# Zero-Shot Vulnerability Detection in Low-Resource Smart Contracts Through Solidity-Only Training

***Abstract*—Smart contracts have transformed decentralized finance, but flaws in their logic still create major security threats. Most existing vulnerability detection techniques focus on well-supported languages like Solidity, while low-resource counterparts such as Vyper remain largely underexplored due to scarce analysis tools and limited labeled datasets. Training a robust detection model directly on Vyper is particularly challenging, as collecting sufficiently large and diverse Vyper training datasets is difficult in practice. To address this gap, we introduce `Sol2Vy`, a novel framework that enables cross-language knowledge transfer from Solidity to Vyper, allowing vulnerability detection on Vyper using models trained exclusively on Solidity. This approach eliminates the need for extensive labeled Vyper datasets typically required to build a robust vulnerability detection model. We implement and evaluate `Sol2Vy` on various critical vulnerability types, including reentrancy, weak randomness, and unchecked transfer. Experimental results show that `Sol2Vy`, despite being trained exclusively on Solidity, achieves strong detection performance on Vyper contracts and significantly outperforms prior state-of-the-art methods.**

***Index Terms*—smart contract, vulnerability detection, Solidity, Vyper**

## I. Introduction

Smart contracts have become the foundation of decentralized applications and finance [1]–[3], yet their vulnerabilities continue to cause severe security and financial incidents. Detecting these vulnerabilities is therefore of paramount importance. Existing approaches fall into two categories. *The first relies on traditional program analysis techniques*, such as Mythril [4], Slither [5], and Securify [6]. However, static analysis tools depend heavily on manually crafted patterns and often suffer from high false positive rates, especially on newer compiler versions, while fuzzing-based dynamic approaches struggle with path explosion, limiting their practical applicability. *The second category leverages deep learning* [7]–[10], which offers greater scalability and accuracy by automatically learning semantic features from large datasets.

Existing deep learning-based approaches, however, overwhelmingly target Solidity, the dominant Ethereum smart contract language, which benefits from large, labeled datasets with well-curated vulnerability samples to support the training of robust detection models. In contrast, other low-resource languages such as Vyper remain severely underexplored. Vyper is a rising alternative with Python-like syntax and a security-oriented design that emphasizes simplicity, readability, and a reduced attack surface, making it attractive to developers who prioritize correctness and safety [11]. It is well known that training deep learning models requires a large amount of labeled data. However, *due to the data scarcity of labeled vulnerability samples in Vyper, no deep learning-based models currently exist for detecting vulnerabilities in Vyper*.

**Our Idea and Approach.** In this work, we take *the first step toward vulnerability detection in Vyper smart contracts*. To address the challenge of data scarcity, we propose *zero-shot vulnerability detection in Vyper via Solidity-only training*, where transferable knowledge is learned from Solidity, enabling a model trained on Solidity to detect vulnerabilities in Vyper. Achieving such cross-language knowledge transfer is non-trivial, however, due to the substantial syntactic and semantic differences between the two languages.

To learn transferable knowledge, we leverage SlithIR [5], a language-agnostic intermediate representation (IR) that serves as a bridge between Solidity and Vyper. SlithIR encodes smart contract semantics in a simplified, language-independent form, enabling effective cross-language representation learning. Building on this, we propose a novel three-stage framework, named `Sol2Vy`, that systematically transfers vulnerability detection knowledge from Solidity to Vyper.

Both Solidity and Vyper contracts are lifted to SlithIR using Slither. The *first stage* is *unsupervised transferable knowledge learning*, employs a multi-view architecture designed to capture both sequential and hierarchical representations of SlithIR. We train this architecture by minimizing the Maximum Mean Discrepancy (MMD) loss between Solidity and Vyper, enabling the model to learn *transferable, language-agnostic knowledge* from unlabeled datasets. In the *second stage*, *supervised vulnerability detection on Solidity*, a classification network is added on top of the pre-trained multi-view architecture. This network is trained only using labeled vulnerable and safe *Solidity contracts*, while the parameters of the multi-view backbone are frozen to preserve the transferable knowledge learned in the first stage. The *third stage*, *testing on Vyper*, applies the trained architecture and classification network directly to Vyper contracts without any modifications. Notably, `Sol2Vy` achieves effective vulnerability detection in Vyper *without requiring any labeled Vyper vulnerability samples for training*. This zero-shot capability addresses the data scarcity challenge in low-resource smart contract languages, overcoming both syntactic and semantic differences that have historically limited cross-language model reuse.

**Results.** We implement our approach, `Sol2Vy`, which learns transferable knowledge and enables effective vulnerability detection in Vyper contracts using models trained solely on Solidity. When the vulnerability detection model is trained and tested on Solidity, it achieves false positive rate (FPR)

/false negative rate (FNR) scores of 0.09/0.13, 0.08/0.12, and 0.11/0.14 for the three vulnerability types: reentrancy (RE), weak randomness (WR), and unchecked transfer (UT), respectively. When `Sol2Vy` is trained on Solidity and directly reused to evaluate Vyper contracts, it attains FPR/FNR scores of 0.11/0.16, 0.10/0.15, and 0.12/0.17 for RE, WR, and UT. The corresponding FPR/FNR increases (0.02/0.03, 0.02/0.03, and 0.01/0.03 for RE, WR, and UT) are minimal, indicating that `Sol2Vy` successfully captures strong cross-language transferable knowledge, enabling a Solidity-trained model to perform reliably on Vyper. Moreover, our approach consistently outperforms baseline methods across all evaluated settings. Below we highlight our contributions:

- We take the first step toward enabling robust vulnerability detection for the low-resource Vyper language. `Sol2Vy` performs zero-shot detection on Vyper contracts using only Solidity training data, removing the need for large labeled Vyper vulnerability datasets.
- We design a novel three-stage pipeline that lifts Solidity and Vyper into SlithIR and learns language-agnostic semantic representations. A multi-view encoder captures both sequential instruction flows and hierarchical structures, providing richer and more comprehensive semantic coverage.
- A classifier trained solely on labeled Solidity vulnerabilities generalizes effectively to Vyper, achieving low FPRs/FNRs (0.10–0.12) across RE, WR and UT. `Sol2Vy` significantly outperforms prior state-of-the-art methods.
- The Vyper vulnerability detector requires no labeled Vyper contracts, yet still accurately identifies vulnerable Vyper smart contracts, demonstrating the practical benefits of our cross-language transferable learning approach.

## II. Related Work

**Ethereum Virtual Machine and Smart Contracts.** Blockchain is a paradigm in distributed computing that underpins various cryptocurrency platforms and decentralized applications [12]–[19]. There are various blockchain platforms with unique features [20]–[22]. Ethereum Virtual Machine (EVM) pioneered programmable blockchains through its robust functionality [23]–[26]. Smart contracts, written in various languages, are self-executing programs deployed on various platforms. Solidity [27] remains the dominant language for EVM, while Vyper [11] provides a more Pythonic alternative with enhanced security features. The variety of smart contract languages introduces fundamental distinctions in their security models, vulnerability patterns, and development paradigms, making cross-language analysis challenging.

**Vulnerability Detection in Solidity Smart Contracts.** Smart contract vulnerabilities pose significant financial risks to blockchain ecosystems, with millions of dollars lost due to code defects [13], [14], [28]–[39]. Vulnerability detection in smart contracts has progressed through multiple approaches. Traditional static approaches include Slither [5] and Smartcheck [40] employ various techniques to identify vulnerabilities without executing the code. Dynamic analysis has evolved through tools like Echidna [41], Mythril [4], ContractFuzzer [42], symvalic [43], DivertScan [44], ILF [45], RLF [46], xFuzz [47], CrossFuzz [48], sFuzz [49], SMARTIAN [50], SCFuzzer [38], FunFuzz [51] and Harvey [52]. Recent work explores deep learning and LLMs for blockchain security analysis [7]–[10], [28], [29], [53]–[56]. Clear [55] captures the correlation between vulnerable and safe programs using a contrastive learning (CL) model. GPTLens [54] introduces two roles of detection: auditor and critic to enhance vulnerability discovery. AmeVulDetector [53] combines vulnerability-specific patterns with neural networks. GPTScan [56] combines LLM's understanding with static analysis checks by breaking vulnerabilities into scenarios and properties to query the LLM.

**Vulnerability Detection in Low-resource language.** Despite the robust ecosystem of vulnerability detection tools for Solidity smart contracts, there remains a critical shortage of analysis capabilities for low-resource languages such as Vyper. For example, Slither and Mythril rely on manually crafted patterns or rules to detect vulnerabilities in Vyper, which has low coverage. Deep learning models often require large labeled datasets, which are scarce in low-resource languages like Vyper. This disparity in security tooling across smart contract languages creates a significant vulnerability gap in the blockchain ecosystem.

To mitigate this gap, we propose *a novel framework that learns transferable, language-agnostic knowledge* across Solidity and Vyper, enabling the use of Solidity's abundant labeled datasets to train a robust detection model that can be directly reused to analyze Vyper contracts

## III. Overview

### A. Two Types of Datasets

`Sol2Vy` aims to develop a robust deep learning model that can extend vulnerability detection capabilities from Solidity to Vyper. We define two types of datasets used in `Sol2Vy`.

**General Dataset.** This dataset is used to learn transferable language-agnostic knowledge shared between Solidity and Vyper. It consists of *unlabeled* Solidity and Vyper smart contracts, which can be easily collected from public blockchain explorers (e.g., Etherscan) or open-source repositories. Since *no annotation effort is required*, this unlabeled corpus enables our system to capture common semantic patterns, structural behaviors, and execution characteristics of smart contracts across the two languages, forming the foundation for effective cross-language transfer in later stages.

