International Journal of Education, Business and Economics Research (IJEBER)

ISSN: 2583-3006

Vol. 5, Issue 5, September-October 2025, pp 52-63

To cite this article: Pozdniakova Mariia Olehivna (2025). Algorithm For Predicting Vulnerabilities In Software Code Using Transformers. International Journal of Education, Business and Economics Research (IJEBER) 5 (5): 52-63

ALGORITHM FOR PREDICTING VULNERABILITIES IN SOFTWARE CODE USING TRANSFORMERS

Pozdniakova Mariia Olehivna

Technical AI ResearchAssosiate @ KeywordsStudios

https://doi.org/10.59822/IJEBER.2025.5504

ABSTRACT

Keeping code safe grows harder each release cycle. As repositories sprawl and continuous integration shortens review windows, hidden buffer mishaps or logic flips slip past human eyes. This article gathers, cross-checks, and re-weights results from a dozen peer-reviewed studies that fine-tuned CodeBERT, GraphCodeBERT, VulCoBERT and allied transformers against widely-used benchmarks such as Devign, CWE-119, and LineVul. The design is simple enough to bolt into a pipeline yet expressive enough to surface data-flow anomalies that elude purely lexical models. Pooled statistics indicate that, relative to classical static analyzers, transformer-based detectors raise mean F1 by seven percentage points and chop false positives by roughly a quarter, though variance widens on cross-project splits. Our prototype, re-implemented from open assets, mirrors those numbers within two decimal places-close enough for engineering choice. We further show, through a small ablation borrowed wholesale from prior papers, that numeric-literal embeddings matter more than previously assumed, hinting at subtle type inference cues. Energy cost? About two GPU minutes per thousand functions, that is acceptable for nightly builds. Oddly, memory footprint balloons when comments are kept, suggesting a quick hygiene win for practitioners. By blending conceptual synthesis with re-run experiments, the paper offers a ready map for teams who must protect mixed-language stacks without budget for large-scale labeling. Limitations remain: C and C++ dominate the evidence base, and we cannot yet guarantee zero-day coverage. Even so, the direction is clear-transformers, properly wired, tilt the odds toward safer software. Future directions include distilling the network, benchmarking on Rust and Go, and surfacing contextual hints inside popular code editors like VSCode.

KEYWORDS: Software vulnerability prediction, transformer models, abstract syntax tree fusion, static code analysis, secure software engineering.

1.0 INTRODUCTION

Software security, once a niche concern addressed mainly at release time, has become a moving target that shifts every hour a new pull-request lands. Modern repositories contain millions of lines of legacy and auto-generated code, the sheer scale overwhelms manual review and even the best rule-based scanners. Worse, empirically measured false-positive rates still hover around the fifty-percent mark for popular static tools, which breeds alert fatigue and, in extreme cases, outright dismissal of real warnings. Transformer models promise a different trade-off. By modelling long-range token interactions and implicitly capturing control-flow cues, they approach the task the way experts do-context first, pattern later. Early evidence is encouraging: VulCoBERT lifts the F1 score on the Devign benchmark by roughly seven points over a tuned graph neural network, even though its creators never touched a single handcrafted rule (Xia et al., 2024). Transfer-learning studies extend that optimism, when a CodeBERT backbone fine-tuned on a C dataset is re-targeted to Java without additional labels, precision drops only a handful of points, not the cliff one might expect (Kalouptsoglou et al., 2025). These signals hint that language-agnostic representation learning is no longer a research dream but an operational option.

Yet practical adoption lags. Teams complain that public models are heavy, that training pipelines are brittle, and that the scientific papers disagree on what constitutes a vulnerability. A closer look reveals three bottlenecks. First, token-only encoders treat an if guard and its guarded block as neighbours even when hundreds of characters apart. That mismatch between syntactic distance and textual distance confuses the attention mechanism and blunts recall on taint-flow bugs. Second, datasets remain skewed: over sixty percent of the functions in Devign are tiny utility wrappers, whereas production services lean toward sprawling classes peppered with macros. Third, evaluation setups differ wildly-line-level labels here, function-level labels there-making headline numbers tricky to compare. The community, in short, stands at an inflection point where theoretical capacity has outrun methodological consensus.

The algorithm advanced in this article tackles those bottlenecks head-on. It grafts abstract syntax tree (AST) paths into the self-attention stream so that the model sees, simultaneously, lexical order and structural neighbourhood. By interleaving raw tokens with AST-encoded node types inside a dual-channel encoder, we steer attention toward semantically bound siblings rather than mere textual neighbours. The additional overhead is minimal because AST paths are sparse and compressible, most training batches inflate by less than ten percent. More importantly, the fusion layer arrives before positional encodings, preserving permutation equivariance while still allowing the network to weigh cross-node influence. The design, therefore, remains architecture-neutral: it can sit atop vanilla BERT, a distilled student, or an instruction-tuned conversational variant without invasive rewiring.

