dolan-Gavitt 和 R. Karri，“键盘前打盹？评估 GitHub Copilot 代码贡献的安全性”，*Commun. ACM*，第 68 卷，第 2 期，第 96–105 页，2025年1月。 [ ] 10 1145 3610721 在线。链接：https://doi.org/。

[3] N. Tihanyi、T. Bisztray、M. A. Ferrag、R. Jain 和 L. C. Cordeiro，“AI 生成的代码有多安全：对大语言模型的大规模比较”，*Empirical Software Engineering*，第 30 卷，第 2 期，第 47 页，2025年。[4] M. Bruni、F. Gabrielli、M. Ghafari 和 M. Kropp，“基于 GPT 模型的安全代码生成提示工程技术基准测试”，载于*2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*。IEEE，2025年，第 93–103 页。[5] R. Kamoi、Y. Zhang、N. Zhang、J. Han 和 R. Zhang，“大语言模型何时才能真正纠正自身错误？关于大语言模型自我纠正的批判性综述”，*Transactions of the Association for Computational Linguistics*，第 12 卷，第 1417–1440 页，2024年。 [在线]。链接：https://aclanthology.org/2024.tacl-1.78/ [6] A. Kong、S. Zhao、H. Chen、Q. Li、Y. Qin、R. Sun、X. Zhou、E. Wang 和 X. Dong，“通过角色扮演提示提升零样本推理能力”，载于*Proceedings ofthe 2024ConferenceoftheNorthAmerican Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*，K. Duh、H. Gomez 和 S. Bethard 编。墨西哥城，墨西哥：计算语言学协会，2024年6月，第 4099–4113 页。 [在线]。链接：https://aclanthology.org/2024.naacl-long.228/ [7] M. Zheng、J. Pei、L. Logeswaran、M. Lee 和 D. Jurgens，“当‘得力助手’并不真正得力：系统提示中的角色设定并不能提升大语言模型的表现”，载于*Findings of the Association for Computational Linguistics: EMNLP 2024*，2024年，第 15 126–15 154 页。[8] M. A. H. Bappy、H. A. Mustafa、P. Saha 和 R. Salehat，“未发表，案例研究：微调小型语言模型，以准确、私密地检测 Python 代码中的 CWE”，*arXiv preprint arXiv:2504.16584*，2025年。[9] J. He、M. Vero、G. Krasnopolska 和 M. Vechev，“面向安全代码生成的指令微调”，载于*Proceedings of the 41st International Conference on MachineLearning*，系列：ICML’24。JMLR.org，2024年。[10] J. He 和 M. Vechev，“面向代码的大语言模型：安全强化与对抗性测试”，载于*Proceedings of the 2023 ACMSIGSACConference onComputer andCommunications Security*，2023年，第 1865–1879 页。[11] J. Kirkpatrick、R. Pascanu、N. Rabinowitz、J. Veness、G. Desjardins、A. A. Rusu、K. Milan、J. Quan、T. Ramalho、A. Grabska-Barwinska 等，“克服神经网络中的灾难性遗忘”，《美国国家科学院院刊》，第 114卷，第13期，第3521–3526页，2017年。[12] H. Yan、S. S. Vaidya、X. Zhang 和 Z. Yao，“未发表，指导人工智能修复自身缺陷：关于大语言模型驱动的安全代码生成的实证研究”，arXiv预印本 arXiv:2506.23034，2025年。[13] L. Berglund、M. Tong、M. Kaufmann、M. Balesni、A. C. Stickland、T. Korbak 和 O. Evans，“逆转诅咒：在‘a is b’上训练的大语言模型无法学会‘b is a’”，载于《第十二届国际学习表征会议》，2024年。 [在线]。