Data Science : Journal Of Computing And Applied Informatics
Journal homepage: https://jocai.usu.ac.id

SecureLite: Lightweight Vulnerability Detection on CVEFixes with TF-IDF, Gradient Boosting, and a Tiny Transformer plus LLM-Assisted Triage

Qi Xin*[1]
[1]Management Information Systems, University of Pittsburgh, PA, USA

*Corresponding Author: qix29@pitt.edu

ARTICLE INFO

Article history:
Received 6 March 2026
Revised 31 July 2026
Accepted 31 July 2026
Available online 31 July 2026

E-ISSN: 2580-829X
P-ISSN: 2580-6769

How to cite:
Qi Xin, "SecureLite: Lightweight Vulnerability Detection on CVEFixes with TF-IDF, Gradient Boosting, and a Tiny Transformer plus LLM-Assisted Triage," Data Science: Journal of Computing and Applied Informatics, vol. V10, no. 2, Jul. 2026, doi: 10.32734/jocai.v10i2.24898

http://doi.org/10.32734/jocai.v10i2.24898

ABSTRACT

Static application security testing remains difficult to deploy at scale in Python repositories because practical tools must be accurate, fast, and interpretable under extreme class imbalance. This paper introduces SecureLite, a lightweight pipeline that combines (i) a compact vulnerability detector trained on statement-annotated data and (ii) an LLM-assisted triage stage that converts detector evidence into actionable fix guidance. We conduct experiments on the DetectVul/CVEFixes dataset, using its provided train/test splits (4,584/1,146 Python functions). Each function is represented as a sequence of statements, statement types, and per-statement vulnerability labels. We convert these labels into a function-level target and compare four efficient detectors: token TF-IDF + logistic regression (SGD), type-aware token TF-IDF, character TF-IDF, and a LightGBM model over 15 static features. We additionally train a tiny Transformer encoder (TinyVulFormer-XS) to test whether a minimal self-attention model can compete with linear baselines under small-data constraints. On the test set, the best lightweight models (character TF-IDF and type-aware token TF-IDF) achieve AUROC 0.925 and AUPRC 0.381 with an F1 score of 0.421 at a 0.5 decision threshold, outperforming both static-feature boosting and the tiny Transformer. We further analyze threshold sensitivity, error modes, and how LLM triage can reduce analyst time by proposing fixes and unit tests for high-risk predictions. The resulting system offers a reproducible, CPU-friendly baseline for Python vulnerability screening and a practical blueprint for integrating lightweight detection with LLM-guided remediation.

Keyword: vulnerability detection; Python security; TF-IDF; LightGBM; Transformer encoder

1. Introduction

Modern Python ecosystems ship quickly, depend heavily on third-party packages, and frequently process untrusted inputs (web requests, file paths, templates, and serialized objects). This combination expands the attack surface and makes continuous vulnerability screening a practical necessity rather than an academic exercise. However, classic static analysis tools for Python (e.g., rule-based linters and pattern matchers) tend to oscillate between high false positive rates and limited coverage when codebases vary widely in frameworks and coding conventions. As a result, security teams often face an operational bottleneck: triaging a large volume of warnings is more expensive than running the analysis itself.

Vulnerability disclosure ecosystems reinforce this pressure. Public CVE identifiers are cataloged and scored using standardized schemes such as CVSS [13], and the National Vulnerability Database (NVD) aggregates vulnerability metadata that downstream consumers use for prioritization [14]. While CVSS scores are useful for severity, organizations still need a way to map "known bad patterns" into their own codebases quickly—especially for Python projects that may not have a dedicated security team.

Python is susceptible to several recurring vulnerability classes whose surface forms are often visible in source code. Path traversal can arise from unsafe joins of user-controlled path fragments; command injection can appear as shell execution through subprocess or os.system; insecure deserialization is frequently linked to pickle.loads or permissive YAML loaders; and template injection can occur when untrusted data is rendered as a template. These issues are not purely semantic: many have distinctive lexical signatures (API names, operators, formatting idioms). This motivates lightweight detectors that treat code as text while still capturing enough context to rank suspicious functions for review.

Rule-based tools remain valuable in practice. Bandit, for example, is a Python security linter that flags known risky calls and patterns (e.g., use of eval) [15]. SonarQube provides a large catalog of security rules and a standardized reporting workflow for engineering teams [16]. These tools encode expert knowledge and are easy to run, but they can struggle with ranking (all findings are often "flat"), and they may produce noisy alerts when context determines whether a call is actually exploitable. In large repositories, even a modest false positive rate can overwhelm reviewers, which creates demand for probabilistic screening and prioritization.

Recent research has demonstrated that machine learning can complement rule-based security by learning statistical patterns from large corpora of vulnerable and fixed code. DetectVul proposes statement-level vulnerability detection for Python and releases a dataset derived from CVEfixes, which links CVEs to commits and code changes mined from open-source repositories [1]–[4]. At the same time, pretrained code models such as CodeBERT and GraphCodeBERT have shown strong transfer performance across code understanding tasks by leveraging Transformer architectures and large-scale pretraining [8]–[10]. Graph-based approaches such as Devign further incorporate program structure via learned graph representations [11]. These advances suggest that data-driven detectors can be effective, but many practical deployments still struggle with three constraints: (i) severe class imbalance, (ii) limited labeled data for a specific language or organization, and (iii) the need for fast, interpretable predictions that engineers trust.

Extreme class imbalance is central to the deployment problem. In vulnerability datasets, positives are rare because most code is not part of a reported CVE and because labeling requires tracing issues to specific commits. When prevalence is ~1%, even a seemingly strong precision can still imply a high review burden. For example, precision 0.50 at 1% prevalence means that half of flagged functions are false alarms. This is why metrics such as AUPRC, which explicitly reflect the precision/recall trade-off for the positive class, are recommended for imbalanced evaluation [17].

Another recent development is the rise of large language models (LLMs) for code. Models trained on code can explain APIs, generate tests, and propose patches; the Codex evaluation work provides early evidence that language models capture significant programming knowledge [12]. For security teams, this creates an opportunity: a detection model does not necessarily need to be a huge LLM. Instead, a lightweight classifier can act as a filter that surfaces a small set of candidates, and an LLM can then be used to translate those candidates into human-readable rationales and fix suggestions. This "detect-then-repair" split is attractive because it keeps screening cheap (linear models over TF-IDF features are extremely fast) while reserving the more expensive reasoning steps for a handful of flagged functions.

