

Peer Review

Review of: "MVD: A Multi-Lingual Software Vulnerability Detection Framework"

Evgenii Arsentev¹

1. Independent researcher

1. What the work claims

The authors argue that function-level vulnerability detectors have been built almost entirely for C/C++, while real projects are polyglot, and they propose MVD: one CodeBERT-based classifier trained jointly on six languages (Python, C/C++, Java, C#, JavaScript, TypeScript). During training, the head has $n+1$ outputs, one "vulnerable" class per language plus a shared "clean" class (SIII-C); at test time, the n vulnerable probabilities are summed and compared with the clean one. Class imbalance is handled by a combined loss, FOLA, which joins focal loss with logit adjustment (Eq. 1). A distillation term (Eq. 2, Eq. 3) lets the model absorb a new language without the old training data.

The data are a new curation built on CVEfixes tooling: Table I lists 779 / 6,311 / 789 / 332 / 2,969 / 151 vulnerable functions for Python, C/C++, Java, C#, JavaScript, and TypeScript. I summed these to 11,331, which matches the "11K+" in the abstract. The headline result is that MVD beats single-language LineVul models "by 83.7% to 193.6% in PR-AUC" (abstract; SV-A), that each component helps in ablation (Table III), and that incremental extension beats a from-scratch single-language model in four of six languages (SV-C, Table IV).

2. Whether the conclusion holds on the data

There is real value here. The problem is practical and under-studied, the dataset is new and covers five languages besides C/C++ that the C/C++-centred literature rarely touches, and the three research questions are well chosen: joint training, component ablation, and extension without old data. I also checked every "% Vuln." cell of Table I against its counts. Python $779/11,580 = 6.7\%$, C/C++ $6,311/123,036 = 5.1\%$, Java $789/11,476 = 6.9\%$, C# $332/1,612 = 20.6\%$, JavaScript $2,969/31,176 = 9.5\%$, TypeScript $151/1,911 =$

7.9%. All six agree with the table, which is more than many papers manage. The decision to report PR-AUC as the main metric under this imbalance is also the right one.

My main concern is how the headline number is built. The 83.7% to 193.6% range (§V-A, third paragraph) compares MVD, evaluated on all six languages, against each LineVul model trained on one language and then also evaluated on all six. A Python-only model scored on C/C++ functions is expected to do badly, so this comparison mostly measures out-of-language failure of the baselines, not the benefit of joint training. The fair comparisons are already in the paper, and they are more modest: per language, MVD improves PR-AUC by 34.9% for C/C++, 30.7% for Java, and 24.9% for TypeScript, and is "on par (within 5% in PR-AUC)" for the other three (§V-A, second paragraph). Against the set of six per-language LineVul models, MVD is "approximately 7% better PR-AUC than that (0.5008)". I would put these numbers in the abstract and the contribution list, and present the 83.7% to 193.6% as a secondary illustration of what happens when a single-language model is deployed on the wrong language. As written, "within 5%" also leaves open that MVD is slightly below LineVul on Python, JavaScript, and C#; the text should say which way.

Second, the sample on the small languages. With an 8:1:1 split per language (§IV-D), and assuming it is stratified, the TypeScript test set holds about $0.1 \times 151 \approx 15$ vulnerable functions, and the C# test set about $0.1 \times 332 \approx 33$. PR-AUC on 15 positives moves a lot when two or three predictions change. This matters most for §V-C, where TypeScript PR-AUC rises "from 0.1653 to 0.2860" after Java is added, and ranges "from 0.1260 to 0.2860" across extension orders. The ratio $0.2860 / 0.1653 = 1.73$ looks large, but I could not tell from the text whether it exceeds run-to-run noise. No result in the paper comes with a seed count, a standard deviation, a confidence interval, or a significance test.

That same gap affects the ablation. §V-B states that full MVD beats the other loss variants "by at least 0.2% in PR-AUC" on average, and beats MVD-binary by 4%. A 0.2% difference from one training run cannot support "These results substantiate the beneficial impact of the FOLA loss function." It may well be real. I would simply like to see it across, say, five seeds before the claim is made.

Third, data leakage deserves a direct check. Labels follow Big-Vul practice (§IV-B): functions touched by a vulnerability-fixing commit are vulnerable, the rest of the functions in those commits are non-vulnerable. The paper does not say whether the 8:1:1 split is random at the function level or grouped by commit, CVE, or project. If it is function-level, near-identical functions from the same project and commit can sit in both train and test. The Python figure of 0.9018 PR-AUC quoted in §V-C (model w/o-C/C++, Python), at a 6.7% positive rate, is high enough that I would want this ruled out. It is also not stated

whether the vulnerable sample is the pre-fix version of the function, and whether the post-fix version is excluded.

The RQ3 conclusion, that extension does little harm to old languages, is reasonably supported by the one example given: Python drops from 0.9018 to 0.8846 when C/C++ is added, i.e., 0.0172 absolute, or $0.0172 / 0.9018 = 1.9\%$ relative. The authors are fair in noting that the result depends on the order and combination of languages.

