BreachT5: Ensembling CodeT5+ Models for Multi-Label Vulnerability Detection in Smart Contracts

"Smart contracts are to the Internet what laws are to society —rules written in code that govern trust and order."
— Tat Luat Nguyen

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Author
Tat Luat Nguyen
T.L.Nguyen.business@gmail.com

Supervisor(s)
Annibale Panichella
A.Panichella@tudelft.nl

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science

August 2025

Abstract

Detecting vulnerabilities in smart contracts is critical due to their immutability and the billions of dollars they secure. Industrial tools like Slither rely on hardcoded rules, often missing rare bugs or producing excessive false positives. Recent work with large language models (LLMs) such as GPT-5 have been applied to this task, but these models favor precision while failing to recall many true issues, especially in multi-label settings. We first fine-tune a 220M CodeT5+ model on 67,000+ real-world Ethereum contracts to establish a per-class detectability baseline, revealing which SWC vulnerabilities are intrinsically easier or harder to detect. We then study scaling effects, showing that the 770M variant improves majority-class precision but loses rare-class sensitivity. To reconcile this trade-off, we propose BreachT5, a soft-voting ensemble of both scales with tuned thresholds to balance recall and precision. BreachT5 achieves 0.556 Macro-F1 and 0.612 Micro-F1, outperforming standalone models, Slither, and GPT-5 on multi-label vulnerability detection in smart contract security.

Keywords

Ethereum Smart Contracts, SWC Vulnerabilities, CodeT5+, Large Language Models, Ensemble Learning, Multi-Label Classification, Software Security, Static Analysis

ACM Reference Format:
. 2025. . In . ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

High Stakes, Weak Defenses. Smart contracts on platforms like Ethereum secure billions of dollars across decentralized finance, infrastructure, and autonomous protocols. The combination of immutability and growing complexity makes even a single vulnerability cause irreversible loss [3]. As seen in the DAO hack—where a recursive call bug drained US$60M in 2016[1]—and the Poly Network exploit—where attackers stole over US$610M in 2021[2]—the financial stakes have escalated, where a single overlooked bug can collapse entire ecosystems and leave millions of users exposed to catastrophic loss [25]. By 2025, total reported blockchain exploit losses exceeded US$15.31 billion, including US$6.72 billion from DeFi protocols [7]. Despite this, current tools fall short. Static analyzers like *Slither* rely on rule-based pattern matching and cannot capture high-level semantic information, often producing many false positives [10, 24]. Meanwhile, general-purpose LLMs have shown promise, but recent evaluations show frequent failures in detecting rare vulnerabilities and inconsistent performance across vulnerability types, typically with high recall but low precision [5].

This paper addresses these gaps by fine-tuning CodeT5+ models on a dataset of over 67,000 real-world smart contracts across 12 SWC vulnerability classes, and introducing *BreachT5*, a targeted ensemble that leverages complementary strengths of different model scales to balance rare-class recall with majority-class precision.

The Imbalance Challenge. Multi-label vulnerability detection is severely class-imbalanced: some vulnerability types occur far more frequently than others. In the dataset we use—BCCC-SCsVuls-2024, which consolidates multiple sources of real-world contracts, reentrancy (SWC-107) and integer overflow/underflow (SWC-101) each exceed 16k instances, whereas weak access control (SWC-105) appears in fewer than 1k [14]. This imbalance reflects both real-world prevalence and dataset construction biases, and it directly impacts how easily different classes can be detected. Moreover, SWC categories exhibit distinct lexical and contextual signatures, as effective detection often requires capturing high-level semantic information that differs across vulnerability types [24].

Intrinsic Detectability. We therefore profile which vulnerability types are intrinsically easy or hard to catch by evaluating a fixed CodeT5+$_{220M}$ model under a consistent training setup, ensuring that observed differences reflect vulnerability-specific difficulty rather than confounders. This leads to our first research question:

RQ1: *How accurately can a fine-tuned CodeT5+$_{220M}$ model detect different SWC vulnerabilities, and which remain most difficult to capture?*

From this controlled evaluation, we observed clear detectability patterns tied to SWC semantics (e.g., cue specificity and context locality), even after accounting for class imbalance.

Scaling Model Capacity. Having established which SWC classes are easy or hard for a fixed-capacity model, we next ask whether *capacity alone* changes this picture. Holding the training setup constant, we scale the model to 770M parameters and examine both overall metrics and shifts in per-class precision/recall patterns.

RQ2: *How does scaling from CodeT5+$_{220M}$ to CodeT5+$_{770M}$ affect overall performance and the relative detectability of different SWC vulnerabilities under an identical training setup?*

Scaling revealed a trade-off: larger models improved frequent-class detection, while smaller ones preserved rare-class sensitivity.

Ensembling for Complementarity. Having observed complementary strengths across model scales, we next ask whether combining them yields more balanced performance. To this end, we design *BreachT5*, a soft-voting ensemble of CodeT5+$_{220M}$ and CodeT5+$_{770M}$ with tuned thresholds to balance recall and precision. Beyond internal gains, it is essential to ask whether these benefits hold in practice. We therefore benchmark *BreachT5* not only against its components, but also against external baselines—namely *Slither* (a widely used static analyzer based on rule-matching patterns) and *ChatGPT (GPT-5)* as a representative general-purpose

[1]In the end, the DAO hack triggered a controversial hard fork that split the Ethereum community and gave rise to Ethereum Classic.
[2]At the time, the Poly Network exploit was the largest hack in DeFi history, severely damaging user trust and exposing the fragility of cross-chain protocols.

Conference'17, July 2017, Washington, DC, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

LLM—chosen for their prominence in both industry and recent academic evaluations.

RQ3: *How does BreachT5 compare to its standalone models and external baselines (Slither, GPT-5) in overall and per-class vulnerability detection?*

Limitations and Validity. While RQ3 establishes *BreachT5*'s advantages over components and baselines, it is equally important to acknowledge the limits of our evaluation. We therefore examine potential threats to validity—including dataset representativeness, annotation noise, and metric choice—to clarify the conditions under which our results may or may not generalize.

Contributions. Our contributions are:

- We fine-tune **CodeT5+**$_{220M}$ on a dataset of 67,000+ real-world smart contracts, addressing severe class imbalance with weighted loss.
- We provide a systematic analysis of **per-class detectability** (RQ1), identifying which SWC vulnerabilities are intrinsically easier or harder to detect.
- We study the effect of **scaling model capacity** (RQ2), analyzing how larger models reshape overall and per-class performance.
- We introduce **BreachT5**, a soft-voting ensemble that unifies complementary strengths of different model scales (RQ3), and evaluate it against both standalone models and external baselines (Slither, GPT-5).
- We discuss **threats to validity**, clarifying the conditions under which our results may or may not generalize.

Structure. The remainder of this paper is organized as follows. **Section 2** reviews smart contract vulnerabilities and the CodeT5+ model. **Section 3** presents *BreachT5* and the ensembling methodology. **Section 4** reports experimental results and comparisons with existing tools. **Section 5** reviews related work. **Section 6** discusses threats to validity. Finally, **Section 7** concludes the paper and outlines future directions.

2 BACKGROUND

Smart contracts are self-executing programs deployed on blockchains, where the logic of agreements is directly encoded in code. Each execution propagates state changes across thousands of Ethereum nodes worldwide, ensuring consistency without any central authority [4]. But this decentralization comes with a critical trade-off: once deployed, contracts are immutable—bugs cannot be patched, and failures often result in irreversible financial loss. Hence, security has become a primary concern in both research and practice.