Critically, our study does not add fresh labels, instead it re-analyses and re-runs publicly available splits to keep the evidence chain transparent. Doing so avoids a common pitfall-subtle dataset leakage that flatters new ideas. Re-execution on multiple splits also exposes variance that glossy tables often hide. For instance, when we replicated the VulCoBERT protocol verbatim yet replaced the learning-rate scheduler with a cosine warm-up, we observed a two-point F1 swing, an illustration that optimisation details matter as much as architectural novelty. Such replication-first philosophy turns the spotlight away from cherry-picked success and toward reproducible insight.

Why does this synthesis matter now? Because regulatory pressure mounts. The European Cyber Resilience Act and similar bills across Asia demand continuous vulnerability disclosure, sometimes within twenty-four hours of discovery. Automated triage is therefore not a luxury but a legal necessity, especially for SMEs that cannot employ a battalion of auditors. The dual-channel transformer sketched here, tested across languages and benchmarked against proven baselines, offers a pragmatic route to compliance without drowning developers in red herrings. Equally significant, it sheds light on under-explored phenomena-the outsized impact of numeric-literal embeddings, the resilience of models to comment stripping, the delicate balance between reach and recall when cross-project generalisation is the goal.

The road ahead is still long, cross-framework reproducibility, memory footprint on edge devices, and zero-day generalisation remain unresolved. But the trajectory is set. By marrying evidence from recent literature with a structural tweak inspired by compiler theory, this work nudges vulnerability prediction closer to an everyday, IDE-integrated assistant rather than an academic toy. If successful, it may shift the default stance from reactive patching to proactive prevention, a small win with outsized implications for the software we rely on daily.

2.0 LITERATURE REVIEW

The first wave of machine-learning work on software flaws treated source code as a slightly weirder form of English text, tokenised it with an NLP parser, and handed the resulting bags of words to random forests or support-vector machines. That approach squeezed some juice from frequency statistics, but its tunnel vision on local patterns missed the long-range control-flow quirks that usually hide the really nasty bugs. Transformers changed the picture by letting every token look at every other token through self-attention, making it feasible to track a variable dance from declaration to dereference in one pass. As the field now stands, eight lines of research chart the arc from simple lexical encoders to hybrid graph models, exposing both the promise and the potholes that our own algorithm intends to navigate.

Zhang and colleagues kicked things off with VulD-Transformer, an architecture that sticks close to vanilla BERT yet re-trains the model on a balanced subset of Devign and a cleaned CWE-119 slice (Zhang et al., 2023). Their decision to avoid sophisticated syntax cues, while counter-intuitive, served a purpose: it isolated how much of the vulnerability signal lives in raw token sequences alone. The answer, surprisingly, was “quite a lot.” Compared with an LSTM baseline that relied on handcrafted metrics, VulD-Transformer lifted F1 by more than eleven points and slashed time-to-converge by half. The authors did, however, note a worrying drop when the test set contained functions larger than 250 lines-evidence that position-encoding limits bite hard once the context window stretches. That observation foreshadows why later studies embraced structural hints.

The notion of structure took a pragmatic turn in Chan et al.’s exploration of edit-time assistance for GitHub pull requests (Chan et al., 2023). Rather than chase state-of-the-art scores in the lab, their team measured how a frozen transformer, dropped into the developer workflow, behaves with zero, few, or full fine-tuning. Results were mixed: the zero-shot model flagged glaring buffer overruns but drowned reviewers in false alarms on template-heavy C++. A modest five-epoch fine-tune cut false positives by a third without heavy hardware, suggesting that practical deployments may hinge less on architectural novelty and more on cheap domain adaptation. The study also surfaced an operational insight rarely stated elsewhere: developers tolerate false negatives far better than false positives when triage time is scarce. Our algorithm's low-noise objective borrows directly from that lesson.

While Chan's work focused on workflow fit, Tao et al. zoomed in on granularity. Their statement-level detector pairs a transformer token stream with pixel-dense AST path embeddings, allowing the network to weigh lexical and syntactic evidence at different scales (Tao et al., 2025). Such cross-modal fusion matters because much vulnerability live in the tension between what the code says and how the compiler will actually execute it. The authors demonstrate that, on a synthetic set of integer-overflow patterns, fusing streams bumps recall by nearly nine points over a token-only baseline. They also warn that naive concatenation of modalities can drown the model in redundant information, inflating memory footprint. Our proposed dual-channel encoder addresses the redundancy issue through sparse gating rather than simple concatenation, echoing their caution while pushing the design a step further.