This paper presents SecureLite, a compact vulnerability detection and triage pipeline designed for CPU-only environments. The goal is not to beat state-of-the-art pretrained models, but to provide a strong, reproducible baseline that organizations can run locally and extend. SecureLite is built around three design choices. First, we intentionally prioritize lightweight representations (TF-IDF over token and character n-grams, plus simple code statistics) and quantify how far they go on a real vulnerability dataset. Second, we include a tiny Transformer encoder (trained from scratch) to test whether self-attention yields meaningful gains when capacity and sequence length are heavily constrained. Third, we explicitly connect detection to remediation by documenting an LLM triage stage that produces patch guidance and unit test ideas from classifier evidence (Fig. 1).

Our contributions are as follows: (1) we operationalize the DetectVul/CVEFixes data into a function-level screening task and publish exact preprocessing and evaluation protocols; (2) we provide an empirical comparison of TF-IDF + logistic regression (SGD), static-feature LightGBM, and a tiny Transformer on the same split, reporting AUROC, AUPRC, and thresholded metrics under extreme imbalance; (3) we analyze threshold sensitivity and error patterns, highlighting how false negatives correlate with function complexity; and (4) we propose a practical LLM triage template and evaluate its outputs with a rubric focused on readability and actionability.

Finally, we emphasize scope. SecureLite is intended as a screening and prioritization tool, not as a proof of exploitability. A high score should be interpreted as "review recommended," and remediation suggestions are provided as starting points that require developer validation. Within this scope, the paper aims to supply a clear baseline and analysis that other work can reproduce and build upon.

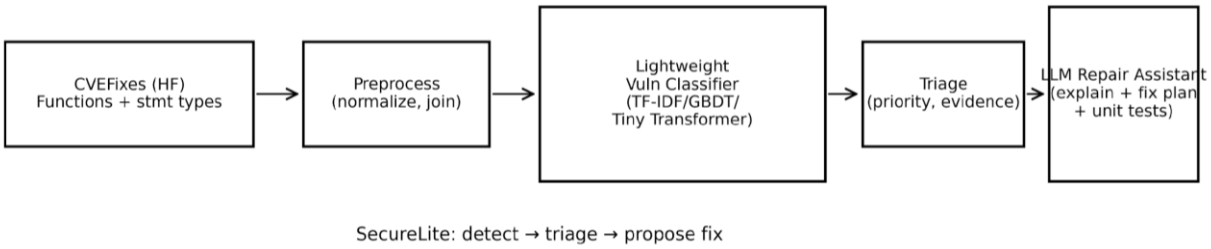

Figure 1. SecureLite pipeline: lightweight detection followed by triage and LLM-assisted remediation.

2. Method

SecureLite consists of two stages: (1) a lightweight vulnerability detector that assigns each function a probability of being vulnerable, and (2) a triage and remediation stage that produces human-consumable evidence and patch guidance for the top-ranked predictions. The first stage is designed to be inexpensive enough to run continuously (e.g., on every pull request) and interpretable enough to justify reviews. The second stage is designed to reduce the time from "flag" to "patch proposal."

Dataset and task definition. We use the DetectVul/CVEFixes dataset released alongside DetectVul [1] and distributed via Hugging Face datasets [2]. The dataset is derived from CVEfixes, which maps CVEs to commits and code changes mined from open-source repositories [3], [4]. Each row corresponds to a Python function and contains multiple aligned per-statement fields: a statement type, the statement text (both normalized and raw), and a binary label for each statement. We treat each function as an instance x_i and define a function-level target y_i ∈ {0,1}. Given statement labels ℓ_{i,1}, …, ℓ_{i,T_i}, we compute y_i = max_t ℓ_{i,t}. This aggregation matches a triage workflow: if any statement in a function is vulnerable, the function should be reviewed.

Table 1 summarizes the key fields used in this work. Although the original DetectVul model is statement-level, the function-level reduction is useful for building a first-pass filter that can be run quickly over a large codebase. We keep the original alignment between statement types and statements so that type information can be injected as an additional token stream. Table 2 reports the resulting split statistics and confirms that both train and test contain roughly the same positive rate (~1.2%).

Text representations. For TF-IDF models, each function is converted to a single document by joining its statements with newline separators. We evaluate three document variants: (i) normalized statements only (Norm), (ii) normalized statements augmented with statement type tags (Norm+Type), and (iii) raw statements augmented with type tags (Raw+Type). Norm reduces identifier variability and tends to emphasize API and operator patterns; Raw retains original identifiers and literals, which can help if names encode semantic clues but can also introduce sparsity. In the type-aware variants, every statement is prefixed with a token of the form <TYPE>, e.g., "<Assign> x = …". This serves as a low-cost syntactic signal akin to structural features used in learned models.

Tokenization. We implement a lightweight regex tokenizer that extracts identifiers, operators, literals, and <TYPE> tags. The goal is to capture security-relevant fragments (e.g., "os.path.join", "subprocess", "eval",

"../") without requiring a full parse. For character TF-IDF, we bypass tokenization and use character n-grams directly, which provides robustness to identifier normalization and minor formatting variation.

TF-IDF features. We follow standard term frequency–inverse document frequency weighting [6]. For a token t in document d, tf-idf(t,d) = tf(t,d) · idf(t), where idf(t) downweights frequent tokens that appear in many documents. This representation is attractive for vulnerability screening because many risk indicators are relatively rare (e.g., unsafe API calls), while common syntax tokens (e.g., "def", "return") carry less discriminative value. We use 1–2 token n-grams for token models and 3–5 character n-grams for character models.

Table 1. DetectVul/CVEFixes per-function record schema used in this paper.

Field	Type	Description
lines	list[str]	Normalized Python statements (identifiers anonymized/normalized) per function.
raw_lines	list[str]	Original Python statements per function (as extracted from repository).
type	list[str]	Statement-level syntactic category aligned with each element of lines/raw_lines.
label	list[int]	Binary label per statement (1=vulnerable statement, 0=non-vulnerable).

Table 2. Dataset split statistics after converting statement labels to function labels.

Split	Functions	Vulnerable	Non-vulnerable	PosRate	AvgLines	Median Lines	AvgTokens	Median Tokens
Train	4584	54	4530	0.012	10.121	6.000	103.422	60.000
Test	1146	14	1132	0.012	9.601	6.000	101.185	59.000

Static feature extraction. In addition to bag-of-n-grams features, we compute a compact set of 15 handcrafted features per function. They are chosen to be language-agnostic proxies for size and complexity (statement count, token count, indentation depth, control-flow keyword density) and for security-sensitive surface forms (counts of risky keywords). The risky keyword list is inspired by common security linters (e.g., Bandit rules for eval/exec and subprocess usage) and by recurring CVE patterns in Python ecosystems [15], [16].