**Vulnerability Detection Dataset.** This dataset is related to the vulnerability detection task. It consists of both safe and vulnerable contracts in Solidity, and is used to train the vulnerability detection model. Our goal is to *reuse* this Solidity-trained model to detect vulnerabilities in Vyper contracts by leveraging the transferable knowledge learned from the general unlabeled dataset.

When referring to data scarcity, we specifically mean the *scarcity of Vyper vulnerability datasets*. While large numbers

of *unlabeled* contracts can be collected for the general dataset, and abundant labeled vulnerability samples exist for Solidity due to its maturity and dominance, *Vyper lacks sufficiently labeled vulnerable contracts* because it is relatively new and supported by fewer security tools. As a result, directly training a robust vulnerability detector for Vyper is difficult. This motivates our approach of training the detector on Solidity and reusing it for Vyper vulnerability detection.

### B. IR Representation

Solidity and Vyper express the same functionality using different source-level syntax. As shown in Figure 1(a) and Figure 1(b), even semantically equivalent implementations (e.g., the `mint()` function) differ substantially in structure, syntax, and language constructs. This motivates the need for a common representation that can abstract away language-specific syntax. To this end, we adopt SlithIR, a language-agnostic intermediate representation produced by the Slither [5]. SlithIR converts both Solidity and Vyper into a standardized three-address code format that captures code core semantics.

However, *directly relying on SlithIR is insufficient*. While SlithIR provides a unified format, the SlithIR generated from Solidity and Vyper can still differ significantly. Figures 1(c) and 1(d) illustrate that even when the source code implements identical functionality, their SlithIR forms remain noticeably different. These discrepancies arise from several factors, including: (1) *Compilation-specific variations*: `solc` and `vyperlang` introduce distinct IR constructs (e.g., Solidity uses `MODIFIER_CALL`, while Vyper inlines modifier-like logic); and (2) *Differences in language features and control flow structures*: Given some unique high-level mechanism in Solidity (i.e., inheritance, overloading), the two languages generate divergent abstract syntax trees (AST) and thus different SlithIR, even for equivalent logic. As a result, in our experiment (Section V-F), a model trained solely on SlithIR derived from Solidity contracts performed poorly when tested on SlithIR derived from Vyper, confirming that SlithIR alone does not automatically align semantics across languages.

To address this misalignment, we design an *unsupervised cross-language transfer learning pipeline* that learns a shared, language-agnostic representation of SlithIR (see Section IV). Our multi-view encoder jointly models sequential instruction patterns and hierarchical structure, while an MMD-based alignment module minimizes the distributional distance between Solidity and Vyper SlithIR corpora. By training this encoder on large unlabeled datasets from both languages, we learn transferable semantic features that bridge the gap between their IRs. These learned transferable representations form the foundation of our downstream vulnerability detector, enabling a classifier trained solely on Solidity vulnerabilities to effectively analyze Vyper contracts, without requiring any labeled Vyper vulnerability data.

### C. Model Architecture

Our approach consists of three stages: (1) learning transferable knowledge in an unsupervised manner using the general dataset, (2) training a classification network on the Solidity vulnerability detection dataset, and (3) reusing the trained model to detect vulnerabilities in Vyper contracts.

Figure 2 shows an overview of `Sol2Vy`. In Step ①, we train the sequential and hierarchical encoders using the general dataset (consisting of *unlabeled Solidity and Vyper contracts*) to learn transferable language-invariant knowledge. In Step ②, we train the vulnerability classification network using the Solidity vulnerability detection dataset, which contains *labeled* safe and vulnerable *Solidity* contracts. Finally, in Step ③, we directly reuse the trained encoders and classifier to test Vyper smart contracts, *without any modification*.

It is important to note that throughout the workflow, `Sol2Vy` has never seen any vulnerable samples in Vyper, yet it still achieves strong vulnerability detection performance on Vyper contracts.

## IV. Model Design

### A. IR Preprocessing

**IR Extraction.** The first step is to lift both Solidity and Vyper source code into SlithIR. SlithIR abstracts away the syntactic differences of the two languages, providing a standardized, three-address code format that captures the core semantics and control flow. We first extract the ASTs through the compilers (`solc` for Solidity and `Vyperlang` for Vyper). Once we have the ASTs, we run Slither on the compiler outputs to generate per-function SlithIR, including (1) linearized IR statements, (2) basic blocks, and (3) call sites/event emissions/storage reads-writes. This produces a uniform IR interface for both languages even when their source constructs differ (e.g., modifiers, inheritance, and Vyper's Pythonic syntax). We extract SlithIR at the function level (and keep function boundaries), because most vulnerability patterns (reentrancy, unchecked transfer, weak randomness) manifest within a function's call/control/data interactions. Contract-level context (e.g., state variables) is retained through referenced storage symbols and types.

**IR Tokenization.** We decompose each SlithIR instruction into constituent components based on operation types, operands, and control flow structures. The tokenization process transforms the three-address code format of SlithIR into discrete tokens while preserving semantic information critical for vulnerability detection: (1) Operation Tokenization: Core operations (e.g., `ASSIGN`, `CONDITION`) are preserved as atomic tokens to maintain instruction semantics. (2) Variable Tokenization: State variables, local variables, and temporary variables (e.g., `TMP_0`) are tokenized separately to distinguish their scope and lifetime. (3) Type Information: Type annotations (e.g., `uint256`, `bool`, `address`) are retained as separate tokens to preserve type-safety semantics. (4) Control Flow Markers: Branch conditions, function boundaries, and expression hierarchies are marked with splitters to maintain structural information.

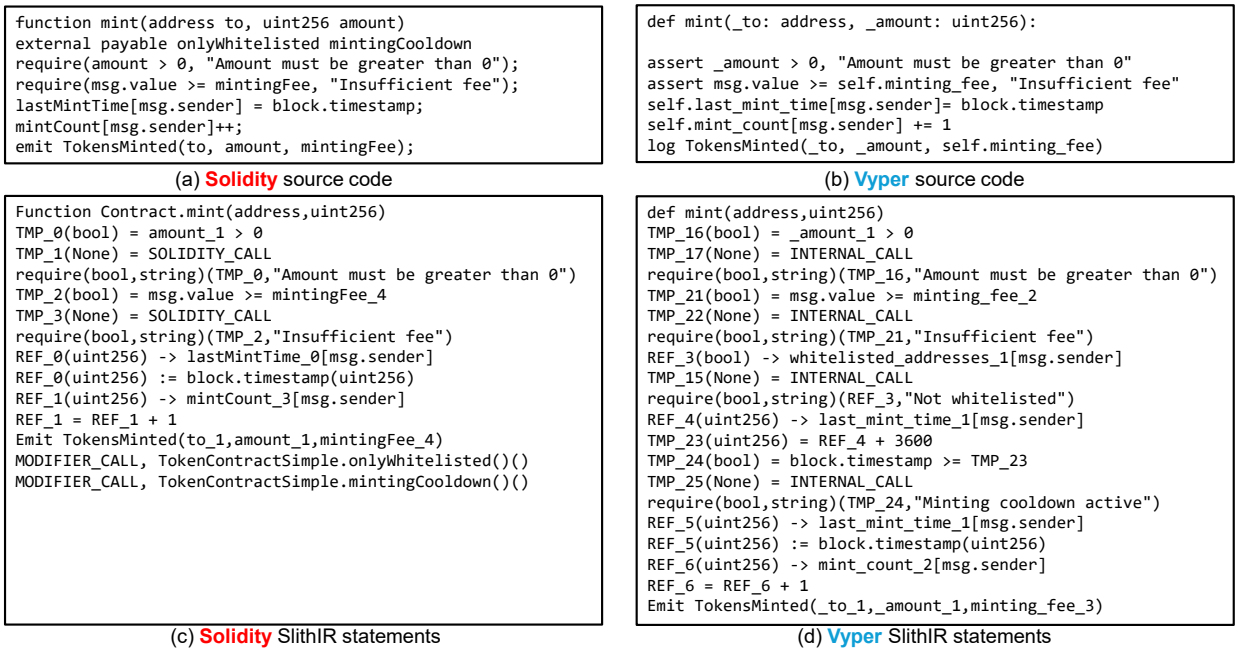


Fig. 1: Comparison of SlithIR generated from Solidity and Vyper versions of the `mint()` function.

**IR Normalization.** After tokenization, we process SlithIR to reduce compilation-specific noise. Specifically, the following normalization rules are applied:

- The names of temporary variables, reference variables and tuple variables are normalized: `TMP_0` → <TMP>, `REF_3` → <REF>, `TUP_2` → <TUP>.
- Simplify the customized variable name: `amount_1` → `amount`, `rate_2` → `rate`.
- We remove messages from statements: `require (<TMP>,"Amount > 0")` → `require (<TMP>)`.
- The string literals are replaced with <STR>.