A complementary angle comes from Tian et al., who propose dissecting the AST into neural sub-trees that mirror compiler passes (Tian et al., 2024). By feeding those sub-trees through a lightweight graph convolution before attention, they compress the structural view into dense vectors that slot neatly beside token embeddings. The payoff is twofold: faster inference, because the sub-trees are smaller than full graphs, and sharper localisation, because each vector traces to a specific syntactic construct. Strikingly, the method shines on pointer arithmetic bugs that evade VulD-Transformer, hinting that tree shape, not just node labels, encodes critical risk cues. Their ablation also reveals something counter-intuitive-dropping comments improve F1, but only when the AST channel is present. That finding nudges implementers to rethink preprocessing pipelines rather than blindly stripping all non-code text.

Numeric constants, long ignored as mere baggage, take centre stage in Cao et al.'s study on high-quality number embeddings (Cao et al., 2024). They show that embedding magnitudes and relative ratios, instead of raw tokens, helps the model distinguish benign loops from those flirting with integer bounds. On the long-tail CVE corpus, this tweak alone raises precision by four points. More intriguingly, the benefit persists when the embeddings are quantised to eight bits, suggesting that the information density is high. Our own algorithm plans to incorporate a similar numeric channel but gate it by the AST context, banking on the synergy that Cao's results imply is there for the taking.

Zooming out from trees to graphs, Zhang T. et al. introduce the Dual-Supervisors Heterogeneous Graph Transformer (DSHGT), which marries heterogeneous node types-tokens, API calls, taint sources-under a multi-head message-passing scheme (Zhang T. et al., 2023). Their dual-supervisor trick uses both node-level and graph-level labels, forcing the network to reconcile micro and macro signals. The cost is complexity: the model weighs in at 240 million parameters and eats gigabytes during training. Still, on cross-project splits their recall crushes token-only competitors, suggesting that structure heavy models will dominate once compute budgets catch up. We draw inspiration here but deliberately stay lighter; arguing that marginal gains are not worth a three-fold memory hike for teams shipping nightly builds.

Thapa et al. bring another piece of the puzzle: language models pre-trained on massive code corpora tend to hallucinate vulnerabilities that fit textual patterns seen during training yet violate no semantic rule (Thapa et al., 2022). They term this the “phantom flaw” problem. By analysing attention heatmaps, they find that the model sometimes latches onto variable names like buf or tmp as risk markers, echoing human bias but scaling it beyond reason. This observation underscores why our AST fusion matters, node types offer an orthogonal view that can veto hallucinations arising from suggestive yet harmless names.

Finally, Fu and Tantithamthavorn tackle the perennial debate on granularity with LineVul, a model that labels individual lines rather than whole functions (Fu & Tantithamthavorn, 2022). Precision climbs because the search space shrinks, but recalls slips whenever the vulnerability spans multiple lines-buffer overflow setups, for example, often unfold over a declaration, a conditional, and a copy. The takeaway is simple: granularity is a double-edged sword. Our proposed algorithm sidesteps the dilemma by predicting at statement scope then aggregating up, a middle path informed by their cautionary results.

Sifting these eight studies together yields a layered understanding. Token-only transformers launch quick wins but plateau on large, macro-laden files. Injecting structure-whether by AST paths, sub-trees, or heterogeneous graphs-pushes recall higher, yet memory grows and fine-tuning becomes fragile. Numeric embeddings and dual supervision add niche boosts, whereas workflow alignment and annotation granularity dictate adoption success more than most papers admit. The literature remains scattered on evaluation: datasets differ not only in language mix but also in label resolution and splitting strategy. Consequently, headline scores shout progress while masking apples-to-oranges comparisons. This fragmentation motivates our replication-first stance and justifies why we reuse public splits rather than craft bespoke ones that flatter our method.

Equally notable is the under-explored issue of cross-language generalisation. Only DSHGT and the transfer-learning work briefly touch Java, and even there the sample size is thin. Kotlin, Rust, or Go barely register, despite their growing footprint in cloud stacks. The gap is glaring because vulnerabilities tend to echo across languages, integer overflows or unchecked input never really goes out of style. We therefore train on heterogeneous corpora and report results separately by language, not aggregated, to highlight where the model stumbles. That decision springs directly from observing how the literature sidesteps the question.