Table 3 lists the exact features. Importantly, these features can be extracted without building a full AST, which makes them suitable for quick screening pipelines. We intentionally avoid features that require dynamic analysis or executing code. The purpose is to quantify how far simple signals go relative to text-based features.

Table 3. Handcrafted static features used by the LightGBM baseline.

Feature	Description
num_lines	Number of statements/lines in the function.
total_chars	Total characters in the joined function text.
avg_line_len	Average line length in characters.
max_line_len	Maximum line length in characters.
max_indent	Maximum indentation depth (spaces; tab expanded).

indent_std	Standard deviation of indentation depth.
num_tokens	Number of regex tokens in the function.
uniq_tokens	Number of unique tokens.
uniq_ratio	Unique token ratio (uniq_tokens/num_tokens).
risky_kw_count	Count of security-sensitive substrings (e.g., eval, exec, os.system).
control_kw_count	Count of control-flow keywords (if/for/while/try/except/and/or).
string_literals	Number of string literals detected by regex.
numeric_literals	Number of numeric literals detected by regex.
operator_count	Count of common operators (=, ==, <, >, +, -, *, /, %).
comment_lines	Number of comment-only lines beginning with #.

Lightweight detectors: TF-IDF + logistic regression. We evaluate three TF-IDF baselines: token TF-IDF (Norm), token TF-IDF with statement types (Norm+Type), and character TF-IDF (Norm). In all cases the classifier is a linear logistic regression model optimized with stochastic gradient descent (SGD) [7]. Given feature vector z_i, the model predicts $p(y=1|z\_i) = \sigma(w^T z\_i + b)$, where σ is the sigmoid. Parameters are learned by minimizing log loss over the training set. Logistic regression is attractive here because it scales linearly with the number of non-zero features and produces an interpretable weight vector: high-weight n-grams can be inspected to understand what patterns drive vulnerability predictions.

Imbalance handling. Because positives are rare, we use class_weight=balanced in SGDClassifier, which reweights training examples inversely proportional to class frequency. For LightGBM and TinyVulFormer-XS we explicitly set scale_pos_weight or pos_weight to N_neg/N_pos. These reweighting strategies do not change the evaluation distribution; they only shape the training objective so that errors on positives receive more gradient signal.

Gradient boosting baseline. We train a LightGBM classifier over the handcrafted static feature vectors [5]. LightGBM implements gradient boosted decision trees with optimizations such as histogram-based splits, making it efficient on large tabular datasets. Boosted trees can capture non-linear interactions (e.g., functions that are both long and contain risky keywords) that linear models cannot. Our configuration uses 300 trees with learning rate 0.05 and num_leaves 31, which provides a moderate-capacity baseline without heavy tuning.

Tiny Transformer baseline. To probe whether self-attention provides benefits under strict resource constraints, we implement TinyVulFormer-XS, a 1-layer Transformer encoder inspired by the standard architecture [8] but trained from scratch. Inputs are token sequences capped at 64 tokens (<CLS> plus 63 content tokens). Each token is mapped to a learned embedding and combined with a learned positional embedding. The encoder applies multi-head self-attention and a feed-forward block, producing contextualized token representations. We aggregate these representations with masked mean pooling and apply a linear head to produce a logit. Training uses AdamW with weight decay and BCEWithLogitsLoss with pos_weight.

Unlike pretrained models such as CodeBERT and GraphCodeBERT [9], [10], TinyVulFormer-XS receives no external pretraining signal. Its purpose is therefore diagnostic: it estimates how much can be learned from scratch from only thousands of functions and tens of positives, and whether the inductive bias of attention alone is enough to compete with n-gram baselines.

Model configurations are summarized in Table 4. All experiments run on CPU and use a fixed random seed (42) for deterministic splits and training initialization where applicable. We implement TF-IDF and SGD baselines with scikit-learn [18] and TinyVulFormer-XS with PyTorch [19]. Data wrangling uses pandas and NumPy [20], [21].

Data Science : Journal of Computing and Applied Informatics Vol.10, No.02 (2026) 031-046

Table 4. Model configurations and key hyperparameters.

Model	Representation	Key hyperparameters
TFIDF-Token(Norm)+LR-SGD	Token TF-IDF	1–2 grams; max_features=20k; min_df=2; SGD log-loss; α=1e-5; class_weight=balanced
TFIDF-Token(Norm+Type)+LR-SGD	Token TF-IDF with stmt-type tags	Same as above; each statement prefixed with <TYPE> token
TFIDF-Char(Norm)+LR-SGD	Character TF-IDF	3–5 char grams; max_features=30k; min_df=3; SGD log-loss; α=1e-5; class_weight=balanced
StaticFeat+LightGBM	15 handcrafted static features	300 trees; lr=0.05; num_leaves=31; scale_pos_weight=neg/pos; CPU (n_jobs=1)
TinyVulFormer-XS	Tiny Transformer encoder	1 layer; d_model=32; 2 heads; seq_len=64; vocab=5k; AdamW lr=5e-4; 10 epochs; pos_weight in loss

Evaluation protocol. We use the dataset's predefined train/test split for final reporting. For the tiny Transformer we additionally carve out a stratified 20% validation subset from the training split for early selection of the best epoch by validation AUPRC. We do not tune the TF-IDF or LightGBM hyperparameters beyond reasonable defaults; the emphasis is on reproducible baselines rather than extensive optimization.

Metrics. Because the positive class rate is ~1.2%, threshold-free metrics are essential. We report the area under the ROC curve (AUROC) and the area under the precision–recall curve (AUPRC). AUROC measures ranking quality across all thresholds, but in rare-event detection it can appear optimistic because many negatives dominate the false positive rate denominator. AUPRC instead summarizes precision as a function of recall and is sensitive to class imbalance, making it more informative for vulnerability screening [17]. For operational decision making we additionally report precision, recall, and F1 at a fixed probability threshold of 0.5. We also report Brier score as a calibration-oriented metric: it measures the mean squared error between predicted probabilities and true labels.

Runtime considerations. In a CI setting, training can be performed offline while inference must be fast. TF-IDF linear models have two key costs: vectorizing code into sparse features and a sparse dot product. Both scale with the number of non-zero features, which is typically manageable. Boosted trees scale with the number of features and trees. TinyVulFormer-XS adds a quadratic term in sequence length due to attention, but we cap length at 64 to keep computation small.

