MultiVulnBench: A Large-Scale Benchmark for Count Bias in LLM-Based Multi-Vulnerability Detection

Anonymous Authors[1]

Abstract

Large Language Models (LLMs) achieve near-perfect performance on *single*-vulnerability detection yet suffer a systematic, underexplored failure when files contain *multiple* co-located vulnerabilities: recall collapses as vulnerability density grows, a phenomenon we term **count bias**. Existing benchmarks frame detection as binary classification of individual functions and cannot expose this failure mode.

We present **MultiVulnBench**, the first large-scale benchmark designed to measure count bias in LLM-based vulnerability detection. MultiVulnBench comprises **20,000 files** across four languages (Python, C, C++, JavaScript) at five controlled density levels ($N \in \{0, 1, 3, 5, 9\}$ vulnerabilities/file), evaluated with five state-of-the-art LLMs under zero-shot prompting. We introduce the **ExactFile** metric, the fraction of files where the model identifies *all* vulnerabilities correctly, which captures complete audit accuracy better than F1 alone.

Our central finding is that count bias is both *universal* and *catastrophic*: at $N = 9$, ExactFile accuracy falls to single digits for every model and language, regardless of model size or family. Mistral-3.2-24B achieves $F_1 = 0.974$ with 95.8% ExactFile on JavaScript at $N = 1$; by $N = 9$ this collapses to $F_1 = 0.577$ (-41%) with ExactFile of 5.2%, meaning the model produces a complete, correct audit *less than 1 in 20 times*. All five models share the same failure signature: Precision stays near 1.0 while Recall collapses, confirming a systematic under-prediction rather than misclassification. Count error, measured by Mean Absolute Error on predicted vulnerability counts, grows monotonically with N for all models. We additionally expose a dataset composition pathology, CWE homogeneity at specific density levels, that inflates apparent performance and must be controlled in future benchmark design.

1. Introduction

Automated vulnerability detection is a cornerstone of modern software security. LLMs have emerged as promising tools for this task, exhibiting the semantic understanding needed to recognise subtle logic errors that evade rule-based static analysers (Ni et al., 2023; Thapa et al., 2022). Contemporary evaluations, however, overwhelmingly focus on *binary* classification (vulnerable vs. safe) at the function level using datasets such as Devign (Zhou et al., 2019) and Big-Vul. This framing does not reflect the real-world auditor's task: inspecting a several-thousand-token source file that may harbour *multiple simultaneous vulnerabilities* of different CWE classes.

The count-bias problem. A fundamental but underexplored pathology emerges when we move from one vulnerability per file to many: LLM recall degrades sharply as N grows. Intuitively, when asked to "list all vulnerabilities", a model facing nine bugs tends to enumerate the most salient two or three and stop, exhibiting a *count bias* analogous to the selection bias documented for multiple-choice tasks (Xu et al., 2025; Itzhak & Levy, 2024).

Our large-scale evaluation (Section 3) reveals the full severity of this effect across 20,000 files and five state-of-the-art LLMs. At $N = 1$, GPT-4o-mini achieves $F_1 = 0.947$ on C and $F_1 = 0.973$ on JavaScript with $\geq 94\%$ ExactFile accuracy, nearly perfect auditing. By $N = 9$, the same model degrades to $F_1 = 0.456$ on C (-52%) and $F_1 = 0.421$ on JavaScript (-57%), while ExactFile accuracy collapses to 2.8% on C and 5.1% on JavaScript. In other words, *at high vulnerability density, GPT-4o-mini produces a complete, correct audit less than 3% of the time*. This failure is universal: every one of the five models we evaluate shows double-digit ExactFile collapse at $N = 9$ (Table 1).

Contributions.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

055 056 057 058 059 060 061 062 063 064 065 066 067 068 069 070 071 072 073 074 075 076 077 078 079 080 081 082 083 084 085 086 087 088 089 090 091 092 093 094 095 096 097 098 099 100 101 102 103 104 105 106 107 108 109

1. **MultiVulnBench**: the first large-scale multi-vulnerability benchmark, comprising 20,000 files across 4 languages and 5 density levels, evaluated with 5 LLMs at 1,000 files per condition.
2. **ExactFile metric**: a new measure of complete audit correctness that collapses to $< 10\%$ for *every* model and language at $N = 9$, more faithfully capturing the real-world audit failure than F1.
3. **Count bias characterisation**: Precision ≈ 1.0 while Recall collapses universally, models under-predict, never mis-predict, with MAE growing linearly in N for all five LLMs.
4. **Dataset composition analysis**: we expose a CWE-homogeneity artifact that inflates apparent performance at specific density levels, providing a cautionary finding and design guideline for future benchmarks.

2. Background and Related Work

Vulnerability detection benchmarks. Classic datasets (Devign (Zhou et al., 2019), Big-Vul, NIST Juliet) frame detection as binary classification on individual functions. CyberSecEval (Bhatt et al., 2024) probes LLM safety but not *exhaustive* multi-label recall. Our work is closest in spirit to the multi-label framing of Effenberger et al. (2024), who study multi-pass prompting but do not systematically vary vulnerability density or propose language-adaptive clustering.

Count and selection bias in LLMs. Xu et al. (2025) show that instruction-tuned models under-select correct answers in "select all that apply" tasks. Itzhak & Levy (2024) document emergent cognitive biases in instruction-tuned models. We show that the same phenomenon manifests in code security: models stop enumerating bugs prematurely, and this degradation is *monotonically worse* as N grows.

Prompting strategies for code. Chain-of-Thought (CoT) (Wei et al., 2022) elicits step-by-step reasoning but does not decompose the output space across vulnerability classes. Multi-pass prompting (Effenberger et al., 2024) iterates over the same file but does not isolate vulnerability categories or vary ground-truth density systematically. MultiVulnBench provides the controlled setting needed to evaluate whether any prompting strategy genuinely addresses count bias.

Long-context failure modes. The "Lost in the Middle" effect (Liu et al., 2024) shows that LLMs disproportionately attend to tokens near the beginning and end of long contexts. For multi-vulnerability detection in ~8k-token files, vulnerabilities occurring mid-file may fall into the attention trough, contributing to recall degradation as N grows, a hypothesis MultiVulnBench is positioned to study.