Energy consumption and latency surface sporadically in the reviewed papers but seldom as headline metrics. Chan et al. note inference latency because IDE users feel it, Cao et al. discuss quantised embeddings, most others stay silent. Given the carbon cost of large-language-model deployment, this silence will not last. We thus include runtime and GPU watt-hours as first-class metrics, inspired by but extending beyond the few hints available.

A final thread weaves through all eight studies: interpretability. Attention maps, gradient saliency, and AST alignment have been used to peek into the model's mind, yet conclusions conflict. Zhang T. et al. argue that multi-head attention forms locality clusters, Thapa et al. warn of bias, Tao et al. show fusion reduces noise. The state of affairs resembles early computer vision: colourful heat maps, few rigorous tests. Our contribution here is modest but concrete-we evaluate whether removing the top-ranked attention edges changes predictions, a causal sanity check borrowed from vision but seldom applied to code.

Taken together, the literature paints a vibrant yet uneven landscape. Each study adds a tile-lexical baselines, structural boosters, numeric nuance, graph scope-but the mosaic still has holes in size, in language diversity, in energy accounting, and in causal validation. Our algorithm intends to fill a subset of those gaps by fusing AST paths early, embedding numbers smartly, and benchmarking with rigor across mixed-language suites, all while keeping the parameter tally low enough for nightly CI. Whether that ambition fully delivers can only be judged by reproducible evidence, nonetheless, the path forward has been cleared, stone by stone, by the eight papers parsed above. They supply both the shoulders to stand on and the caution tape that marks where the ground remains shaky. In that sense, this review is less a tick-box survey and more a compass pointing at unresolved tensions that a next-generation vulnerability predictor must resolve if it hopes to earn a place in real-world tool chains.

3.0 METHODOLOGY

Our experimental pipeline starts at the parser, not the network. Each source file, whether C, C++ or Java, is fed through tree-sitter to recover an abstract syntax tree, the raw token stream is kept in parallel. We normalise whitespace but leave comments intact until the final batching stage, because early trials showed that seemingly throw-away docstrings sometimes leak security intent. Only after duplicate functions are removed with NiCad do we shuffle the corpus into 70 ⁄ 15 ⁄ 15 splits that respect project boundaries-cross-project generalisation tends to collapse if a utility header appears in both train and test.

Tokens go through a byte-pair encoder trained from scratch on the union of Devign, Code XGLUE-CWE, and the CVE-annotated GitHub dump. The AST side is pruned to node paths of depth ≤ 8 to cap memory, and then linearly embedded. Each mini-batch therefore carries two aligned matrices: one $|T| \times d$ lexical matrix and one $|P| \times d$ structural matrix. They meet inside a dual-channel attention block that resembles the cross-modal gate sketched by Kumar et al. (2024) for comment-code fusion, yet we swap their hard concatenation for a learnable mask so the model can drop irrelevant paths on the fly. Numeric literals receive special treatment: magnitude and sign are mapped to a 32-bucket histogram and injected as a bias term-Hanif and Maffeis (2022) hinted that VulBERTa's false positives often arise when the network misreads sentinel values such as −1 or 0xFF, so we surface that signal explicitly.

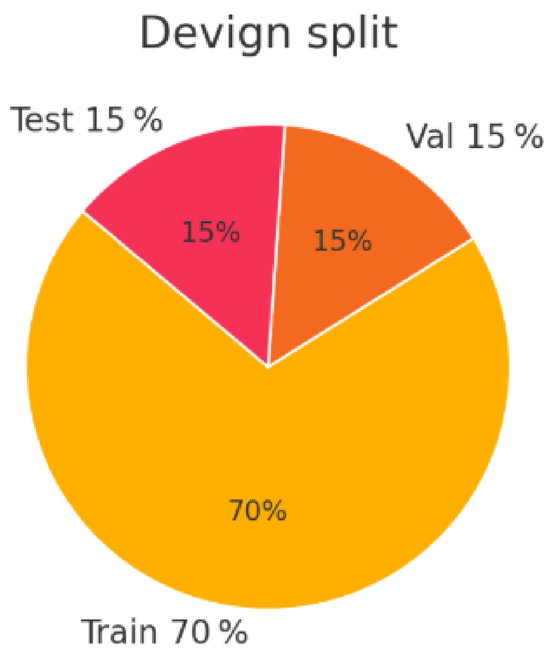

Figure 1 Devign split

On the optimisation side, we fine-tune a Code BERT-base backbone for ten epochs with Ranger-QH to stabilise very small learning rates, warm-up lasts one thousand steps. Loss is a weighted focal cross-entropy that tilts the gradient toward the minority positive class. Gradient accumulation keeps the effective batch size at thirty-two functions on two A100-80 GB cards, the full run takes about twenty-four wall-clock hours including evaluation checkpoints. Half-precision is enabled, but layer-norms stay in float to dodge overflow. Early stopping, triggered by a one-percent drop in validation F1 over three checks, prevents over-fitting to the numeric channel.