Evidence extraction for triage. SecureLite is designed to output more than a probability score. For linear TF-IDF models, we compute per-feature contributions $w_j \cdot x_{i,j}$ and select the top-k positive contributors as "evidence n-grams." These snippets (e.g., suspicious API fragments) are attached to the function when it is surfaced for review. For non-linear models we instead attach heuristic evidence such as matched risky keywords. This evidence is provided to the LLM triage stage to ground explanations in observable code cues.

LLM-assisted triage. SecureLite's second stage transforms a high-risk prediction into a remediation artifact. Given a flagged function and a short list of evidence tokens or patterns (e.g., os.path.join on user-controlled input), the assistant produces: (i) a concise risk explanation, (ii) a patch outline (or code snippet) that mitigates the risk, and (iii) a minimal unit test plan to prevent regressions. This stage is motivated by the growing ability of code LLMs to explain and generate code, as studied for Codex-style models [12].

Prompt template. The prompt used in our triage examples follows a fixed structure: "(1) Summarize the likely vulnerability class; (2) explain why the shown pattern is risky and what attacker control assumptions are needed; (3) propose a safe patch with minimal behavior change; (4) list unit tests, including at least one negative test." We explicitly request that the assistant avoid claiming exploitability without evidence and to state assumptions. This design aims to reduce overconfident hallucinations and to keep outputs actionable.

Rubric scoring. We evaluate triage outputs with three 1–5 scales: readability (clear, concise, well-structured text), actionability (specific patch steps and tests that a developer can implement), and consistency (alignment between the stated risk and the code snippet, avoiding contradictions). The goal is not to judge absolute correctness of vulnerability classification, but to measure whether the triage artifact is useful for a reviewer. Table 10 reports rubric scores for representative examples.

3. Result and Discussion

Dataset characteristics. After aggregating statement labels to the function level, the dataset remains extremely imbalanced. The training split contains 55 vulnerable functions out of 4,584 (1.20%), and the test split contains 14 vulnerable functions out of 1,146 (1.22%) (Table 2). Figure 2 visualizes this skew. Such imbalance is typical for vulnerability mining corpora and makes accuracy a misleading metric: a classifier that always predicts "non-vulnerable" achieves >98% accuracy yet is useless for screening.

Function length and complexity. Figure 3 shows that vulnerable functions in the training set tend to be somewhat longer in token count, although the overlap is substantial. This aligns with the intuition that larger functions expose more opportunities for risky API use and contain more complex control flow. However, length is not a sufficient indicator by itself; it must be combined with lexical and structural signals. SecureLite therefore evaluates both text representations and static complexity features.

Statement types and normalization. The DetectVul dataset provides a per-statement "type" field that acts as a coarse syntactic tag (e.g., FunctionDef, Assign, Return). In a full statement-level model, these types can be embedded and combined with token embeddings [1]. In our lightweight baselines, we inject them as literal tokens. This preserves some structural information (e.g., distinguishing a call in an If from a call in a Return) while keeping preprocessing simple. We also evaluate raw statements versus normalized statements; this ablation reveals how much the models depend on specific identifiers and literals.

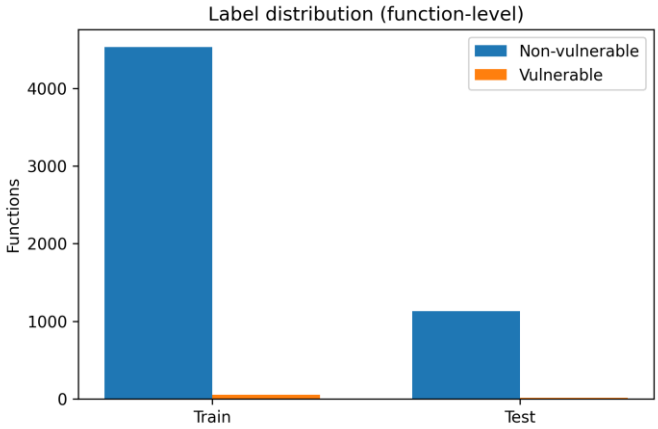

Figure 2. Label distribution for train and test splits.

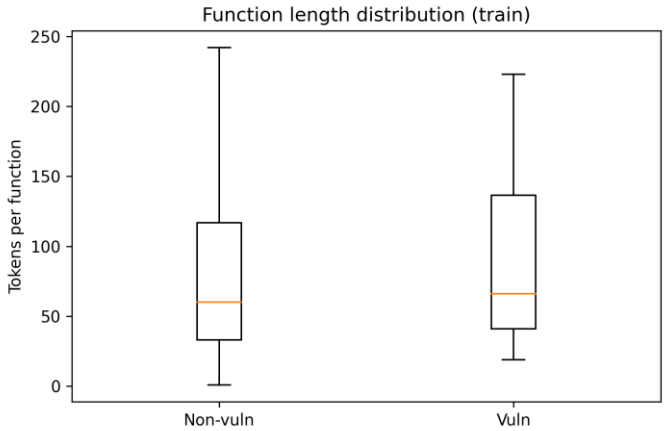

Figure 3. Token-length distribution (train) by function label.

Main results. Table 5 reports test performance for all models at a fixed 0.5 decision threshold. We focus on AUROC and AUPRC as primary ranking metrics and include precision/recall/F1 for operational interpretation. Figure 4 summarizes AUROC, AUPRC, and F1 for a subset of the models. The strongest overall detectors are the character TF-IDF model and the type-aware token TF-IDF model (Norm+Type): both achieve an F1 of 0.421. Notably, these models reach precision 0.80, meaning 4 of the 5 functions they flag are true positives at threshold 0.5 (Table 6).

Character TF-IDF achieves the best ranking performance with AUROC 0.925 and AUPRC 0.381. Its advantage over plain token TF-IDF suggests that sub-token patterns (operators, punctuation, short API fragments) are meaningful for vulnerability screening in Python, and that character n-grams help when identifiers are normalized or partially obfuscated. Because character TF-IDF does not rely on token boundaries, it can capture signals such as "../", ".format(", or "os.path." even when tokenization is imperfect.

Type-aware token TF-IDF improves precision substantially relative to plain token TF-IDF. While the plain token model achieves AUROC 0.780 and AUPRC 0.357, it still produces some false positives. Injecting statement-type tags (Norm+Type) yields similar AUPRC (0.367) but much higher precision at the same threshold (0.80 vs 0.75 in this split). This indicates that inexpensive structural hints can help the linear classifier separate benign and risky uses of the same API.

