CLEP: A Novel Contrastive Learning Method for Evolutionary Reentrancy Vulnerability Detection

Jie Chen[1], Liangmin Wang[1]*, Huijuan Zhu[2], Victor S. Sheng[3]

[1]School of Cyber Science and Engineering, Southeast University
[2]School of Computer Science and Communication Engineering, Jiangsu University
[3]Department of Computer Science, Texas Tech University
{jiechensimon, liangmin}@seu.edu.cn, huijuanzhu@ujs.edu.cn, victor.sheng@ttu.edu

Abstract

Reentrancy vulnerabilities in smart contracts have been exploited to steal enormous amounts of money, thus detecting reentrancy vulnerabilities is a hotspot issue in security research. However, a new attack is emerging in which attackers continuously release new reentrancy patterns to exploit fresh vulnerabilities and obfuscate existing ones. Existing detection methods neglect the time-series evolution of vulnerabilities across different smart contract versions, leading to a gradual decline in their effectiveness over time. We investigate the time-series correlations among vulnerabilities in various versions and refer to these as Evolutionary Reentrancy Vulnerabilities (ERVs). We summarize that ERVs detection faces two key challenges: (i) capturing the evolving pattern of ERVs along a complete evolutionary chain and (ii) detecting fresh reentrancy vulnerabilities in new versions. To address these challenges, we propose `CLEP`, a novel Contrastive Learning with Evolving Pairs detection method. It can effectively capture the evolving patterns by discerning similarities and differences across versions. Specifically, we first modified the sample distribution by incorporating version declarations as time-series evolution information. Then, leveraging the hierarchical similarity, we design an evolving pairs scheme to form negative and positive contract pairs across versions. Finally, we build a complete evolutionary chain by proposing a version-aware contrastive sampler. Our experimental results show that `CLEP` not only outperforms state-of-the-art baselines in version-specific scenarios but also shows promising performance in cross-version evolution scenarios.

Introduction

Smart contracts are deployed in blockchain systems to facilitate the implementation of nearly arbitrary business logic in areas involving high-value assets (Kalodner et al. 2018). Attackers can illicitly steal cryptocurrencies or tokens by exploiting underlying vulnerabilities in smart contracts. Reentrancy, one of the most notorious smart contracts vulnerabilities (SCVs), has led to millions of dollars in financial loss. For example, the most severe breach so far is the "TheDAO" incident, wherein hackers stole over $50 million worth of Ether (Rodler et al. 2019). Smart contracts suffer from the inherent risk of being prone to errors, owing to they are not patchable like traditional software, making it exceptionally challenging to update or modify smart contracts once they are deployed (Jiao et al. 2020).

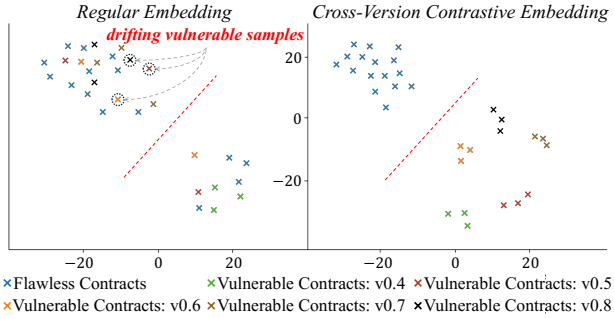

Figure 1: Regular Embedding vs. Cross-Version Contrastive Embedding. The embeddings are visualized using T-SNE, showing that new vulnerable contracts tend to become drifting samples and are misclassified as flawless. We aggregate the sub-versions under their major versions, such as grouping version 0.6.3 under 0.6.

The detection of SCVs has attracted significant interest from both industry and academia. Existing efforts can be roughly categorized into two groups: static analysis and dynamic analysis. These two types of analysis methods mainly involve software testing techniques and neural networks, with neural networks primarily present in static analysis. A critical branch of static analysis involves the semantic and syntactic analysis of smart contract source code. In recent years, researchers have increasingly delved into various deep learning-based methods (Zhuang et al. 2021; Wu et al. 2021; Zhang et al. 2022; Sendner et al. 2023) to detect SCVs, driven by the neural network's capacity to learn hidden representations from extensive data.

However, due to the evolution of SCVs when programming language (Solidity) upgrades, existing methods have become out of date and ineffective. Our study finds that, a well-trained SCV detector exhibits a decrease (from 0.92 to 0.65) in precision when tested on $<$0.5 smart contracts compared to those with 0.7 versions. Specifically, the Ethereum network operates multiple versions of smart contracts simultaneously. Moreover, reentrancy vulnerabilities have evolved from the traditional same-function pat-

*Corresponding author.

tern into various increasingly sophisticated variants (Rodler et al. 2019). It initially exploited by invoking an external contract and updating the state within the same function. It subsequently branched into several new patterns, i.e., re-entering through different functions, hiding within DELEGATECALL or CALLCODE instructions, and creating a new contract that make external calls in its constructor to an attacker-controlled contract. To counter these emerging threats, a considerable number of breaking changes and several defense techniques (i.e., *checks-effects-interactions* pattern and *ReentrantLock*) have been introduced with Solidity language updates. Hence, the semantic and syntactic features of smart contracts have continuously altered over time. Fig. 1 reveals these breaking changes cause the new vulnerable contract samples tend to drift away from the distribution of prior vulnerable contract samples in the embedding space. In other words, new versions introduce many breaking changes involving semantic and syntactic changes compared to their predecessors (depicted in Fig. 2). Consequently, newly vulnerable smart contracts are more likely to become drifting samples and be misclassified as flawless. Existing deep learning-based approaches treat each input contract as a separate entity, causing them to fall short in capturing the evolving vulnerability patterns (shown in Fig. 3). Besides, existing sampling methods fail to cover samples from all available versions in a training batch, leading to a broken evolutionary chain for multiple versions of contracts.

