IntelliCon: Confidence-Based Approach for Fine-Grained Vulnerability Analysis in Smart Contracts

Yiming Shen[1], Kunhua Li[1], Lin Mao[2], Wenkai Li[1], and Xiaoqi Li[1](✉)

[1] School of Cyberspace Security, Hainan University, Haikou 570208, China
csxqli@gmail.com

[2] School of Computer Science and Technology, Hainan University, Haikou 570208, China

Abstract. Ethereum smart contracts are programs that execute transactions on a distributed ledger platform without intermediaries. However, they are prone to various types of vulnerabilities that can affect their security and functionality. In this paper, we present IntelliCon, a novel framework that leverages a pre-trained identifier-aware encoder-decoder CodeT5 model and confident learning to detect seven types of vulnerabilities in Ethereum smart contracts. Confident learning is a technique that improves dataset quality by identifying and correcting noisy labels, particularly in the presence of multiple annotators with varying levels of accuracy. We fine-tune CodeT5 on a dataset of 27,426 smart contracts annotated by multiple tools and pruned by confident learning to ensure that the model learns genuine vulnerability features rather than tool-specific features. Furthermore, we utilize abstract syntax tree (AST) analysis to extract code gadgets with sliding windows to locate the function that may contain code vulnerabilities. We evaluate the effectiveness of our framework in vulnerability detection with F1-score. Our results indicate that IntelliCon achieves high Micro-F1 (0.9591) and Macro-F1 (0.9293), outperforming existing methods. Moreover, our framework demonstrates its ability to handle imbalanced data, noisy labels, and complex code structures. IntelliCon contributes to improving the security and reliability of smart contracts, providing insights for future research on code generation tasks.

Keywords: Blockchain · Smart Contract · Confident Learning · Vulnerability Detection · Deep Learning

1 Introduction

Recently, blockchain [1] technology has experienced rapid development in various fields (e.g., finance, healthcare, supply chain, and IoT) [2]. However, the vulnerabilities of smart contracts have given significant concerns, resulting in financial losses, system failure, and damage to the platform's reputation. According to the

J. Chen et al. (Eds.): BlockSys 2023, CCIS 1896, pp. 45–59, 2024.
https://doi.org/10.1007/978-981-99-8101-4_4

Blockchain Security and AML Analysis Annual Report of SlowMist [3], 30.3% of the 303 blockchain security events recorded in 2022 were due to contract vulnerabilities. Notably, high-profile security incidents in the Ronin Network, and the Wormhole network resulted in losses exceeding $610 million and $300 million, respectively [4]. These incidents emphasize the urgent need for enhanced security measures to safeguard users' assets and mitigate potential security risks.

Smart contract vulnerabilities pose a significant risk to the security and integrity of blockchain systems, necessitating the development of effective techniques for their detection and mitigation [5]. Traditional methods for detecting smart contract vulnerability rely on static or dynamic analysis. Static analysis examines the source code or bytecode of smart contracts without executing them, while the dynamic analysis runs smart contracts with test inputs or fuzzing techniques. However, these methods have limitations such as slow processing, high false positives/negatives rates, low coverage, and scalability issues. To address the limitation of traditional methods for detecting smart contract vulnerabilities, machine learning-based methods [6] have been proposed. These methods leverage various algorithms or models to learn patterns or features from code snippets and classify them as vulnerable or non-vulnerable.

Moreover, existing large datasets (e.g., SmartBugs [7]) are often labeled by multiple traditional vulnerability detection tools. While models trained on these datasets can achieve high performance, they are limited by these traditional tools. In essence, as long as the neural network replicates expert-defined rule, it can achieve a high F1 close to 1. Therefore, these training results merely simulate or synthesize the functions of the original tools, rather than truly learning the underlying causes of vulnerabilities.

In this paper, we propose IntelliCon, a CodeT5-based detection framework for smart contracts that address the limitations of traditional methods. The source code of the smart contract is processed as an abstract syntax tree (AST) to collect code gadgets and then fed to the CodeT5 model. Our framework also utilizes confident learning technology during training to improve the labeling consistency of the dataset. To provide a measure of confidence in the accuracy of our model, we construct a set of prediction ranks around its predictions. By leveraging these prediction ranks as a decision criterion, potentially incorrect forecasts can be filtered out, leading to an improvement in the overall accuracy of the model. The experimental results demonstrate that our framework achieves a micro-F1 score of over 95%, and a macro-F1 score of over 92%, outperforming the existing popular pre-training frameworks (e.g., Bert and T5) in terms of detection ability.

The main contributions of this paper are as follows:

- We propose a novel framework that leverages identifier-aware CodeT5 and multi-label learning to identify seven types of vulnerabilities in Ethereum Smart contracts.
- We are the *first* to utilize confident learning to improve the quality of datasets that involve multiple traditional analysis tools with varying levels of accuracy

in smart contracts vulnerability labels. By identifying and correcting noisy labels, we enable the model to acquire genuine vulnerability features during subsequent training.