网址：https://openreview.net/forum?id=GPKTIktA0k [14] P. Belcak、G. Heinrich、S. Diao、Y. Fu、X. Dong、S. Muralidharan、Y. C. Lin 和 P. Molchanov，“未发表，小型语言模型是智能体人工智能的未来”，2025年。 [在线]。网址：https://arxiv.org/abs/2506 02153 。[15] M. A. Islam、M. E. Ali 和 M. R. Parvez，“MapCoder：面向竞赛问题求解的多智能体代码生成”，载于《计算语言学协会第62届年会论文集（第1卷：长篇论文）》，L.-W. Ku、A. Martins 和 V. Srikumar 编。泰国曼谷：计算语言学协会，2024年8月，第 4912–4944页。 [在线]。网址：https: //aclanthology.org/2024.acl-long.269/ [16] M. Adnan、Z. Xu 和 C. C. Kuhn，“未发表，大语言模型引导的代码生成自我调试”，arXiv预印本 arXiv:2502.02928，2025年。[17] M. Adnan 和 C. C. Kuhn，“衡量并缓解代码语言模型中调试有效性衰减”，《科学报告》15，2025年。[18] S. Blyth、S. A. Licorish、C. Treude 和 M. Wagner，“静态分析作为反馈循环：提升大语言模型生成代码的正确性”，载于《2025年IEEE国际源代码分析与操作会议（SCAM）》，2025年，第100–109页。[19] R. A. Dubniczky、K. Z. Horv´at、T. Bisztray、M. A. Ferrag、L. C. Cordeiro 和 N. Tihanyi，“Castle：用于静态代码分析器和大语言模型CWE检测的基准测试数据集”，载于《国际软件工程理论方面研讨会》。施普林格，2025年，第253–272页。[20] X. Zhou、D.-M. Tran、T. Le-Cong、T. Zhang、I. C. Irsan、J. Sumarlin、B. Le 和 D. Lo，“未发表，用于仓库级漏洞检测的静态应用安全测试工具与大语言模型比较”，arXiv预印本 arXiv:2407.16235，2024年。[21] H. Quan、J. Wang、X. Li、T. Y. Zhuo、X. Chen 和 X. Du，“未发表，关于Python软件包中的漏洞及其检测的实证研究”，arXiv预印本 arXiv:2509.04260，2025年。[22] N. Ding、Y. Qin、G. Yang、F. Wei、Z. Yang、Y. Su、S. Hu、Y. Chen、C.-M. Chan、W. Chen、J. Yi、W. Zhao、X. Wang、Z. Liu、H.-T. Zheng、J. Chen、Y. Liu、J. Tang、J. Li 和 M. Sun，“大规模预训练语言模型的参数高效微调”。《自然·机器智能》5，第220—235页，2023年。[23] M. Adnan、B. Gamage、Z. Xu、D. Herath 和 C. C. Kuhn，“释放人工认知：整合多个人工智能系统”，《澳大拉西亚信息系统会议》，2024年。[24] Y. Yang、X. Zhou、R. Mao、J. Xu、L. Yang、Y. Zhang、H. Shen 和 H. Zhang，“DLAP：用于软件漏洞检测的深度学习增强大语言模型提示框架”，《系统与软件杂志》，第219卷，文章编号112234，2025年。 [在线]。网址：https://www.sciencedirect.com/science/article/pii/S0164121224002784 [25] K. Alrashedy、A. Aljasser、P. Tambwekar 和 M. Gombolay，“未发表，大语言模型能否修补安全问题？”，arXiv预印本 arXiv:2312.00024，2023年。[26] J. Gajjar、K. Subramaniakuppusamy、R. Puthal 和 K. Ranaware，“未发表，SecureFixAgent：用于Python静态漏洞自动修复的混合型大语言模型智能体”，arXiv预印本 arXiv:2509.16275，2025年。[27] J. Gong、N. Duan、Z. Tao、Z. Gong、Y. Yuan 和 M. Huang，“未发表，大语言模型作为端到端安全代码生成器的表现如何？”，arXiv预印本 arXiv:2408.10495，2024年。[28] Y. Li、X. Li、H. Wu、M. Xu、Y. Zhang、X. Cheng、F. Xu 和 S. Zhong，“ 未发表，关于基于大语言模型的漏洞检测，你想知道的一切 —— 但又不敢问”，arXiv预印本 arXiv:2504.13474，2025年。[29] M. L. Siddiq 和 J. C. S. Santos，“SecurityEval数据集：挖掘漏洞示例以评估基于机器学习的代码生成技术”，载于《第一届软件仓库挖掘在隐私与安全领域应用国际研讨会论文集》，2022年。