To overcome these challenges, we propose CLEP, a Contrastive Learning detection method based on Evolving Pairs for effectively hunting ERVs. It can capture the evolving vulnerability patterns along the complete evolutionary chain by learning the cross-version commonalities among vulnerable contracts and cross-version differences between vulnerable and flawless contracts. Our idea has three aspects: 1) Vulnerable contracts within the same version are closely aligned in the embedding space. 2) Vulnerable contracts across different versions remain hierarchically distinguishable, forming distinct sub-clusters. 3) Vulnerable and flawless contracts are separated as far apart as possible. Specifically, CLEP initially refines a pre-trained encoder on extensive historical smart contracts with cross-version contrastive loss to form a specialized model. Then we further fine tune this specialized model with a small number of contracts from the new versions, enabling CLEP to adapt to new contracts in the future.

The main contributions can be summarized as follows:

- We uncover the time-series correlations between vulnerabilities across various versions, which we refer to as evolutionary reentrancy vulnerabilities (ERVs). To detect ERVs, we propose CLEP, a Contrastive Learning with Evolving Pairs detection method to capture the evolving vulnerability patterns.
- In CLEP, we design a hierarchical similarity-based evolving pair scheme to capture the evolving vulnerability patterns. We also propose a version-aware contrastive sampler to construct a complete evolutionary chain for integrating the time-series correlations among versions.
- Experimental results prove the effectiveness of CLEP on version-specific and cross-version evolution scenarios.

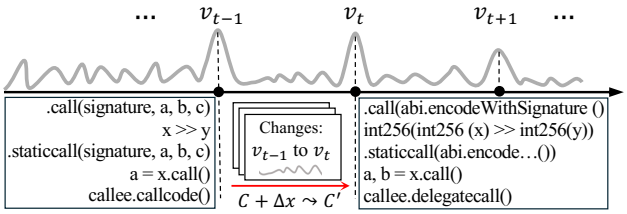

Figure 2: Breaking changes in the evolutionary chain when the Solidity language upgrades. We take the main breaking changes introduced in Solidity version 0.5.0 as an example.

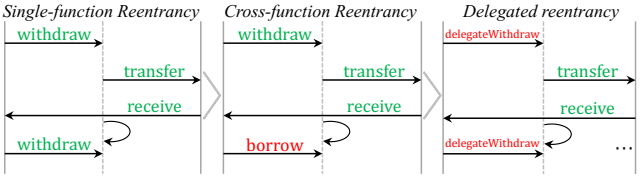

Figure 3: The evolving patterns: there are commonalities (green) and differences (red) among various variants.

Related Works

The detection of SCVs can be roughly divided into dynamic analysis and static analysis, depending on whether the programs need to be executed. Dynamic analysis typically involves simulating program execution and observing its dynamic behavior to determine the existence of vulnerabilities. For example, ReGuard (Liu et al. 2018) is a fuzzing-based analyzer designed to systematically detect reentrancy vulnerabilities in smart contracts by generating a series of random but strategically varied transactions. Sereum (Rodler et al. 2019) is a security tool that employs real-time monitoring and validation techniques to protect deployed contracts from reentrancy. Static analysis (Tsankov et al. 2018; Feist, Grieco, and Groce 2019; Zhuang et al. 2021; Wu et al. 2021; Zhang et al. 2022; Sendner et al. 2023) focuses on the semantic and syntactic levels of programs. For example, Oyente (Luu et al. 2016) uses symbolic execution to detect vulnerabilities in smart contracts by exploring feasible execution paths. Securify (Tsankov et al. 2018) is a lightweight smart contract verifier that detects vulnerabilities by analyzing dependency graphs for precise semantic information. Slither (Feist, Grieco, and Groce 2019) leverages Solidity's AST to uncover vulnerabilities by analyzing control and inheritance graphs.

Advancements in deep learning have spurred a rise in research utilizing neural networks for SCVs detection. For example, DR-GCN (Zhuang et al. 2021) employs graph neural network to detect SCVs by leveraging a contract graph to capture both syntactic and semantic information within smart contract source code. Peculiar (Wu et al. 2021) adopts a pre-training-based detection approach that extracts crucial data flow graphs from source codes. ReVulDL (Zhang et al. 2022) introduces a graph-based pre-training model to detect and locate reentrancy vulnerabilities by learning complex relationships in propagation chains within smart contracts. ESCORT (Sendner et al. 2023) is a transfer learning-based