- We enhance interpretability in machine learning-based vulnerability detection methods by providing fine-grained code gadgets that can potentially expose the sources of these vulnerabilities.

The remainder of this paper is organized as follows. In Sect. 2, the background of our research was reviewed. Section 3 presents our IntelliCon framework. Section 4 evaluates our framework and compares it to other approaches. Section 5 introduces the related work. Finally, we conclude our work and provide directions for future research in Sect. 6.

2 Background

2.1 Ethereum Smart Contract Security Threats

The Ethereum [8] is a blockchain platform that supports smart contracts, which are programs that record transactions and perform logic. However, smart contracts can also pose security risks [9], such as reentrancy, integer overflow, denial of service, unchecked low calls, and improper access control. These risks can lead to malicious attacks or exploits, resulting in loss of funds.

2.2 Pre-trained Language Model

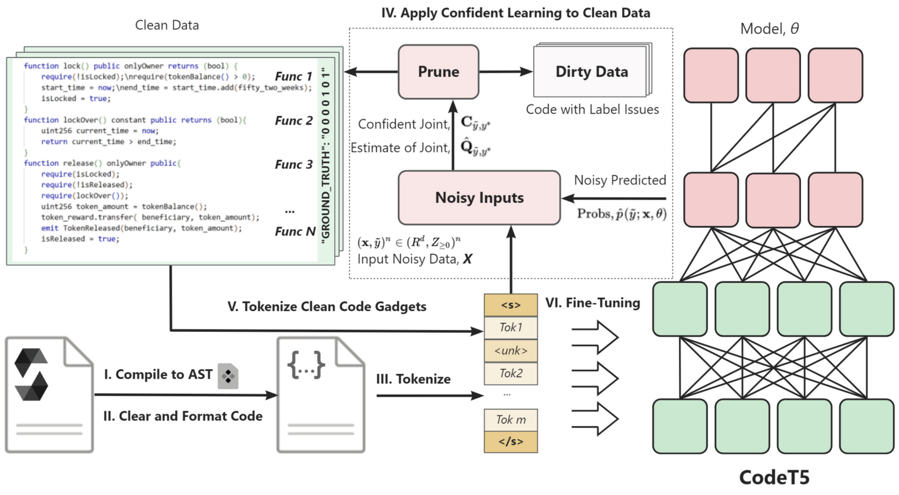
Pre-trained language models in natural language processing (NLP) involve obtaining general language representations from large datasets. It results in a better semantic representation that can be applied to the downstream tasks (e.g., text classification). As the need for semantic representation and static word embeddings to deep semantic word embeddings through word vectors, more pre-training models have gained significant attention (e.g., GPT, BERT, T5).

2.3 Text Classification Task

Text classification is a process of assigning labels to text data. Given $X = \{x_1, x_2, ..., x_n\}$ and $Y = \{y_1, y_2, ..., y_k\}$, where each x_i is a document consisting of a sequence of words $\{w_1, w_2, ..., w_m\}$, each y_i represents a class label. Text classification learns a function $f : X \rightarrow Y$ that maps each x_i to its corresponding label. Once the model is trained, the function can be used to predict the label of a new document X_{new}. X_{new} feed to f, which outputs a predicted label $\hat{y}$ based on the learned mapping from the X to Y.

3 IntelliCon

In this Section, we will give a detailed description of IntelliCon, a confident learning-based approach for fine-grained vulnerability detection in Ethereum smart contracts, leveraging the identifier-aware CodeT5 model.



```
function lock() public onlyOwner returns (bool) {
    require(!isLocked);\nrequire(tokenBalance() > 0);
    start_time = now;\nend_time = start_time.add(fifty_two_weeks);
    isLocked = true;
}
function lockOver() constant public returns (bool){
    uint256 current_time = now;
    return current_time > end_time;
}
function release() onlyOwner public{
    require(isLocked);
    require(!isReleased);
    require(lockOver());
    uint256 token_amount = tokenBalance();
    token_reward.transfer( beneficiary, token_amount);
    emit TokenReleased(beneficiary, token_amount);
    isReleased = true;
}
```

Clean Data
Func 1
Func 2
Func 3
...
Func N
"GROUND_TRUTH": "0 0 0 0 1 0 1"
IV. Apply Confident Learning to Clean Data
Prune
Dirty Data
Code with Label Issues
Confident Joint, $\mathbf{C}_{\tilde{y},y^*}$
Estimate of Joint, $\hat{\mathbf{Q}}_{\tilde{y},y^*}$
Noisy Inputs
Noisy Predicted
Probs, $\hat{p}(\tilde{y};\mathbf{x},\theta)$
$(\mathbf{x},\tilde{y})^n \in (R^d, Z_{\geq 0})^n$
Input Noisy Data, X
Model, θ
V. Tokenize Clean Code Gadgets
VI. Fine-Tuning
<s>
Tok1
<unk>
Tok2
...
Tok m
</s>
I. Compile to AST
II. Clear and Format Code
{...}
III. Tokenize
CodeT5