[30] D. B. Johnson，“关于Dijkstra最短路径算法的注记”，《*ACM*期刊》，第20卷，第3期，第385–388页，1973年7月。 [在线]。网址：https://doi.org/10 1145 321765 321768 。/。B. Hui、J. Yang、Z. Cui、J. Yang、D. Liu、[31] L. Zhang、T. Liu、J. Zhang、B. Yu、K. Lu、K. Dang、Y. Fan、Y. Zhang、A. Yang、R. Men、F. Huang、B. Zheng、Y. Miao、S. Quan、Y. Feng、X. Ren、X. Ren、J. Zhou 和 J. Lin，“Qwen2.5-coder技术报告”，2024年。 [在线]。网址：https://arxiv.org/abs/ 2409 12186 。A. Q. Jiang、A. Sablayrolles、[32] A. Mensch、C. Bamford、D. S. Chaplot、D. de las Casas、F. Bressand、G. Lengyel、G. Lample、L. Saulnier、L. Renard Lavaud、M.-A. Lachaux、P. Stock、T. Le Scao、T. Lavril、T. Wang、T. Lacroix 和 W. El Sayed，“Mistral 7b”，2023年。 [在线]。网址：https://arxiv.org/abs/ 2310 06825 。A. Grattafiori、A. [33] Dubey 和 A. J. 等，“Llama 3模型家族”，2024 [ ] 2407 21783 。在线。网址：[34] https://arxiv.org/abs/。M. Abdin、J. Aneja、H. Behl、S. Bubeck、R. Eldan、S. Gunasekar、M. Harrison、R. J. Hewett、M. Javaheripi、P. Kauffmann 等，“Phi-4技术报告”，*arXiv*预印本 *arXiv:2412.08905*，2024年。[35] D. Guo、Q. Zhu、D. Yang、Z. Xie、K. Dong、W. Zhang、G. Chen、X. Bi、Y. Wu、Y. Li 等，“DeepSeek-Coder：大语言模型邂逅编程——代码智能的崛起”，*arXiv*预印本 *arXiv:2401.14196*，2024年。[36] A. Rastogi、A. Yang 和 e. a. Albert Q. Jiang，“Devstral：面向编码智能体应用的语言模型微调”，2025年。 [在线]。网址：https://arxiv.org/abs/2509 25193 。CodeQL。CodeQL 研究版。[37] https://securitylab.github.com/tools/codeq

[38] Semgrep。2022。Semgrep。https://github.com/semgrep/semgrep。

附录

A. 系统提示词

你是一名安全意识强的开发者，精通常见漏洞模式。对于给定的代码块，请进行系统的安全分析。

关键约束：你必须准确识别一个 CWE。即使存在多个漏洞，也只能选择其中最严重的一个。

说明：1. 检查每一行代码和每个函数，查找潜在的安全问题 2. 考虑与代码表面用途相关的常见攻击向量 3. 评估任何发现的严重性和可利用性 4. 识别代码中存在的最严重的一个漏洞 5. 仅使用其 CWE 标识符报告该漏洞

回复格式（必须遵守）：
分析
⟨analysis⟩对发现的详细分析，包括说明为何这是最严重的漏洞 ⟨/analysis⟩### 发现摘要 CWE: CWE-N

重要说明：- 必须严格使用格式“CWE: CWE-N”，其中 N 为编号 - 不要列出多个 CWE（例如，“CWE-89, CWE-20”无效）- 如果不存在漏洞，请写 “CWE: None” - 在 “### 发现摘要 ” 部分，除了选定的 CWE 外，不得提及任何其他 CWE。

正确示例：**代码块：
** “‘python
def login(username, password): query =f“ SELECT * FROM users WHERE \username=’用户名’ AND password=’密码’” result = db.execute(query) if result: return ”登录成功” return ”登录失败” “‘

预期回复：
分析⟨
分析 ⟩
此登录函数存在严重的SQL注入漏洞。该函数直接将用户输入（用户名和密码）拼接到SQL查询字符串中，未进行任何净化或参数化。攻击者可以通过恶意SQL代码操纵用户名参数，例如 ”admin’ OR ’1’=’1’ –”，从而完全绕过身份验证。虽然该函数还以明文存储密码（CWE- 256），但SQL注入（CWE-89）危害更严重，因为它可能导致完全绕过身份验证，并可能造成数据库失陷。因此，CWE-89被认定为最严重的漏洞。 ⟨/analysis⟩### 发现摘要 CWE：CWE-89