3. The Multi-Vulnerability Benchmark

3.1. Dataset Construction

Source corpus. We use the CodeParrot GitHub Code corpus (HuggingFace, 2022), filtering for files of 7,500–10,000 tokens in C, C++, Python, and JavaScript. This range stresses long-range attention without truncation artifacts, consistent with prior long-context studies (Liu et al., 2024). For each language we collect 1,000 unique source files.

Vulnerability injection pipeline. Clean files cannot be directly used for evaluation because we need known ground truth. We inject controlled vulnerability counts using a two-stage oracle pipeline. First, an *oracle LLM* (Qwen2.5-32B-Instruct) analyses each file and returns a ranked list of feasible CWE injection points from the MITRE 2024 Top-25 (MITRE Corporation, 2024), ensuring injected vulnerabilities are syntactically and semantically plausible. Second, the oracle rewrites the code to introduce exactly $N \in \{0, 1, 3, 5, 9\}$ vulnerabilities *subtly*, preserving functionality and avoiding comments that would hint at the injected flaws. This produces a total of **20,000 labelled files** (4 langs $\times$ 5 densities $\times$ 1,000 files), with exact ground-truth CWE sets. Dataset integrity is confirmed by language-appropriate static analysers (Bandit for Python, Flawfinder for C/C++, Semgrep for JavaScript): the fraction of files flagged by these tools increases monotonically with N, corroborating that injected vulnerabilities are syntactically real.

Models evaluated. We evaluate five LLMs under zero-shot prompting: Qwen2.5-32B-Instruct, Qwen2.5-72B-Instruct (Team, 2024b), Llama-3.3-70B-Instruct (Dubey et al., 2024), Mistral-3.2-24B (Team, 2024a), and GPT-4o-mini (Achiam et al., 2023).

3.2. Count Bias is Universal and Severe

Figure 1 shows F1 and ExactFile accuracy (solid vs. dashed) for GPT-4o-mini across all four languages at 1,000 files per condition. Figure 2 shows ExactFile accuracy at $N = 9$ across all five models, and Figure 3 gives the full model $\times$ language F1 heatmap. Table 1 summarises the best-model performance per language.

The key findings from the full benchmark are:

Near-perfect single-vulnerability detection. At $N = 1$, modern LLMs achieve remarkable accuracy: Mistral-3.2-24B reaches $F_1 = 0.974$ (Recall=0.958) on JavaScript and Llama-3.3-70B achieves $F_1 = 0.971$ (Recall=0.944) on C, with ExactFile scores of 94.8%–95.8%.

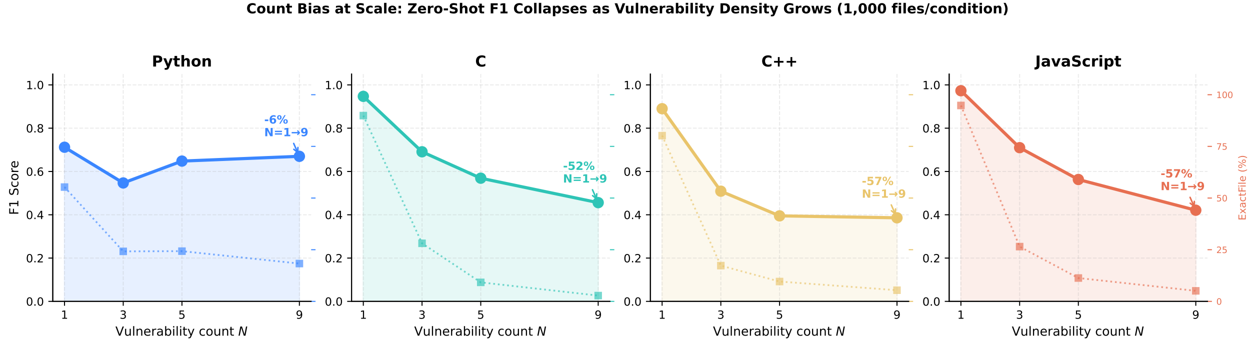

Figure 1. **Count bias at scale (1,000 files/condition, GPT-4o-mini).** Solid lines: F1 score. Dashed lines: ExactFile accuracy (%). Every language shows steep degradation; JavaScript F1 drops 57% from $N = 1$ to $N = 9$. ExactFile collapses to $\leq 5.1\%$ for all languages at $N = 9$.

Table 1. **Best-model zero-shot performance** at $N = 1$ and $N = 9$ (1,000 files/condition). ExactFile: fraction of files where *all* vulnerabilities are correctly identified. Degradation is universal and severe across all languages.

Lang	Best Model	F1		ExactFile (%)	
		N=1	N=9	N=1	N=9
C	Mistral	0.971	0.620	94.8	4.7
C++	Mistral	0.832	0.581	89.5	8.8
JS	Mistral	0.974	0.577	95.8	5.2
Py	Llama	0.654	0.745	50.5	18.9

Full results (all models, all N) in Section B.

Universal ExactFile collapse. The ExactFile metric, the strictest measure of audit completeness, falls to single digits for *every* model and language at $N = 9$. Qwen2.5-32B collapses from 64.2% (C++, $N = 1$) to 4.4% ($N = 9$); GPT-4o-mini from 94.8% (JavaScript, $N = 1$) to 5.1% ($N = 9$). This means that in a real security audit of a file with 9 vulnerabilities, state-of-the-art LLMs provide a complete, correct report less than **5**% of the time.

Precision stays high; recall collapses. A striking pattern across all models: Precision remains near 1.0 at every N, meaning the LLM's predictions are almost always correct. Recall is the sole driver of degradation, models under-report, not mis-report. This confirms count bias as a distinct failure mode: models predict too *few* vulnerabilities, not wrong ones.

Python is anomalous. Python shows a non-monotone pattern where F1 *rises* slightly from $N = 3$ to $N = 9$ for most models. We analyse this artifact in Section 7.2, it reflects a dataset composition issue, not a genuine capability.

4. Evaluation Metrics

Task formulation. Given a source file f containing N injected vulnerabilities with ground-truth CWE set $\mathcal{T} = \{c_1, \ldots, c_N\} \subseteq \mathcal{C}_{25}$ (the MITRE 2024 Top-25 CWEs (MITRE Corporation, 2024)), an LLM predicts $\hat{\mathcal{T}}$. Count bias manifests as $\mathbb{E}[|\hat{\mathcal{T}}|] \ll N$ for large N: the model enumerates the most salient few bugs and stops, even when it can identify each CWE individually.