Finally, as rendered on Qeios, Tables II, III, and IV are not legible in the text I reviewed (they appear as captions only), so I could only check the numbers quoted in the prose. I could not verify the red/blue rankings or the per-metric claims such as "only except Recall in JavaScript".

3. Reproducibility

The authors promise data, models, and code on figshare (§I contributions; Data Availability). The link is a private share link (figshare.com/s/...); I did not open it and cannot say what it contains. A permanent DOI-backed record would be better for a published version.

What the text gives: base model CodeBERT with 12 transformer layers (§III-C), AdamW, an initial learning rate of 2×10^{-5} with cosine annealing (§IV-D), 8:1:1 splits, model selection by best validation PR-AUC, and the LineVul repository used for baselines (footnote 4). Training ran on the Phoenix HPC service (Acknowledgments).

What is missing, and fairly easy to add:

- The values of α_t , γ , and τ in Eq. 1, and how α_t is set per class. These are the core of the paper's loss.
- Batch size, number of epochs, maximum input length, and how functions longer than CodeBERT's window were handled (truncation affects long C/C++ functions far more than short Python ones).
- Random seeds and the number of runs per configuration.
- The exact checkpoint name of CodeBERT and its tokenizer.
- Whether the plain, unweighted sum in Eq. 3 is what was used or a weight between the two terms was tuned, and which inputs feed the old model during distillation when old data are unavailable.
- GPU type and training time. The text says MVD saves "nearly five times more resources" (§V-A), but no time or memory figure supports it, and six models versus one would naively suggest six.
- The CVEfixes snapshot date and NVD cut-off used for curation, plus the list of projects per language.

4. What specifically to fix

1. **Abstract and §I, second contribution.** Lead with the per-language gains (34.9% C/C++, 30.7% Java, 24.9% TypeScript, on par elsewhere) and the -7% gain over the set of six LineVul models (0.5008), and describe the 83.7% to 193.6% range for what it is: a comparison against single-language models applied outside their language.
2. **Eq. 1, the logit adjustment term.** As written, $\tau \log(q_t)$ is added to the loss outside the focal term. Since q_t is a class frequency and does not depend on the model parameters, that term contributes no gradient, and the loss reduces to focal loss. In Menon et al., the adjustment is applied to the logits inside the softmax. If the implementation does that, please correct the equation; if it does exactly what Eq. 1 says, MVD-focal and MVD should coincide up to run-to-run noise, which would fit the small 0.2% gap in §V-B.
3. **Eq. 2.** The squared difference is taken between raw new logits z_i and softmax-normalised old logits $\sigma(z_i^{\text{old}})$. Those live on different scales. I suspect σ should apply to both, perhaps with a temperature as in Learning without Forgetting, which the paper cites; please check against the code.
4. **§III-B.** The paper says CodeBERT uses WordPiece. To my knowledge, the public CodeBERT checkpoint ships a RoBERTa byte-level BPE tokenizer, so the WordPiece versus BPE discussion may describe the wrong tokenizer. Worth checking and correcting.
5. **§III-A** says CodeBERT was pre-trained on CodeSearchNet. That corpus does not include C/C++, C#, or TypeScript, which I think is an interesting finding hiding in your results: three of your six languages are unseen in pre-training. A sentence analysing whether gains differ between seen and unseen languages would add a lot.
6. **§IV-D.** State whether the split is grouped by commit, CVE, or project, and report a deduplication check across splits. If the split is random by function, a project-grouped rerun of RQ1 would make the results much harder to question.
7. **Tables II, III, IV.** Repeat each configuration with several seeds and report the mean and standard deviation, or bootstrap intervals on the test set. This is most needed for TypeScript and C#, where the test sets hold roughly 15 and 33 positives.
8. **§V-B, FOLA paragraph.** Soften "substantiate" unless the 0.2% average gain survives multiple seeds.
9. Table I, JavaScript row: ".jss" is probably meant to be ".jsx". Consider also whether .mjs and .cjs files were captured.

10. **SV-A, third paragraph.** The list "Python, C/C++, Java, and JavaScript, C# and TypeScript" does not make clear which of the six percentages belongs to which baseline. A small table would remove the ambiguity.
11. Explain or remove "nearly five times more resources" (SV-A); a measured training-time comparison would do.
12. **Baselines.** LineVul is a sensible anchor, but the conclusions would be stronger with one newer code model (for example, a CodeT5 or UniXcoder variant fine-tuned the same way) and a simple joint-binary CodeBERT trained with plain cross-entropy, so that the multi-class head and the loss can be separated from the effect of pooling data.
13. **SVI-A** refers to removing "code irrelevant to vulnerability" using the checklist in [48] but does not say which checks were applied or how many functions each removed. A short filtering funnel per language would help.

5. Recommendation

Minor revisions. This is honest, useful work with a new multi-language dataset, clear research questions, and a sensible design, and the per-language gains for C/C++, Java, and TypeScript are encouraging. The fixes I ask for are mostly about framing the headline comparison fairly, correcting Eq. 1 and Eq. 2 so they match the code, and adding seeds and a leakage check. None of them requires a new method, and I expect the core findings to hold.

Declarations

Potential competing interests: No potential competing interests to declare.