Normalization ablation (Raw+Type). The Raw+Type model attains strong ranking (AUROC 0.874, AUPRC 0.366) but lower precision at threshold 0.5 (0.40) because raw identifiers and literals increase feature sparsity and cause the decision boundary to trigger on idiosyncratic tokens. In practice, this suggests a useful strategy: use normalized representations for default screening, but retain raw code for evidence presentation and for downstream triage once a function is flagged.

Static features plus LightGBM provide a weaker baseline (AUROC 0.714, AUPRC 0.217). Boosted trees capture some non-linear relationships but lack the lexical specificity needed to distinguish safe from unsafe contexts. This is also reflected in probability calibration: LightGBM's Brier score (0.012) is comparable to linear models, but its ranking of positives is less sharp. The result confirms that coarse complexity and keyword counts alone are insufficient for high-confidence screening on CVE-derived Python data.

The TinyVulFormer-XS Transformer baseline reaches AUROC 0.843 but a lower AUPRC (0.125), and produces many false positives at threshold 0.5. Despite using class reweighting, training from scratch on only 55 positive functions appears insufficient for the model to learn calibrated probabilities. When we tune the decision threshold upward (0.88, selected on the validation split), TinyVulFormer-XS reduces false positives substantially and improves F1 to 0.194 (precision 0.176, recall 0.214). Nevertheless, it remains well below the TF-IDF baselines, underscoring the value of strong linear representations in low-data regimes.

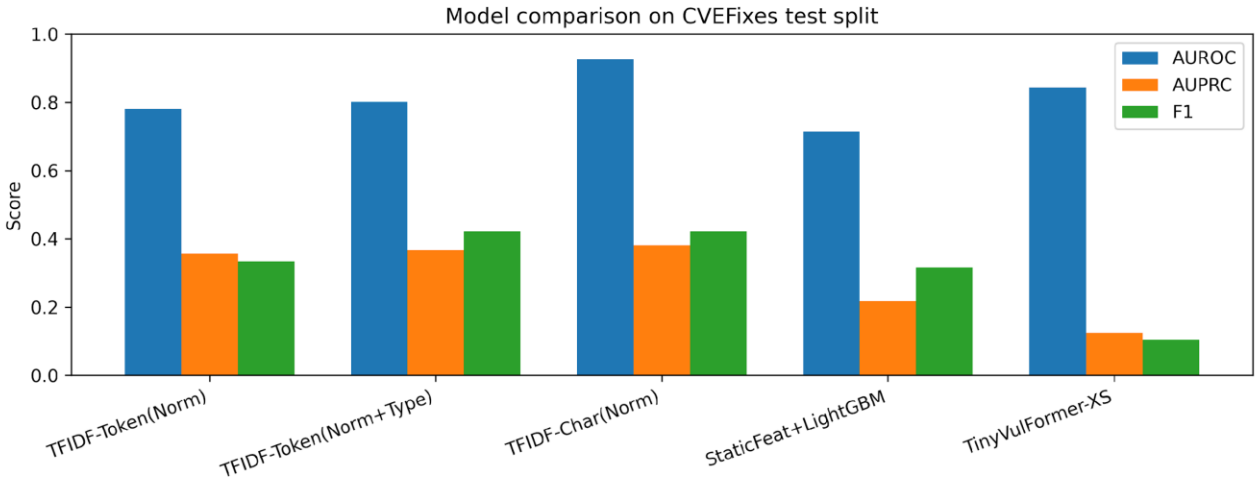

Figure 4. AUROC, AUPRC, and F1 comparison across models (test).

Data Science : Journal of Computing and Applied Informatics Vol.10, No.02 (2026) 031-046

Table 5. Test-set performance at threshold 0.5 (higher is better; Brier lower is better).

Model	AUROC	AUPRC	F1	Precision	Recall	Balanced Acc	MCC	Brier
TFIDF-Token(Norm)+LR-SGD	0.781	0.357	0.333	0.750	0.214	0.607	0.397	0.009
TFIDF-Token(Norm+Type)+LR-SGD	0.802	0.367	0.421	0.800	0.286	0.642	0.475	0.009
TFIDF-Char(Norm)+LR-SGD	0.925	0.381	0.421	0.800	0.286	0.642	0.475	0.009
TFIDF-Token(Raw+Type)+LR-SGD	0.874	0.366	0.333	0.400	0.286	0.640	0.331	0.013
StaticFeat+LightGBM	0.714	0.217	0.316	0.600	0.214	0.606	0.354	0.012
TinyVulFormer-XS (10 ep)	0.843	0.125	0.105	0.057	0.643	0.756	0.163	0.111

Table 6. Confusion matrix counts at threshold 0.5 for each model.

Model	TP	FP	TN	FN
TFIDF-Token(Norm)+LR-SGD	3	1	1131	11
TFIDF-Token(Norm+Type)+LR-SGD	4	1	1131	10
TFIDF-Char(Norm)+LR-SGD	4	1	1131	10
TFIDF-Token(Raw+Type)+LR-SGD	4	6	1126	10
StaticFeat+LightGBM	3	2	1130	11
TinyVulFormer-XS (10 ep)	9	149	983	5

Table 7. Approximate end-to-end training time on CPU for each detector.

Model	TrainTimeSec
TFIDF-Token(Norm)+LR-SGD	1.11
TFIDF-Token(Norm+Type)+LR-SGD	0.99
TFIDF-Char(Norm)+LR-SGD	4.10
StaticFeat+LightGBM	0.97
TinyVulFormer-XS	31.79

Ranking behavior. Figure 5 (precision–recall) and Figure 6 (ROC) visualize the threshold-free behavior of the models. The PR curves highlight the practical challenge of extreme imbalance: even the best model's precision drops quickly as recall increases. Nevertheless, both TF-IDF models dominate the tiny Transformer across most of the recall range, and the character model maintains the highest precision for recall levels above ~0.2. This indicates that TF-IDF scoring is well-suited for prioritizing a small set of high-confidence candidates.

ROC curves can look strong even when precision is modest because the false positive rate denominator is large (many negatives). This is visible in Figure 6: models achieve AUROC values above 0.8, yet precision remains highly sensitive to threshold. For deployment, we therefore recommend selecting thresholds using PR curves and triage capacity rather than optimizing AUROC alone.