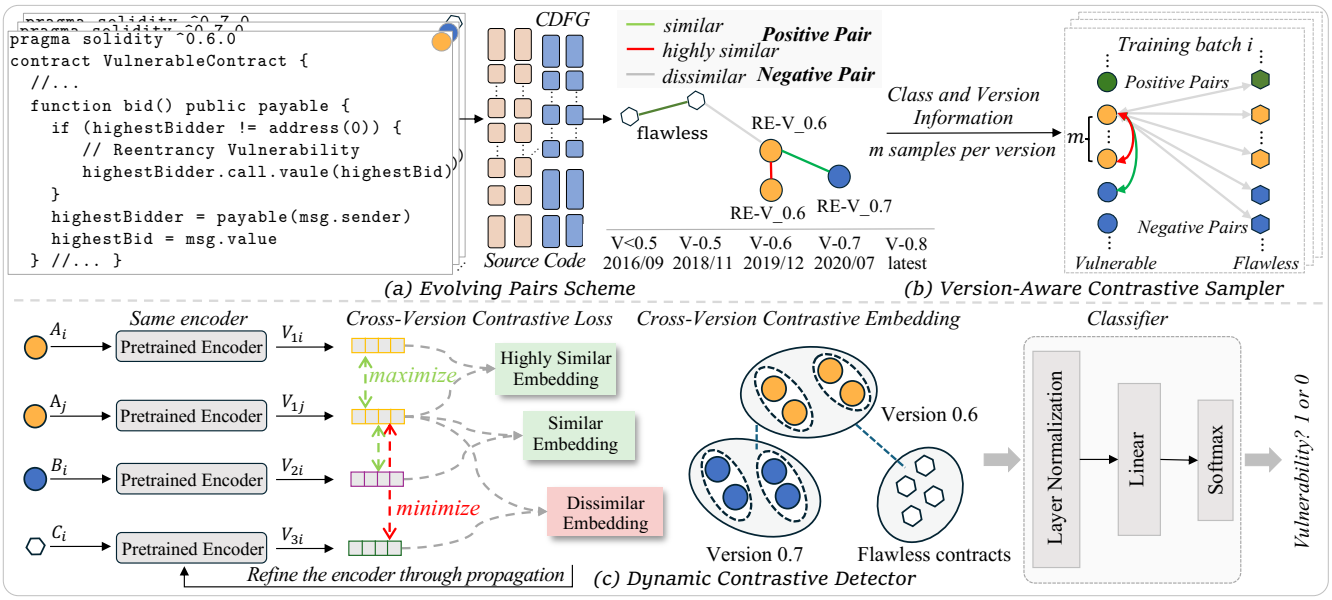

Figure 4: The architecture of CLEP. Sub-figure (a) shows the process of constructing the evolving pairs. Sub-figure (b) depicts the version-aware contrastive sampler, which ensures that each batch contains samples from various versions. Sub-figure (c) is the dynamic contrastive detector. The construction of crucial data flow graph (CDFG) is followed by Peculiar (Wu et al. 2021).

vulnerability detection method to detect new vulnerability types with limited data. However, these deep learning-based approaches consider each input contract as a separate entity, falling short in capturing the evolving vulnerability patterns.

Methodology

Fig. 4 shows the architecture of CLEP, including evolving pairs scheme, version-aware contrastive sampler and dynamic contrastive detector.

Evolving Pairs Scheme

Hierarchical Similarity. Smart contracts from multi-version exhibit hierarchical similarities from the semantic and syntactic perspectives. We assume hierarchical similarities such that: 1) Vulnerable contracts within the same version are highly similar. 2) Vulnerable contracts across different versions are similar. 3) Vulnerable and flawless contracts are distinctly dissimilar. For example, two reentrant smart contracts with the same version share a high degree of similarity. The underlying reason is twofold: i) they adhere to the same programming principles, and ii) contracts with reentrancy typically involve fund transfers and the invocation of external functions in terms of program business logic. Besides, as depicted in Fig. 2, the introduced breaking changes, including both semantic and syntactic changes, reduce the level of similarity. Thus, two reentrant contracts from different versions are similar, while reentrant and non-reentrant contracts are distinctly different.

Forming Evolving Pairs. Contrastive learning can enhance the similarity among positive samples (bringing them closer) and the dissimilarity among negative samples (pushing them apart). Inspired by studies (Khosla et al. 2020; Chen, Ding, and Wagner 2023; Chen et al. 2024), we discern positive and negative samples based on labels. Apart from the standard class label, we further introduce a version label as an extra label to jointly form the evolving pairs.

Let x_i be an arbitrary sample from a batch containing $2N$ samples, with its ground truth binary class label denoted as $y \in 0, 1$ and its version label denoted as v. We define the following three pair type sets based on the hierarchical similarity to track samples across various versions.

- Both samples are vulnerable with the same version.

$$P\left(i, y_i, v_i\right) \equiv \{j \mid y_i = y_j = 1 \wedge j \neq i \wedge v_i = v_j\} \quad (1)$$

- Both samples are vulnerable, but with different versions; or, both samples are flawless.

$$P_z\left(i, y_i, v_i\right) \equiv \{j \mid j \neq i \wedge y_i = y_j = 1 \wedge v_i \neq v_j \vee y_i = y_j = 0\} \quad (2)$$

- No matter which version, one is flawless and the other is vulnerable.

$$N\left(i, y_i\right) \equiv \{j \mid y_j \neq y_i\} \quad (3)$$

where $P\left(i, y_i, v_i\right)$ captures pairs that are considered highly similar, $P_z\left(i, y_i, v_i\right)$ contains pairs that are similar, and $N\left(i, y_i\right)$ includes pairs that are dissimilar. Based on these three sets, we derive following three properties that sample pairs are required to satisfy.

- If x_1, x_2 are both vulnerable with the same versions, then their embeddings should be highly similar: specifically, $\|enc\left(x_1\right) - enc\left(x_2\right)\|_2$ should be as small as possible.