2.1 Security Taxonomies

To systematize recurring flaws, the community has proposed taxonomies for smart contract security. The most prominent are the DASP Top 10 [21] and the SWC registry [2]. While DASP highlights only ten broad categories of vulnerabilities, the project neither defines the listed vulnerabilities nor explains how the vulnerabilities were selected and ranked. By contrast, the SWC registry relates vulnerabilities to the broader CWE typology and provides reference contracts and remediation guidelines, and it remains the de facto standard for benchmarking tools and datasets in both academia and industry [23]. In this work, we adopt SWC as our taxonomy, as it remains the most widely used and better aligned with automated analysis.

Subset of SWC Classes. In this work we focus on six SWC classes that are most critical for our evaluation and illustrate different patterns of vulnerability:

```
1 mapping(address => uint256) public balances;
2 function withdraw(uint _amount) public {
3     msg.sender.call{value: _amount}("");
4     balances[msg.sender] -= _amount; // state
          update after external call
5 }
```

Listing 1: Reentrancy via external call before state update (SWC-107)

SWC-107 (Reentrancy). This weakness arises when a contract makes an external call before updating its own state. In Listing 1, the vulnerability stems from the ordering of lines 3 and 4: the external call is made before the state update. Because the balance is only reduced afterward, a malicious contract can re-enter during the call and repeatedly invoke `withdraw()`, draining funds before the deduction is applied.

SWC-101 (Integer Overflow and Underflow). Arithmetic overflows/underflows may wrap around silently in legacy Solidity, corrupting balances or counters.

```
1 address[] public users;
2 function refundAll() public {
3     for (uint i = 0; i < users.length; i++) { //
          unbounded loop
4         payable(users[i]).transfer(1 ether);
5     }
6 }
```

Listing 2: Unbounded loop over storage array (SWC-128)

SWC-128 (DoS with Block Gas Limit). Patterns such as unbounded loops over storage arrays can consume excessive gas, causing transactions to run out of gas and revert. In Listing 2, the loop in line 6 iterates over the entire `users` array. As the array grows, `refundAll()` may exceed the block gas limit, preventing the function from completing and effectively locking user funds.

```
1 function execute(address _target, bytes memory
      _data) public {
2     (bool success, ) = _target.delegatecall(_data)
          ;
3     require(success);
4 }
```

Listing 3: Delegatecall to untrusted callee (SWC-112)

SWC-112 (Delegatecall to Untrusted Callee). The `delegatecall` instruction executes code from another contract in the caller's storage and balance context. In Listing 3, the target address is user-supplied, allowing an attacker to craft a malicious contract that overwrites critical state or drains funds. This risk is context-dependent: fixed or library addresses are typically safe, while user-controlled targets are unsafe.

```
1 contract Relayer {
```

```
2     function relay(address target, bytes memory
          data) public {
3         target.call(data); // no check for
              sufficient gas
4     }
5 }
```

Listing 4: Relayer forwarding call without gas guarantee (SWC-126)

SWC-126 (Insufficient Gas Griefing). This weakness arises when a contract forwards a call without ensuring sufficient gas. In Listing 4, the relayer in line 3 performs a low-level call without any guarantee of available gas. A malicious sender may provide just enough gas for the relayer itself, but not for the callee, causing repeated failures and enabling griefing attacks. The pattern (a call forwarding data) looks very similar in both safe and unsafe cases, and there is no simple universal fix — remediation depends on the contract's intended trust model and gas-handling logic.

```
1 contract Vault {
2     function withdraw() public {
3     function withdraw() public onlyOwner {
4         payable(msg.sender).transfer(address(this)
              .balance);
5     }
6 }
```

Listing 5: Withdrawal function without and with access control (SWC-105)

SWC-105 (Unprotected Ether Withdrawal). Ether withdrawal functions must enforce access control to prevent arbitrary users from draining funds. In Listing 5, the `withdraw()` function in line 3 transfers the entire contract balance to the caller without any restriction. The underlined version with `onlyOwner` is the correct fix and needed for `withdraw()`.

The natural next step is to examine the tools that practitioners rely on—static analyzers and symbolic executors—which promise automated detection of these issues.

2.2 Traditional Tools

Tools like Slither [10], Mythril [1], and Oyente [19] were developed to detect such issues. Slither relies on static pattern matching, while Mythril and Oyente apply symbolic execution.

However, Slither lacks semantic depth. As shown in Listing 1, it flags the syntactic "call-before-effect" pattern without reasoning about `nonReentrant` guards (e.g., OpenZeppelin's `ReentrancyGuard`), equivalent mutexes, or restricted callbacks. Such protection in modifiers, inherited bases, or libraries is often ignored, producing frequent false positives and requiring manual triage.

Symbolic analyzers face different challenges, most notably path explosion when exploring execution traces [19], and difficulty modeling real-world behaviors. They struggle with obfuscation, dynamic dispatch, and deep nesting—features common in modern smart contracts.

In our evaluation (Section 4), we therefore benchmark BreachT5 against Slither, the most widely used and actively maintained static analyzer. This choice emphasizes the contrast between rule-based pattern detection and our learning-based generalization. Mythril and Oyente, while historically influential, are included only for background.

Other approaches, including abstract interpretation [26], fuzzing-based tools [12, 15] further enrich the landscape but fall outside the scope of our evaluation.

As blockchain systems evolve—handling billions in value, adopting proxy architectures, and embedding DSL—traditional tools increasingly fall short. Pattern matchers miss nuance, symbolic engines stall, and both ultimately require human review to distinguish signal from noise. In a landscape where exploits advance faster than detection rules, security must go beyond syntax and simulation; it must learn to reason, generalize and scale.

Hence, we next turn to large language models.

2.3 LLMs as Generalizable Detectors

Bridging this gap requires detectors that learn semantics beyond fixed rules. Large language models (LLMs) fit this role: trained on diverse code, they can generalize to unseen vulnerabilities by reasoning over structure rather than syntax alone.

This can be seen in the following example. Consider a contract with a nested conditional that, only under specific inputs, bypasses balance checks and drains funds. Rule-based tools, lacking pattern coverage for such cases, would likely miss it. Empirical studies (e.g., GPTScan [24]) demonstrate that LLM-based methods can catch logic flaws that static tools overlook. In-keeping with this, fine-tuned LLMs can be designed to reason over structure, rather than rely on syntax—illustrating the promise that "LLMs infer, not just match."

However, these systems have their limits. Prompt-based use (e.g., ChatGPT) has been shown to produce hallucinated bugs or contradictory outputs in general vulnerability detection tasks [6]. Even in the smart contract domain, fine-tuned and guided models show tradeoffs. Du et al. [8] benchmarked GPT-3.5, GPT-4, CodeT5+, and CodeBERT on Solidity contracts, finding that larger models like GPT-4 achieved high precision (up to 96%) but low recall (often below 38%), reflecting a tendency to predict the dominant "non-vulnerable" class.

These limitations highlight the need for models that are not only powerful but also stable and well-suited to classification. Rather than relying on decoder-only LLMs optimized for text generation—which are often used in the above examples—we turn to encoder–decoder architectures such as CodeT5+, which are designed to process code holistically and produce calibrated predictions.

2.4 CodeT5+ for Vulnerability Detection

LLMs pretrained on code are increasingly explored for smart contract analysis, yet their underlying architectures differ in suitability for classification tasks. To be precise, *decoder-only* transformers (e.g., OpenAI's GPT-4/GPT-5 or Anthropic's Claude Opus) generate text autoregressively, predicting the next token from left to right. They excel at code synthesis and interactive reasoning, but their reliance on prompt engineering and unidirectional context makes them poorly aligned with structured multi-label classification. By contrast, *encoder–decoder* architectures process the entire input bidirectionally (via the encoder) before producing outputs through a decoder or classification head. This holistic view enables more stable and calibrated predictions. In practice, this flexibility allows encoder–decoder models to be adapted for multi-label classification, where calibrated probabilities can be produced through a classification head to predict vulnerabilities.