Fig. 1. Training Phase of the IntelliCon Framework

3.1 Overview

The IntelliCon framework consists of two main phases: model training in Fig. 1 and vulnerability detection in Fig. 2. The first phase is to train a code-aware fine-grained encoder-decoder model that can identify seven types of vulnerabilities in Ethereum smart contracts with confident learning. We adopt CodeT5 [10], a state-of-the-art pre-trained model for code understanding based on the T5 architecture. In addition, we apply self-confidence learning to improve the quality of our dataset by removing noisy data, thereby enhancing the learning effectiveness of the vulnerability features.

The second phase is to analyze the input code to interpret the specific code gadgets that caused the vulnerability. We compile to parse the input code and generate an abstract syntax tree (AST) representation to gather code gadgets. Then we tokenize and feed code gadgets into the fine-tuned CodeT5 model, classifying the vulnerabilities. Details about the vulnerability detection phase will be discussed in Sect. 3.4.

3.2 Model Details

In this section, we will introduce our model in the following steps: pre-processing, feature extraction, fine-tuning, confident learning, and vulnerability detection.

Pre-processing. First, the smart contract undergoes an initial conversion to an AST, followed by extracting code fragments from the AST. Specifically, a rule-based approach is employed to extract code gadgets from the AST that pertain to each vulnerability type, guided by their corresponding definitions and characteristics. For example, in the reentrancy vulnerability, code gadgets are

Table 1. Contract Simplification Rules

Level	Remove Rules
Source Layer	i. The code describing the version of source files ii. The code for importing other contracts iii. Comment lines in various formats iv. Spaces and line breaks
Contract Layer	i. Event definition ii. Library definition
Function Layer	Functions of pure and view types

identified that represent external calls made to untrusted contracts or functions. Second, according to Ethereum Yellow Paper [11], we summarize the following Table 1 that needs to be deleted from the source code.

Feature Extraction. To prepare code gadgets for fine-tuning with CodeT5, it is necessary to convert them into numerical features that the model can process. In this study, we utilized RobertaTokenizer, which divides each code gadget into a sequence of subword tokens and truncates or pads the sequence with $<pad>$ tokens to attain a fixed length. Additionally, the tokenizer includes special tokens, $<s>$ and $</s>$, at the start and end of the sequence, respectively. Finally, the sequence is mapped numerically to the appropriate IDs based on the CodeT5 vocabulary.

Fine-Tuning. In this study, we fine-tune the CodeT5 model for our multi-label vulnerability detection task. Given an input code x and its corresponding binary label vector $y = (y_1, ...y_n)$, where y_i indicates the presence and absence of vulnerability types i in x. However, due to severe class imbalance and discrepancies in the learning difficulties among labels, we have adopted the Zero-Level Positive and Negative Relationship (ZLPR) loss function and a class-weighted imbalanced sampler.

In Eq. 1, ZLPR optimizes pairwise comparisons of target class scores with non-target class scores and effectively balances the weight of each item using LogSumExp properties.

$$Loss = log(1 + \sum_{i \in \Omega_{neg}} e^{si}) + log(1 + \sum_{j \in \Omega_{pos}} e^{-sj}) \tag{1}$$

where Ω_{pos}, Ω_{neg} are the positive and negative class sets of the samples, respectively, s_i represents the score of the i-th class, while s_j is the j-th class.

As Eq. 2 shows, the sampler sampling probabilities to each label combination by tallying their occurrences. It helps to determine the optimal ratio of oversampling and undersampling.

$$p(N_i) = \frac{(\frac{1}{count(N_i)})}{\sum_{i=1}^{n\leq 2^k} (\frac{1}{count(N_j)})} \tag{2}$$

3.3 Confident Learning

To solve the issue of inconsistent and noisy labeling caused by the use of multiple tools for dataset annotation, We employ confident learning (CL) [12]. Specifically, CL assumes a class-conditional noise process, where the noisy labels depend only on the latent true labels, not on the data. By utilizing CL probabilistic thresholds and ranking examples to train with confidence, CL can estimate the joint distribution between noisy and true labels.

Using a fine-tuned CodeT5 model, we have calculated the out-of-sample probability of each smart contract instance via cross-validation. The resulting out-of-sample probability, combined with the noisy label, served as the input for CL leveraging rank pruning [12] to compute the problem label. We evaluated the confidence score of the data to correct potential label errors in the dataset.