Algorithm 1: Version-Aware Contrastive Sampler

Input: Smart contracts X, class labels L, version labels V, samples per version m, batch size bs, buffer size b
Output: An sample iterator ξ

1: $ls \leftarrow b, losp \leftarrow m \times |V_{unique}| \times |L_{unique}|$
2: **if** bs is None **then**
3: $\quad ls \leftarrow ls - k, k \leftarrow ls \mod losp$
4: $\quad \mathcal{N} \leftarrow \lfloor ls/losp \rfloor$
5: **else**
6: $\quad ls \leftarrow ls - k, k \leftarrow ls \mod bs$
7: $\quad \mathcal{N} \leftarrow \lfloor ls/bs \rfloor$
8: **end if**
9: Initialize a nested dict $\mathcal{I}[l][v]$
10: **for** each contract x_i in X **do**
11: $\quad$ **if** l_i not in $\mathcal{I}$ **then**
12: $\quad\quad \mathcal{I}[l_i] \leftarrow \{\},\ l_i \in L$
13: $\quad$ **end if**
14: $\quad$ **if** v_i not in $\mathcal{I}[l_i]$ **then**
15: $\quad\quad \mathcal{I}[l_i][v_i] \leftarrow \{\},\ v_i \in V$
16: $\quad$ **end if**
17: $\quad$ Append idx_i to the list $\mathcal{I}[l_i][v_i]$
18: **end for**
19: **for** $a = 1, a \leq \mathcal{N}, a++$ **do**
20: $\quad$ **for** each l, v in $\mathcal{I}$ **do**
21: $\quad\quad sampleSize \leftarrow |\mathcal{I}[l][v]|$
22: $\quad\quad replace \leftarrow sampleSize \geq m$
23: $\quad\quad \mathcal{P} \leftarrow$ *RandomSample*$(\mathcal{I}[l][v], m, replace)$
24: $\quad$ **end for**
25: **end for**
26: **return** an iterator ξ for evolving pairs $\mathcal{P}[:ls]$

- If x_1, x_2 are both vulnerable with different versions; or, both samples are flawless, then their embeddings should be similar: specifically $\|enc(x_1) - enc(x_2)\|_2 \leq \delta$. Notably, despite varying semantics, flawless contracts are typically aligned closely in embedding space to establish robust decision boundaries.
- If one of x_1, x_2 is vulnerable and the other is flawless, then their embedding should be highly dissimilar: specifically, $\|enc(x_1) - enc(x_2)\|_2 \geq 2\delta$.

As shown in the Fig. 4, sample A_i and A_j are both containing reentrancy vulnerability from the same version 0.6 while sample B_i has reentrancy vulnerability from 0.7 and sample C_i is flawless from 0.5. These four vulnerable contract samples (A_i, A_j B_i, and C_j) are passed through an encoder to get their representations (i.e., v_{1i}, v_{1j}, v_{2i} and v_{3i}). Then the goal of our contrastive learning is to hierarchically maximize the similarity between positive samples (i.e., (v_{1i}, v_{1j}), (v_{1j}, v_{2i}) and minimize the similarity between negative samples (such as (v_{1j}, v_{3i})).

Version-Aware Contrastive Sampler

The goal of proposed sampler is to build a complete evolutionary chain by covering samples from all available versions, facilitating our dynamic contrastive detector to capture the evolving patterns by learning version-specific and cross-version features. We empirically observed existing sampling methods (e.g., Triplet-Sampler (Li, Wu, and Zheng 2021), MPerClassSampler (Shah, Reddy, and Bhattacharyya 2023)) fail to cover contracts from all available versions in a training batch, typically limited to only a partial set of versions of contracts, leading to a broken evolving chain for multiple versions of contracts. Therefore, we propose MPVSampler, a version-aware contrastive sampling method by integrating version labels alongside class labels.

For each smart contract x_i, we leverage three attributes: index (idx), version label (v), and class label (l). We define m as the number of samples per version within each class. MPVSampler, as detailed in Algorithm 1, includes a preparation phase (lines 1-8) and a sampling phase (lines 9-25). It returns an iterator ξ with sampled pairs for the *dataloader*. In the preparation phase, MPVSampler initializes the pair list size (ls) with a fixed buffer size and computes the *length of single pass* (lines 1-2). If the batch size (bs) is not specified, MPVSampler calculates an optimal buffer length $\mathcal{N}$ based on *losp* to ensure sample diversity in each batch (lines 3-5). Otherwise, it dynamically adjusts the buffer size based on the bs (lines 6-8). In the sampling phase, MPVSampler checks if the class and version label of each sample x_i already exists in $\mathcal{I}$. For unseen class or version, it creates new lists in $\mathcal{I}$ based on l_i and v_i (lines 9-18). MPVSampler then calculates the size (*sampleSize*) of each class-version list (line 21). The *replace* flag determines whether sampling should be repetitive (if *sampleSize* $\leq m$) or non-repetitive to meet the required number of samples per version (lines 22-23). Thus, each training batch includes two sets of cross-version samples: one of vulnerable samples and one of flawless samples. An anchor sample is selected from one set, with the remaining samples in that set forming positive pairs, while samples from the other set form negative pairs.

Dynamic Contrastive Detector

We train a evolving contrastive detector f to detect vulnerabilities in smart contracts, which includes two subnetworks. The first, a contrastive pre-trained encoder enc, refines GraphCodeBERT (Guo et al. 2021) with contrastive loss to generate embeddings $z = enc(x_i)$. The second subnetwork is the classifier g, which takes the contrastive embedding z and predicts a class label $g(z)$.