### *B. Learning Transferable Knowledge*

**Multi-View Architecture.** To capture the sequential and structural semantic information inside SlithIR, we design a multi-view architecture to encode *both the sequential and structural* architecture of SlithIR. The multi-view architecture consists of two components that operate in parallel on SlithIR:

The first component is a ***sequential encoder***, which processes SlithIR instructions as a sequence to capture temporal dependencies and execution flow patterns. We employ a Transformer-based architecture for the sequential encoder, leveraging its proven effectiveness in modeling long-range dependencies in sequential data. It consists of the following parts: (1) Token embedding layer: The tokenized SlithIR instructions are first converted into dense vector representations. Each token (operation, variable, type annotation) is mapped to an embedding vector via a learnable embedding matrix $E \in R^{|V| \times d_{\text{seq}}}$, where $|V|$ is the vocabulary size and $d_{\text{seq}}$ is the embedding dimension (we set it to 128 in implementation). (2) Multi-head self-attention mechanism: The core of our sequential encoder employs multi-head self-attention to capture dependencies between the instructions regardless of their distance in the sequence. This allows the model to jointly attend to information from different representation subspaces, capturing diverse semantic relationships such as data flow dependencies, control flow patterns and variable usage patterns. (3) Feed-forward networks: Each Transformer layer includes a position-wise feed-forward network consisting of two linear transformations with ReLU activation.

The second component is a ***hierarchical encoder***, which processes the Abstract Syntax Tree (AST) derived from SlithIR to capture structural relationships and architectural patterns. We construct a graph representation $G = (V, E)$ where nodes $V$ correspond to syntactic elements such as functions, control flow constructs, variable declarations, and expression components. The edges $E$ in this graph represent three types of relationships: (1) AST structural relationships $E_{\text{ast}}$, (2) Variable definition-use chains $E_{\text{data}}$ and (3) Execution flow between basic blocks $E_{\text{ctrl}}$.

To process this graph structure, we employ Graph Attention Networks (GATs), which provide the capability to learn adaptive attention weights for different types of structural relationships. For node feature initialization, each node $v \in V$ is initialized with a feature vector $x_v \in R^{d_{\text{hie}}}$ (where we use $d_{\text{hie}} = 128$ in our implementation) that encodes: (1) Node type embedding, (2) Operation type and (3) Variable scope information (state/local/temporary).

After node initialization, the GAT layers aggregate information from neighboring nodes through learned attention weights, enabling the propagation of contextual information throughout the hierarchical structure. This aggregation process captures multi-hop relationships in the graph, allowing the model to understand complex structural dependencies that may span multiple levels of the syntax tree. The hierarchical encoder

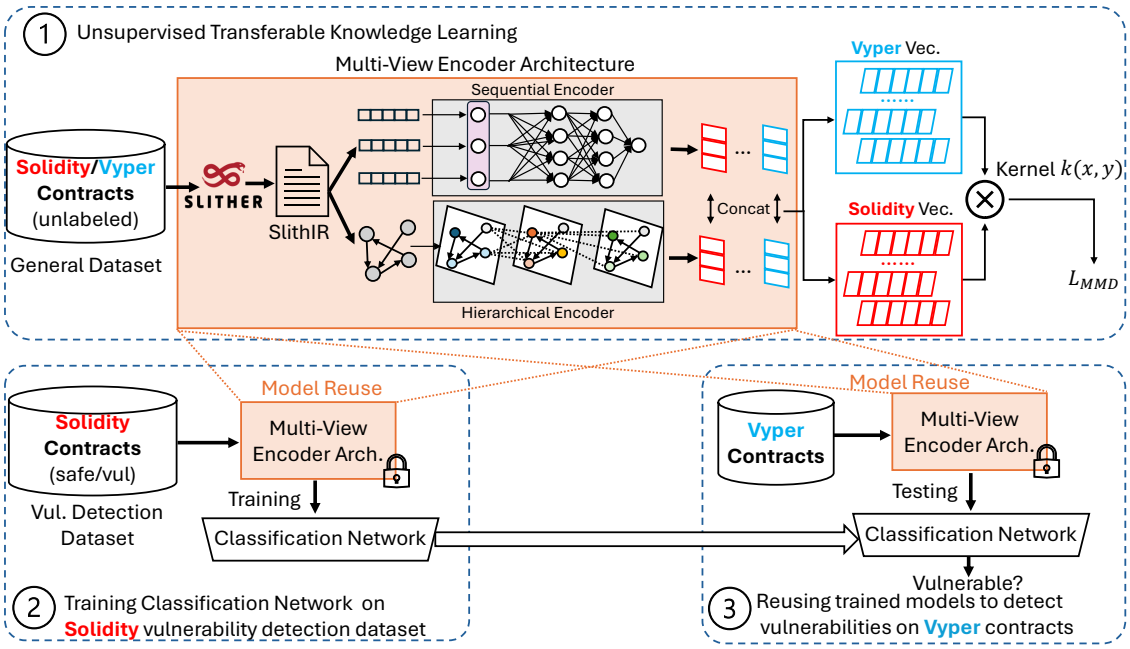


Fig. 2: Applying `Sol2Vy` to detect vulnerability in Vyper smart contracts by learning transferable knowledge from Solidity &Vyper corpus and training a classification network on Solidity.

thus produces node-level representations that encode both local structural properties and global architectural patterns.

**Feature Extraction and Combination.** For each Solidity and Vyper smart contract in the general dataset, we lift it to SlithIR and preprocess it. Then we feed the processed SlithIR through our multi-view architecture. The sequential encoder processes the tokenized SlithIR instructions and outputs a feature vector $h_{\text{seq}} \in \mathbb{R}^{d_{\text{seq}}}$. The hierarchical encoder processes the AST-derived graph structure and outputs a feature vector $h_{\text{hie}} \in \mathbb{R}^{d_{\text{hie}}}$. We concatenate these presentations to form the final feature vector: $z = [h_{\text{seq}}, h_{\text{hie}}]$ (see step ① in Figure 2, where $d = d_{\text{seq}} + d_{\text{hie}}$ is the total dimensionality of the combined feature representation. We set both $d_{\text{hie}}$ and $d_{\text{seq}}$ to 128 so the total dimensionality is $d = 256$.

**Maximum Mean Discrepancy (MMD) Loss.** The fundamental challenge is the domain gap of the SlithIR code between Solidity and Vyper. Even though they both follow a similar format, their SlithIR representations exhibit distributional differences due to compiler-specific optimizations, language-specific idioms, and syntactic variations.

To address this challenge, we need a learning objective that explicitly encourages the encoder to learn representations that are *invariant to the source language* while preserving the semantic information necessary for vulnerability detection. This is the role of Maximum Mean Discrepancy (MMD) loss in our framework. We employ MMD to minimize the distributional distance between Solidity and Vyper feature representations. MMD is a non-parametric statistical method that measures the distance between two probability distributions by comparing their mean embeddings in a Reproducing Kernel Hilbert Space (RKHS). Specifically, given a batch of feature vectors from Solidity $(z_1^S, z_2^S, \ldots, z_{n_S}^S)$ and Vyper $(z_1^V, z_2^V, \ldots, z_{n_V}^V)$. The MMD loss $\mathcal{L}_{\text{MMD}}$ is calculated in Equation (1).

$$\mathcal{L}_{\text{MMD}} = \left\| \frac{1}{n_S} \sum_{i=1}^{n_S} \phi(\mathbf{z}_i^S) - \frac{1}{n_V} \sum_{j=1}^{n_V} \phi(\mathbf{z}_j^V) \right\|_{\mathcal{H}}^2 \tag{1}$$

where $\phi(\cdot) : \mathbb{R}^d \to \mathcal{H}$ is the feature mapping function that maps input features into the RKHS $\mathcal{H}$,

**Kernel Design and Implementation.** The feature mapping function $\phi(\cdot)$ is implicitly defined through the kernel trick using a combination of linear and Gaussian Radial Basis Function (RBF) kernels. We use a composite kernel defined in Equation (2).

$$k(x, y) = \alpha k_{\text{Linear}}(x, y) + (1 - \alpha) k_{\text{RBF}}(x, y) \tag{2}$$

where $k_{\text{Linear}}(x, y) = x^T y$ captures linear relationships between features. And $k_{\text{RBF}}(x, y) = \exp(-\gamma \|(x-y)\|^2)$ captures non-linear similarities. $\alpha$ is the bandwidth hyperparameter that balances the contribution of linear and non-linear components. In practice, we use Equation (3) to calculate $\mathcal{L}_{\text{MMD}}$ to avoid explicit computation of the feature mapping $\phi(\cdot)$ while still measuring the distance between language distributions in RKHS $\mathcal{H}$. This transferable knowledge learning stage operates in a fully unsupervised manner, *requiring no labeled Vyper data*. This aligns perfectly with our problem setting, where labeled Vyper datasets are scarce. The loss only requires unlabeled samples from both domains, making it practical for low-resource language scenarios.

$$\begin{aligned} \mathcal{L}_{\text{MMD}} = & \frac{1}{n_S^2} \sum_{i=1}^{n_S} \sum_{i'=1}^{n_S} k(\mathbf{z}_i^S, \mathbf{z}_{i'}^S) + \frac{1}{n_V^2} \sum_{j=1}^{n_V} \sum_{j'=1}^{n_V} k(\mathbf{z}_j^V, \mathbf{z}_{j'}^V) \\ & - \frac{2}{n_S n_V} \sum_{i=1}^{n_S} \sum_{j=1}^{n_V} k(\mathbf{z}_i^S, \mathbf{z}_j^V) \end{aligned} \tag{3}$$

### C. Vulnerability Detection Module

Once the multi-view encoders are trained to produce language-agnostic representations, we proceed to the vulnerability detection task, leveraging the learned transferable knowledge. This process involves the following stages: supervised training on Solidity, and cross-language testing on Vyper.