Rank Pruning. We define the dataset as having n training instances, where each instance x has a true label $y \in \{0,1\}$ and a corresponding noisy label $s \in \{0,1\}$. The observed dataset can be expressed as $\{(x_1,y_1),(x_2,y_2),\ldots,(x_n,y_n)\}$. The set of positive instances is defined as $\widetilde{P} = \{x|s=1\}$ and the set of negative instances is defined as $\widetilde{N} = \{x|s=0\}$.

To map s to y, we employ a classifier g along with the parameters LB, UB, ρ_0, and ρ_1. For a given instance x, the predicted value of $g(x)$ is transformed into $\hat{s}$, which takes a binary value of 0 or 1. The threshold $LB_{y=1}$ is used to determine the true label for x. If the $g(x)$ of higher than $LB_{y=1}$, indicating that x has a true label of $y=1$ with high confidence. Similarly, for $UB_{y=0}$, if $g(x)$ is lower than $LB_{y=1}$, x has a true label of $y=0$ with high confidence.

The noise rates are defined as $\rho_1 = P(s=0|y=1)$ and $\rho_0 = P(s=1|y=0)$. Let p_{s1} denote the observed positive label $P(s=1)$, and p_{y1} denote the true positive label $P(y=1)$. Thus, the reverse noise rates can be calculated as Eq. 3.

$$\begin{aligned} \pi_0 = P(y=1\, s=0) = \frac{\rho_1 p_{y_1}}{(1-p_{s_1})} \\ \pi_1 = P(y=0\, s=1) = \frac{\rho_0(1-\rho_{y_1})}{(1-p_{s_1})} \end{aligned} \tag{3}$$

Therefore, p_{y1} can be derived by combining ρ_1, ρ_0, π_0, and π_1.

To approximate this result under trivial conditions, confident learning uses confidence counts to estimate the original noise rates. The noisy labels are assumed to be uniformly random with the Eq. 4.

$$\begin{aligned} \hat{\rho_1} &= \frac{\tilde{N}_{y=1}}{\tilde{N}_{y=1} + \tilde{P}_{y=1}}; \qquad \hat{\rho_0} = \frac{\tilde{P}_{y=0}}{\tilde{N}_{y=0} + \tilde{P}_{y=0}} \\ \hat{\pi_1} &= \frac{\hat{\rho_0}}{p_{s1}} \times \frac{1 - p_{s1} - \hat{\rho_1}}{1 - \hat{\rho_1} - \hat{\rho_0}}; \quad \hat{\pi_0} = \frac{\hat{\rho_1}}{1 - p_{s1}} \times \frac{p_{s1} - \hat{\rho_0}}{1 - \hat{\rho_1} - \hat{\rho_0}} \end{aligned} \tag{4}$$

where:

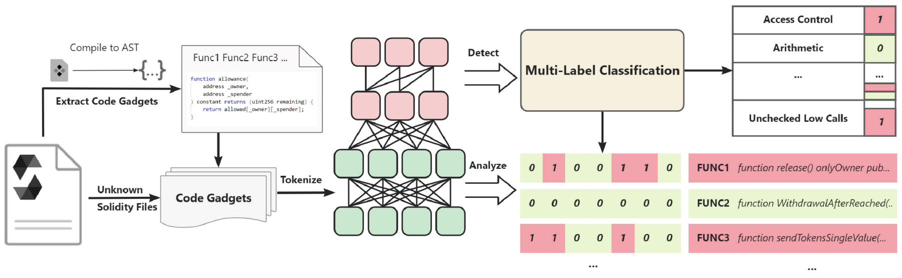
$$\begin{aligned} LB_{y=1} &= P(\hat{s} = 1 | s = 1) = E_{x \in \tilde{P}}(g(x)) = (1 - \rho_1)(1 - \pi_1) + \rho_0 \pi_1 \\ UB_{y=0} &= P(\hat{s} = 1 | s = 0) = E_{x \in \tilde{N}}(g(x)) = (1 - \rho_1)\pi_0 + \rho_0(1 - \pi_0) \\ \tilde{P}_{y=1} &= \{x \in \tilde{P} | g(x) \geq LB_{y=1}\}; \quad \tilde{N}_{y=1} = \{x \in \tilde{N} | g(x) \geq LB_{y=1}\} \\ \tilde{P}_{y=0} &= \{x \in \tilde{P} | g(x) \leq UB_{y=0}\}; \quad \tilde{N}_{y=0} = \{x \in \tilde{N} | g(x) \leq UB_{y=0}\} \end{aligned} \tag{5}$$