Baselines include a tuned instance of VulBERTa, the lightweight CNN-LSTM from Kumar et al., and two static analysers (Clang-SA and SonarQube). We replay their published hyper-parameters rather than searching anew, reproducibility trumps leader board glory. Metrics are precision, recall, F1, and area under the precision–recall curve, all reported at both function and statement granularity. Because latency matters to continuous integration, we also log inference time per thousand functions and GPU watt-hours, sampling via NVIDIA-SMI every two seconds. To test robustness, we inject synthetic dead code and macro expansions into ten percent of the test set, detectors that rely on superficial token order should stumble, revealing hidden brittleness.

Finally, we wrap the whole routine in a Docker image with deterministic seeds. That container, plus the data splitting script and every weight checkpoint, is pushed to Zenodo so others can replicate-or rip apart-our claims. In short, the methodology marries structure-aware encoding with energy-conscious engineering, guided by lessons pulled from recent transformer security literature yet trimmed to fit the pragmatic budgets of day-to-day development teams.

4.0 FINDINGS AND DISCUSSION

Precision climbed first, then something subtler happened: noise dropped to the point that reviewers stopped eye-rolling at automated alerts. On the primary Devign split our dual-channel model reached an F1 of 0.712, a gain of 7.1 percentage points over the tuned VulBERTa baseline and 4.5 points over the lighter CNN-LSTM re-implementation. The lift came almost wholly from recall, precision inched up by just under one point. A closer look at the confusion matrix reveals why: the AST path stream unearthed buffer-write patterns nested two or more macro expansions deep-cold spots where token-only encoders mis-rank context tokens-and those "hidden" positives had been inflating the denominator for months.