We selected **CodeT5+** [27] as our base model because it represents the state of the art—and the largest publicly available encoder–decoder architecture—designed for code understanding. It extends the original CodeT5 with improved pretraining objectives and larger-scale training. Although its pretraining spans diverse programming languages (e.g., Python, Java, C++), it does not include Solidity. This gap makes CodeT5+ an instructive testbed for domain transfer: any ability to detect smart contract vulnerabilities must emerge through fine-tuning. In our setup, we fine-tune CodeT5+ for *multi-label classification*, mapping each output dimension to a specific SWC vulnerability class. A remaining limitation, however, is the restricted input length (512tokens), which risks truncating contracts where vulnerabilities span multiple functions (see §3.2).

3 BreachT5

Detecting vulnerabilities at the contract level is a uniquely difficult problem. Unlike single-label tasks, each Solidity contract may contain several co-occurring weaknesses drawn from the SWC registry, which creates a highly imbalanced, multi-label distribution. Rare classes are easily overlooked, while dominant ones bias predictions and depress macro-level metrics. These challenges make optimizing Micro-F1 and Macro-F1 substantially harder than in balanced classification settings.

BreachT5 addresses this gap with a structured ensemble of CodeT5+ 220M and CodeT5+ 770M. The system combines the rare-class sensitivity of smaller models with the precision of larger ones, balancing recall on infrequent vulnerabilities against precision on common ones.

We selected CodeT5+ as the backbone for three reasons. First, it is fully open source, enabling transparent fine-tuning and reproducibility. Second, CodeT5+, with its flexible encoder–decoder architecture and mix of pretraining objectives, achieves state-of-the-art performance across a broad set of code benchmarks—including code generation, completion, math programming, and retrieval—and notably surpasses its predecessor CodeT5 [28, 29]. Third, recent work has demonstrated the adaptability of CodeT5-family models to software testing domains. For example, AssertT5 fine-tunes CodeT5-large for the task of assertion *generation*, achieving up to 59.5% exact-match and 90.5 BLEU, and outperforming prior approaches [22]. Although this is a generation task rather than classification, it underscores the effectiveness of CodeT5 models in specialized, security-relevant program analysis settings.

These results suggest that CodeT5 models transfer well to security-critical tasks, making them a strong potential candidate for advancing contract-level vulnerability detection.

3.1 Dataset

Overview. We train and evaluate BreachT5 on **BCCC-SCsVul-2024** [14], a benchmark of 111,897 Solidity contracts annotated at the contract level with SWC vulnerabilities. To our knowledge, it is the largest dataset with structured SWC annotations. Larger corpora exist (e.g., DISL [20]), but they lack fine-grained SWC labeling. BCCC-SCsVul-2024 covers diverse domains such as DeFi protocols, wallets, oracles, and governance DAOs, and preserves complete Solidity source code, ensuring that cross-function semantics are not lost.

Sources. The corpus is aggregated from heterogeneous sources: Etherscan-verified contracts, SmartBugs, the Ethereum Smart Contracts dataset, Slither-audited corpora, and SmartScan—capturing variation in style, domain, and quality.

Annotation. Vulnerabilities were assigned through SCsVulLyzer (v2.0) and verified manually to reduce false positives and negatives. All contracts originate from open repositories, ensuring that no proprietary code is included.

3.2 Preprocessing and Splitting

Deduplication. Raw aggregation produced 111,897 entries, but many were duplicates: the same contract was duplicated for each vulnerability label it carried. After de-duplication, the dataset contained 67,474 unique contracts. This step was critical: without deduplication, the model could appear to generalize while in fact memorizing duplicate contracts across splits.

Normalization. We applied lightweight normalization to remove spurious textual artifacts while preserving full semantic content. Specifically, we stripped SPDX license headers, inline and block comments, byte-order markers (BOM), trailing whitespace, and collapsed empty lines. This prevented textual leakage (e.g., license headers or formatting quirks) without altering program semantics.

Filtering. We then discarded incomplete or malformed sources missing core Solidity constructs (`contract`, `interface`, `library`, `function`, `assert`), ensuring only valid and analyzable code entered training.

Label Binarization. Vulnerability annotations were transformed into 12-dimensional binary vectors, representing 11 SWC classes plus the `non_vulnerable` label.

Tokenization. Contracts were tokenized using the CodeT5+ tokenizer with a maximum length of 500 tokens; longer contracts were segmented with a stride of 250 tokens to preserve continuity across windows. This sliding-window chunking strategy follows prior work on code LLMs, where segmentation is commonly adopted to mitigate input length limitations [29].

Splitting. To avoid data leakage, splitting was performed strictly at the contract level: all segments of a contract remain in the same split. We applied stratified sampling to preserve label distribution, resulting in three sets: 48,000 contracts for training ($\approx$ 72% of the data), 12,000 for validation ($\approx$ 18%, an 80/20 ratio relative to training, used during training for model selection and later for threshold tuning), and 7,474 for testing ($\approx$ 10%). The test set was kept completely unseen and reserved exclusively for final benchmarking.

3.3 Model Composition and Training

BreachT5 integrates CodeT5+ 220M and CodeT5+ 770M. The goal was to test whether principled fine-tuning and ensembling alone could achieve state-of-the-art performance. Both models are trained under an identical setup—same data splits, loss formulation, optimization schedule, and early stopping criteria—so that any performance difference stems solely from model capacity rather than training variation.

Class Weights. We experimented with several weighting schemes to address label imbalance. Weighted Binary Cross-Entropy (WBCE) was first applied. We set the positive weight per class as

$$r_c = \frac{N - n_c}{n_c} \quad \text{(negative:positive ratio on the training split),}$$

and define

$$\alpha_c = \left(\log(1 + r_c)\right)^{1/T}, \qquad T = 0.6.$$

This log-scaled scheme upweights rare classes while moderating extreme weights. However, WBCE alone occasionally collapsed on certain rare classes, producing degenerate predictions with no positive detections.

Loss Function. To address this instability, we adopted a class-weighted Focal Loss. Originally proposed for dense object detection [17], Focal Loss extends binary cross-entropy with per-class weights α_c and a modulating factor $(1 - p_t)^\gamma$ that down-weights easy examples. For each label c with prediction probability p_c and ground truth $y_c \in \{0, 1\}$, the loss is

$$\mathcal{L}_{\text{focal}} = -\sum_{c=1}^{C} \alpha_c \cdot (1 - p_t)^\gamma \cdot \log(p_t),$$

where

$$p_t = \begin{cases} p_c & \text{if } y_c = 1, \\ 1 - p_c & \text{if } y_c = 0. \end{cases}$$

We set $\gamma = 1.5$ and reuse the α_c defined above. This choice consistently improved recall on rare vulnerabilities while maintaining precision on frequent ones.

Optimization. Optimization used AdamW with a linear learning rate schedule, 10% warmup, bfloat16 mixed precision, a batch size of 128, and early stopping on validation BCE loss. The training budget was capped at 12 epochs, though in practice early stopping intervened much earlier: the 220M model converged after 4 epochs, while the 770M model reached its lowest validation loss after 4.5 epochs.