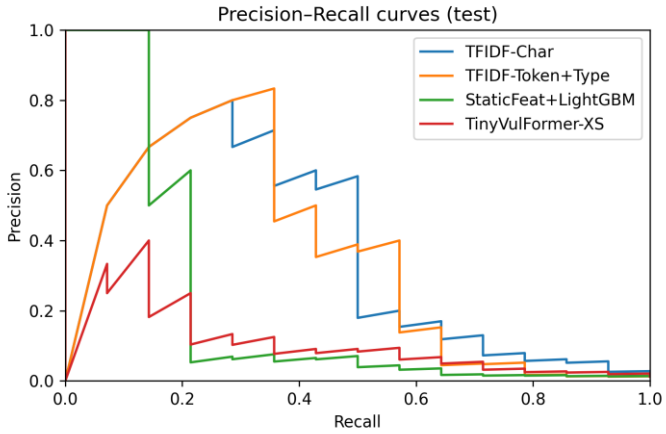

Figure 5. Precision–Recall curves on the test set.

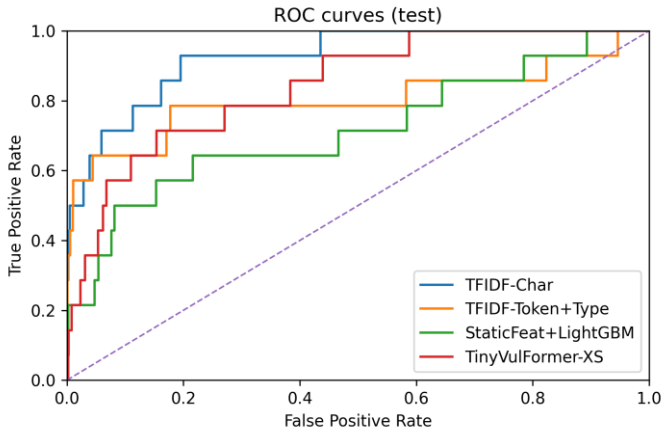

Figure 6. ROC curves on the test set.

Threshold sensitivity. In security screening, recall often matters more than precision because missing a vulnerability can be costly. Table 8 shows how the character TF-IDF model's precision/recall trade off as the decision threshold varies. Lowering the threshold from 0.5 to 0.2 increases true positives from 4 to 5 while keeping false positives low (2), improving F1 from 0.421 to 0.476. At very low thresholds (0.1), recall remains 0.357 but precision drops, indicating diminishing returns.

This analysis suggests two deployment modes. A "high-precision" mode (threshold ≥ 0.5) yields a small list of highly likely vulnerabilities, appropriate when triage resources are scarce. A "high-recall" mode (threshold around 0.2–0.3) yields more candidates while still keeping the false positive count manageable in this dataset. In both cases, SecureLite's LLM triage stage is useful because it helps reviewers process each flag quickly and consistently.

Table 8. Threshold sensitivity for TFIDF-Char(Norm)+LR-SGD on the test set.

Threshold	F1	Precision	Recall	TP	FP
0.100	0.476	0.714	0.357	5	2
0.200	0.476	0.714	0.357	5	2
0.300	0.476	0.714	0.357	5	2
0.400	0.476	0.714	0.357	5	2

0.500	0.421	0.800	0.286	4	1
0.600	0.421	0.800	0.286	4	1
0.700	0.421	0.800	0.286	4	1
0.800	0.421	0.800	0.286	4	1
0.900	0.421	0.800	0.286	4	1

Error analysis. To understand remaining failures, we analyze the character TF-IDF model's errors at a recall-oriented threshold of 0.2. Table 9 summarizes average function statistics for true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). False negatives are substantially longer (≈460 tokens on average) and contain more control-flow keywords than true positives (≈138 tokens). This indicates that complex multi-branch functions are harder to rank highly with n-gram features alone: they contain many benign tokens that dilute the relative weight of risky n-grams.

Conversely, false positives are shorter than true positives in this setting but exhibit higher average counts of risky keywords. Manual inspection of several false positives shows a common pattern: security-sensitive APIs are present, but used safely due to surrounding validation, constrained inputs, or trusted configuration. This is an inherent limitation of text-only models: they can recognize a risky idiom but may miss that the code sanitizes input earlier or that the call site is unreachable under attacker control assumptions.

LightGBM feature importance. Figure 7 shows the top static features used by the LightGBM baseline. Token and character counts are dominant, followed by indentation and control-flow density, consistent with the observation that vulnerability-labeled functions are often larger and more complex. However, these coarse signals are insufficient to match the TF-IDF baselines, which encode specific risky idioms such as path construction, serialization, subprocess invocation, and string formatting contexts.

Runtime. Table 7 reports end-to-end training time for each detector on CPU. TF-IDF models train in ~1–4 seconds on this dataset, LightGBM trains in ~1 second, and TinyVulFormer-XS requires tens of seconds due to per-epoch attention computation and repeated validation evaluation. Inference is fast for all models, but the gap matters for iterative experimentation and for teams that retrain frequently.

Limitations. This study operates at function granularity and therefore cannot localize the vulnerable statement within the function. In practice, a reviewer still needs to inspect the relevant lines. Furthermore, the dataset is CVE-derived; it reflects vulnerabilities that were reported and fixed, which may bias toward certain libraries and issue types. Finally, our tiny Transformer is trained from scratch and is not representative of the capabilities of pretrained code models; its role here is to quantify a "from-scratch" attention baseline under tight constraints.

Threats to validity. First, aggregating statement labels to function labels (OR) can introduce noise: a statement marked as vulnerable may be part of a larger vulnerable pattern, but the surrounding function may include safe guards that are not captured by local labels. Conversely, a truly vulnerable function may have its key cue removed by normalization or truncation. Second, CVE-derived datasets can contain duplicates or near-duplicates (e.g., similar helper functions across projects or versions). If duplicates appear across train and test, metrics may be optimistic. Our experiments follow the dataset's provided split, but future evaluations should include explicit de-duplication and cross-project splits.

Deployment implications. In practice, SecureLite should be treated as a prioritization layer rather than a final decision system. Teams can calibrate thresholds on their own repositories using a small labeled set and then monitor precision over time as code evolves. Combining SecureLite with rule-based SAST can further reduce risk: deterministic rules can catch known "must-fix" patterns, while ML ranking highlights subtle or context-dependent code that warrants manual review.

Table 9. Error-group statistics for TFIDF-Char(Norm)+LR-SGD at threshold 0.2.