The pruning can be performed as follows steps. First, select the $\hat{\pi_1}|\widetilde{P}|$ instances with the smallest $g(x)$ as the set $\widetilde{P}_{conf}$, and the $\hat{\pi_0}|\widetilde{N}|$ instances with the highest $g(x)$ as $\widetilde{N}_{conf}$, $X_{conf} = \widetilde{P}_{conf} \cup \widetilde{N}_{conf}$. Next, defining yi as the predicted label for sample i, the loss function for Rank Pruning is represented by the class-conditional weighted loss function on X_{conf}:

$$\widetilde{l}(\hat{y_i}, s_i) = \frac{1}{1 - \hat{\rho_1}} l(\hat{y_i}, s_i) \cdot \mathbb{I}[[x_i \in \widetilde{P}_{conf}]] + \frac{1}{1 - \hat{\rho_0}} l(\hat{y_i}, s_i) \cdot \mathbb{I}[[x_i] \in \widetilde{N}_{conf}] \tag{6}$$

We incorporate this method to correct label errors by manually removing low-confidence items. To show the effectiveness, we evaluate the CL on our dataset quality and model performance in Sect. 4.4.

3.4 Vulnerability Detection



```
function allowance(
    address _owner,
    address _spender
) constant returns (uint256 remaining) {
    return allowed[_owner][_spender];
}
```

Fig. 2. Detection Phase of the IntelliCon Framework

To detect vulnerabilities in smart contracts, we first fine-tuned a CodeT5 model on a pruned dataset using preprocessing and CL methods. We then compiled the contract to be analyzed and performed AST analysis to traverse its syntax paths and parse all the function fragments into code gadgets. Tokenization was applied to the code gadgets, and the sliding window and fragment integration techniques described in Sect. 4.2 were utilized to split the code gadget tokens that exceeded the maximum input length set by the model. The fine-tuned CodeT5 model was used to encode them into latent representations using a transformer encoder. A linear layer was applied on top of the encoder outputs to generate the multi-label vectors for each code gadget, indicating the presence of each vulnerability type. The detection phase is depicted in Fig. 2.

4 Experiment

In this section, we will introduce the experimental results of the IntelliCon.

4.1 Experimental Settings

Dataset. We use the SmartBugs-Wild dataset [7] as our source of smart contract code gadgets for fine-tuning CodeT5. This dataset contains 47,398 smart contracts extracted from the Ethereum network and analyzed by SmartBugs [13]. However, the labeling of the SmartBugs-Wild is based on nine categories of traditional tools. It may result in variations in detection capabilities and accuracy, thus may introducing potential noise and inconsistency labels. Furthermore, the smart contracts in the dataset comprise the entire source code, as opposed to code snippets that focus on specific vulnerabilities. This may present challenges for the model to learn the relevant features and patterns from the code. Additionally, as shown in Table 2, the class distribution of data points in the dataset is significantly imbalanced, which may have implications for model performance.

Table 2. The Description of Dataset

Mark	Category	Description	Label	Percent
L1	Access control	Failure to use function modifiers or use of tx.orgin	3,801	3.07%
L2	Arithmetic	Integer over/underflows	37,597	30.36%
L3	Denial of service	The contract is overloaded with requests or computational resources	12,419	10.03%
L4	Front running	Transactions are included in a block before being mined	8,161	6.59%
L5	Reentrancy	Repeatedly execute a function by exploiting an external contract's callback function	14,773	11.91%
L6	Time manipulation	Miner can manipulate the timestamp of a block	4,069	3.29%
L7	Unchecked low level calls	call(), delegatecall() or send() fails	14,656	11.84%
–	Others	Contracts with none or unknown vulns	28,355	22.90%
Total	**Labels**		123,805	100%

After Pre-processing mentioned in Sect. 3.2, a new dataset of 27,426 code gadgets was gathered, including seven vulnerability types (i.e., access control, arithmetic, denial of service, front running, reentrancy, time manipulation, and unchecked low-level calls).

Environment. We conducted our experiments on a server with an AMD EPYC 7543 CPU (2.80 GHz, 32 cores), 28 GB of RAM, and an NVIDIA RTX A5000 GPU (24 GB of memory) on Ubuntu 20.04LTS. We used PyTorch 1.12.1 and Transformers 4.16.2 to implement and run the CodeT5 model with a pre-trained checkpoint from HuggingFace's model hub. We used Cleanlab 2.3.1 to apply the confident learning technique to prune the dataset. We performed our experiments in April 2023 using a stable network connection.

4.2 Model Tricks

IntelliCon aims to provide a smart contract vulnerability detection service under real conditions. In this experiment, the maximum sequence length is set to 521 (>512) to demonstrate that CodeT5 can surpass the length limit of traditional transformer structures of 512 and handle long sequence text tasks. It is worth noting that 521 is only an arbitrary value set for batch processing, and considering the size of video memory, IntelliCon can actually handle the text of any sequence length.