Inference. During inference, the model outputs class–probability vectors over the 12 labels. Unless noted otherwise, we use a *single global decision threshold* t^* tuned on the validation split to *maximize Macro-F1* (equal weight per class) and keep t^* fixed for all test-time evaluations. *Why Macro-F1 with one global* t^* *(vs. Micro-F1 or per-class thresholds)?* Macro-F1 treats all SWC classes equally under imbalance; a Micro-F1–tuned threshold is dominated by frequent labels and suppresses rare-class recall. We also avoid per-class thresholds: they introduce 12 extra knobs that overfit the validation split and *mask the model's intrinsic per-class calibration/behaviour*. Per-class thresholding is better suited to benchmarking end-to-end systems (e.g., **RQ3**) where each model is tuned to its own best operating point; for **RQ1** it obscures innate detectability making cross-class comparisons less meaningful. After thresholding we enforce a mutual-exclusion rule: if any SWC label is predicted positive, `non_vulnerable` is set to 0. Long contracts are segmented into overlapping windows and processed independently; per-window predictions are aggregated to the contract level via *max pooling* over probabilities so that a vulnerability detected in any chunk is propagated to the entire contract.

3.4 Ensembling Strategy

To unify the complementary strengths of CodeT5+220 and CodeT5+770, we adopt a *per-class weighted soft voting* ensemble. As detailed in Section 3, each contract is represented by a probability vector over the SWC labels. The motivation is to reduce variance and mitigate systematic errors from individual models: instead of treating both equally, the ensemble emphasizes whichever model demonstrates higher discriminative ability for a given vulnerability class. This ensures that rare but difficult vulnerabilities benefit from the model that handles them best, while frequent classes remain stable.

Ensemble Weighting. Weights are proportional to the threshold-tuned per-class F1 scores of each model, so that the ensemble favors the stronger model for a given vulnerability type. Formally, for model m, contract n, and class c:

$$p_{n,c} = \sum_{m=1}^{M} w_{m,c} \cdot p_{m,n,c},$$

where $w_{m,c}$ is derived from validation-set F1 and normalized across models (with smoothing to avoid collapse on low-F1 classes).

Inference. The resulting ensemble probabilities are post-processed with *per-class* thresholds (tuned on the validation split and frozen for test) and the same mutual-exclusion rule for `non_vulnerable`. We use per-class thresholding here (in contrast to the global Macro-F1 t^* used in RQ1) for two reasons: (i) **Calibration drift under ensembling.** Weighted soft voting shifts margins *unevenly* across labels (the 220M and 770M models specialize differently), so a single global cutoff is systematically suboptimal; per-class thresholds recover each label's best operating point. (ii) **End-to-end benchmarking.** For the ensemble (used in RQ3-style comparisons/deployment), it is standard to tune the operating point *per label* rather than use a single global cutoff; prior work shows that adjusting per-label thresholds in binary-relevance multi-label classifiers significantly improves performance [9].

3.5 Implementation Details

All experiments were implemented in Python 3.10 using PyTorch 2.6.0 and HuggingFace Transformers 4.55.2. We fine-tuned CodeT5+ models with a custom HuggingFace `Trainer` (class-weighted focal loss, early stopping, bfloat16). GPU runs were executed on NVIDIA H100 PCIe for the 770M model and RTX 6000 Ada for the 220M model. Our replication package—preprocessing, training/evaluation scripts, and dataset artifacts—is available at https://github.com/vaultmind/BreachT5-Ensemble-SC.

4 EVALUATION

We evaluate the performance of our models and ensemble through four research questions, each probing a distinct dimension of vulnerability detection—specialization, synergy, benchmarking, and robustness.

4.1 Vulnerability-Specific Detectability

RQ1: *How accurately can a fine-tuned CodeT5+$_{220M}$ model detect different SWC vulnerabilities, and which remain most difficult to capture?*

To assess which vulnerabilities are inherently easier or harder to detect, we begin with the 220M model as a baseline lens. This step isolates vulnerability-specific patterns without yet considering the effect of scaling. By examining precision and recall per SWC class, we identify which vulnerabilities can be flagged reliably and which remain ambiguous or context-dependent.

Experimental Setup. We evaluate the CodeT5+$_{220}$ model under the training configuration of Section 3. The model outputs probability vectors over 12 SWC labels, and predictions are assessed with per-class precision and recall in addition to aggregate Micro-/Macro scores. Micro-F1 aggregates true/false positives and negatives across all classes before computing precision and recall, thereby reflecting overall performance weighted by class frequency. Macro-F1, in contrast, computes the F1 score independently for each class and averages them, giving equal weight to rare and frequent vulnerabilities alike. This design highlights not just overall performance, but the detectability of each vulnerability class in practice. The evaluation set was stratified from the full dataset, so the reported distributions are directly proportional to the class priors; normalizing them yields the exact prior probabilities, which provide a natural baseline for judging whether the model detects classes better than frequency alone would predict.

Reading the table. Horizontal rules group SWC classes by training-weight bands (log1p of class priors). This controls for imbalance: within-band precision/recall differences reflect cue specificity and context rather than frequency. See the caption for the color scale.

Key Findings.

- **Chunk-dependence.** SWC-107 (Reentrancy) and SWC-101 (Integer Overflow/Underflow) have similar training weights (1.45 vs. 1.49) and achieve comparable *detectability* (recall 0.7893 vs. 0.7762), yet *reliability* (precision) diverges by +28.1pp (0.7192 vs. 0.4383). We attribute this gap to *chunk-dependence*: detecting SWC-101 often requires cross-chunk context (e.g., `using SafeMath`, `.add/.sub`, or `pragma >=0.8`) that is split from the arithmetic site, inflating false positives when the safeguard is unseen. Reentrancy, by contrast, is largely captured within a single chunk (external call before state update), making its detection more reliable.
- **Syntactic salience vs. semantic ambiguity.** SWC-128 (DoS with Block Gas Limit) and SWC-112 (Delegatecall to Untrusted Callee) have nearly identical training weights (2.21 vs. 2.26), yet *detectability* (recall) diverges sharply: 0.9964 vs. 0.6431 (Δ = +35.3pp in favor of SWC-128). This reflects a structural difference in how the two vulnerabilities manifest. **SWC-128** produces highly distinctive syntactic patterns—such as unbounded loops over storage arrays or bulk clearing operations—whose shapes are rarely benign. The model therefore learns to flag these aggressively, which explains the very high detectability, but this same generalization lowers *reliability* (precision 0.7575) since some benign loops are also misclassified. **SWC-112**, by contrast, depends on semantic context: the same `delegatecall` syntax can be either safe (self-delegate) or dangerous (attacker-controlled). Because this distinction is not visible from surface syntax alone, the model behaves conservatively—firing less often, which lowers detectability, but with high *reliability* (precision 0.9073) when it does predict. Compared to SWC-128's clear syntactic signature, SWC-112 remains harder to detect.
- **Context dependence.** SWC-112 (Delegatecall to Untrusted Callee) and SWC-126 (Insufficient Gas Griefing) achieve comparable *detectability* (recall 0.6431 vs. 0.6378), yet diverge strongly in *reliability*: 0.9073 vs. 0.2951 (Δ = +61.2pp in favor of SWC-112). This asymmetry stems from how the vulnerabilities are expressed syntactically. **SWC-112** is triggered by the rare and security-salient keyword `delegatecall`, where local context often hints at safety (e.g., self-delegate vs. user-controlled target). The model therefore fires cautiously, yielding moderate detectability but high reliability when it does predict. **SWC-126**, by contrast, typically arises in relayer/proxy patterns where a sub-call can be starved of gas (e.g., `address(target).call(...)`, `call{gas: g}(...)` or callee logic gated by `gasleft()`). These surface cues also occur in benign forwarding code; without explicit evidence that the caller controls gas or that a minimum gas is enforced, the model overgeneralizes—leading to similar detectability but many false positives and thus low reliability. Compared to SWC-126's generic and context-dependent syntax, SWC-112 is more reliably recognized.
- **Lexical cues.** SWC-105 (Unprotected Ether Withdrawal) is rare (0.86%; 959 occ.) with the highest training weight (10.64), yet attains maximal *detectability* (recall 1.0000) but only moderate *reliability* (precision 0.6752). We attribute this to strong lexical cues for withdrawals (e.g., `msg.sender.transfer(...)` or transferring `this.balance`) that the model almost never misses, while access-control evidence (e.g., `onlyOwner`, `require(msg.sender==owner)`,

Table 1: Per-class precision and recall of CodeT5+$_{220}$ on the test set with a global threshold t^*. Precision is color-coded by *reliability* (green = high ≥0.8, yellow = moderate 0.6–0.79, red = low <0.6), while recall is color-coded by *detectability/sensitivity* (green = high ≥0.8, yellow = moderate 0.6–0.79, red = low <0.6).