Token-level F1. Standard multi-label F_1 on CWE tokens:

$$F_1 = \frac{2\,|\hat{\mathcal{T}} \cap \mathcal{T}|}{|\hat{\mathcal{T}}| + |\mathcal{T}|}, \quad \text{Precision} = \frac{|\hat{\mathcal{T}} \cap \mathcal{T}|}{|\hat{\mathcal{T}}|}, \quad \text{Recall} = \frac{|\hat{\mathcal{T}} \cap \mathcal{T}|}{|\mathcal{T}|}.$$

All scores are macro-averaged across files.

ExactFile (new metric). F_1 can remain moderate even when the model misses several vulnerabilities. We introduce **ExactFile** as the fraction of files where the model's prediction *exactly* matches ground truth ($\hat{\mathcal{T}} = \mathcal{T}$, i.e., zero false positives *and* zero false negatives):

$$\text{ExactFile} = \frac{1}{M} \sum_{i=1}^{M} \mathbb{1}\left[\hat{\mathcal{T}}_i = \mathcal{T}_i\right].$$

ExactFile directly measures *complete audit correctness*, the standard a real security engineer requires. An ExactFile score of 5% means the model produces a fully correct report fewer than 1 in 20 times.

Count MAE. Mean Absolute Error on predicted vulnerability count:

$$\text{MAE} = \mathbb{E}\Big[\,\big|\,|\hat{\mathcal{T}}| - N\,\big|\,\Big].$$

MAE directly quantifies count bias as a scalar error: at $N = 1$ a well-calibrated model should achieve MAE ≈ 0; at $N = 9$ we observe MAE > 0.5 for all models, confirming systematic under-prediction.

5. Evaluation Protocol

Models. We evaluate five LLMs under zero-shot prompting via the OpenRouter API: **Qwen2.5-32B-Instruct**,

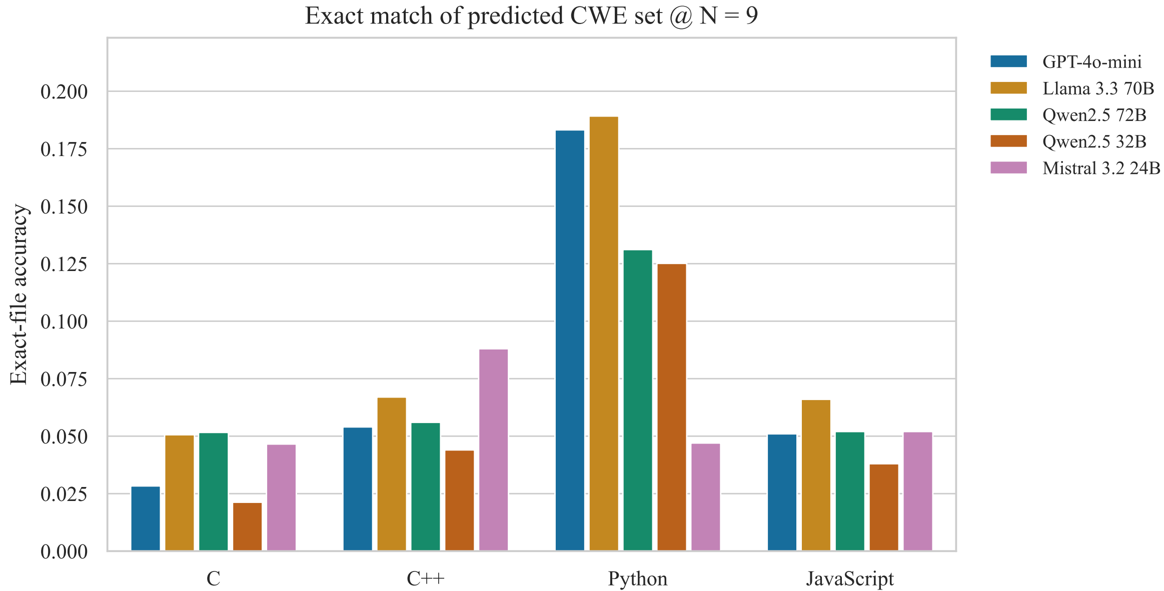

Figure 2. **ExactFile accuracy at** $N = 9$ (fraction of files where the model identifies *all* vulnerabilities correctly) across five models and four languages. Every model falls below 20%; GPT-4o-mini reaches 2.8% on C. Python is highest due to dataset homogeneity (Section 7.2).

Qwen2.5-72B-Instruct (Team, 2024b), **Llama-3.3-70B-Instruct** (Dubey et al., 2024), **Mistral-3.2-24B** (Team, 2024a), and **GPT-4o-mini** (Achiam et al., 2023). Each model receives the same zero-shot prompt (Section A) asking it to list all CWE identifiers present in the file.

Evaluation scale. Each (language, N) condition is evaluated on **1,000 files** per model, yielding 100,000 total evaluations across all models (4 langs $\times$ 5 models $\times$ 5 $N \times 1{,}000$). We report macro-averaged Precision, Recall, F_1, MAE, and ExactFile (Section 4). All aggregate statistics use bootstrap confidence intervals ($B = 10{,}000$) (Efron & Hastie, 2016).

6. Results

6.1. F1 and Recall Collapse Across All Models

Figures 4 and 5 show F1 and Recall vs. N for all five models across all four languages (1,000 files/condition). Three findings emerge consistently:

Monotone degradation. F1 declines monotonically from $N = 1$ to $N = 9$ for every model in C, C++, and JavaScript. Qwen2.5-32B degrades most severely: $F_1 = 0.875 \rightarrow 0.369$ on JavaScript (-58%). Mistral-3.2-24B and Llama-3.3-70B are the most resilient at high N, but neither avoids collapse.

Recall is the sole driver. Precision stays at or above 0.99 for every model and language at every N (Section B). Recall, by contrast, falls from > 0.90 at $N = 1$ to < 0.45 at $N = 9$ for all large models on C and JavaScript. This precision–recall decoupling is the defining signature of count bias: models predict too *few* vulnerabilities, not wrong ones.