We define $f(x_i)$ as the prediction score for class $y = 1$ from the softmax layer, and $1 - f(x_i)$ as the score for flawless contracts. The predicted class label $\hat{y}$ is assigned $\hat{y} = 1$ when $f(x_i) \geq 0.5$, and $\hat{y} = 0$ otherwise. We formulate a joint loss function that encourages $f(x_i)$ to correctly predict the label y, while ensuring the encoder meets the aforementioned three specified properties. This should guide the flawless samples clustering together, while vulnerable samples from each version also cluster together. The latter cluster should comprise multiple sub-clusters, each corresponding to a specific version. This will guide the encoder to discern version-specific and cross-version features in evolving vulnerability patterns. Consequently, the classifier naturally becomes robust against drifting samples. To achieve this, the joint loss function is a weighted sum of cross-version contrastive loss and classification loss. We then refine our

model, including the encoder and classifier, end-to-end using this loss. The joint loss function is defined as following equation:

$$\mathcal{L} = \mathcal{L}_{ce} + \lambda \mathcal{L}_{vc} \tag{4}$$

where $\mathcal{L}_{ce}$ is the classification loss and $\mathcal{L}_{vc}$ is the cross-version contrastive loss. $\lambda = \frac{\alpha \mathcal{L}_{ce}}{\mathcal{L}_{ce}+\mathcal{L}_{vc}}$, which is a dynamic weight and α is a hyperparameter, is strategically employed to dynamically adjust the balance between these two losses during training. The classification loss employs the cross-entropy loss:

$$\begin{aligned} \mathcal{L}_{\text{ce}} &= \sum_i \mathcal{L}_{\text{ce}}\left(x_i, y_i\right) \\ \mathcal{L}_{\text{ce}}\left(x_i, y_i\right) &= -\, y_i \log f(x_i) \\ &\quad -\left(1-y_i\right) \log(1-f(x_i)) \end{aligned} \tag{5}$$

where i is the index of samples in the batch. The cross-version contrastive loss is calculated on pairs of samples in a batch of size $2N$ ($N = m * n$), where m is the sample number per version, n is the sample number per class. The first N samples $\{x_k, v_k, y_k\}_{k=1,\ldots,N}$ are sampled from vulnerable contracts across different versions using our sampler, followed by extra N samples $\{x_{k+N}, v_{k+N}, y_{k+N}\}_{k=1\ldots N}$ from flawless contracts. These $2N$ samples together form positive and negative pairs in the evolutionary chain.

Our cross-version contrastive loss leverages a class and version mutual exclusive strategy regularized by a contrastive margin. The goal of our cross-version contrastive loss is to refine the model's feature space, aiming to closely align the features of vulnerable contracts while ensuring that samples from different versions remain distinguishable and that contracts with and without vulnerabilities are maximally separated. Let i denote the index of an arbitrary sample within a batch of $2N$ samples. Let d_{ij} represent the euclidean distance between two arbitrary samples i and j in the embedding space, given by $d_{ij} = \|enc\left(x_1\right) - enc\left(x_2\right)\|_2$. Denoting a fixed margin as δ and cross-version contrastive loss is defined as:

$$\begin{aligned} \mathcal{L}_{vc} &= \sum_i \mathcal{L}_{vc}(i) \\ \mathcal{L}_{vc}(i) &= \frac{1}{\left|P\left(i, y_i, v_i\right)\right|} \sum_{j \in P(i, y_i, v_i)} d_{ij} \\ &+ \frac{1}{\left|P_z\left(i, y_i, v_i\right)\right|} \sum_{j \in P_z(i, y_i v_i)} \max\left(0, d_{ij}-\delta\right) \\ &+ \frac{1}{\left|N\left(i, y_i\right)\right|} \sum_{j \in N(i, y_i)} \max\left(0, 2\delta - d_{ij}\right) \end{aligned} \tag{6}$$

The cross-version contrastive loss comprises three terms. The first term emphasizes samples which are both vulnerable with same version should be highly similar, and we impose a penalty on any non-zero distance d_{ij} between these samples. The second term asks positive pairs from $P_z\left(i, y_i, v_i\right)$ (both samples are vulnerable with different versions or both samples are flawless) to be relatively close together. We penalize the distance between these pairs only if it exceeds a threshold of δ, which is set to 10. Specifically, this means that if the distance between sample pair is less than the δ, as we expect, there is no penalty. However, if the distance between the sample pairs is larger than δ, a penalty is applied, leading to an increase in the local loss. This term guides us to learn cross-version features that are common to vulnerable smart contracts with different versions or all flawless smart contracts. The last term aims to separate vulnerable samples from flawless ones by at least 2δ, without penalizing for greater distances.

Fine-tuning the CLEP. We implement three fine-tuning strategies: version-specific fine-tuning (VSFT), cross-version fine-tuning (CVFT) and sequential fine-tuning (SFT). VSFT produces independent models for each version by fine-tuning a pre-trained model on each version's training data. CVFT fine-tunes the model on a comprehensive training data covering all versions, allowing it to analyze arbitrary version of smart contracts. In contrast, SFT refines a pre-trained model with historical training data (<0.5 to 0.7) and then fine-tunes it using few latest samples (0.8), enabling `CLEP` to adapt to new contracts in the future.

Experiments

Experiment Setup

Datasets. 1) `SmartBug Wild` (Durieux et al. 2020) is a widely recognized smart contract dataset. It contains roughly $47,398$ *.sol* files with a total of $203,716$ contracts, including 1,197 with reentrancy vulnerabilities. Specifically, 89.65% $(182,636/203,716)$ of these contracts are from versions <0.5. This dataset was manually labeled by Wu et al. (2021). Due to the limited number of contracts in higher versions, we also utilize the following dataset. 2) `ReTranStudy` (Zheng et al. 2023) is a recently cross-version smart contract dataset, which contains $139,424$ contracts. There are $23,461$ contracts for version 0.5 (3805 with reentrancy), $24,761$ for 0.6 ($2,040$ with reentrancy), $8,292$ for 0.7 (275 with reentrancy), and $14,714$ for 0.8 (125 with reentrancy). Following Peculiar (Wu et al. 2021) and ReVulDL (Zhang et al. 2022), we split the datasets into training (20%), validation (10%), and testing sets (70%).