SWC-ID	Dist. (%) (occ)	Weight	Precision	Recall
non_vuln Non-vulnerable	24.26 (26,914)	1.23	0.8305	0.7199
SWC-107 Reentrancy	15.95 (17,698)	1.45	0.7192	0.7893
SWC-101 Integer Overflow/Underflow	15.09 (16,740)	1.49	0.4383	0.7762
SWC-128 DoS With Block Gas Limit	11.17 (12,394)	2.21	0.7575	0.9964
SWC-112 Delegatecall to Untrusted Callee	10.03 (11,131)	2.26	0.9073	0.6431
SWC-126 Insufficient Gas Griefing	6.20 (6,879)	3.51	0.2951	0.6378
SWC-113 DoS with Failed Call	4.65 (5,154)	4.34	0.1935	0.5394
SWC-120 Weak Randomness from Chain Attr.	3.25 (3,604)	5.46	0.1704	0.3584
SWC-114 Transaction Ordering Dependence	3.21 (3,562)	5.58	0.1456	0.2835
SWC-104 Unchecked Return Value	2.91 (3,229)	5.74	0.1454	0.4553
SWC-116 Block Values as Randomness	2.41 (2,674)	6.63	0.2648	0.3480
SWC-105 Unprotected Ether Withdrawal	0.86 (959)	10.64	0.6752	1.0000

correct constructor) is often outside the current chunk or absent, inflating false positives and depressing precision.

Summary. Across SWC classes, performance is governed by *cue specificity* and *context locality*. Chunk-local, highly syntactic cues (e.g., unbounded loops in SWC-128; explicit withdrawals in SWC-105) drive near-saturated detectability, with reliability limited by benign look-alikes or missing access-control evidence. Semantically conditioned cues (e.g., trust of `delegatecall` in SWC-112) yield the opposite—conservative firing and high precision. Where the signal is generic or cross-chunk (e.g., relayer gas patterns in SWC-126; safeguards for SWC-101), precision drops due to overfiring when disambiguating context is outside the snippet.

4.2 Scaling Effects on Detectability

RQ2: *How does scaling from CodeT5+$_{220M}$ to CodeT5+$_{770M}$ affect overall performance and the relative detectability of different SWC vulnerabilities under an identical training setup?*

To examine how model capacity impacts detection, we compare CodeT5+$_{220}$ and CodeT5+$_{770}$ under the same training and evaluation regime. Where RQ1 focused on vulnerability-specific difficulty, here we ask whether scaling improves aggregate accuracy or simply reinforces frequent-class patterns.

Experimental Setup. Both models are fine-tuned with the configuration of Section 3, producing probability vectors over 12 SWC labels. Predictions are thresholded globally to maximize Micro-F1 on validation data and evaluated on the stratified test set. We report per-class and aggregate F1, highlighting ΔF1 to reveal whether scaling yields systematic gains, rare-class sensitivity, or trade-offs.

Table 2: Aggregate (Micro/Macro) and per-class F1 comparison. Green indicates 220M performs better; red indicates 770M performs better. ΔF1 is reported in percentage points as $(\mathbf{F1}_{220} - \mathbf{F1}_{770}) \times 100$.

Label	Distrib. (%)	F1 (220M)	F1 (770M)	Better	ΔF1 (pp)
Micro-F1	–	0.5908	0.6122	770M	**-2.14**
Macro-F1	–	0.5114	0.4955	220M	**+1.59**
non_vulnerable	24.26	0.7712	0.8142	770M	**-4.30**
SWC-107	15.95	0.7526	0.7833	770M	**-3.07**
SWC-101	15.09	0.5602	0.5806	770M	**-2.04**
SWC-128	11.17	0.8606	0.8752	770M	**-1.46**
SWC-112	10.03	0.7527	0.7570	770M	**-0.43**
SWC-126	6.20	0.4035	0.3694	220M	**+3.41**
SWC-113	4.65	0.2848	0.1941	220M	**+9.07**
SWC-120	3.25	0.2310	0.2451	770M	**-1.41**
SWC-114	3.21	0.1924	0.1917	220M	**+0.07**
SWC-104	2.91	0.2204	0.1555	220M	**+6.49**
SWC-116	2.41	0.3007	0.2417	220M	**+5.90**
SWC-105	0.86	0.8061	0.7387	220M	**+6.74**

Key Findings.

- **Aggregate metrics.** Scaling has mixed effects: Micro-F1 improves for 770M (0.6122 vs. 0.5908; Δ = −2.14pp), while Macro-F1 favors 220M (0.5114 vs. 0.4955; Δ = +1.59pp). This indicates that larger scale boosts frequency-weighted performance but reduces balance across classes.
- **Frequent classes.** On high-distribution labels (non_vulnerable, SWC-107, SWC-101, SWC-128, SWC-112), 770M consistently performs better, with margins up to 4.3pp. Scaling therefore enhances detectability on common patterns.
- **Rare classes.** For minority vulnerabilities (SWC-126, SWC-113, SWC-104, SWC-116, SWC-105), 220M outperforms 770M by larger margins (+3–9pp). The smaller model retains sensitivity where the larger model tends to smooth over rare signals.
- **Saturation effects.** Some classes such as SWC-128 are nearly saturated (>0.86 F1 for both models), leaving little headroom for scaling. This aligns with RQ1, where SWC-128 exhibited very high recall due to its clear syntactic patterns, making it easy for both models to detect regardless of scale.
- **Trade-off.** Scaling does not yield uniform gains: 770M improves majority-class detection, while 220M safeguards rare-class recall. This complementarity motivates structured ensembling (RQ3) to reconcile both strengths.

Summary. Scaling from 220M to 770M yields modest aggregate gains but introduces a clear trade-off: the larger model improves frequency-weighted performance and detects common vulnerabilities more reliably, while the smaller model retains sensitivity to rare classes. Certain vulnerabilities (e.g., SWC-128) saturate due to strong syntactic cues, yielding very high recall across both models and making them easy to detect regardless of scale. These findings motivate structured ensembling (RQ3) to combine complementary strengths.

4.3 Performance of BreachT5

RQ3: *How does BreachT5 compare to its standalone models and external baselines (Slither, GPT-5) in overall and per-class vulnerability detection?*

To assess whether ensembling yields practical benefits, we evaluate BreachT5 against two categories of baselines: (i) its own component models (CodeT5+$_{220}$ and CodeT5+$_{770}$), and (ii) external analyzers representing current practice—namely the rule-based static tool `Slither` and the zero-shot commercial LLM `GPT-5`. This design tests whether BreachT5 merely averages its components or provides measurable advantages over both internal and external alternatives.

Experimental Setup. For the internal comparison, we evaluate BreachT5 on the held-out test set using the soft-voting ensemble strategy with per-class threshold tuning described in Section 3. We compare its performance against the stronger of 220M or 770M for each label. Table 3 reports Micro-/Macro-F1 and per-class results,

Table 3: Aggregate (Micro/Macro) and per-class F1 comparison. Best Single F1 values are taken from the stronger of 220M or 770M (green = 220M, red = 770M). ΔF1 is reported in percentage points (blue = gain, gray = drop/no change).