**Training the Classification Network Using *Solidity* Vulnerability Detection Dataset.** In this stage, the weights of the sequential and hierarchical encoders from Stage ① are frozen to preserve the learned transferable knowledge. This ensures that the vulnerability-specific patterns learned from Solidity do not corrupt the language-agnostic representations. A classification network is added on top of the frozen encoders for vulnerability detection. We train the network using labeled Solidity smart contracts (safe and vulnerable), handling each vulnerability type separately.

The classification network consists of a multi-layer perceptron (MLP). It takes the concatenated feature vector $z = [h_{\text{seq}}, h_{\text{hie}}] \in \mathbb{R}^d$ from the frozen encoders as input, where $d = d_{\text{seq}} + d_{\text{hie}}$ represents the combined dimensionality. We employ two fully connected hidden layers with ReLU activation functions. The final output layer produces vulnerability predictions using a sigmoid function for binary classification (vulnerable or safe). For each vulnerability type (RE, WR, UT), we train a separate classification network.

**Reusing the Trained Model to Test on Vyper.** After training, the model can be applied directly to test Vyper smart contracts. The multi-view encoders and classification network process each contract to predict whether it is safe or vulnerable, without any additional fine-tuning or modification.

## V. Evaluation

We conduct extensive experiments to assess `Sol2Vy` in terms of vulnerability detection effectiveness. Additionally, we perform hyperparameter studies, runtime and few-shot analysis. We address seven research questions (**RQs**):

**Research Questions**

**RQ1:** How effective is the language-agnostic features alignment for learning transferable knowledge?
**RQ2:** What is the vulnerability detection performance of `Sol2Vy`?
**RQ3:** How does `Sol2Vy` compare with baseline methods?
**RQ4:** How do different hyperparameter settings impact vulnerability detection performance?
**RQ5:** How does each component in `Sol2Vy` affect the overall vulnerability detection performance?
**RQ6:** What are the effects of adding few-shot Vyper samples in Stage ②?
**RQ7:** What is the runtime overhead for `Sol2Vy`?

### A. Experimental Setting

We implement `Sol2Vy` using Transformer and Graph Attention Network in Pytorch 2.0. The sequential encoder is built with a 6-layer Transformer architecture with 8 attention heads and 128-dimensional embeddings, while the hierarchical encoder employs a 3-layer Graph Attention Network (GAT) with 128-dimensional node features. The classification network consists of a 2-layer MLP with ReLU activation and dropout ($p = 0.3$). For MMD computation, we use a composite kernel with $\alpha = 0.8$ and $\gamma = 0.005$. All the experiments were conducted on a computer with Intel Core i9-12900K CPU (16 cores, 3.2GHz), 64GB RAM, and an NVIDIA RTX 4090 GPU with 24GB VRAM.

**General Dataset.** We collect unlabeled Solidity and Vyper smart contracts from diverse and authoritative sources, including Etherscan [57] and GitHub repositories [58]. We collect 1842 Solidity contracts and 1193 Vyper contracts. It is important to note that the general dataset used to learn transferable language-agnostic knowledge has *no overlap* with the vulnerability detection dataset used in the vulnerability detection task.

**Learning Transferable Knowledge.** We use the general dataset to train the multi-view architecture to learn language-invariant transferable knowledge. We train the sequential and hierarchical encoder by minimizing the MMD loss between Solidity and Vyper. We employ a curriculum learning strategy where we gradually increase the complexity of the unlabeled contract dataset throughout training. We measure the complexity based on a combination of AST depth, SlithIR instruction diversity and contract scale. This approach helps the model learn fundamental transferable knowledge before tackling more challenging distributional differences. The training stops until the validation MMD loss drops below 0.2.

### B. RQ1: Evaluating Quality of Transferable Knowledge

To validate our assumption that MMD-based alignment captures meaningful semantic correspondence, we conduct a quantitative analysis of the learned feature representations. Specifically, our goals are threefold: (**G1**) Semantically equivalent Solidity-Vyper pairs have a high similarity; (**G2**) The learned features are truly language-agnostic; (**G3**) Prevent representation collapse, where the encoder maps all inputs to nearly identical vectors to trivially minimize MMD loss.

**G1: Semantic Equivalency.** We construct a set of semantically equivalent SlithIR statement pairs from Solidity and Vyper, respectively. The pairs are chosen from six representative patterns: balance update, authorization check, external call, array push, reentrancy pattern and randomness condition. For each pattern, we select the most frequent statement pairs for assessment. For example, `<REF>->balances[sender]` and `<REF>->self.balances[sender]` are an equivalent pair between Solidity and Vyper SlithIR that belongs to the balance update category. Table I shows that for all six representative patterns, the cosine similarity scores are significantly higher after MMD training, indicating that semantically equivalent statements are closer in feature space, regardless of their source language (**Achieving G1**).

TABLE I: Semantic similarity of equivalent SlithIR patterns.

| Pattern Type | Cos. Sim. (Before) | Cos. Sim. (After) |
|---|---|---|
| Balance Update | 0.21 | **0.42** |
| Authorization Check | 0.12 | **0.39** |
| External Call | 0.13 | **0.40** |
| Array Push | 0.15 | **0.37** |
| Reentrancy Pattern | 0.18 | **0.43** |
| Randomness Condition | 0.17 | **0.39** |

**G2: Language Discriminability.** To quantitatively measure whether the features are truly language-agnostic, we train a simple logistic regression classifier to predict SlithIR statements from Solidity or Vyper. Before MMD training, this classifier achieves 94.3% accuracy, indicating that features contain strong language-specific signals. After MMD training, classification accuracy drops to 52.1% (near random chance), confirming that language-specific patterns have been effectively suppressed (**Achieving G2**).

**G3: Representation Collapse Prevention.** A critical concern in domain alignment methods is representation collapse, where the encoder might learn trivial solutions by mapping all inputs to identical or near-identical representations. To verify that our representations remain expressive, we measure the variance within each language's feature distribution. Results show that our intra-class variance remains stable, decreasing only slightly from 32.1 to 28.9 for Solidity and 39.5 to 37.6 for Vyper. This modest reduction indicates that MMD training compresses language-specific noise while preserving semantic diversity (**Achieving G3**). Representation collapse is prevented through two complementary mechanisms inherent to our framework design:

- First, our multi-view architecture provides implicit regularization against collapse. The self-attention mechanisms within the sequential encoder preserve positional and ordering information that could be lost in collapsed representations. Similarly, the hierarchical encoder processes graph structures through GAT layers that aggregate neighborhood information, which requires diverse node representations. If representations collapsed, the attention weights in both encoders would become uninformative, preventing the model from learning useful patterns in the unlabeled corpora.
- Second, we employ ***curriculum learning*** during Stage ①, gradually increasing dataset complexity. This progressive training strategy encourages the model to first learn coarse-grained structural patterns before refining fine-grained semantic distinctions, naturally preventing premature convergence to degenerate solutions.

In summary, semantically equivalent Solidity-Vyper statements are mapped to a closer region in the feature space (**G1**). After MMD-based training, the encoders largely encode language-agnostic artifacts (**G2**). Furthermore, this alignment is achieved without sacrificing representational expressiveness (**G3**). These results collectively verify that our MMD-based multi-view alignment module ***learns language-agnostic features*** that form a reliable foundation for zero-shot vulnerability detection on Vyper contracts.

### C. RQ2: Vulnerability Detection

After training the multi-view model (consisting of the sequential and hierarchical encoders) on the general datasets to learn transferable knowledge, we then use the *Solidity* vulnerability detection dataset to train the classification network. Once trained, we directly reuse the encoders together with the Solidity-trained classifier to evaluate Vyper smart contracts, enabling cross-language vulnerability detection *without requiring any Vyper-labeled data*.

**Vulnerability Detection Dataset in *Solidity*.** We build this dataset from multiple popular smart contract repositories like SmartBugs [59], SWC-registry [60], DAppSCAN [61], Etherscan [57]. Since existing datasets often contain inconsistent or incorrect annotations, we implement a systematic label verification process to ensure high data quality. For initial vulnerability labeling, we employ multiple established tools, including Slither [5], Securify [6], Mythril [4], Echidna [41], which provide automated detection capabilities for known vulnerability patterns. We apply a consensus criterion where vulnerabilities must be identified by at least two independent tools to go through the subsequent manual verification.

Then, we manually verify every candidate Solidity label with a two-reviewer, auditable checklist: for collected samples, a contract is first flagged only when at least two tools agree on the same vulnerability type, after which two reviewers independently confirm the label by recording (1) the vulnerability type (RE/WR/UT), (2) exact evidence locations (function + line range / IR snippet), and (3) a brief exploit rationale. RE requires identifying an external call enabling re-entry before critical state update, WR requires a security-relevant decision using manipulable entropy (e.g., timestamp/block data), and UT requires a value-transfer whose success is not checked or enforced by revert/require. Labels are accepted only when both reviewers agree on type and evidence. Otherwise, the sample is marked disputed and excluded.

After the process, we collect 1275, 753, 785 and 813 Solidity contracts that are labeled as safe, reentrancy (RE), weak randomness (WR) and unchecked transfer (UT), respectively. We allocate 80% of the data for training (denoted as $D_{\text{Sol-Train}}$) and 20% for testing (denoted as $D_{\text{Sol-Test}}$). Note that the vulnerability detection dataset *has no overlap* with the general dataset used for learning transferable knowledge.

**Training Classification Layer on Solidity.** We freeze the sequential and structural encoders and append a classification layer to the combined output. The classification network is trained on $D_{\text{Sol-Train}}$ until the loss falls below 0.5.