Hyper-Parameter Setting. We conduct a grid search to find out the best settings of hyper-parameters. The learning rate l tuned amongst $\{1e-6, 1e-5, 2e-5, 5e-5\}$, the epoch ep is searched in $\{4, 6, 8, 10, 15\}$, code length c in $\{256, 512\}$, the batch size bs in $\{24, 32, 64, 128\}$ and data flow length d in $\{64, 128\}$. The performance of all neural network models, unless specifically stated, are reported on the following settings: for VSFT, we set l to $1e-5$, ep to 5, bs to 24, c to 256, and d to 64; for CVFT, l was set to $1e-6$, ep to 10, bs to 32, c to 256, and d to 64; and for SFT, l, ep, and bs were set to $1e-5$, 10, and 32 respectively in the refinement phase, and to $2e-5$, 4, and 48 in the fine-tuning phase. All experiments were conducted with two Intel Xeon 6148 CPUs, two 3090Ti GPUs, and 512GB RAM.

Evaluation metrics, and Baselines. We use *recall*, *precision*, and *F1-score* as the evaluation metrics. On version-specific scenario, we compare `CLEP` with 11 state-of-the-art baseline methods (Oyente (Luu et al. 2016), Mythril

Approaches	SmartBug Wild (<0.5)			RetranStudy (0.5)			RetranStudy (0.6)			RetranStudy (0.7)			RetranStudy (0.8)		
	R (%)	P (%)	F1 (%)	R (%)	P (%)	F1 (%)	R (%)	P (%)	F1 (%)	R (%)	P (%)	F1 (%)	R (%)	P (%)	F1 (%)
Oyente	54.11	65.63	56.44	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Mythril	51.69	50.24	49.74	42.88	37.17	39.82	35.92	30.57	33.3	15.68	11.84	14.70	8.03	5.11	6.25
Securify	54.81	52.63	53.36	17.66	15.73	16.64	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Slither	65.41	51.97	52.60	56.21	50.07	52.97	40.33	31.71	35.50	20.10	14.93	17.13	12.25	7.68	9.44
Smartian	57.60	55.37	56.46	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Sailfish	62.27	60.06	61.15	38.08	37.51	37.79	14.10	12.31	13.14	N/A	N/A	N/A	N/A	N/A	N/A
Vanilla-RNN	59.12	49.32	51.46	53.78	43.82	45.26	45.28	38.59	41.67	38.01	30.49	33.84	34.32	27.91	30.78
LSTM	68.27	52.35	59.26	57.90	50.21	53.78	52.25	46.30	49.09	47.70	42.33	44.85	42.65	39.09	40.79
GRU	71.27	53.20	61.92	61.08	54.91	57.83	56.70	51.05	53.73	50.27	47.74	48.97	44.00	41.37	42.64
Peculiar (VSFT)	92.40	91.80	92.10	83.32	81.42	82.36	79.19	77.87	77.53	68.72	64.50	66.54	65.56	58.72	61.95
ReVulDL (VSFT)	93.44	92.58	93.01	85.01	83.70	84.35	80.98	79.82	80.40	69.54	66.07	67.76	66.09	59.26	62.49
CLEP (VSFT)	**95.29**↑	**95.12**↑	**95.21**↑	**90.07**↑	**89.68**↑	**89.87**↑	**86.34**↑	**84.93**↑	**85.63**↑	**78.25**↑	**76.37**↑	**77.30**↑	**76.08**↑	**73.96**↑	**75.00**↑
													ALL (<0.5 to 0.8)		
Peculiar (CVFT)	*Cross-version fine-tuning:*												70.25	71.27	70.76
ReVulDL (CVFT)	*fine tuning on all cross-version training data, and test on corresponding cross-version test data.*												71.06	73.21	72.12
CLEP (CVFT)													**84.73**↑	**85.20** ↑	**84.96**↑
													Unseen New Contracts		
Peculiar (SFT)	*Sequential fine-tuning: refining on historic contracts from <0.5-0.7 and fine-tuning with a*												55.46	62.91	58.95
ReVulDL (SFT)	*small number of latest contracts from 0.8, then conduct few-shot prediction on 0.8 unseen test data.*												57.87	63.03	60.34
CLEP (SFT)													**74.45**↑	**79.30**↑	**76.80**↑

Table 1: Reentrancy detection results on different scenarios. The top part shows the results on version-specific vulnerability detection with version-specific fine-tuning (VSFT), while the bottom part shows the results on cross-version vulnerability detection with cross-version fine-tuning (CVFT) and sequential fine-tuning (SFT), respectively.

(Mueller 2017), Securify (Tsankov et al. 2018), Slither (Feist, Grieco, and Groce 2019), Smartian (Choi et al. 2021), Sailfish (Bose et al. 2022), Vanilla-RNN (Goller and Kuchler 1996), LSTM (Sak, Senior, and Beaufays 2014), GRU (Chung et al. 2014), Peculiar (Wu et al. 2021), ReVulDL (Zhang et al. 2022)). On cross-version scenarios, we compare CLEP with Peculiar, and ReVulDL on all existing versions. The selection rationale lies in CLEP, Peculiar, and ReVulDL can be fine-tuned in cross-version scenarios.