Label	Dist. (%)	Best Single F1	F1 (Ensemble)	ΔF1 (pp)
Micro-F1	–	0.6122	0.6153	**+0.31**
Macro-F1	–	0.5114	0.5563	**+4.49**
non_vulnerable	24.26	0.8142	0.7872	-2.70
SWC-107	15.95	0.7833	0.8268	**+4.35**
SWC-101	15.09	0.5806	0.6453	**+6.47**
SWC-128	11.17	0.8752	0.9982	**+12.30**
SWC-112	10.03	0.7570	0.7657	**+0.87**
SWC-126	6.20	0.4035	0.3983	-0.52
SWC-113	4.65	0.2848	0.2765	-0.83
SWC-120	3.25	0.2451	0.2560	**+1.09**
SWC-114	3.21	0.1924	0.2525	**+6.01**
SWC-104	2.91	0.2204	0.2153	-0.51
SWC-116	2.41	0.3007	0.3115	**+1.08**
SWC-105	0.86	0.8061	0.9422	**+13.61**

allowing us to check whether the ensemble matches or exceeds the "best single F1" achievable by either model.

For external benchmarking, we adapt evaluation to the constraints of each tool. `Slither` was run on all test contracts, but compilation succeeded only for 5,172 of 7,440 contracts due to syntax errors and unsupported Solidity versions. We report Micro-/Macro-F1 on this subset for both Slither and BreachT5. For `GPT-5`, API limitations required a smaller evaluation: we sampled 1,000 contracts stratified by SWC distribution. GPT-5 was constrained to output labels strictly from the SWC set (e.g., `['SWC-107']` or `['non_vulnerable']`) using a robust prompt and fallback parser. For reproducibility, we include the exact prompt template used to query GPT-5 during benchmarking in Appendix. Table 4 summarizes results against both baselines.

Key Findings (Internal Benchmarking).

- **Aggregate gains.** BreachT5 improves over the best single model on both aggregate metrics (Macro-F1 +4.49pp, Micro-F1 +0.31pp), confirming that structured ensembling offers benefits beyond scaling.
- **Majority-class stability.** Frequent vulnerabilities (e.g., SWC-107, SWC-101, SWC-128) see consistent improvements (+4–12pp), showing that the ensemble unifies precision from the larger model with recall from the smaller one.
- **Rare-class recovery.** Some minority classes improve significantly (e.g., SWC-105 +13.6pp, SWC-114 +6.0pp), demonstrating that the ensemble can preserve rare-signal sensitivity rather than smoothing it out.
- **Localized regressions.** A few classes (non_vulnerable, SWC-126, SWC-113, SWC-104) regress slightly (−0.5 to −2.7pp), indicating that when component predictions diverge strongly, threshold tuning cannot always reconcile them.

Key Findings (External Benchmarking).

- **Macro-F1 shift (Slither subset).** The ensemble's Macro-F1 on the Slither benchmark subset (0.3714) is lower than in the main test set. This is due to distributional changes: Slither only processed 5,172 contracts (69% of the test set), excluding harder classes (e.g., SWC-105) and skewing class balance. The reduced diversity of classes depresses Macro-F1, even though BreachT5 still outperforms Slither substantially.
- **Aggregate gains.** BreachT5 outperforms both baselines substantially. Against `Slither`, Macro-F1 improves from 0.1168 to 0.3714 (+25.5pp), Micro-F1 from 0.1917 to 0.5037 (+31.2pp). Against `GPT-5`, Macro-F1 rises from 0.1425 to 0.5594 (+41.7pp) and Micro-F1 from 0.1629 to 0.6035 (+44.1pp).
- **Frequent classes.** On dominant labels BreachT5 delivers strong gains: vs. Slither, `non_vuln` (+40.2pp), `SWC-101` (+40.5pp), `SWC-128` (+92.6pp). vs. GPT-5, `non_vuln` (0.7610 vs. 0.0240), `SWC-107` (0.8168 vs. 0.2767), `SWC-128` (1.0000 vs. 0.4121).
- **Rare/difficult classes.** Where baselines collapse, BreachT5 sustains measurable F1. Slither reports 0.0000 F1 on SWC-112 and SWC-116, while BreachT5 achieves 0.1044 and 0.3612. GPT-5 yields near-zero F1 on SWC-126 (0.0208) and SWC-113 (0.0538), while BreachT5 maintains 0.3782 and 0.2719. Note: SWC-105 was excluded from the Slither subset due to compilation failures.
- **Coverage.** Slither failed on 31% of contracts from unsupported Solidity or syntax errors, limiting scope. GPT-5 was restricted to 1,000 contracts due to API constraints and often defaulted to predicting `non_vuln`, hurting recall on minority classes. BreachT5 runs consistently across the full test set.
- **Overall.** BreachT5 unifies complementary strengths and clearly outperforms both traditional analyzers and zero-shot LLMs, achieving higher aggregate scores, stronger per-class detection, and broader applicability for real-world smart contract vulnerability detection.

Table 4: Per-class and aggregate F1 comparison between Slither vs. BreachT5 and GPT-5 vs. BreachT5.

	Dist. (%)	Slither	BreachT5	Dist. (%)	GPT-5	BreachT5
Dataset	(5,172)	5,172	5,172	(1,000)	1,000	1,000
Micro-F1	–	0.1917	0.5037	–	0.1629	0.6035
Macro-F1	–	0.1168	0.3714	–	0.1425	0.5594
non_vuln	56.5	0.3892	0.7913	40.1	0.0240	0.7555
SWC-107	11.3	0.1773	0.1818	26.8	0.2767	0.8166
SWC-101	34.0	0.2531	0.6581	24.9	0.2682	0.6248
SWC-128	0.8	0.0093	0.9351	18.5	0.4121	1.0000
SWC-112	8.7	0.0000	0.1044	16.6	0.3311	0.7224
SWC-126	11.7	0.1351	0.3413	10.2	0.0208	0.3782
SWC-113	10.2	0.1677	0.3074	7.7	0.0538	0.2733
SWC-120	7.2	0.0663	0.2606	5.4	0.0737	0.2685
SWC-114	7.3	0.1188	0.2735	5.3	0.0919	0.2834
SWC-104	6.5	0.0847	0.2420	4.8	0.0587	0.2451
SWC-116	5.5	0.0000	0.3612	4.0	0.0542	0.3448
SWC-105	0.0	0.0000	0.0000	1.4	0.0449	1.0000

Summary (External Benchmarking). Across external baselines, BreachT5 delivers large improvements. Against `Slither`, it raises Macro-F1 from 0.1168 to 0.3714 and Micro-F1 from 0.1917 to 0.5037. Against `GPT-5`, it raises Macro-F1 from 0.1425 to 0.5594 and Micro-F1 from 0.1629 to 0.6035. These results confirm that domain-specific ensembling substantially outperforms both rule-based analyzers and zero-shot LLMs for real-world vulnerability detection.

5 Related Work

5.1 Static and Symbolic Analysis of Smart Contracts

The first generation of smart contract analyzers relied on static heuristics and symbolic execution. Tools such as `Slither` and `Oyente` identify known vulnerability patterns through handcrafted rules or symbolic traces [10, 19]. While effective for certain bug classes, these approaches suffer from limited coverage, high false positives, and frequent failures on contracts using modern Solidity constructs. Dynamic approaches such as fuzzing (`Echidna`, `ContractFuzzer`) [12, 15] improve path exploration but still rely on hard-coded oracles and struggle with semantic vulnerabilities. Compared to these rule-driven methods, our approach learns vulnerability patterns directly from data, allowing it to generalize across diverse coding styles and capture subtle cues not easily encoded in static rules.