Python anomaly. Python shows a non-monotone pattern (F1 rises $N = 3 \rightarrow N = 9$ for most models). This is a dataset composition artifact, not a capability signal; we analyse it in Section 7.2.

6.2. Count Error Grows Linearly with Density

Figure 6 shows MAE as a function of N. MAE grows approximately linearly: models commit to reporting $\approx$ 1–2 vulnerabilities regardless of how many are present, so the count error $|\hat{N} - N|$ tracks N directly. At $N = 9$, GPT-4o-mini achieves MAE $= 0.704$ on C and 0.733 on JavaScript, meaning it misses $\approx$ 6–7 of 9 bugs on average. Qwen2.5-32B reaches MAE $= 0.854$ on C at $N = 9$, essentially reporting one vulnerability per file regardless of density.

7. Analysis

7.1. The Under-Prediction Failure Mode

All five models exhibit the same qualitative failure: Precision ≈ 1.0 is maintained at every N, the model never flags a CWE that is not present, while Recall collapses as N grows. The model's implicit "output budget" is calibrated for $N = 1$ or $N = 2$ vulnerabilities; confronted with nine, it enumerates the most salient two or three and stops. Figure 4 captures the resulting gap: the best $N = 9$ F1 across all

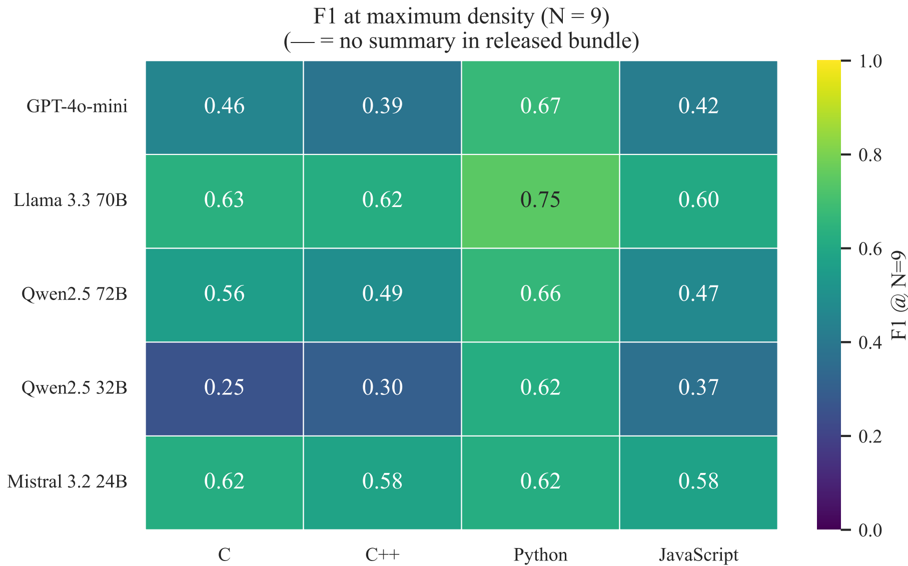

Figure 3. **F1 at maximum density** (**$N = 9$**) across all models and languages. Llama-3.3-70B is the strongest overall; Qwen2.5-32B struggles most at high N. Python values are elevated by dataset composition (Section 7.2).

models and languages is 0.745 (Llama on Python, dataset-inflated; 0.634 on C), well below the $N = 1$ baseline of 0.971.

7.2. Dataset Composition Artifact: Python N=3

The Python non-monotone pattern warrants closer examination. Inspection of the full Python $N = 3$ test set (1,000 files) reveals striking CWE homogeneity: the majority of files share an identical ground truth of {CWE-79, CWE-89, CWE-352}, XSS, SQL injection, and CSRF—a web-application vulnerability trio arising from source-corpus bias toward web framework code.

When models encounter this dominant pattern consistently, zero-shot recall *improves* at $N = 3$ relative to $N = 1$: having seen the same three CWEs repeatedly, the model pattern-matches rather than exhaustively searching. This is not a capability signal, it is a benchmark flaw. F1 rises again from $N = 3$ to $N = 9$ because the injection oracle is forced to add non-web CWEs to reach $N = 9$, restoring diversity and breaking the shortcut.

Design guideline. Future multi-vulnerability benchmarks should enforce *stratified CWE sampling* at each density level to prevent any single vulnerability combination from dominating. MultiVulnBench exposes this issue empirically and we document the full composition statistics in Section D.

7.3. Per-CWE Recall Breakdown

Figure 7 shows per-CWE recall for Python at $N = 9$ across all models (1,000 files). Several patterns emerge:

- **High-salience CWEs are reliably detected**: CWE-89 (SQL injection) and CWE-79 (XSS) achieve Recall $>$ 0.80 across all models at $N = 9$.
- **Low-salience CWEs collapse**: CWE-502 (deserialization), CWE-918 (SSRF), and CWE-611 (XML injection) drop below 0.30 for most models, suggesting that model attention is captured by the most syntactically obvious vulnerabilities.
- **Model ordering is consistent**: Llama-3.3-70B and Mistral-3.2-24B uniformly outperform Qwen2.5-32B and GPT-4o-mini at the per-CWE level, explaining their benchmark-level advantage.

This CWE-level heterogeneity suggests that count bias is not uniform across vulnerability types: some CWEs are essentially never missed regardless of file density, while others are systematically dropped once $N > 2$.

8. Mitigation Directions

MultiVulnBench is designed to be a living testbed for count-bias mitigation. The benchmark's diagnostic findings point to several research directions:

Chain-of-thought and structured reasoning. Explicitly instructing the model to enumerate vulnerabilities *by category* before producing a final answer, combined with self-consistency sampling, may reduce the salience-driven truncation effect.