**Vulnerability Detection Dataset in *Vyper*.** Due to the limited availability of existing labeled Vyper datasets, we construct a vulnerability detection dataset for *Vyper* from scratch in two ways. First, we collect Vyper contracts from public repositories and apply Slither [5] and Mythril [4] (both of which support Vyper) to label them. Second, we create additional vulnerable contract variants by randomly introducing the three

TABLE II: Vyper contract length (Line of Code, LoC).

| Type | Total | 20–100 LoC | 101–200 LoC | ≥201 LoC |
|---|---|---|---|---|
| Safe | 434 | 120 | 140 | 174 |
| RE | 236 | 60 | 80 | 96 |
| WR | 226 | 55 | 75 | 96 |
| UT | 307 | 70 | 100 | 137 |
| All | 1203 | 305 | 395 | 503 |

target vulnerability patterns into safe contracts. We go through the same labeling and manual verification process as the Solidity dataset. For injected samples, reviewers additionally confirm the injected code is reachable and semantically consistent (not dead/unexecutable) before inclusion.

Our final dataset comprises 434, 236, 226 and 307 Vyper contracts that are labeled as safe, RE, WR and UT, respectively. The details are shown in Table II. All the safe contracts and 160, 143, and 205 RE, WR and UT vulnerable contracts are collected from public repositories, despite our significant efforts to gather Vyper vulnerable samples.

**Results.** We first evaluate the Solidity-trained classifier on the Solidity test dataset, $D_{Sol-test}$, which represents *the upper bound of achievable performance* since the classifier is trained and tested on the same language. In this setting, no cross-language knowledge transfer is involved, and thus no inter-language information loss occurs. The classifier, trained and evaluated exclusively on Solidity contracts, achieves FPR/FNR values of 0.09/0.13, 0.08/0.12 and 0.11/0.14 for RE, WR and UT, respectively.

We next reuse the Solidity-trained classifier to evaluate Vyper contracts using $D_{Vy-all}$. The results are shown in Table III. It achieves FPR/FNR scores of 0.13/0.15, 0.12/0.14 and 0.13/0.15 for RE, WR and UT. Note that this evaluation is performed in a zero-shot setting; when few-shot training with a balanced combination is applied, both FPR and FNR decrease substantially (see Section V-G).

Comparing the performance on Solidity and Vyper, we observe *only minimal degradation*: an increase of just 0.04/0.02, 0.04/0.02, and 0.02/0.01 in FPR/FNR for RE, WR, and UT, respectively. This minimal degradation demonstrates the high quality of the transferable knowledge learned and the strong semantic preservation achieved by our approach.

To further understand why our model maintains strong performance across languages, we analyze how `Sol2Vy` leverages transferable knowledge during vulnerability detection. `Sol2Vy` effectively connects the learned cross-language semantic representations to the three vulnerability types (RE, WR, and UT) through pattern-type alignment. As shown in Table I in RQ1 (V-B), the model aligns critical vulnerability-specific patterns such as "Balance Update" for RE and "Authorization Check" for UT across Solidity and Vyper. This alignment ensures that semantically equivalent vulnerability cues retain similar representations even when implemented with language-specific constructs. For example, improved alignment enables the classifier to correctly identify Reentrancy vulnerabilities despite differences such as the use of `MODIFIER_CALL` in Solidity versus `INTERNAL_CALL` in Vyper. These aligned representations allow the downstream classifier to generalize effectively, explaining the negligible performance drop when transferring from Solidity to Vyper.

### *D. RQ3: Baselines Comparison*

We next compare the vulnerability detection performance of `Sol2Vy` against several baselines. Existing approaches for detecting vulnerabilities in Vyper smart contracts rely primarily on traditional program analysis techniques, and no deep learning–based models currently exist for Vyper. To provide a comprehensive evaluation, we design three neural baselines adapted from state-of-the-art models. Additionally, we include an LLM-based baseline using direct prompting. Below are the baselines included in our evaluation:

- Slither [5]: A static analysis tool that extracts ASTs and detects vulnerabilities via manually crafted pattern matching
- Mythril [4]: A dynamic analysis tool that applies symbolic execution to EVM bytecode generated from Vyper contracts.
- Vyper-Trained Classification Network: A baseline where we directly train and test our classification network on Vyper, without any transferable knowledge learning.
- CodeT5+ [62]: A Transformer-based model pre-trained on a large multi-language corpus of source code.
- GraphCodeBERT [63]: A Transformer-based model pre-trained on multiple programming languages using both sequential context and data-flow information.
- LLM Prompting: We provide vulnerability descriptions for each vulnerability type and prompt the LLM for detection. We select three LLM backends that demonstrate high performance on code: GPT-4.1 [64], CodeLlama-7b [65], and Qwen-Coder-7b [66].
- Joint Training Baseline: To validate whether freezing is necessary for cross-language generalization, we evaluate a joint training variant where we do not freeze the multi-view encoders during Stage ②. Instead, we fine-tune the encoders and classifier end-to-end on the Solidity vulnerability dataset while concurrently applying the MMD alignment objective on unlabeled Solidity/Vyper batches. Specifically, we use the loss function $\mathcal{L}_{\text{Joint}} = \mathcal{L}_{\text{CLS}} + \lambda \mathcal{L}_{\text{MMD}}$ instead of original loss function $\mathcal{L}_{\text{MMD}}$. This baseline tests whether supervised gradients improve discriminative power at the cost of language invariance, compared to the frozen strategy.

For both CodeT5+ and GraphCodeBERT, we append a sigmoid classification layer on top of the encoder and fine-tune the model for the vulnerability detection task.

For baseline comparison, the Vyper vulnerability detection dataset $D_{Vy-all}$ is split into two parts: 80% for training (denoted as $D_{Vy-1}$), and 20% for testing (denoted as $D_{Vy-2}$).

**Results.** The results are presented in Table IV. We observe that `Sol2Vy` consistently outperforms all baseline methods. Below, we provide a detailed analysis of these results.

***Comparison with Program Analysis-based Methods***. For Slither, which is based on static analysis techniques, consider reentrancy as an example. `Sol2Vy` achieves an FPR/FNR

TABLE III: A Solidity-trained model evaluated on both Solidity and Vyper samples.

| | RE | | WR | | UT | |
|---|---|---|---|---|---|---|
| | FPR | FNR | FPR | FNR | FPR | FNR |
| Train and Test on Solidity (Optimal Setting) | 0.09 | 0.13 | 0.08 | 0.12 | 0.11 | 0.14 |
| Train on Solidity & Test on Vyper | 0.13 | 0.15 | 0.12 | 0.14 | 0.13 | 0.15 |

TABLE IV: Comparison between `Sol2Vy` and Baseline Methods.

| | | RE | | WR | | UT | |
|---|---|---|---|---|---|---|---|
| | | FPR | FNR | FPR | FNR | FPR | FNR |
| Program Analysis-based | Slither | 0.46 | 0.58 | 0.48 | 0.55 | 0.45 | 0.52 |
| | Mythril | 0.38 | 0.49 | 0.40 | 0.46 | 0.42 | 0.48 |
| Deep Learning-based (Train & Test on Vyper) | Vyper-trained Classification Network | 0.24 | 0.31 | 0.29 | 0.35 | 0.26 | 0.32 |
| | CodeT5+ | 0.28 | 0.34 | 0.32 | 0.38 | 0.30 | 0.36 |
| | GraphCodeBERT | 0.32 | 0.38 | 0.35 | 0.42 | 0.33 | 0.39 |
| LLM-based Methods | GPT-4.1 | 0.41 | 0.52 | 0.44 | 0.56 | 0.40 | 0.50 |
| | CodeLlama-7b | 0.45 | 0.55 | 0.47 | 0.59 | 0.43 | 0.53 |
| | Qwen-Coder-7b | 0.39 | 0.50 | 0.42 | 0.54 | 0.38 | 0.49 |
| Joint Training Baseline | `Sol2Vy` w/o freezing | 0.48 | 0.53 | 0.68 | 0.78 | 0.47 | 0.56 |
| Our Approach | `Sol2Vy` | **0.11** | **0.16** | **0.10** | **0.15** | **0.12** | **0.17** |

of 0.11/0.16 compared to Slither's 0.46/0.58. This large performance gap arises from the *pattern-matching limitation* in Slither's approach when applied to Vyper contracts. Slither relies heavily on manually crafted syntactic patterns, which fail in Vyper due to the language's different syntactic structures and programming paradigms. Our `Sol2Vy` abstracts away these syntactic differences, enabling pattern recognition at the semantic level.

For Mythril, which is based on symbolic execution, `Sol2Vy` still outperforms it. Mythril's effectiveness is constrained by the path explosion problem, where the number of execution paths grows exponentially with program complexity. As a result, Mythril often fails to explore deep or complex contract behaviors and incurs significant computational overhead.

***Comparison with Deep-Learning-based Models***. Note that no deep learning–based models currently exist for Vyper, so we design three neural baselines. The first is a *Vyper-trained classification network*, where we directly train and test the classifier on Vyper contracts without any transferable-knowledge learning. As expected, this Vyper-only model performs worse than `Sol2Vy`, primarily due to the *data scarcity problem*: the available Vyper training set is too small to support training a robust detection model. Moreover, collecting a large number of vulnerable Vyper samples is difficult, which motivates our approach of reusing a well-trained Solidity model to detect vulnerabilities in Vyper contracts.