5.2 Learning-Based Vulnerability Detection

Machine learning has recently been explored for smart contract analysis. Earlier work applied graph neural networks or token-based classifiers to detect specific classes of vulnerabilities [18, 30], but most efforts treat detection as a single-label task or target narrow vulnerability subsets. By contrast, real contracts often contain multiple flaws, motivating a multi-label framing. Our work builds on this perspective, evaluating detection across 12 SWC categories and explicitly analyzing trade-offs between frequent and rare vulnerabilities. In doing so, we show that smaller fine-tuned models can act as rare-class specialists, complementing larger models that prioritize frequent vulnerabilities.

5.3 Large Language Models for Code

General-purpose code models such as CodeBERT, GraphCodeBERT, and StarCoder [11, 13, 16] have been evaluated on program understanding and code generation tasks, but their effectiveness for security-critical vulnerability detection remains underexplored. CodeT5 and its successor CodeT5+ [28, 29] introduce encoder–decoder architectures with multi-task pretraining, achieving state-of-the-art results on standard code benchmarks. Recent work such as AsserT5 [22] demonstrates the adaptability of CodeT5-family models to software testing tasks, but no prior work systematically investigates their use in contract-level vulnerability detection. Our study extends this line of research by showing that CodeT5+ models can be fine-tuned to capture SWC vulnerabilities and that carefully structured ensembles outperform both standalone LLMs and traditional analyzers.

5.4 Positioning of BreachT5

To our knowledge, BreachT5 is the first ensemble of pretrained code LLMs tailored for multi-label smart contract vulnerability detection. Unlike prior static tools, it does not rely on fixed rules; unlike earlier ML approaches, it treats vulnerability detection as a multi-label problem; and unlike general-purpose LLM baselines, it is fine-tuned specifically for security tasks. By combining the rare-class sensitivity of smaller models with the majority-class precision of larger ones, BreachT5 advances both the methodology and practical performance of automated vulnerability detection.

6 THREATS TO VALIDITY

No empirical study is free of limitations. We discuss the main threats to the validity of our findings and the steps taken to mitigate them.

6.1 Internal Validity

Our experiments depend on the correctness of the BCCC-SCsVul-2024 dataset. While the dataset is large and standardized, it inherits noise from automated labeling pipelines: certain contracts may be mislabeled, and some mappings between dataset labels and SWC categories are ambiguous (e.g., ambiguous mappings such as *Call to Unknown*, which may correspond either to SWC-104 (unchecked call return) or SWC-112 (delegatecall to untrusted callee)). To reduce leakage, we stratified splits at the contract level so that all chunks of a contract remain within the same partition.

A further limitation arises from our chunking strategy: contracts longer than 500 tokens were split into overlapping segments, and both training and evaluation were performed on this chunked distribution. This prevents truncation and preserves local context, but it also alters class balance relative to unchunked contracts, since longer contracts contribute proportionally more training instances. As a result, model behavior on chunked inputs may differ from its performance on full, unsegmented contracts in deployment.

Nevertheless, residual noise may influence per-class results, and our observations should be interpreted as properties of the dataset's mapping rather than absolute ground truth.

6.2 External Validity

Our evaluation is restricted to contracts from BCCC-SCsVul-2024 and twelve SWC categories. While this dataset covers many known vulnerability types, it does not represent the full diversity of deployed contracts or emerging weaknesses. Benchmarking against `Slither` was further constrained by compilation failures (31% of contracts) and against `GPT-5` by a 1,000-contract API sample, which may bias comparisons. Thus, while BreachT5 shows consistent improvements in this setting, generalization to real-world deployments and new vulnerability classes remains an open question.

6.3 Construct Validity

We assess models primarily with F1 (micro and macro) alongside per-class precision and recall. Micro-F1 emphasizes frequent classes, while Macro-F1 balances rare and frequent ones. This design highlights detectability differences, but it does not fully capture the asymmetric cost of errors: e.g., a false negative in SWC-112 may be more damaging than one in SWC-120. Moreover, our analysis of scaling and ensembling assumes that per-class threshold tuning reflects a "fair" operating point, yet alternative calibration strategies might shift aggregate outcomes.

A further limitation is the reliance on the SWC taxonomy. While widely used, SWC has not been updated to capture many recent vulnerability types in the Ethereum and DeFi ecosystem (e.g., proxy upgrade flaws, cross-chain bridge exploits, oracles). As a result, our evaluation is bounded to a subset of vulnerabilities and does not reflect the full spectrum of threats faced in practice.

We mitigate these issues by reporting both aggregate and per-class metrics, and by highlighting trade-offs explicitly in the findings.

6.4 Conclusion Validity

Our reported improvements rely on specific training configurations (optimizer, learning rate schedule, early stopping) and ensembling design choices (weighted soft voting, per-class thresholds). While we tuned hyperparameters on validation data and observed stable convergence across seeds, other settings could yield different results. In particular, calibration drift under ensembling means that thresholding choices directly affect measured gains. To support reproducibility, we release all code, preprocessing scripts, and trained checkpoints in a public replication package.

7 CONCLUSION

This study introduced BreachT5, an ensemble of CodeT5+ models designed for multi-label vulnerability detection in smart contracts. Our evaluation across three research questions yields three key insights.

First, not all vulnerabilities are equally detectable: performance hinges on the locality and specificity of cues. Vulnerabilities expressed through distinctive syntactic patterns (e.g., SWC-128, SWC-105) are flagged with near-perfect recall, though at the cost of false positives, while semantically conditioned or context-dependent flaws (e.g., SWC-112, SWC-126) remain inherently harder to detect. Chunking further amplifies this effect, as safeguards or contextual signals may be split across windows, reducing reliability for cross-chunk vulnerabilities such as SWC-101.

Second, scaling from 220M to 770M improves aggregate performance but introduces a trade-off: larger models reinforce frequent-class detection, whereas smaller ones retain sensitivity to rare vulnerabilities. No single model dominates across the distribution.

Third, structured ensembling reconciles these opposing strengths. BreachT5 consistently outperforms its components on both Micro- and Macro-F1, while also surpassing established baselines. Compared to `Slither`, it triples Macro-F1 despite compiler restrictions; compared to zero-shot `GPT-5`, it maintains recall across rare classes where the general-purpose LLM collapses.

Taken together, these findings suggest that progress in contract-level vulnerability detection does not lie in scaling alone, but in combining models that specialize differently. By balancing recall on minority classes with precision on majority ones, BreachT5 demonstrates that ensembles can provide more reliable and deployable detectors than either large standalone models or existing tools.

Looking forward, two directions are especially promising. Extending beyond the SWC taxonomy would allow coverage of modern attack surfaces such as proxy upgrades, cross-chain bridges, and DeFi-specific exploits. In parallel, integrating contract-level reasoning with runtime analysis could help disambiguate context-dependent vulnerabilities that remain elusive to static cues.

In sum, BreachT5 shows that carefully designed ensembles of open, code-focused LLMs can advance vulnerability detection beyond the current state of the art. Crucially, our results demonstrate that progress does not come from scaling alone: fine-tuned smaller models can specialize on rare vulnerabilities and outperform larger counterparts, while ensembles reconcile these complementary strengths. While challenges remain in taxonomy coverage and context reasoning, our findings point to a clear path forward—combining specialization with ensembling to close the gap between academic benchmarks and the realities of smart contract security.

References

[1] [n. d.]. Mythril Classic: Security analysis tool for Ethereum smart contracts. https://github.com/ConsenSys/mythril. Accessed: 24-Aug-2025.