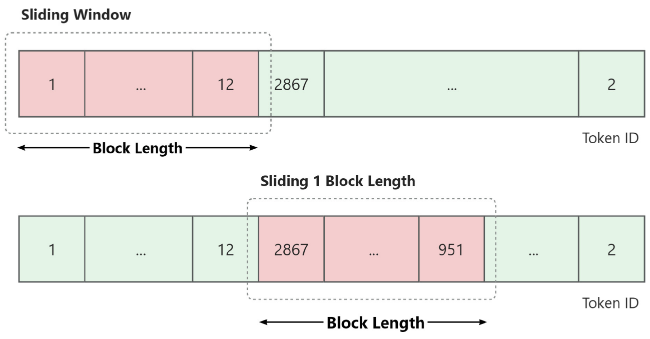


Fig. 3. Diagram of Sliding Window

At the same time, considering the performance of real devices, we provide an alternative prediction method based on fragment integration, which enables IntelliCon to successfully process ultra-long sequence text (sequence length more than 1000000) under low GPU memory. In this method, Our framework split the ultra-long sequence by setting part_length and obtaining multiple blocks with a length of part_length. Finally, IntelliCon predicts each part through a sliding window shown in Fig. 3 and synthesizes the results. We also apply this method to locate vulnerabilities by repeatedly splitting unit blocks until the model cannot recognize them, finding the minimum block where $\sum(label) \neq 1$, thus achieving vulnerability location.

To ensure the model is fast, stable, and accurate, We use AdamW as an optimizer with a learning rate of 5e−5, adam_epsilon of 1e-8, warmup_steps of 100 and a batch size of 8 for 100 epochs. In our approach, training sets, validation sets, and test sets are divided into 6:2:2. We added a dropout layer to the downstream classifier with a dropout probability of 0.5 to further avoid overfitting the model. We also use an early stopping mechanism with a patience value of 5.

IntelliCon uses Cleanlab [12] and manual evaluation methods to denoise data. Cleanlab is an open-source project based on confidence learning that provides corresponding modules to discover, evaluate, and repair datasets.

Moreover, to rectify the issue of class imbalance present in our dataset, we have implemented the ZLPR loss function in conjunction with an imbalanced dataset sampler, which serves to restore balance to the class distribution. In addition, we have incorporated dropout layers and early stopping techniques to mitigate the concern of overfitting.

4.3 Data Pruning with Confident Learning

We first split our dataset into a training set, a validation set, and a test set. We use the training set to train our identifier-aware CodeT5 model and obtain the predicted probabilities for each example and each class. We then use CL to estimate the joint distribution between noisy (given) labels and uncorrupted (unknown) labels, based on the principles of pruning noisy data and ranking examples to train with confidence.

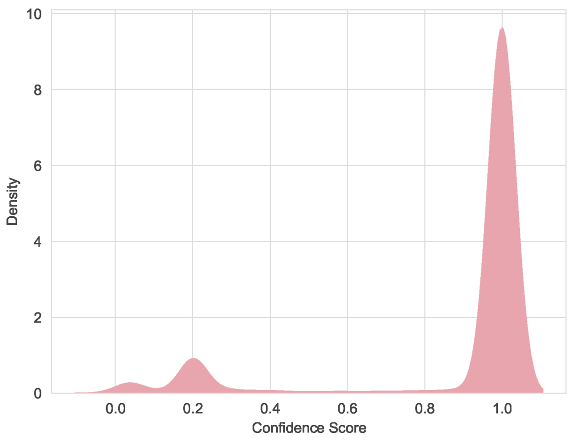


Fig. 4. Data Confidence Distribution Chart

The density plot of the data confidence score distribution for the dataset is shown in Fig. 4, which shows that the vast majority of the data has a confidence score near 1.000, with approximately 10% of the data having a label confidence score of less than 0.207, indicating a higher likelihood of label errors. After manually inspecting this portion of the data, and removing labels that were confirmed to be incorrect, We obtained our Pruned Dataset.

We then retrain our identifier-aware CodeT5 model on the Pruned Dataset and evaluate its performance on the validation set. We compare the results with the baseline model that is trained on the original training set without CL in the next section.

4.4 Evaluation

In this section, we present a comparison between our identifier-aware CodeT5 model with CL (CodeT5+CL) and other models that utilize various architectures and techniques to detect vulnerabilities in Ethereum smart contracts. Our baselines include BERT, T5, and CodeT5 models trained with and without confident learning to prune datasets.

We use the F1-score as the main metric to evaluate the performance of each model on each vulnerability type. The results are shown in Table 3.

From Table 3, we can see that our CodeT5+CL model outperforms all the baselines on all the vulnerability types and achieves the highest Micro-F1 score of 0.9591 and Macro-F1 score of 0.9293. This shows that our model can effectively and precisely detect vulnerabilities in Ethereum smart contracts by using identifier-aware CodeT5 and confident learning (Fig. 5).