For the two state-of-the-art pretrained code models: GraphCodeBERT and CodeT5+. Both models are pretrained on large-scale multi-language code corpora and have demonstrated strong performance on various code understanding tasks. We attach a classification head to each model, fine-tune them on $D_{Vy-1}$, and evaluate them on $D_{Vy-2}$. GraphCodeBERT achieves FPR/FNR scores of 0.32/0.38, 0.35/0.42, and 0.33/0.39 for RE, WR, and UT, respectively, while CodeT5+ performs slightly better with scores of 0.28/0.34, 0.32/0.38, and 0.30/0.36 for FPR/FNR. In comparison, `Sol2Vy` substantially outperforms both models.

***Comparison with LLM-based Models***. We further compare `Sol2Vy` with three LLM-based baselines (GPT-4.1, CodeLlama-7b, and Qwen-Coder-7b) using direct prompting for vulnerability detection. Specifically, we provide the full contract code together with a vulnerability description prompt and ask the model to determine whether the contract is vulnerable. As shown in Table IV, `Sol2Vy` consistently outperforms all LLM-based methods across RE, WR, and UT. For example, on RE detection, `Sol2Vy` achieves an FPR/FNR of 0.11/0.16, significantly lower than GPT-4.1 (0.41/0.52), CodeLlama-7b (0.45/0.55), and Qwen-Coder-7b (0.39/0.50). This performance gap indicates that while LLMs possess general code understanding capabilities, simple prompting without explicit structural modeling or cross-language alignment is insufficient for accurate vulnerability detection in Vyper.

***Comparison with Joint Training Baseline***. Instead of freezing the multi-view encoders in Stage ②, we additionally evaluate a joint-training baseline that does not freeze encoder parameters. In this baseline, we fine-tune the encoders and classifier end-to-end on the labeled Solidity vulnerability dataset while simultaneously applying the MMD alignment objective on unlabeled Solidity/Vyper batches. The overall objective is $\mathcal{L}_{\text{Joint}} = \mathcal{L}_{\text{CLS}} + \lambda \mathcal{L}_{\text{MMD}}$, which tests whether freezing is essential for superior cross-language generalization. As shown in Table IV, the joint training only achieves FPR/FNR scores of 0.48/0.53, 0.68/0.78, 0.47/0.56 for RE, WR and UT, which is significantly higher than `Sol2Vy`. These results suggest that in the joint training paradigm, end-to-end supervised updates partially overwrite language-agnostic features

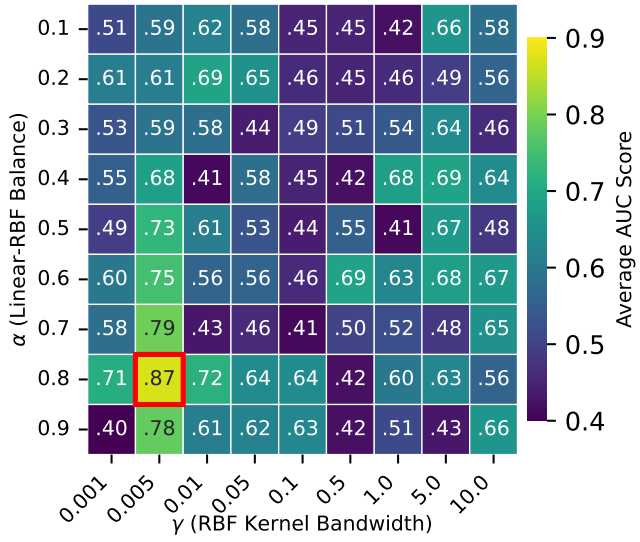


Fig. 3: Heatmap showing the average AUC value when varying the hyperparameters $\alpha$ and $\gamma$.

learned in Stage ①. In contrast, freezing preserves the aligned representation space, enabling the Solidity-trained classifier to generalize with minimal degradation on Vyper.

### E. RQ4: Hyperparameter Study

To understand the impact of kernel composition on transferable knowledge learning effectiveness, we conduct a hyperparameter study focusing on $\alpha$ and $\gamma$ parameters in Equation (2). We systematically vary $\alpha$ and $\gamma$ to evaluate their combined effect on vulnerability detection performance. For each parameter combination, we train `Sol2Vy` using the same general dataset and Solidity vulnerability detection dataset, and evaluate on all three vulnerability types using the same Vyper test dataset. We report the AUC score for each setting.

**Results.** Figure 3 presents the average AUC scores across three vulnerability types for different hyperparameter combinations. We observe that the optimal combination is $\alpha = 0.8$ and $\gamma = 0.005$, achieving an average AUC of 0.87. This configuration can be explained in two aspects: (1) *High linear component emphasis:* The optimal configuration heavily favors the linear kernel component ($\alpha = 0.8$), suggesting that the domain gap between Solidity and Vyper is primarily characterized by linear transformations rather than complex nonlinear mappings. This indicates that while the two languages have syntactic differences, their semantic structures maintain largely linear relationships in the representation space. (2) *Fine-grained RBF Bandwidth:* The very small $\gamma$ value (0.005) captures long-range semantic similarities while being robust to local syntactic variations. This bandwidth setting ensures that semantically similar code patterns across languages are mapped to nearby points in feature space. Larger $\gamma$ values (over 0.1) create overly narrow kernels that fragment the feature space and fail to generalize across language boundaries.

TABLE V: Ablation Study Results.

| | RE | | WR | | UT | |
|---|---|---|---|---|---|---|
| | FPR | FNR | FPR | FNR | FPR | FNR |
| `Sol2Vy` | **0.11** | **0.16** | **0.10** | **0.15** | **0.12** | **0.17** |
| `Sol2Vy` w/o seq. | 0.42 | 0.48 | 0.29 | 0.25 | 0.31 | 0.39 |
| `Sol2Vy` w/o hie. | 0.28 | 0.34 | 0.55 | 0.52 | 0.28 | 0.35 |
| `Sol2Vy` w/o both | 0.45 | 0.52 | 0.58 | 0.54 | 0.46 | 0.73 |

**Sensitivity Analysis.** The performance landscape exhibits significant variability, indicating high sensitivity to parameter selection. (1) $\alpha$ *sensitivity:* Performance shows a sharp peak at $\alpha = 0.8$, with substantial degradation at both higher ($\alpha = 0.9$) and lower ($\alpha \leq 0.7$) values. This suggests a critical threshold where linear alignment mechanisms become dominant for effective cross-language transfer. (2) $\gamma$ *sensitivity* The optimal $\gamma = 0.005$ represents a narrow sweet spot. Both smaller ($\gamma = 0.001$) and larger ($\gamma \geq 0.01$) values show decreased performance, indicating that $\gamma$ must be precisely tuned to capture the appropriate scale of semantic similarities.

### F. RQ5: Ablation Study

To isolate the contribution of each component in `Sol2Vy`, we conduct an ablation study that systematically removes the key elements of `Sol2Vy`. Specifically, we consider the following four scenarios for comparison:

- `Sol2Vy`: The complete framework with both sequential and hierarchical encoders, trained with MMD-based transferable knowledge learning.
- `Sol2Vy` without sequence encoder: This variant uses only the hierarchical encoder during both the unsupervised transferable knowledge learning stage and the vulnerability detection stage, without the sequential encoder.
- `Sol2Vy` without hierarchical encoder: This variant uses only the sequential encoder throughout the pipeline, without the hierarchical encoder.
- `Sol2Vy` without both sequence and hierarchical encoder (no transferable learning): This variant is trained directly on Solidity and tested on Vyper without learning the transferable knowledge.

For each configuration, we train the model using the same Solidity vulnerability detection dataset and evaluate on the Vyper test set across all three vulnerability types (RE, WR, UT). We report both FPR and FNR metrics on the vulnerability detection performance. The results are shown in Table V. From the results, we have the following findings:

**Finding 1: Complementary Contributions of the Multi-View Encoders.** From Table V, we can see that `Sol2Vy` with both hierarchical encoder and sequential encoders perform best. While remocostumed variable nameving either encoder results in a performance degradation for all the vulnerability types and metrics. Removing both hierarchical and sequential encoders leads to the most severe performance decrease. These results indicate that the sequential and hierarchical encoders provide ***non-redundant*** views of SlithIR semantics, and that the full multi-view architecture is necessary to fully realize the benefits of transferable knowledge.

**Finding 2: Different Vulnerability Types Respond Differently to Component Removal.** Another pattern revealed is that the three vulnerability categories respond differently to the removal of specific components, reflecting their inherent semantic structures and the type of invariances required for cross-language transfer.

For example, reentrancy attacks are fundamentally defined by a ***temporal dependency pattern*** of: external call, state update and reentrant invocation path. Even minor deviations in SlithIR order can obscure this signal. Vyper's aggressive inlining of modifier-like logic causes substantial operation-order drift, which the sequential encoder compensates for. Removing the sequential encoder increases FPR/FNR of RE from 0.11/0.16 to 0.42/0.48. While removing the hierarchical encoder only increases FPR/FNR from 0.11/0.16 to 0.28/0.34, which is relatively minimal. Thus, the RE detection requires ***mostly temporal invariance***, which only the ***sequential encoder*** can provide.

On the other hand, WR vulnerabilities rely on identifying flawed entropy sources buried inside ***nested conditions***, including multi-branch logic or deep AST regions. These patterns produce ***long-range structural dependencies*** but are not highly sensitive to local instruction order. This is consistent with the results: Removing the hierarchical encoder makes the FPR/FNR of WR increase from 0.10/0.15 to 0.55/0.52, which is nearly identical to the worst scenario (removing both encoders) of 0.58/0.54. So WR detection requires ***structural invariance***, which heavily depends on the ***hierarchical encoder***. For UT, an unchecked transfer combines a structural check and a sequential misuse of balance transfer. This hybrid nature explains why performance on UT drops moderately, but not catastrophically when either encoder is removed.