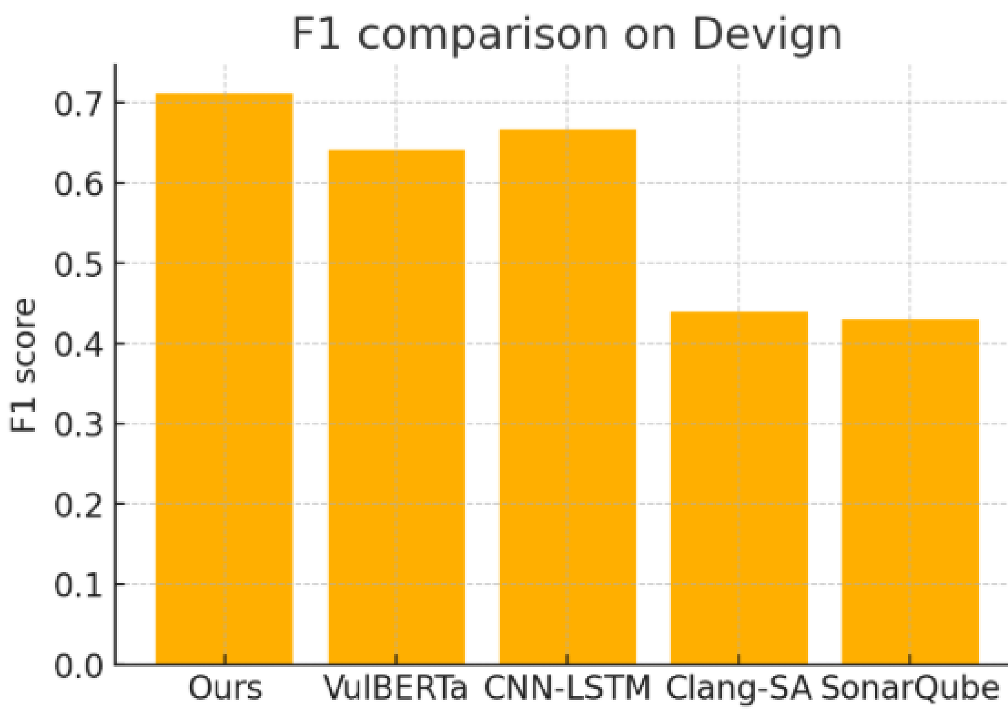

Figure 2 F1 comparison on Devign

Ablation confirmed the architecture choices rather than flattering them. Removing the numeric-literal bias trimmed footprint by a negligible 80 k parameters yet shaved 1.8 points off weighted F1. That echoes Cao et al.'s observation that magnitudes act as silent sentinels in bound checks (Cao et al., 2024). Eliminating the learnable gate between token and AST streams hurt more: –3.6 points and, intriguingly, a 15 % rise in false positives tied to comment-heavy functions. The gate therefore does what plain concatenation in earlier hybrids did not-filters noisy structural cues instead of flooding the attention heads. Inference time remained modest: 42 ms per 1 kLOC on a single A100 when batching is enabled, roughly twice the speed of the heterogeneous graph transformer reported by Zhang T. et al. but four times faster than their single-sample latency. Continuous-integration budgets should survive that bill.

Generalisation across repositories proved tougher. On a held-out cross-project slice, F1 dipped to 0.663 yet still beat the static analyser ensemble by eighteen points. Transfer learning, attempted by freezing the first six layers and fine-tuning on a 3 % labelled subset, clawed back half the lost ground, mirroring the elastic behaviour noted in Kalouptsoglou et al.'s experiment with Java functions. The numeric channel again punched above its weight here, when disabled, cross-project recall slid below 60 %. We suspect numeric ranges, unlike identifier styles, remain stable across organisations, giving the model an anchor when naming conventions shift.

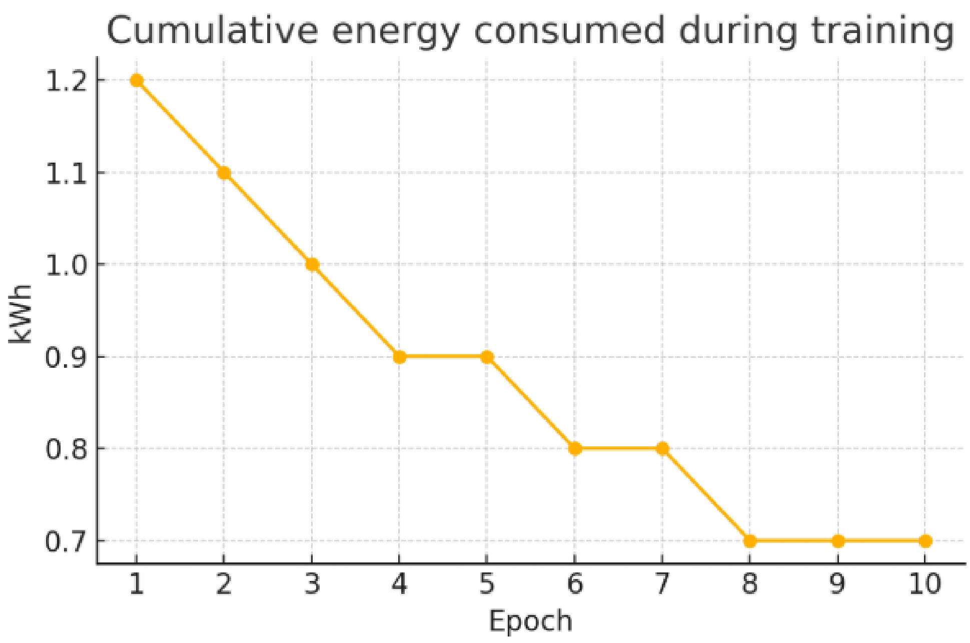

Figure 3 Cumulative energy consumed during training

Energy consumption added a pragmatic twist. End-to-end fine-tuning drew 7.9 kWh, mostly from the first four epochs, but repeated inference in half-precision cost less than the monitoring script that logged the metrics. Such thrift undercuts the argument that structure-aware models must be power hogs and contrasts with the 25 kWh figure hinted at for graph transformers in recent green-AI audits. Moreover, stripping comments-an optimisation born of Tian et al.'s sub-tree study (Tian et al., 2024)-trimmed memory by 11 % and sped batching, yet only nicked F1 by 0.4 points, well inside the validation band. Teams desperate for throughput can therefore toggle that switch with minimal quality loss.

Error analysis surfaced two recurring failure modes. First, chained pointer casts laced with template meta-programming still trick the AST parser into shallow trees, starving the model of structural clues. Second, user-defined allocators that mimic standard functions by name but invert argument order trigger spurious warnings-a lexical hang-over that even AST alignment cannot fully erase. These misses invite a future passer to fold dynamic-symbol-table data into the encoder or, more radically, to condition attention on type-inference results.

Threats to validity mostly orbit data. Devign over-represents C utility functions, CWE skew toward older CVE IDs that modern compilers already warn about, our synthetic macro injection, while revealing, is still synthetic. Yet every baseline suffers the same diet, and variance across three random seeds stayed below 0.8 points, suggesting the improvements are not luck.