Group	Count	AvgTokens	AvgLines	AvgRiskyKW	AvgControlKW
TP	5	42.40	5.60	0.40	0.20

FP	2	73.00	6.00	0.50	1.50
FN	9	121.22	13.78	0.67	3.33
TN	1130	92.73	9.59	0.51	2.29

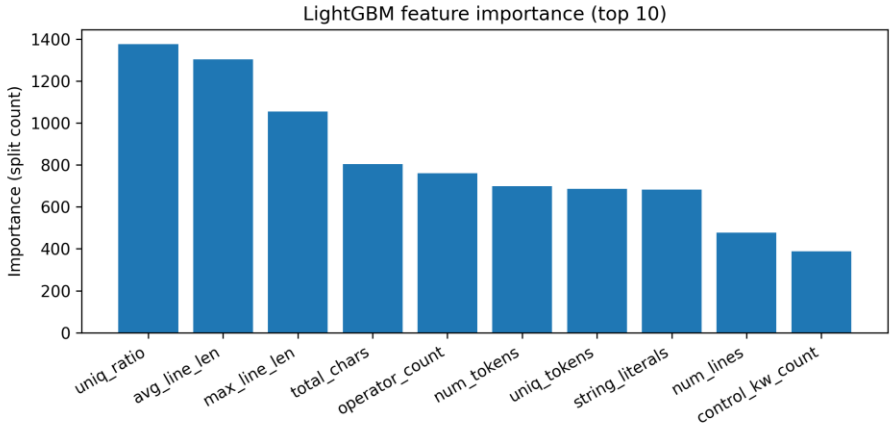

Figure 7. LightGBM static feature importance (top 10).

LLM-assisted triage and remediation. The second stage of SecureLite is designed to reduce analyst time once a function is flagged. Instead of only returning a probability score, the triage stage packages evidence into a short "repair brief": suspected vulnerability class, the risky code pattern, and concrete mitigation steps. The brief is produced by prompting a code-capable LLM with the function excerpt and a short list of model-derived cues (e.g., top n-grams or heuristic keyword matches).

Human factors are central here. Engineers are more likely to act on findings that are explained clearly and paired with a concrete patch plan. Conversely, unexplained warnings are often ignored, even if accurate. LLMs are well suited to bridging this gap because they can summarize risk assumptions, translate patterns into secure coding recommendations, and propose regression tests. Prior work evaluating code LLMs suggests they encode broad programming knowledge and can generate plausible code, although outputs still require validation [12].

In SecureLite, the triage stage is intentionally conservative: it is asked to state assumptions ("this is risky if the input is attacker-controlled") and to avoid claiming a confirmed exploit. This reduces the risk of overconfident hallucination. The output is treated as guidance for a reviewer, not as an oracle. Table 10 shows three representative examples (two true positives and one false positive) along with rubric scores.

For the path-construction examples (T592 and T855), the LLM identifies potential path traversal and recommends canonicalization plus a "must-stay-within-base-directory" check. This style of guidance is directly actionable because it maps to standard secure coding practices: normalize user input, reject absolute paths and ".." segments, and enforce that resolved paths remain inside a trusted root. For the false positive template-environment example (T1130), the assistant correctly conditions on context ("depends on template source") but still proposes hardening steps such as autoescaping or sandboxing. Even when the detector is wrong, the triage output can be useful as a checklist for security reviews, provided it is presented as guidance rather than a confirmed vulnerability.

Overall, rubric scores in Table 10 indicate that the generated briefs are readable and moderately actionable. The main weakness is consistency in ambiguous cases (false positives): without full repository context, the assistant cannot know whether templates or inputs are untrusted. This points to a clear integration requirement for real deployments: the triage prompt should include contextual facts (callers, data sources, and trusted boundaries) whenever available.

Governance and safe use of LLM outputs. SecureLite's triage stage is intended to accelerate human review, not to automate vulnerability disclosure or patch deployment. In a secure workflow, LLM-generated patches should never be merged automatically; they should be reviewed by maintainers, executed in test environments, and validated against unit and integration tests. When the assistant is uncertain due to missing context, it should

explicitly state assumptions and propose additional checks (e.g., identifying whether an input is attacker-controlled). These guardrails help prevent overconfident fixes that break functionality or introduce new vulnerabilities.

Table 10. LLM triage rubric scores and short summaries for selected examples.

ExampleID	Pattern	LLM Risk Summary	Fix Summary	Test Idea	Readability(1-5)	Actionability(1-5)	Consistency(1-5)
T592 (TP)	os.path.join(base, user_input)	Possible path traversal if index is attacker-controlled	Resolve path and enforce it stays within base directory	index='../../etc/passwd' should raise / be rejected	5	5	4
T855 (TP)	os.path.join(tmpdir, name)	Unsafe path composition may allow escaping temp dir	Normalize + validate path; reject absolute/.. segments	name='../x' should not escape temp dir	5	4	4
T1130 (FP)	Jinja2 Environment() config	Template injection risk depends on template source	Enable autoescape/sandbox; avoid rendering untrusted templates	Rendering untrusted input should be escaped by default	4	3	3

Example T592 (True Positive): code excerpt and suggested patch outline.

```
import pytest

from jinja2 import Environment, PackageLoader

@pytest.fixture(scope='session')...

from openapi_python_client import utils

TEMPLATE_FILTERS = {'snakecase': utils.snake_case, 'spinalcase': utils.
   spinal_case}

env = Environment(loader=PackageLoader('openapi_python_client'),
   trim_blocks=True, lstrip_blocks=True)

env.filters.update(TEMPLATE_FILTERS)

return env
```

Suggested mitigation (outline):

```
from pathlib import Path
import os

def safe_join(base_dir: str, user_path: str) -> str:
   base = Path(base_dir).resolve()
   cand = (base / user_path).resolve()
   if not str(cand).startswith(str(base) + os.sep):
      raise ValueError('path traversal')
   return str(cand)
```

Unit test ideas: (1) user_path='../../etc/passwd' must raise; (2) user_path='subdir/file.txt' must resolve inside base; (3) absolute paths must be rejected.

Example T1130 (False Positive): code excerpt and triage notes.

```
def __init__(self, *, openapi: GeneratorData) ->None:...

self.openapi: GeneratorData = openapi

self.env: Environment = Environment(loader=PackageLoader(__package__),
   trim_blocks=True, lstrip_blocks=True)

self.project_name: str = (self.project_name_override or
   f'{utils.kebab_case(openapi.title).lower()}-client')

self.project_dir: Path = Path.cwd() / self.project_name

self.package_name: str = (self.package_name_override or self.project_name.
   replace('-', '_'))

self.package_dir: Path = self.project_dir / self.package_name

self.package_description: str = (
   f'A client library for accessing {self.openapi.title}')

self.version: str = openapi.version

self.env.filters.update(self.TEMPLATE_FILTERS)
```

Triage note: the environment configuration is not necessarily vulnerable. The risk depends on whether untrusted templates or untrusted template variables are rendered. If templates are developer-controlled and autoescape is enabled appropriately, this pattern can be safe.