### *G. RQ6: Few-shot Analysis*

To investigate the potential benefits of incorporating limited Vyper data during training, we conduct a few-shot analysis to examine how small amounts of labeled Vyper contracts influence `Sol2Vy`'s performance. This analysis captures a practical scenario in which practitioners may have access to only a handful of labeled Vyper samples and wish to leverage them to further improve detection accuracy beyond the pure zero-shot transfer setting.

We evaluate fifteen different few-shot configurations by adding varying numbers and compositions of Vyper smart contracts to the training set, in addition to the original Solidity vulnerability detection data. The configurations span from 4-shot to 20-shot scenarios with three composition strategies: (1) **Safe-only additions**: Adding only safe Vyper smart contracts (4S, 8S, 12S, 16S, 20S), (2) **Vulnerable-only additions**: Adding only vulnerable Vyper smart contracts (4V, 8V, 12V, 16V, 20V), and (3) **Balanced additions**: Adding equal numbers of safe and vulnerable Vyper smart contracts (2S+2V, 4S+4V, 6S+6V, 8S+8V, 10S+10V). Note that the added Vyper smart contracts *have no overlap* with the Vyper contracts for testing. For each configuration, we retrain the classification network while maintaining the same frozen multi-view encoder architecture from the unsupervised transferable knowledge learning stage. To ensure statistical robustness, we repeat each few-shot configuration with three different random samples and report the average performance.

**Results.** Table VI presents the few-shot analysis results across all three vulnerability types. We have the following findings:

- **Progressive Performance Gains**: Adding small amounts of balanced Vyper training data consistently improves performance across all vulnerability types. The gains are most pronounced when moving from zero-shot to few-shot scenarios, with diminishing returns as more contracts are added.
- **Saturation Effects**: Performance improvements exhibit strong saturation effects beyond 16-20 examples, suggesting that `Sol2Vy` can effectively leverage small amounts of target-domain data without requiring extensive labeled datasets. This saturation behavior validates our transferable knowledge learning approach: the frozen encoders already capture most cross-language semantic patterns, requiring only minimal fine-tuning through target-domain examples to bridge the remaining gap. The practical implication is significant since practitioners can achieve near-optimal performance by annotating just 12-16 carefully selected Vyper contracts. While further adding more Vyper samples may lead to overfitting.
- **Composition Sensitivity**: The composition of few-shot examples significantly impacts performance outcomes, with balanced datasets containing both safe and vulnerable contracts striking a good trade-off between FNR and FPR. This outcome is because a balanced combination can ***promote inter-class discrimination***, allowing encoder decision boundaries to be calibrated against both normal and risky samples. Additionally, balanced sampling minimizes feature space collapse towards a ***single-class direction***, leading to more stable classifier gradients in Stage ② fine-tuning. While including only vulnerable samples or safe samples will lead to a relatively high FPR or FNR rate.

**Shot Selection Strategy.** In high-risk environments where omission of vulnerabilities is unacceptable, slightly biasing few-shot samples toward vulnerable contracts (e.g., 0S+8V) may further minimize the FNR rate, even though such configurations could introduce a moderate increase in FPR rate. For example, configurations such as 0S+8V in the 8-shot setting and 4S+8V in the 12-shot setting prioritize exposure to abnormal execution behaviors, encouraging the classification layer to adopt a more conservative decision boundary. This design shifts the model toward recognizing subtle vulnerability patterns rather than requiring high confidence from structural normality cues. As a result, the classifier becomes more strict on potential risk, effectively *detecting more true vulnerabilities* at the cost of occasionally *misclassifying safe contracts*. This pattern is aligned with practitioners' intent in high-risk environments, such as DeFi protocols or DAO treasury management, where missing a vulnerability can cause irreversible financial impact, while falsely alerting a safe contract typically incurs only minor manual validation overhead.

TABLE VI: Few-shot analysis results (S = safe Vyper contracts; V = vulnerable Vyper contracts).

| | | **0-shot** | **4-shot** | | | **8-shot** | | | **12-shot** | | | **16-shot** | | | **20-shot** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | (base) | 4S | 4V | 2S+2V | 8S | 8V | 4S+4V | 12S | 12V | **6S+6V** | 16S | 16V | 8S+8V | 20S | 20V | 10S+10V |
| RE | FPR | 0.11 | 0.10 | 0.27 | 0.09 | 0.09 | 0.22 | 0.08 | **0.05** | 0.17 | 0.08 | 0.07 | 0.20 | 0.08 | 0.06 | 0.29 | 0.08 |
| | FNR | 0.16 | 0.25 | 0.13 | 0.12 | 0.23 | 0.11 | 0.10 | 0.18 | **0.07** | 0.08 | 0.29 | 0.10 | 0.07 | 0.28 | 0.09 | 0.08 |
| WR | FPR | 0.10 | 0.09 | 0.28 | 0.08 | 0.08 | 0.22 | 0.07 | **0.05** | 0.19 | 0.06 | 0.09 | 0.28 | 0.07 | 0.08 | 0.31 | 0.07 |
| | FNR | 0.15 | 0.24 | 0.13 | 0.12 | 0.23 | 0.11 | 0.11 | 0.20 | **0.08** | 0.09 | 0.29 | 0.10 | 0.09 | 0.29 | 0.10 | 0.08 |
| UT | FPR | 0.12 | 0.11 | 0.27 | 0.10 | 0.10 | 0.23 | 0.09 | **0.07** | 0.19 | 0.08 | 0.10 | 0.32 | 0.07 | 0.09 | 0.31 | 0.07 |
| | FNR | 0.17 | 0.25 | 0.14 | 0.13 | 0.24 | 0.12 | 0.11 | 0.18 | 0.08 | **0.06** | 0.30 | 0.11 | 0.08 | 0.27 | 0.10 | 0.08 |

TABLE VII: Runtime overhead: average time (s) per contract. We report the mean ± CI over five runs.

| Contract Size | `Sol2Vy` | Mythril | Slither |
|---|---|---|---|
| Small (20-100 lines) | 0.28± 0.16 | 24.3 ± 21.1 | 0.15 |
| Medium (101-200 lines) | 0.42± 0.13 | 87.0 ± 43.6 | 0.21 |
| Large ($\geq$ 201 lines) | 0.65± 0.21 | 303 ± 264.4 | 0.31 |

**Cross-Vulnerability Transfer.** We further explore whether few-shot examples for one vulnerability type can benefit the detection of other types. To investigate this, we conduct experiments where we add few-shot examples labeled for RE and measure the impact on WR and UT detection. Our results show limited cross-vulnerability transfer: adding 8 reentrancy-specific Vyper contracts improves reentrancy detection by 0.05 AUC but provides only 0.01 improvement for WR and UT. This suggests that vulnerability-specific patterns are largely orthogonal in the feature space, and practitioners should allocate annotation effort proportionally across the vulnerability types they care about rather than focusing on a single type.

### *H. RQ7: Runtime Overhead Analysis*

To assess the practical applicability of `Sol2Vy`, we conduct a runtime overhead analysis comparing our approach against traditional static and dynamic methods. We measure the end-to-end analysis time for each tool on the Vyper vulnerability detection dataset, which contains contracts ranging from 50 to 800 lines of code. For each tool, we measure: (1) preprocessing time (including compilation and IR generation where applicable), (2) core analysis time, and (3) total wall-clock time per contract. We exclude dataset loading and initialization overhead, focusing solely on per-contract analysis time.

To reduce noise, we select different seeds and run `Sol2Vy` and Mythril five times for each contract. Then we report the mean runtime with 95% confidence intervals (CI). Slither is a static deterministic tool so we only run it once. This provides a statistically supported estimate of the typical runtime and quantifies uncertainty due to run-to-run variation.

**Results.** Table VII presents the runtime overhead comparison across different contract size categories. `Sol2Vy` demonstrates superior efficiency compared to dynamic analysis while maintaining competitive performance with static methods. For small contracts (20-100 lines), `Sol2Vy` achieves an average analysis time of 0.28 seconds, slightly higher than Slither's 0.15 seconds but orders of magnitude faster than Mythril's 24.3 seconds. For medium-sized contracts (101-200 lines), `Sol2Vy`'s analysis time increases sublinearly to 0.42 seconds, while Mythril's time explodes to 87.0 seconds due to path explosion in symbolic execution. For large contracts (over 201 lines), the efficiency gap widens further: `Sol2Vy` requires only 0.65 seconds on average, compared to Mythril's over 300 seconds. Notably, Slither still maintains the fastest analysis time for all scales. But as shown in Table IV, this speed comes at the cost of significantly lower detection accuracy (FPR of 0.45-0.48 compared to `Sol2Vy`'s around 0.11).

**Analysis.** The efficiency of `Sol2Vy` stems from the architectural advantage of one-pass encoding: Unlike Mythril's iterative symbolic execution, `Sol2Vy` performs a single forward pass through the multi-view encoders and classification network. These results demonstrate that `Sol2Vy` achieves an optimal balance between detection accuracy and computational efficiency, making it suitable for both on-demand analysis and large-scale automated scanning pipelines.

## VI. Discussion on Generalization

Considering that Solidity is the most common language in blockchain, we have developed techniques to transfer knowledge from Solidity to Vyper. To validate the effectiveness of the techniques, we conducted an evaluation that tested the transferability from Solidity to Vyper for the downstream vulnerability detection task. In theory, the techniques could apply generally. However, it is important to note that our results are specific to the evaluated scenarios. In order to ascertain the effectiveness of this approach across other downstream tasks and other smart contract languages, further investigation and comprehensive testing are needed.