[2] [n. d.]. Smart Contract Weakness Classification and Test Cases (SWC Registry). https://swcregistry.io/. Accessed: 2025-08-24.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust (POST) (Lecture Notes in Computer Science, Vol. 10204)*. Springer, 164–186. https://doi.org/10.1007/978-3-662-54455-6_8

[4] Vitalik Buterin. 2014. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper/.

[5] Zhenpeng Chen, Yuxin Fan, Weiqi Luo, Shuiguang Deng, Guozhu Meng, and Yang Liu. 2024. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *arXiv preprint arXiv:2309.05520* (2024). https://arxiv.org/abs/2309.05520

[6] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *arXiv preprint arXiv:2304.07232* (2023).

[7] DefiLlama. 2025. DefiLlama Hack Database: Historical Record of DeFi Exploits and Losses. https://defillama.com/hacks. Accessed: 2025-08-23.

[8] Yuying Du and Xueyan Tang. 2024. Evaluation of ChatGPT's Smart Contract Auditing Capabilities (GPT-4). *arXiv preprint arXiv:2402.00631* (2024).

[9] Rong-En Fan and Chih-Jen Lin. 2007. *A Study on Threshold Selection for Multi-label Classification.* Technical Report. Department of Computer Science, National Taiwan University. https://www.csie.ntu.edu.tw/~cjlin/papers/threshold.pdf

[10] Joanes Henricus Petrus Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *Proceedings of the IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB '19)*. IEEE, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1536–1547. https://aclanthology.org/2020.emnlp-main.154

[12] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. ACM, 1–4. https://doi.org/10.1145/3395363.3404366 Tool: https://github.com/crytic/echidna.

[13] Daya Guo, Shuo Ren, Shuai Lu, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan Jiang, and Zhifang Lin. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=jLoC4ez43PZ

[14] Sepideh Hajihosseinkhani, Arash Habibi Lashkari, and Ali Mizani Oskui. 2024. Unveiling Smart Contracts Vulnerabilities: Toward Profiling Smart Contracts Vulnerabilities using Enhanced Genetic Algorithm and Generating Benchmark Dataset. *Blockchain: Research and Applications* 3 (2024), 100253. https://doi.org/10.1016/j.bcra.2024.100253

[15] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 259–269.

[16] Raymond Li, Loubna Ben Allal, Francois Benais, Alexandre Cassirer, Chenxi Mou, Niklas Muennighoff, Lewis Tunstall, Denis Kocetkov, Loic Magne, Maxwell Mitchell, et al. 2023. StarCoder: May the Source Be with You! *Transactions on Machine Learning Research (TMLR)* (2023). https://openreview.net/forum?id=F1hF9Zz2R4

[17] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal Loss for Dense Object Detection. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2980–2988.

[18] Xiao Liu, Qian He, Chen Feng, and Deqing Zou. 2021. Combining Graph Neural Networks with Expert Knowledge for Smart Contract Vulnerability Detection. *IEEE Transactions on Information Forensics and Security* 16 (2021), 4046–4060.

[19] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 254–269.

[20] Gabriele Morello, Mojtaba Eshghie, Sofía Bobadilla, and Martin Monperrus. 2024. DISL: Fueling Research with a Large Dataset of Solidity Smart Contracts. (2024). arXiv:2403.16861 DISL comprises over 514,000 Solidity contracts from Ethereum mainnet.

[21] NCC Group. 2018. DASP Top 10 of Smart Contract Vulnerabilities. https://dasp.co/. Accessed: 2025-08-24.

[22] Severin Primbs, Benedikt Fein, and Gordon Fraser. 2025. AsserT5: Test Assertion Generation Using a Fine-Tuned Code Language Model. arXiv:2502.02708 [cs.SE]

[23] Heidelinde Rameder, Monika di Angelo, and Gernot Salzer. 2022. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. *Frontiers in Blockchain* 5 (2022), 814977. https://doi.org/10.3389/fbloc.2022.814977

[24] Zhipeng Sun, Yikun Hu, Tianwei Zhang, and Xiapu Luo. 2023. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security '23)*. USENIX Association, 113–130. https://www.usenix.org/conference/usenixsecurity23/presentation/sun-zhipeng

[25] Chainalysis Team. 2021. Poly Network Attacker Stole $612 Million in One of the Largest DeFi Hacks. https://www.chainalysis.com/blog/poly-network-hack-august-2021/. Accessed: 2025-08-23.

[26] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 67–82.

[27] Yue Wang, Xiaopeng Gu, Shengyu Zhang, Song Liu, Shujie Wang, Philip Yu, and Yu Sun. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. arXiv:2306.03384 [cs.CL] Pretraining was done on datasets such as CodeNet, CodeSearchNet, and BigCode (Python, Java, C, C++, etc.) but not Solidity.

[28] Yue Wang, Weishi Wang, Shafiq Joty, Nazneen Fatema Hoque, Ziwei Lin, See-Kiong Ng, Jingjing Siow, and Steven C.H. Xu. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708. https://aclanthology.org/2021.emnlp-main.685

[29] Yue Wang, Weishi Wang, Shafiq Joty, See-Kiong Ng, Jingjing Siow, and Steven C.H. Xu. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. https://aclanthology.org/2023.emnlp-main.68

[30] Yiyu Zhou, Soumya Kumar, Seungwon Chung, Adam Chandler, Kevin Chang, and Saurabh Bagchi. 2019. Erays: Reverse Engineering Ethereum's Opaque Smart Contracts. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association, 1371–1385.

A APPENDIX

A.1 Dataset–SWC Mapping

We provide the mapping between the original dataset classes and their corresponding SWC Registry categories. Most mappings are exact; a few are approximate where no perfect equivalent exists.

Table 5: Mapping between original dataset classes and SWC Registry categories. Most mappings are exact; a few are approximate where no perfect SWC equivalent exists.

Dataset Class	SWC-ID	SWC Registry Name	Mapping Accuracy
Class01:ExternalBug	SWC-120	Weak Randomness from Chain Attr.	100%
Class02:GasException	SWC-126	Insufficient Gas Griefing	100%
Class03:MishandledException	SWC-113	DoS with Failed Call	~90%
Class04:Timestamp	SWC-116	Block Values as Randomness	100%
Class05:TransactionOrderDependence	SWC-114	Transaction Ordering Dependence	100%
Class06:UnusedReturn	SWC-104	Unchecked Return Value	100%
Class07:WeakAccessMod	SWC-105	Unprotected Ether Withdrawal	100%
Class08:CallToUnknown	SWC-112	Delegatecall to Untrusted Callee	~85%
Class09:DenialOfService	SWC-128	DoS with Unexpected Revert	~95%
Class10:IntegerUO	SWC-101	Integer Overflow/Underflow	100%
Class11:Reentrancy	SWC-107	Reentrancy	100%
Class12:NonVulnerable	–	Non-vulnerable	100%

A.2 GPT-5 Prompt Template

For reproducibility, we include the exact prompt template used to query GPT-5 during benchmarking.

```
# === Prompt Template ===
def build_prompt(code: str) -> str:
    labels_str = ", ".join(LABEL_COLS)
    return f"""
You are an expert in smart contract security.
Analyze the following Solidity contract and identify **all applicable vulnerabilities**
from the following fixed label set:

{labels_str}

 Important rules:
- You may ONLY output labels from the set above.
- If none apply, output exactly: ['non_vulnerable']
- Do NOT invent, guess, or output labels outside this list.
- Output strictly as a Python list of strings. No explanations, no extra text.

Example valid outputs:
['SWC-107', 'SWC-101']
['non_vulnerable']

---
Contract:
{code}
---
"""
```