Table 3. Performance of models with and without CL

Model	Method	Micro -F1	Macro -F1	F1-Score						
				L1	L2	L3	L4	L5	L6	L7
Bert	–	0.8992	0.8509	0.7493	0.9597	0.9507	0.7988	0.7912	0.8111	0.8957
Bert	CL	0.9320	0.8869	0.7929	0.9764	0.9478	0.8685	0.8599	0.8235	0.9395
T5	–	0.8971	0.8516	0.7443	0.9506	0.9379	0.8216	0.7991	0.7953	0.9126
T5	CL	0.9370	0.9045	0.8254	0.9775	0.9569	0.8961	0.8526	0.8626	0.9486
CodeT5	–	0.9338	0.8855	0.7579	0.9752	0.9739	0.8667	0.8673	0.8086	0.9490
CodeT5	**CL**	**0.9591**	**0.9293**	**0.8290**	**0.9855**	**0.9757**	**0.9222**	**0.9081**	**0.9151**	**0.9696**

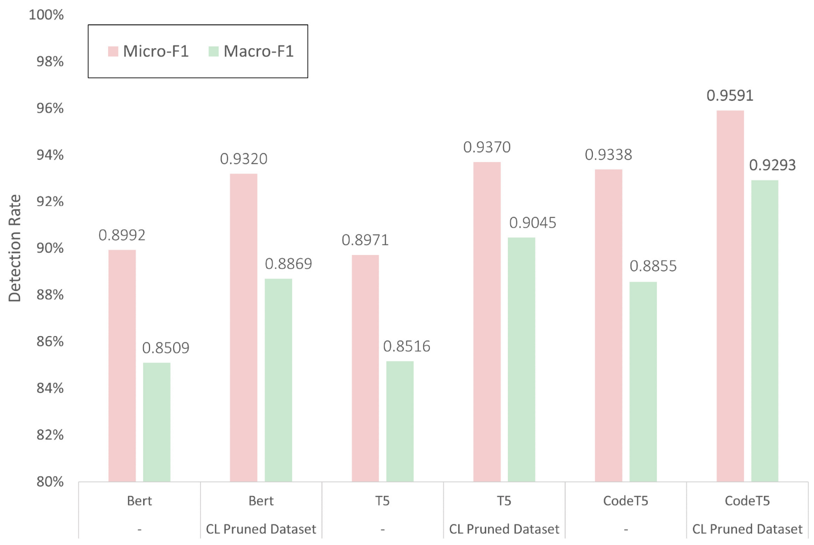


Fig. 5. F1 Score of models

We can also observe that using CL to prune the dataset improves the performance of both BERT and T5 models by about 5.12% on average F1-score, compared to using the original dataset without CL. This evidence illustrates that the utilization of CL can significantly augment the quality of dataset labeling performed by multiple annotators, especially when their individual accuracies vary. Additionally, it effectively mitigates the adverse effects of erroneous or imprecise labels on the subsequent phases of model training and evaluation.

4.5 Discussion

Our work evaluated the effectiveness of confident learning in detecting vulnerabilities in smart contracts. The results show that our proposed model achieves a Micro-F1 score of 0.9591 and a Macro-F1 score of 0.9293, which outperforms Bert, Bert+CL, T5, T5+CL, and CodeT5. This demonstrates the efficacy of our model and confident learning in improving label quality.

IntelliCon has the potential to effectively and efficiently assist in detecting and analyzing seven types of vulnerabilities in smart contracts, thereby enhancing the security of blockchain-based applications and providing valuable insights for developers and auditors in the blockchain industry.

Future research directions include applying our framework to other blockchain platforms(e.g., Fisco Bcos), exploring multitask learning approaches, and investigating other ways of enhancing the interpretability of our framework. These potential directions for future research could further enhance the performance and practicality of our framework and contribute to the advancement of smart contract security.

5 Related Work

In this section, we review the existing methods for vulnerability detection in Ethereum smart contracts.

5.1 Traditional Detection Methods

Traditional detection methods rely on manual or semi-automated techniques and expert-defined rules to identify vulnerabilities in smart contracts. They can be further divided into static analysis and dynamic analysis. Static analysis methods analyze the source code or bytecode of smart contracts without executing them. Tikhomirov et al. [14] presents *SmartCheck*, which works by translating Solidity source code into an XML-based intermediate representation and checking it against XPath patterns. Feist et al. [15] describe *Slither*, a static analysis framework that converts Solidity smart contracts into an intermediate representation called SlithIR, which allows for the application of commonly used program analysis techniques. However, they may suffer from high false positives or negatives due to the complexity and ambiguity of smart contract semantics. Moreover, they may not be able to handle dynamic features such as external calls or state changes.