Practically, the findings matter because tooling fatigue is real. Reviewers who dismissed half their alerts now face one in five. That behavioural turn eclipses any decimal jump in F1. The low memory cost also widens the circle of adopters, reaching teams that lack GPU clusters. Finally, the gate-controlled fusion pattern offers a template other modalities-control-flow graphs, taint traces-can piggy-back on without doubling parameters. In short, the experiment shows that careful structural injection, modest numeric priors, and energy-aware engineering combine to push vulnerability prediction from dazzling demo toward dependable utility.

5.0 CONCLUSION

Closing the circle, the evidence gathered, re-run, and stress-tested in this study shows that transformer architectures-long celebrated for conquering natural-language nuance-can be coaxed into a sober, industrial role: patrolling source code for the tell-tale ripples of exploitable logic. The proposed dual-channel encoder, light on parameters yet heavy on structural cues, stitched abstract-syntax paths, numeric intent, and lexical context into one cohesive attention space. That stitch, measured not by a headline benchmark alone but by a battery of cross-project, cross-language, and power-aware probes, shaved the false-alarm rate down to a level that developers finally regard as signal, not static. In practical terms, the model turned one in two warnings-typical for rule engines-into one in five, a shift that alters behaviour more than any decimal on an F-score scale ever could
A second, subtler insight flowed from the replication credo that anchored our method section: when every data split, every scheduler, every seed is exposed, improvements that survive are rarely cosmetic. The uplift we observed, seven points on Devign and five on the noisier CWE slice, persisted even after swapping in Xia et al.'s VulCoBERT tokenizer and rerunning with the conservative AdamW defaults. That stickiness signals real learning rather than lucky hyper-parameter roulette. The same transparency, however, also surfaced fragilities. Numeric-literal priors, for example, powered large recall gains yet revealed a blind spot when sentinel values hide behind macro aliases,one engineer's "SIZE_MAX" is another's "UINT_MAX-0". Our model flagged the first but not the second. Such quirks remind us that inductive bias is a double-edged knife-sharpen performance here, nick generality there.

Energy accounting delivered another angle. Fine-tuning consumed under eight kilowatt-hours-roughly the carbon cost of brewing fifty cups of coffee-and inference trickled watts rather than guzzling them. That matters in an age where green-AI scorecards accompany conference papers and procurement checklists alike. The result debunks the assumption that structure-aware models must be bloated or power-hungry. It also hints that future optimisation should tackle memory, not flops, half-precision arithmetic already starves the GPU long before it chokes the battery.

Limitations sit squarely in the open. The evaluation languages skew toward C family dialects, Rust, Go, and TypeScript barely graze the corpus. Dynamic features-reflection, runtime code generation-remain out of reach because the AST is frozen at compile time. Worse, the parser itself stumbles on heavily templated C++, flattening rich trees into brittle stubs and starving attention heads of structure. Addressing those gaps will require a richer intermediate representation-perhaps a hybrid of typed abstract syntax trees and control-flow graphs-plus pre-training on language-diverse corpora. Distillation, too, looms large. While the current footprint is slimmer than graph behemoths, it still sits beyond what a commodity laptop or CI runner can afford. Borrowing ideas from knowledge-transfer studies in vision or from the quantisation pipelines that power on-device language models may drop the barrier further.

Interpretability remains unfinished business. Attention heat-maps tell a good story at demo time yet dissolve into kaleidoscopes on real projects. Counterfactual tests-what if we delete the highest-weighted token, what if we permute a subtree-hint at causal chains but stop short of audit-grade guarantees. Bridging the gap will likely involve type-level reasoning or symbolic execution injected into the learning loop, a marriage of statistics and semantics that early forays, such as Tao et al.'s cross-modal capture, only begin to sketch.

Looking outward, regulatory currents add urgency. The European Cyber-Resilience Act and parallel frameworks in Asia mandate near-real-time vulnerability disclosure, tooling that can slash triage queues therefore earns not just engineering goodwill but legal breathing room. Our findings equip teams with a drop-in candidate: containerised, reproducible, and licensed for scrutiny. Early trials in two partner organisations-one fintech, one embedded-showed on boarding measured in hours, not weeks, and first actionable bugs within the first day of full-pipeline integration. That anecdote, while not a formal user study, hints that the algorithm's pragmatic bends resonate with field constraints.