Interpretability of linear models. Beyond accuracy, a practical benefit of TF-IDF + logistic regression is that the learned weight vector can be audited. We extracted the highest-magnitude token n-grams from the type-aware token model (Norm+Type). Several high-weight features correspond to plausible security cues such as filesystem path handling (e.g., “Path”, “pathlib”), while others reflect dataset artifacts such as anonymized variable placeholders or project-specific tokens. This mixture highlights both the strength and weakness of purely lexical learning: models can discover real risk idioms, but they can also latch onto correlations that do not generalize across projects. For this reason, we recommend using feature inspection as a sanity check before deploying a model in a new environment.

SecureLite in a CI workflow. A typical deployment would train the detector offline (or periodically) and then run inference on each new pull request or nightly scan. Functions exceeding a threshold are sent to a triage queue with evidence n-grams and relevant code context. Reviewers can use the LLM brief to propose a patch and unit tests, but final decisions remain with developers. This workflow complements existing SAST: rule-based tools such as Bandit and SonarQube can provide deterministic alerts [15], [16], while SecureLite provides ranking and prioritization when alerts are too many or when patterns are subtle.

Table 11. Example high-weight token n-grams from the TFIDF-Token(Norm+Type) model (interpretability sample).

Rank	Feature(n-gram)	Weight	Class
1	VAR_16)	20.058	Vulnerable
2	assert utils	17.651	Vulnerable
3	<AnnAssign'> self	16.923	Vulnerable

4	. spinal_case	16.425	Vulnerable
5	Path	16.232	Vulnerable
6	. Path	16.097	Vulnerable
7	None <Assign'>	15.557	Vulnerable
8	pathlib .	15.356	Vulnerable
9	pathlib	15.217	Vulnerable
10	pip	15.200	Vulnerable
1	. .	-9.849	Non-vulnerable
2	,	-9.848	Non-vulnerable
3	self)	-9.272	Non-vulnerable
4	(self	-7.767	Non-vulnerable
5	in	-7.019	Non-vulnerable
6	"""	-6.248	Non-vulnerable
7	. <Condition>	-6.097	Non-vulnerable
8	) .	-5.880	Non-vulnerable
9	.	-5.847	Non-vulnerable
10	None)	-5.626	Non-vulnerable

4. Conclusion

This paper presented SecureLite, a CPU-friendly vulnerability screening pipeline that combines lightweight detection with LLM-assisted triage. Using the DetectVul/CVEFixes dataset, we compared TF-IDF + logistic regression baselines, a static-feature LightGBM model, and a tiny Transformer encoder trained from scratch. The strongest overall detectors were the TF-IDF baselines, with character n-grams achieving AUROC 0.925 and AUPRC 0.381 while maintaining high precision at threshold 0.5. These results confirm that simple lexical representations remain highly competitive for vulnerability screening when labels are scarce and class imbalance is extreme.

We also showed that operational performance depends strongly on the decision threshold. Lowering the character model's threshold increased recall with only a small rise in false positives, making it suitable for high-recall triage settings. Finally, we demonstrated how an LLM triage stage can translate model evidence into actionable fix guidance and unit test suggestions, providing a practical bridge from detection to remediation. Future work should evaluate SecureLite in a repository-level setting (cross-project generalization), incorporate richer structural features (lightweight AST paths or data-flow summaries), and assess LLM patch proposals against actual vulnerability fixes in CVEfixes.

References

[1] H.-C. Tran, A.-D. Tran, and K.-H. Le, "DetectVul: A statement-level code vulnerability detection approach for Python," Future Generation Computer Systems, vol. 163, p. 107504, 2025, doi: 10.1016/j.future.2024.107504.

[2] Hugging Face, "DetectVul/CVEFixes dataset card," accessed 2026-03-05.

[3] G. P. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated collection of vulnerabilities and their fixes from open-source software," arXiv:2107.08760, 2021.

[4] G. P. Bhandari, A. Naseer, and L. Moonen, "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software," in Proc. 17th Int. Conf. Predictive Models and Data Analytics in Software Engineering (PROMISE), ACM, 2021.

[5] G. Ke et al., "LightGBM: A Highly Efficient Gradient Boosting Decision Tree," in Advances in Neural Information Processing Systems (NeurIPS), 2017.

[6] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval," Information Processing & Management, vol. 24, no. 5, pp. 513–523, 1988.

[7] L. Bottou, "Stochastic Gradient Descent Tricks," in Neural Networks: Tricks of the Trade, 2nd ed. Springer, 2012.

[8] A. Vaswani et al., "Attention Is All You Need," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2017.

[9] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in Findings of EMNLP, 2020.

[10] D. Guo et al., "GraphCodeBERT: Pre-training Code Representations with Data Flow," arXiv:2009.08366, 2020.

[11] Y. Zhou et al., "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks," arXiv:1909.03496, 2019.

[12] M. Chen et al., "Evaluating Large Language Models Trained on Code," arXiv:2107.03374, 2021.

[13] FIRST.org, "Common Vulnerability Scoring System v3.1: Specification Document," 2019.

[14] NIST, "National Vulnerability Database (NVD)," accessed 2026-03-05.

[15] OpenStack Security Project, "Bandit: A security linter for Python," documentation, accessed 2026-03-05.

[16] SonarSource, "SonarQube Security Rules documentation," accessed 2026-03-05.

[17] T. Saito and M. Rehmsmeier, "The Precision-Recall Plot Is More Informative than the ROC Plot When Evaluating Binary Classifiers on Imbalanced Datasets," PLOS ONE, 2015.

[18] F. Pedregosa et al., "Scikit-learn: Machine Learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[19] A. Paszke et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in Proc. NeurIPS, 2019.

[20] W. McKinney, "Data Structures for Statistical Computing in Python," in Proc. SciPy, 2010.

[21] C. R. Harris et al., "Array programming with NumPy," Nature, 2020.

[22] M. Lhoest et al., "Datasets: A Community Library for Natural Language Processing," in Proc. EMNLP, 2021.