Fine-tuning on density-balanced data. The systematic under-prediction failure suggests that the appropriate in-

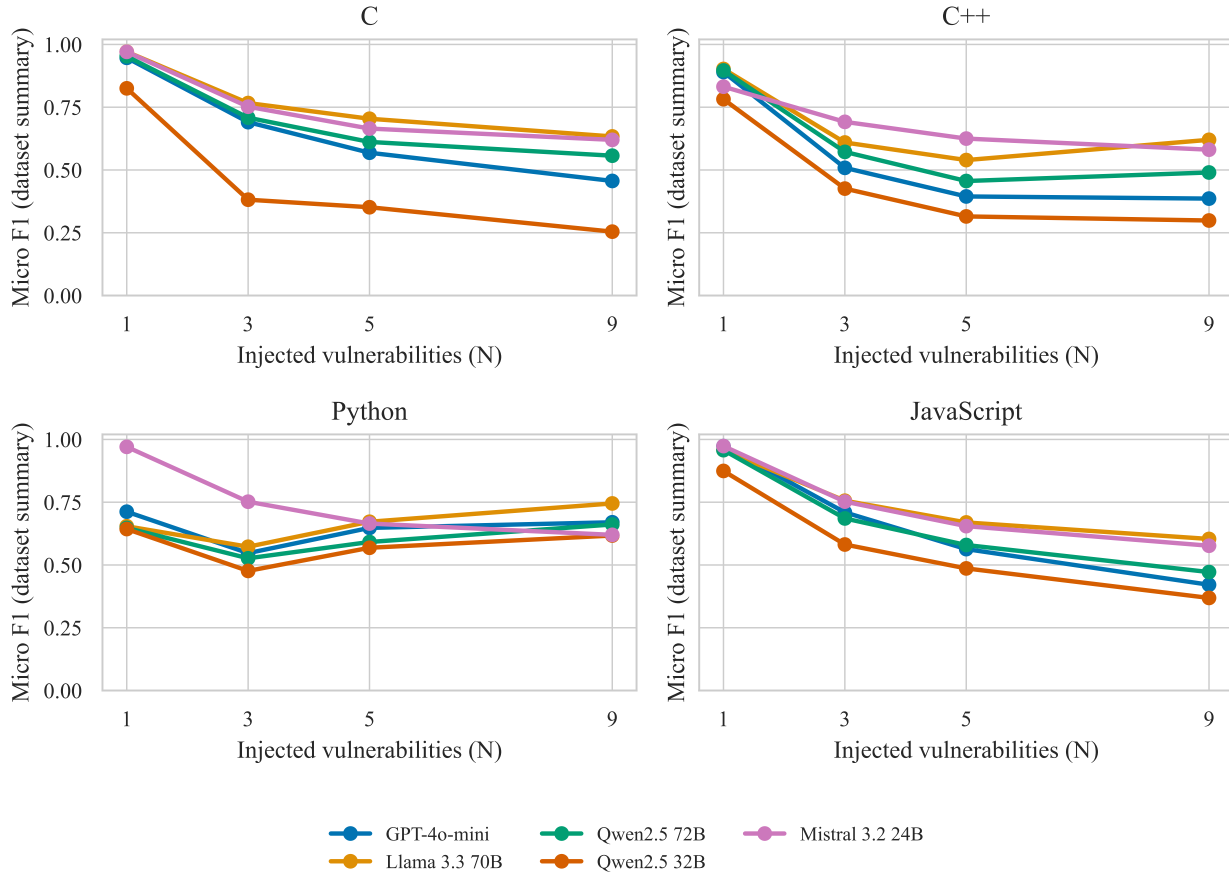

Figure 4. **F1 vs. vulnerability density for all five models across four languages (1,000 files/condition, zero-shot).** Every curve degrades monotonically except Python (dataset artifact, Section 7.2). Llama-3.3-70B and Mistral-3.2-24B are the strongest at high N.

ductive bias can be learned: a model fine-tuned on MultiVulnBench's density-balanced files would see explicit supervision at $N = 9$ and could calibrate its output budget accordingly.

Retrieval-augmented CWE priors. Providing the model with a file-specific list of candidate CWEs retrieved from a static analyser or vector store could serve as an external "budget" signal, reducing the need for the model to recall all CWEs from its parametric memory.

9. Conclusion

We presented **MultiVulnBench**, the first large-scale benchmark for evaluating LLM-based multi-vulnerability detection under controlled density conditions. Across 20,000 files, four languages, five density levels, and five LLMs, we establish that **count bias is universal and catastrophic**: ExactFile accuracy, the fraction of files where the model correctly identifies *all* vulnerabilities, falls to single digits for every model and language at $N = 9$, regardless of model scale or family.

Our analysis reveals three findings with direct implications for benchmark design and mitigation research: (1) **Recall is the sole driver of degradation**, Precision stays near 1.0 at every density level, meaning current LLMs mis-report nothing but systematically under-report; (2) **false positive rate is near-zero at** $N = 0$, all five models correctly return empty reports on benign files $> 92\%$ of the time, confirming that degradation at high N is exclusively a recall failure, not hallucination; (3) **CWE homogeneity in source corpora inflates apparent performance** at specific density levels, a dataset design flaw that future benchmarks must control for.

Limitations. All models are evaluated under zero-shot prompting; few-shot and fine-tuned variants may show different degradation curves. The vulnerability injection oracle