## VII. Conclusion

In this work, we introduced `Sol2Vy`, the first framework enabling deep learning–based vulnerability detection for Vyper smart contracts without requiring any labeled Vyper training data. The core insight is that SlithIR, a language-agnostic intermediate representation, bridges Solidity and Vyper by abstracting away syntactic differences while preserving essential semantics. Leveraging this, we designed a three-stage framework that integrates unsupervised transferable knowledge learning, supervised vulnerability detection on Solidity, and zero-shot testing on Vyper. Experiments across key vulnerability types, including reentrancy, weak randomness, and unchecked transfer, show that `Sol2Vy` achieves strong zero-shot performance on Vyper and significantly outperforms prior methods lacking cross-language transfer.

## REFERENCES


[1] P. De Rosa, S. Queyrut, Y.-D. Bromberg, P. Felber, and V. Schiavoni, "Phishinghook: Catching phishing ethereum smart contracts leveraging evm opcodes," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, 2025, pp. 265–266.

[2] P. De Rosa, P. Felber, and V. Schiavoni, "Scamdetect: Towards a robust, agnostic framework to uncover threats in smart contracts," in *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. IEEE, 2025, pp. 242–244.

[3] J. Liao, B. Gong, W. Sun, F. Zhang, Z. Ning, M. H. Au, and W. Shi, "Bft-rand: Low-latency random number provider for bft smart contracts," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 389–402.

[4] N. Sharma and S. Sharma, "A survey of mythril, a smart contract security analysis tool for evm bytecode," *Indian J Natural Sci*, vol. 13, no. 75, pp. 39–41, 2022.

[5] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 67–82.

[7] B. Boi, C. Esposito, and S. Lee, "Smart contract vulnerability detection: The role of large language model (llm)," *ACM SIGAPP Applied Computing Review*, vol. 24, no. 2, pp. 19–29, 2024.

[8] Y. Liu, Y. Xue, D. Wu, Y. Sun, Y. Li, M. Shi, and Y. Liu, "Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation," *Network and Distributed System Security (NDSS) Symposium*, 2024.

[9] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning." in *NDSS*, 2023.

[10] S. So, S. Hong, and H. Oh, "{SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1361–1378.

[11] Vyper. Vyper: A contract-oriented, pythonic programming language that targets the ethereum virtual machine. https://docs.vyperlang.org/en/stable/.

[12] A. Ranchal-Pedrosa and V. Gramoli, "Zlb: A blockchain to tolerate colluding majorities," in *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2024, pp. 209–222.

[13] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[14] T. Sharma, Z. Zhou, A. Miller, and Y. Wang, "A {Mixed-Methods} study of security practices of smart contract developers," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2545–2562.

[15] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[16] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "Ethir: A framework for high-level analysis of ethereum bytecode," in *International symposium on automated technology for verification and analysis*. Springer, 2018, pp. 513–520.

[17] X. Yi, D. Wu, L. Jiang, Y. Fang, K. Zhang, and W. Zhang, "An empirical study of blockchain system vulnerabilities: Modules, types, and patterns," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 709–721.

[18] L. Jin, Y. Cao, Y. Chen, D. Zhang, and S. Campanoni, "Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 1, pp. 650–664, 2022.

[19] C. Ruggiero, P. Mazzini, E. Coppa, S. Lenti, and S. Bonomi, "Sok: a unified data model for smart contract vulnerability taxonomies," in *Proceedings of the 19th International Conference on Availability, Reliability and Security*, 2024, pp. 1–13.

[20] polkadot. Polkadot: A powerful, secure heart of web3. https://polkadot.com/.

[21] EOSIO. Eosio: A free, open-source blockchain software protocol. https://developers.eos.io/.

[22] binance. Binance smart chain: An evm-compatible blockchain. https://www.bnbchain.org/en/bnb-smart-chain.

[23] P. Ma, N. He, Y. Huang, H. Wang, and X. Luo, "Abusing the ethereum smart contract verification services for fun and profit," *Network and Distributed System Security (NDSS) Symposium*, 2023.

[24] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols." 2023.

[25] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[26] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.

[27] Solidity. Solidity: A statically-typed curly-braces programming language designed for developing smart contracts that run on ethereum. https://soliditylang.org/.

[28] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame, and K. Pohl, "Fuzz on the beach: Fuzzing solana smart contracts," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1197–1211.

[29] S. Cui, G. Zhao, Y. Gao, T. Tavu, and J. Huang, "Vrust: Automated vulnerability detection for solana smart contracts," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 639–652.

[30] C. Liu, Z. Sang, L. Duan, J. Wang, W. Ni, and W. Wang, "Anomaly detection services for blockchain smart contracts with unknown vulnerabilities," *ACM Transactions on Software Engineering and Methodology*, 2025.

[31] C. Chen, J. Su, J. Chen, Y. Wang, T. Bi, J. Yu, Y. Wang, X. Lin, T. Chen, and Z. Zheng, "When chatgpt meets smart contract vulnerability detection: How far are we?" *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 4, pp. 1–30, 2025.

[32] Y. Wu, X. Xie, C. Peng, D. Liu, H. Wu, M. Fan, T. Liu, and H. Wang, "Advscanner: Generating adversarial smart contracts to exploit reentrancy vulnerabilities using llm and static analysis," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1019–1031.

[33] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "Scvhunter: Smart contract vulnerability detection based on heterogeneous graph attention network," in *Proceedings of the IEEE/ACM 46th international conference on software engineering (ICSE)*, 2024, pp. 1–13.

[34] Z. Zheng, N. Zhang, J. Su, Z. Zhong, M. Ye, and J. Chen, "Turn the rudder: A beacon of reentrancy detection for smart contracts on ethereum," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 295–306.

[35] C. Bräm, M. Eilers, P. Müller, R. Sierra, and A. J. Summers, "Rich specifications for ethereum smart contract verification," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[36] W. Ma, D. Wu, Y. Sun, T. Wang, S. Liu, J. Zhang, Y. Xue, and Y. Liu, "Combining fine-tuning and llm-based agents for intuitive smart contract auditing with justifications," *arXiv preprint arXiv:2403.16073*, 2024.

[37] S. Grossman, J. Toman, A. Bakst, S. Arora, M. Sagiv, and C. Nandi, "Practical verification of smart contracts using memory splitting," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 2402–2433, 2024.

[38] S. Wu, Z. Li, L. Yan, W. Chen, M. Jiang, C. Wang, X. Luo, and H. Zhou, "Are we there yet? unraveling the state-of-the-art smart contract fuzzers," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[39] T. Wong, C. Zhang, Y. Ni, M. Luo, H. Chen, Y. Yu, W. Li, X. Luo, and H. Wang, "Confuzz: Towards large scale fuzz testing of smart contracts in ethereum," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1691–1700.

[40] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[41] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 557–560.

[42] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.

[43] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, "Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[44] Y. Liu, W. Meng, and Y. Zhang, "Detecting smart contract state-inconsistency bugs via flow divergence and multiplex symbolic execution," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 22–43, 2025.

[45] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 531–548.

[46] J. Su, H.-N. Dai, L. Zhao, Z. Zheng, and X. Luo, "Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[47] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 2, pp. 515–529, 2022.

[48] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, "Crossfuzz: Cross-contract fuzzing for smart contract vulnerability detection," *Science of Computer Programming*, vol. 234, p. 103076, 2024.

[49] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[50] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 227–239.

[51] M. Ye, Y. Nan, H.-N. Dai, S. Yang, X. Luo, and Z. Zheng, "Funfuzz: A function-oriented fuzzer for smart contract vulnerability detection with high effectiveness and efficiency," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–20, 2024.

[52] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[53] Z. Liu, P. Qian, X. Wang, L. Zhu, Q. He, and S. Ji, "Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Z.-H. Zhou, Ed. International Joint Conferences on Artificial Intelligence Organization, 8 2021, pp. 2751–2759, main Track. [Online]. Available: https://doi.org/10.24963/ijcai.2021/379

[54] S. Hu, T. Huang, F. Ilhan, S. F. Tekin, and L. Liu, "Large language model-powered smart contract vulnerability detection: New perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2023, pp. 297–306.

[55] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–11.

[56] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[57] Etherscan. Etherscan: The ethereum blockchain explorer. https://etherscan.io/.

[58] github. Github: A proprietary developer platform. https://github.com/.

[59] smartbugs, "Sb curated: A curated dataset of vulnerable solidity smart contracts," 2021, https://github.com/smartbugs/smartbugs-curated.

[60] ——, "Smart contract weakness classification registry," 2020, https://github.com/SmartContractSecurity/SWC-registry.

[61] DAppSCAN, "Dappscan: Building large-scale datasets for smart contract weaknesses in dapp projects." 2023, https://github.com/InPlusLab/DAppSCAN.

[62] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proceedings of the 2023 conference on empirical methods in natural language processing*, 2023, pp. 1069–1088.

[63] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. LIU, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *International Conference on Learning Representations*.

[64] OpenAI, "Gpt-4.1," 2025.

[65] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[66] J. Bai, S. Bai, Y. Chu, Z. Cui, K. Dang, X. Deng, Y. Fan, W. Ge, Y. Han, F. Huang *et al.*, "Qwen technical report," *arXiv preprint arXiv:2309.16609*, 2023.