Performance on Version-Specific Scenario

We compare our proposed CLEP with 11 state-of-the-art baselines in the version-specific scenario. Specifically, CLEP implements version-specific fine-tuning for each version, employing MPerClassSampler as the sampler, and SupCon loss as the loss function. The experiment results in Tab. 1 (top part) show that CLEP outperforms all state-of-the-art baselines across all versions. It also can be observed that most non-deep learning methods (e.g., Oyente, Mythril, Securify and Smartian) have not yielded satisfactory results in detecting reentrancy vulnerabilities. Compared to these baselines, CLEP shows progressively larger performance gains. For versions <0.5, CLEP achieves F1 score improvements of 3.11% over Peculiar and 2.20% over ReVulDL. On version 0.8, these gains increase to $13.05\%$and 12.51%, respectively. The performance of all approaches exhibited a noticeable decline with the continuous upgrades of the smart contracts, particularly for versions 0.7 and 0.8. Specifically, Peculiar's F1 score fell by 25.56% and 30.15% in versions 0.7 and 0.8, respectively, when benchmarked against version <0.5. ReVulDL experienced a similar trend, with its F1 score diminishing by 25.25% and 30.52%. Notably, CLEP exhibit the least performance drop, with a decrease of 17.91% for version 0.7 and 20.21% for version 0.8.

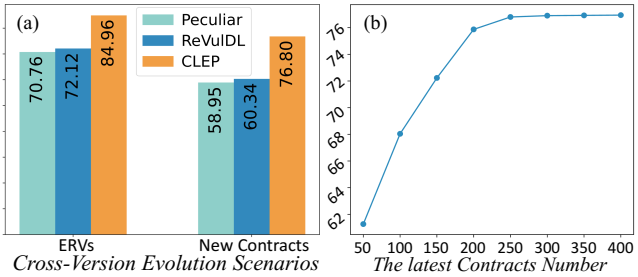

Figure 5: The visualization of CLEP's performance (F1 score) on cross-version evolution scenarios.

Performance on Cross-Version Scenarios

We utilize cross-version and sequential fine-tuning for cross-version smart contracts in cross-version scenarios. Besides, CLEP employs MPVSampler as the sampler and cross-version contrastive loss as the loss function. Peculiar and ReVulDL are fine-tuned in the same manner as CLEP.

Evolved Reentrancy Detection. In this scenario, we fine-tune CLEP with the CVFT strategy, allowing it to analyze arbitrary version of smart contracts. Specifically, The training contracts from each version are aggregated into a cross-version fine-tuning dataset and we form it as the evolving pairs. we then train the encoder of the CLEP on them with cross-version contrastive loss and test on corresponding cross-version test data. The experiment results in Tab. 1 (bottom part) and Fig. 5(a) show that CLEP outperforms Peculiar and ReVulDL across all cross-version datasets, achieving F1 improvements of 14.20% and 12.84%, respectively.

Adaptability to Future Versions. We treat contracts from versions 0.5 to 0.7 as historical data and contracts from 0.8

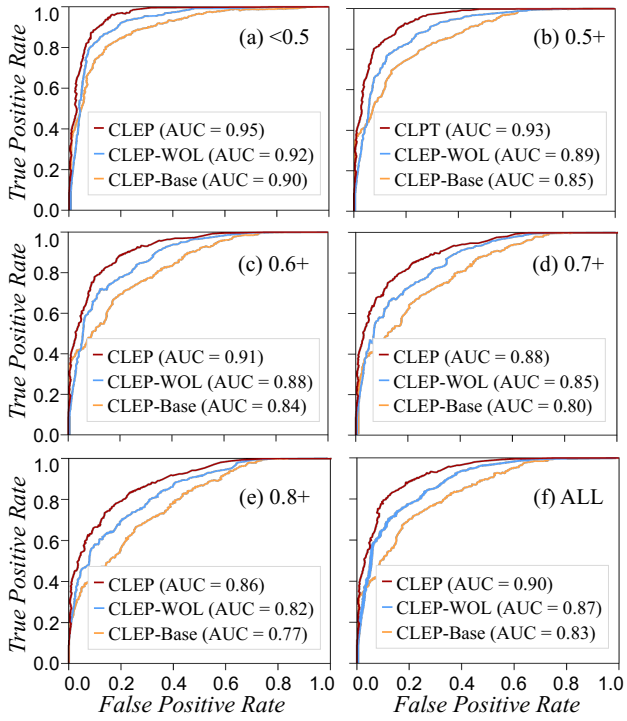

Figure 6: An ablation study of ROC curve comparisons on each version and across all versions. A higher AUC (Area Under the Curve) vaule indicates better performance.

as future data, assessing CLEP's ability to adapt to new contracts in the future. We fine-tune CLEP with the SFT strategy. Specifically, we initially refining CLEP with historical smart contracts. Subsequently, we conducted fine-tuning using a small number of contracts from version 0.8. Then, we evaluate the performance of CLEP on unseen testing dataset from version 0.8. Tab. 1 and Fig. 5(a) shows that CLEP achieved an F1 score of 76.80%, with gains of 17.85% and 16.46% over Peculiar and ReVulDL, respectively. Fig. 5(b) further indicates that CLEP achieves promising adaptability when fine-tuned on only about 250 of the latest contracts. These results suggest that CLEP has the ability to adapt to future smart contracts without the need to train from scratch.

Ablation Study

In cross-version evolution scenarios, we conduct a ablation study to investigate how the the MPVSampler (MPV) and cross-version contrastive (CVC) loss contribute to the CLEP. As shown in Tab. 2, the default CLEP is configured using the MPV in conjunction with CVC loss. The MPerClassSampler (MPC) and supervised contrastive loss (SupCon) (Khosla et al. 2020) are respectively considered as candidate sampler and loss function. Building upon CLEP, CLEP_WOL maintained MPV but substituted CVC with SupCon, leading to a 2.45% reduction in F1 score. Additionally, CLEP_Base further replaced MPV with MPC, which breaks the evolutionary chain by sampling only partial versions in each batch. The experimental results also show that removing the MPV further degrades CLEP's performance,