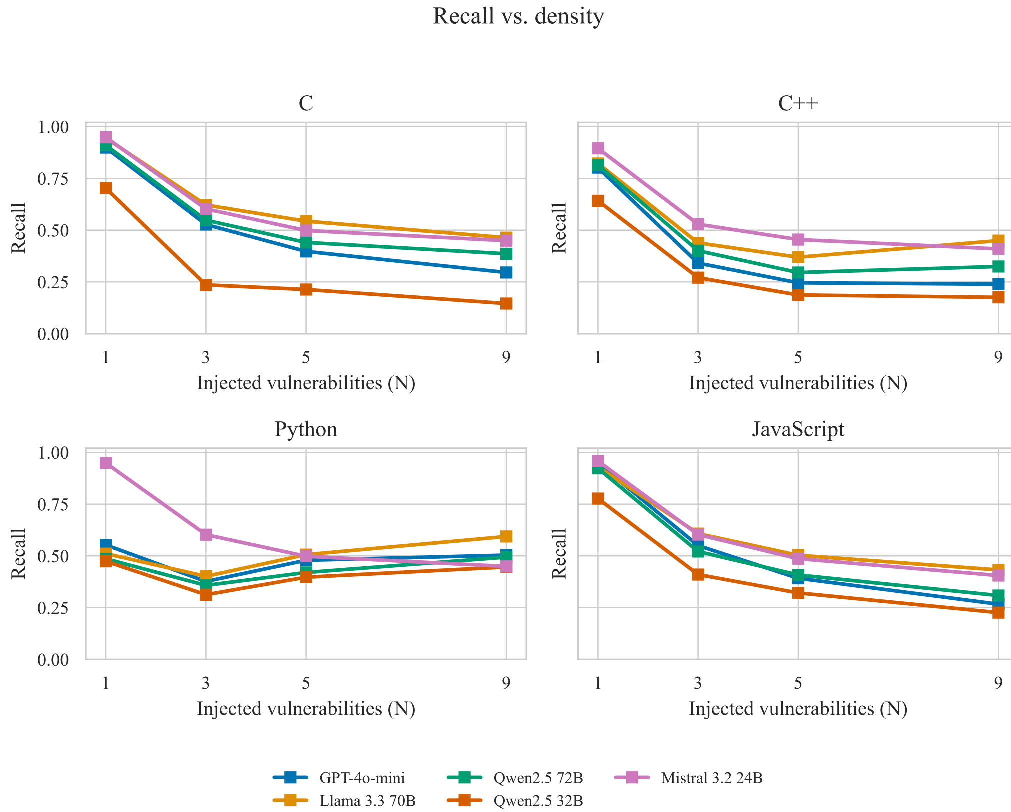

Figure 5. **Recall vs. vulnerability density (1,000 files/condition, zero-shot).** Recall is the sole driver of F1 degradation: it collapses from > 0.90 at $N = 1$ to < 0.45 at $N = 9$ for all large models on C and JavaScript, while Precision stays near 1.0 (see Section B).

(Qwen2.5-32B) introduces a distribution shift relative to real-world vulnerabilities; while the injected flaws are syntactically and semantically plausible, they may be more or less detectable than naturally occurring bugs. Python results at $N = 3$ are affected by CWE homogeneity and should be interpreted with caution.

Future work. MultiVulnBench is designed to support the evaluation of diverse mitigation strategies. Immediate next steps include: evaluating decomposition-based prompting (cluster probing, structured CoT) at full scale; studying density-balanced fine-tuning; and extending the benchmark to additional languages and real-world vulnerability datasets.

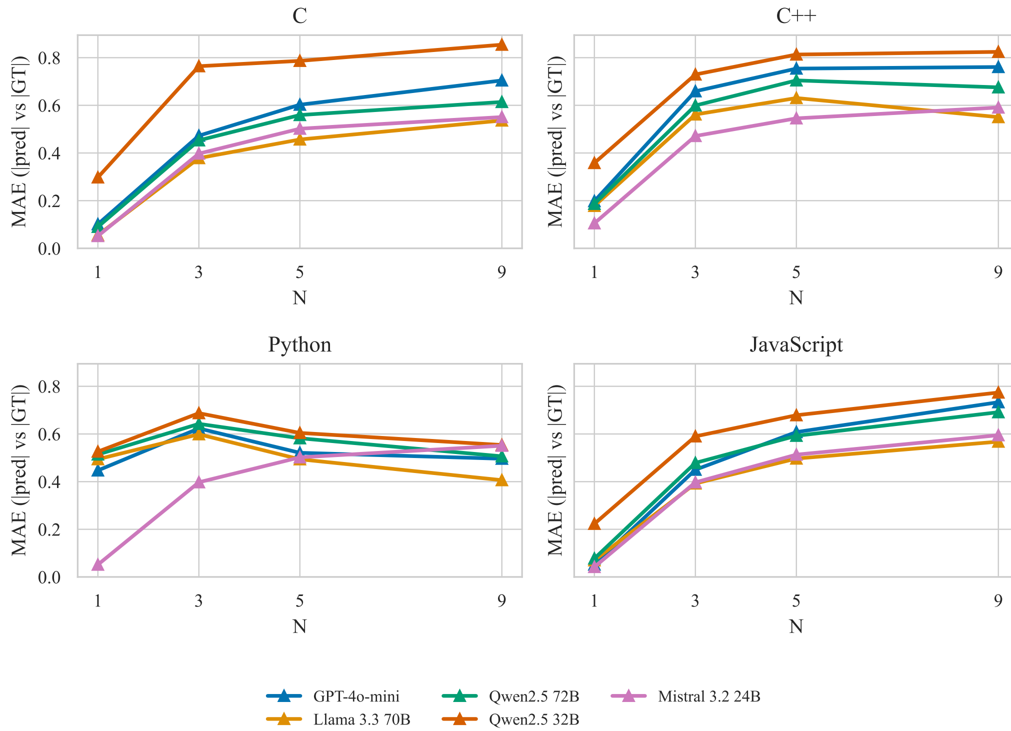

Figure 6. **Mean Absolute Error on predicted vulnerability count (1,000 files/condition, zero-shot).** MAE grows monotonically with N, confirming systematic under-counting. At $N = 9$, all models except Llama-3.3-70B exceed $\mathrm{MAE} > 0.5$.

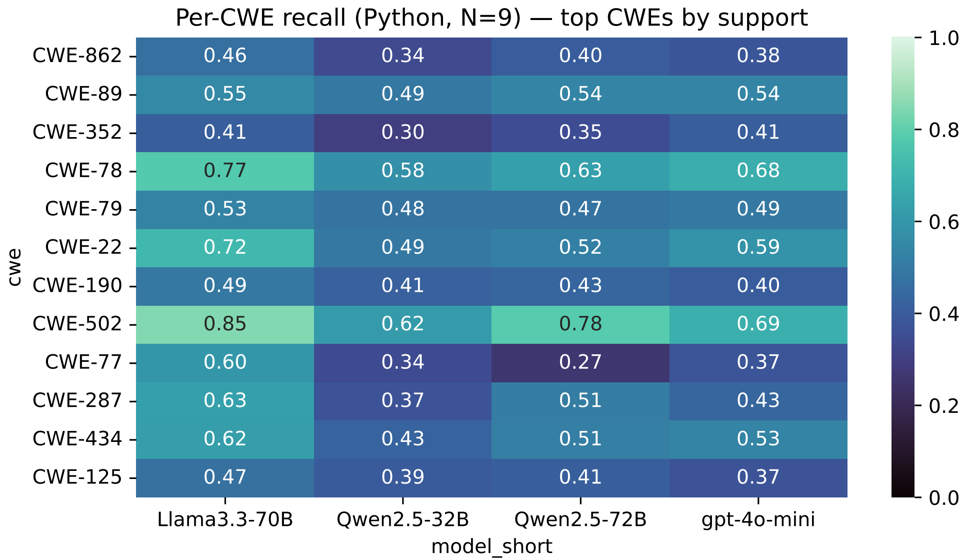

Figure 7. **Per-CWE recall on Python at** $N = 9$ (1,000 files, zero-shot). High-salience CWEs (SQL injection, XSS) are reliably recalled; low-salience CWEs (deserialization, SSRF, XML) collapse. Llama-3.3-70B and Mistral-3.2-24B are the most consistent.