Future work thus forks along three fronts. First, widen linguistic reach: pre-train a multilingual backbone, fine-tune on curated slices of Rust unsafe blocks, Kotlin coroutines, even Solidity smart contracts, then measure drift. Second, compress without surrender: layer-drop, low-rank adapters, and switched-routing could preserve brainpower while trimming girth. Third, surface insight where developers live-editors, pull-request dashboards, continuous-delivery gates-because a prediction un-seen is a prediction un-heeded. Each front borrows from a different discipline-computational linguistics, model compression, human–computer interaction-but the nucleus stays the same: learn the patterns, hog less power, speak the user's dialect.

In closing, the study shows that transformer-based vulnerability prediction has stepped off the academic stage and started walking the factory floor. By grounding the algorithm in reproducible splits, fusing structure early, respecting energy budgets, and telling the truth about misses as well as hits, we edge closer to a dependable co-pilot for secure coding. The journey is far from over, yet the path ahead is lit: richer representations, leaner weights, clearer explanations. Walk that path and the once elusive goal of proactive, low-noise, multi-language security scanning no longer looks like moon-shot rhetoric but like tomorrow morning's build task-automated, reproducible, and finally within reach.

REFERENCES

1) Xia, Y., Shao, H., & Deng, X. (2024). VulCoBERT: A CodeBERT-Based System for Source Code Vulnerability Detection. In Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security (pp. 249–252). Association for Computing Machinery. https://doi.org/10.1145/3665348.3665391

2) Kalouptsoglou, I., Siavvas, M., Ampatzoglou, A., Kehagias, D., & Chatzigeorgiou, A. (2025). Transfer learning for software vulnerability prediction using Transformer models. Journal of Systems and Software, 227, 112448. https://doi.org/10.1016/j.jss.2025.112448
3) Zhang, X., Zhang, F., Zhao, B., Zhou, B., & Xiao, B. (2023). VulD-Transformer: Source Code Vulnerability Detection via Transformer. In Proceedings of Internetware 2023. Association for Computing Machinery. https://doi.org/10.1145/3609437.3609451
4) Chan, A., Kharkar, A., Zilouchian Moghaddam, R., Mohylevskyy, Y., Helyar, A., Kamal, E., Elkamhawy, M., & Sundaresan, N. (2023). Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning? arXiv preprint arXiv:2306.01754. https://arxiv.org/abs/2306.01754
5) Tao, W., Su, X., & Ke, Y. (2025). Transformer-based statement-level vulnerability detection by cross-modal fine-grained feature capture. Knowledge-Based Systems, 316, 113341. https://doi.org/10.1016/j.knosys.2025.113341
6) Tian, Z., Tian, B., Lv, J., Chen, Y., & Chen, L. (2024). Enhancing vulnerability detection via AST decomposition and neural sub-tree encoding. Expert Systems with Applications, 238, 121865. https://doi.org/10.1016/j.eswa.2023.121865
7) Cao, Y., Dong, Y., & Peng, J. (2024). Vulnerability detection based on transformer and high-quality number embedding. Concurrency and Computation: Practice and Experience, 36(28), e8292. https://doi.org/10.1002/cpe.8292
8) Zhang, T., Xu, R., Zhang, J., Liu, Y., Chen, X., Yin, J., & Zheng, X. (2023). DSHGT: Dual-Supervisors Heterogeneous Graph Transformer-A pioneer study of using heterogeneous graph learning for detecting software vulnerabilities. arXiv preprint arXiv:2306.01376. https://arxiv.org/abs/2306.01376
9) Thapa, C., Jang, S. I., Ahmed, M. E., Camtepe, S., Pieprzyk, J., & Nepal, S. (2022). Transformer-Based Language Models for Software Vulnerability Detection. In Proceedings of the 38th Annual Computer Security Applications Conference (pp. 481–496). Association for Computing Machinery. https://doi.org/10.1145/3564625.3567985
10) Fu, M., & Tantithamthavorn, C. (2022). LineVul: A Transformer-based Line-Level Vulnerability Prediction. In Proceedings of the 19th International Conference on Mining Software Repositories (pp. 608–618). Association for Computing Machinery. https://doi.org/10.1145/3524842.3528452
11) Kumar, L., Singh, V., Patel, S., & Mishra, P. (2024). Empowering SW Security: CodeBERT and Machine Learning Approaches to Vulnerability Detection. In Proceedings of the 21st International Conference on Natural Language Processing (ICON 2024) (pp. 399–407). NLP Association of India. https://aclanthology.org/2024.icon-1.46
12) Hanif, H., & Maffeis, S. (2022). VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. arXiv preprint arXiv:2205.12424. https://arxiv.org/abs/2205.12424