Model	Sampler-Loss	R (%)	P (%)	F1 (%)
CLEP	MPV-CVC	**84.73**	**85.20**	**84.96**
CLEP_WOL	MPV-SupCon	82.03↓	83.01↓	82.51↓
CLEP_Base	MPC-SupCon	77.04↓	78.63↓	77.83↓
CLEP_CD	MPV-CVC	81.62↓	80.97↓	81.29↓

Table 2: An ablation study of CLEP on smart contracts from all versions. CLEP is our target model. Compared with CLEP, CLEP_WOL replaces CVC with SupCon. CLEP_Base futher replaces MPV with MPC relative to CLEP_WOL. MPC and CVC are not compatible for the lack of a complete evolutionary chain.

with a 4.68% drop compared to CLEP_WOL. In general, when both MPV and CVC are removed, CLEP_WOL shows a significant overall decrease of 7.13% in F1 score compared to the default CLEP. We also investigate the impact of distance metrics by replacing euclidean distance with cosine distance (CLEP_CD). The results show a 3.67% decrease in F1 score, suggesting that euclidean distance is more effective in our task. Fig. 6 further plots the Receiver Operating Characteristic (ROC) curves to analyze the impact of MPV and CVC. Figs. 6(a-e) show the performance of CLEP and its variants on corresponding test data, while Fig. 6(f) presents their performance on the overall test set. The ROC curves reveal that removing either MPV or CVC leads to a notable decline in performance, with a more substantial decrease when both are removed. For example, in Fig. 6(a), after removing CVC and MPV, the AUC of CLEP on contracts from <0.5 decreases by 0.03 and 0.02 respectively. These experimental results demonstrate their critical roles in improving the performance of CLEP.

Conclusion

In this paper, we find a new evolving pattern which reveals the time-series correlations between vulnerabilities across various versions. Our key insight is that, while vulnerability patterns in smart contracts undergo continuous changes over time, hierarchical similarities persist among variants in the evolutionary sequence. Inspired by this, to detect the evolutionary reentrancy vulnerabilities, we propose CLEP, a Contrastive Learning with Evolving Pairs detection method, to capture the evolving patterns along a complete evolutionary chain. Moreover, CLEP has the ability to adapt to the contracts in the future by fine-tuning with a small number of contracts from the new versions. At present, CLEP is built in a supervised fashion, focusing on reentrancy vulnerability due to its relatively rich annotated dataset collected across multiple versions. Extensive experiments have been conducted to demonstrate the effectiveness of CLEP.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (62372105, 62272204), the Leading-edge Technology Foundation of Jiangsu (BK20202001) and the Science and Technology Achievement Transformation Special Foundation of Jiangsu (SBA2022050016).

References

Bose, P.; Das, D.; Chen, Y.; Feng, Y.; Kruegel, C.; and Vigna, G. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*, 161–178.

Chen, Y.; Ding, Z.; and Wagner, D. 2023. Continuous Learning for Android Malware Detection. In *32nd USENIX Security Symposium (USENIX Security 23)*, 1127–1144.

Chen, Y.; Sun, Z.; Gong, Z.; and Hao, D. 2024. Improving Smart Contract Security with Contrastive Learning-based Vulnerability Detection. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 940–940.

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

Chung, J.; Gulcehre, C.; Cho, K.; and Bengio, Y. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Durieux, T.; Ferreira, J. F.; Abreu, R.; and Cruz, P. 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering (ICSE)*, 530–541.

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

Goller, C.; and Kuchler, A. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of international conference on neural networks (ICNN'96)*, volume 1, 347–352.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Shujie, L.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations (ICLR)*.

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

Kalodner, H.; Goldfeder, S.; Chen, X.; Weinberg, S. M.; and Felten, E. W. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, 1353–1370.

Khosla, P.; Teterwak, P.; Wang, C.; Sarna, A.; Tian, Y.; Isola, P.; Maschinot, A.; Liu, C.; and Krishnan, D. 2020. Supervised contrastive learning. *Advances in neural information processing systems*, 33: 18661–18673.

Li, H.; Wu, G.; and Zheng, W.-S. 2021. Combined depth space based architecture search for person re-identification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 6729–6738.

Liu, C.; Liu, H.; Cao, Z.; Chen, Z.; Chen, B.; and Roscoe, B. 2018. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 65–68.

Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; and Hobor, A. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 254–269.

Mueller, B. 2017. Mythril-Reversing and bug hunting framework for the Ethereum blockchain.

Rodler, M.; Li, W.; Karame, G. O.; and Davi, L. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *26th Annual Network and Distributed System Security Symposium (NDSS)*.

Sak, H.; Senior, A. W.; and Beaufays, F. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *15th Annual Conference of the International Speech Communication Association, Singapore*, 338–342.

Sendner, C.; Chen, H.; Fereidooni, H.; Petzi, L.; König, J.; Stang, J.; Dmitrienko, A.; Sadeghi, A.-R.; and Koushanfar, F. 2023. Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning. In *30th Annual Network and Distributed System Security Symposium (NDSS)*.

Shah, S.; Reddy, S.; and Bhattacharyya, P. 2023. Retrofitting Light-weight Language Models for Emotions using Supervised Contrastive Learning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 3640–3654.

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

Wu, H.; Zhang, Z.; Wang, S.; Lei, Y.; Lin, B.; Qin, Y.; Zhang, H.; and Mao, X. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, 378–389.

Zhang, Z.; Lei, Y.; Yan, M.; Yu, Y.; Chen, J.; Wang, S.; and Mao, X. 2022. Reentrancy vulnerability detection and localization: A deep learning based two-phase approach. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1–13.

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

Zhuang, Y.; Liu, Z.; Qian, P.; Liu, Q.; Wang, X.; and He, Q. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI)*, 3283–3290.