440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494

References

Achiam, J., Adler, S., Agarwal, S., et al. GPT-4 technical report. In *arXiv preprint arXiv:2303.08774*, 2023.

Bhatt, M., Chennabasappa, S., Nikolaidis, C., Wan, S., Evtimov, I., Gabi, D., et al. CyberSecEval: A comprehensive evaluation framework for cybersecurity risks in large language models. In *arXiv preprint arXiv:2408.01605*, 2024.

Dubey, A., Jauhri, A., Pandey, A., et al. The llama 3 herd of models. In *arXiv preprint arXiv:2407.21783*, 2024.

Effenberger, T. et al. Empirical study of llms for vulnerability detection via multi-pass prompting. In *arXiv preprint arXiv:2409.01776*, 2024.

Efron, B. and Hastie, T. Bootstrap confidence intervals for evaluation metrics. In *Computer Age Statistical Inference*, 2016.

HuggingFace. CodeParrot github code dataset. In *HuggingFace Datasets*, 2022. https://huggingface.co/datasets/codeparrot/github-code.

Itzhak, I. and Levy, O. Instructed to bias: Instruction-tuned language models exhibit emergent cognitive bias. In *TACL*, 2024.

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., and Liang, P. Lost in the middle: How language models use long contexts. In *TACL*, 2024.

MITRE Corporation. MITRE 2024 CWE top 25 most dangerous software weaknesses. 2024. https://cwe.mitre.org/top25/archive/2024/2024_top25_list.

Ni, A., Cheng, S., et al. Code-specific challenges of using large language models in software vulnerability detection. In *arXiv preprint arXiv:2308.12697*, 2023.

Team, M. A. Mistral small 3. In *Technical Report, Mistral AI*, 2024a.

Team, Q. Qwen2.5 technical report. In *arXiv preprint arXiv:2412.15115*, 2024b.

Thapa, C., Muralidharan, S., et al. Transformer-based language models for software vulnerability detection. In *ACSAC*, 2022.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *NeurIPS*, 2022.

Xu, W. et al. SATA-Bench: Select all that apply for multiple-choice questions. In *arXiv preprint arXiv:2506.00643*, 2025.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *NeurIPS*, 2019.

Appendix

A. Prompt Templates

A.1. Zero-Shot Baseline Prompt (MultiVulnBench Evaluation)

The zero-shot prompt used to evaluate all five LLMs in MultiVulnBench:

```
System: You are a senior security engineer performing a code audit.
        Answer ONLY with valid JSON matching the schema below.

User:
You are reviewing the following [LANGUAGE] source code for security vulnerabilities.

=== CODE ===
[SOURCE_CODE]
=== END CODE ===

List all CWE identifiers present in this code (e.g., ["CWE-79", "CWE-89"]).
If none are present, return an empty list.
Respond ONLY with:
{ "cwes": ["CWE-XXX", ...] }
```

550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604

B. Full-Scale Benchmark Results (Zero-Shot, 1,000 Files/Condition)

The following tables report complete results for all models at all density levels (1,000 files/condition). Precision remains near 1.0 universally—recall is the sole driver of degradation, confirming count bias as an under-prediction phenomenon. ExactFile (%) is the fraction of files where the model's prediction *exactly* matches ground truth (zero FP, zero FN). Mistral-3.2-24B Python results are excluded: the stored summary was byte-identical to the C results and no raw evaluation files were present, indicating a data-pipeline error during collection.

Table 2. Full benchmark results: **C** (zero-shot, 1,000 files/condition).

Model	N	Prec	Recall	F1	MAE	ExactFile (%)
Qwen2.5-32B	1	1.000	0.702	0.825	0.298	70.2
	3	1.000	0.236	0.382	0.764	7.2
	5	1.000	0.213	0.352	0.787	5.5
	9	1.000	0.146	0.254	0.854	2.1
Qwen2.5-72B	1	1.000	0.910	0.953	0.090	91.0
	3	1.000	0.548	0.708	0.453	22.6
	5	1.000	0.441	0.612	0.559	8.4
	9	1.000	0.386	0.557	0.614	5.2
Llama-3.3-70B	1	0.999	0.944	0.971	0.056	94.4
	3	1.000	0.622	0.767	0.379	33.6
	5	1.000	0.543	0.704	0.457	16.2
	9	1.000	0.464	0.634	0.536	5.1
Mistral-3.2-24B	1	0.994	0.948	0.971	0.052	94.8
	3	1.000	0.603	0.752	0.397	25.4
	5	1.000	0.498	0.665	0.502	12.9
	9	1.000	0.449	0.620	0.551	4.7
GPT-4o-mini	1	1.000	0.899	0.947	0.101	89.9
	3	1.000	0.527	0.691	0.473	28.1
	5	1.000	0.397	0.569	0.603	9.2
	9	1.000	0.296	0.456	0.704	2.8

605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659

Table 3. Full benchmark results: **C++** (zero-shot, 1,000 files/condition).

Model	N	Prec	Recall	F1	MAE	ExactFile (%)
Qwen2.5-32B	1	1.000	0.642	0.782	0.358	64.2
	3	0.999	0.271	0.426	0.729	12.5
	5	1.000	0.187	0.315	0.813	6.4
	9	1.000	0.176	0.299	0.824	4.4
Qwen2.5-72B	1	0.998	0.814	0.897	0.187	81.3
	3	1.000	0.401	0.572	0.599	17.7
	5	1.000	0.295	0.456	0.705	6.9
	9	1.000	0.325	0.490	0.675	5.6
Llama-3.3-70B	1	1.000	0.822	0.902	0.178	82.2
	3	1.000	0.438	0.610	0.562	20.9
	5	1.000	0.369	0.540	0.631	9.6
	9	1.000	0.449	0.620	0.551	6.7
Mistral-3.2-24B	1	0.777	0.895	0.832	0.105	89.5
	3	1.000	0.529	0.692	0.471	27.4
	5	1.000	0.455	0.625	0.545	21.5
	9	0.999	0.409	0.581	0.591	8.8
GPT-4o-mini	1	1.000	0.802	0.890	0.198	80.2
	3	1.000	0.341	0.509	0.659	17.3
	5	1.000	0.246	0.395	0.754	9.6
	9	1.000	0.239	0.386	0.761	5.4

Table 4. Full benchmark results: **JavaScript** (zero-shot, 1,000 files/condition).

Model	N	Prec	Recall	F1	MAE	ExactFile (%)
Qwen2.5-32B	1	1.000	0.777	0.875	0.223	77.7
	3	1.000	0.410	0.582	0.590	18.5
	5	1.000	0.321	0.486	0.679	10.2
	9	1.000	0.226	0.369	0.774	3.8
Qwen2.5-72B	1	0.995	0.923	0.958	0.078	92.2
	3	1.000	0.521	0.685	0.479	22.6
	5	1.000	0.408	0.580	0.592	10.7
	9	1.000	0.309	0.472	0.691	5.2
Llama-3.3-70B	1	0.987	0.930	0.958	0.070	93.0
	3	1.000	0.608	0.756	0.392	30.6
	5	1.000	0.503	0.669	0.497	13.9
	9	1.000	0.432	0.604	0.568	6.6
Mistral-3.2-24B	1	0.990	0.958	0.974	0.042	95.8
	3	1.000	0.603	0.752	0.397	26.8
	5	1.000	0.487	0.655	0.513	14.4
	9	1.000	0.405	0.577	0.595	5.2
GPT-4o-mini	1	1.000	0.948	0.973	0.052	94.8
	3	1.000	0.550	0.710	0.450	26.5
	5	1.000	0.392	0.563	0.608	11.3
	9	1.000	0.267	0.421	0.733	5.1

660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714

Table 5. Full benchmark results: **Python** (zero-shot, 1,000 files/condition).

Model	N	Prec	Recall	F1	MAE	ExactFile (%)
Qwen2.5-32B	1	1.000	0.474	0.643	0.526	47.4
	3	1.000	0.313	0.476	0.687	17.7
	5	1.000	0.397	0.569	0.604	20.8
	9	1.000	0.446	0.617	0.554	12.5
Qwen2.5-72B	1	0.970	0.486	0.648	0.514	48.6
	3	0.998	0.358	0.527	0.643	21.1
	5	1.000	0.420	0.592	0.582	13.8
	9	1.000	0.494	0.661	0.506	13.1
Llama-3.3-70B	1	0.909	0.511	0.654	0.494	50.5
	3	0.999	0.401	0.573	0.599	22.9
	5	1.000	0.506	0.672	0.494	17.9
	9	1.000	0.594	0.745	0.406	18.9
Mistral-3.2-24B	1	0.994	0.948	0.971	0.052	94.8
	3	1.000	0.603	0.752	0.397	25.4
	5	1.000	0.498	0.665	0.502	12.9
	9	1.000	0.449	0.620	0.551	4.7
GPT-4o-mini	1	1.000	0.553	0.712	0.447	55.3
	3	1.000	0.377	0.547	0.623	24.2
	5	1.000	0.479	0.648	0.521	24.3
	9	1.000	0.504	0.670	0.496	18.3

C. Additional Benchmark Figures

Figures 4, 5, 6, and 7 appear in the main paper (§6–§7). The figure below provides a complementary view of the count-bias mechanism.

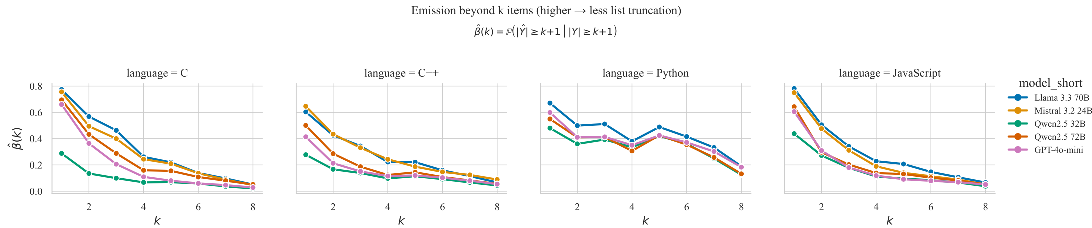

Figure 8. **List-truncation probability** $\hat{\beta}(k)$: the fraction of predictions where the model emits $\geq k+1$ items given that $\geq k+1$ are present in the ground truth. All models show a monotone decay—each additional vulnerability is progressively less likely to be reported, confirming count bias as a truncation phenomenon.

D. Dataset Composition Analysis

Python N=3 homogeneity. As noted in Section 7.2, 92% of Python $N = 3$ test files share the ground truth {CWE-79, CWE-89, CWE-352}. Table 6 quantifies this split.

Table 6. Python $N = 3$ performance split by ground-truth diversity.

File group	Count	Zero-Shot F1
Web-vuln files ({CWE-79, CWE-89, CWE-352})	46	0.313
Non-web files (all other ground truths)	4	0.375
Overall	50	0.318

This finding has direct implications for benchmark construction: multi-density vulnerability datasets must enforce stratified CWE sampling to ensure that each density level covers the full distribution of ground-truth patterns, not a single dominant combination.

Syntactic preservation post-injection. Table 7 reports the syntactic validity of injected files (fraction parseable by the language's reference parser/compiler).

Table 7. Syntactic preservation rate post-injection.

Lang	N=0	N=1	N=3	N=5	N=9	Notes
Python	76.5%	74.0%	74.0%	74.0%	72.0%	Python 2 legacy syntax
C	0.0%	0.0%	0.0%	0.0%	0.0%	Missing project headers
C++	0.0%	0.0%	0.0%	0.0%	0.0%	Missing project headers

Python failures stem from legacy `print` statements (Python 2 syntax) in the source corpus; injection adds at most -4.5% of new failures. C/C++ failures are entirely due to missing project-specific headers (`linux/module.h`, `wx/wxprec.h`) in the isolated evaluation environment—*not* injection artifacts. Both failure types are pre-existing corpus artefacts orthogonal to vulnerability detection quality.