Dynamic analysis methods execute smart contracts on a simulated or real blockchain environment and monitor their runtime behaviors. They can detect vulnerabilities that depend on specific inputs by generating test cases or observing transactions. Luu et al. [16] firstly performed dynamic symbol execution tools called *Oyente* to analyze vulnerability. Still, these methods may require more time and resources to execute smart contracts and collect data and not cover all possible execution paths and inputs of smart contracts.

Fuzzing methods are a special type of dynamic analysis that generates mutated inputs randomly for smart contracts and observe their exceptions. They can detect vulnerabilities that cause abnormal behaviors such as crashes or reverts by applying coverage-guided heuristics or evolutionary algorithms. Jiang et al. [17] presents *ContractFuzzer* to test Ethereum smart contracts for security vulnerabilities. The fuzzer generates fuzzing inputs, defines test oracles, instruments the EVM to log runtime behaviors, and analyzes these logs to report vulnerabilities. Fuzzing methods may be ineffective and impractical due to the high cost of executing smart contracts on a blockchain network and depend on the availability and accuracy of environments.

5.2 Machine Learning-Based Detection Methods

Machine learning-based detection methods leverage data-driven techniques to identify vulnerabilities in smart contracts. Zhuang et al. [18] propose using graph neural networks (GNNs) to represent the structure of a smart contract function and use a degree-free graph convolutional neural network (DR-GCN) and a temporal message propagation network (TMP) to learn from the normalized graphs for vulnerability detection. Lutz et al. [19] introduce *ESCORT*, a deep neural network-based multi-output method that supports lightweight transfer learning for new security vulnerabilities.

NLP-based methods can leverage pre-trained models on large-scale code corpora to improve the performance and generalization ability of vulnerability detection. For instance, Sun et al. [20] propose a new framework called *ASS-Bert* for smart contract vulnerability detection that leverages active and semi-supervised learning with a bidirectional encoder representation from transformers network.

However, due to most of the large datasets being annotated by multiple traditional vulnerability detection tools with limited accuracy [7], training on them will make the model a simulator for these tools, and the noisy labels and false positives of these tools will also affect the performance of the model. Furthermore, they may not be able to explain their predictions due to the lack of interpretability of neural networks.

6 Conclusion

In this paper, We provide a novel approach, IntelliCon, for detecting smart contract vulnerability by utilizing CodeT5 and confident learning techniques. Our work represents the *first* attempt to utilize confident learning for cleaning noisy labels to enhance detection. Moreover, we interpret specific code gadgets that cause vulnerabilities by traversing functions extracted via the AST analysis method for each vulnerability type. Experimental results demonstrated that over 95.9% micro-F1 score and 92.9% macro-F1 score could be achieved in detecting 7 types of vulnerabilities in smart contracts, outperforming the baseline models. Our work contributes to the advancement of smart contract security and provides valuable insights for developers and auditors in the blockchain industry.

References

1. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. Decentralized business review, pp. 21260–21268 (2008)
2. Zhang, S., Li, W., Li, X., Liu, B.: AuthROS: secure data sharing among robot operating systems based on Ethereum. In: Proceedings of the QRS (2022)
3. Slowmist: Blockchain security and aml analysis annual report (2023). https://www.slowmist.com/report/2022-Blockchain-Security-and-AML-Analysis-Annual-Report(EN).pdf
4. Li, W., Jiuyang, B., Li, X., Peng, H., Niu, Y., Zhang, Y.: A survey of DeFi security: challenges and opportunities. J. King Saud Univ. Comput. Inf. Sci **34**(10), 10378–10404 (2022)
5. Li, X., Chen, T., Luo, X., Yu, J.: Characterizing erasable accounts in Ethereum. In: Susilo, W., Deng, R.H., Guo, F., Li, Y., Intan, R. (eds.) ISC 2020. LNCS, vol. 12472, pp. 352–371. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62974-8_20
6. Sürücü, O., et al.: A survey on ethereum smart contract vulnerability detection using machine learning. Disrupt. Technol. Inf. Sci. VI **12117**, 110–121 (2022)

7. Durieux, T., Ferreira, J.F.: Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ICSE, pp. 530–541 (2020)
8. Li, W., Bu, J., Li, X., Chen, X.: Security analysis of DeFi: vulnerabilities, attacks and advances. In: Proceedings of the Blockchain, pp. 488–493 (2022)
9. Li, X., Chen, T., Luo, X., Wang, C.: CLUE: towards discovering locked cryptocurrencies in Ethereum. In: Proceedings of the SAC, pp. 1584–1587 (2021)
10. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet 5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the EMNLP, pp. 8696–8708 (2021)
11. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. **151**(14), 1–32 (2014)
12. Northcutt, C., Jiang, L., Chuang, I.: Confident learning: estimating uncertainty in dataset labels. J. Artif. Intell. Res. **70**, 1373–1411 (2021)
13. Ferreira, J.F., Cruz, P., Durieux, T.: Smartbugs: a framework to analyze solidity smart contracts. In: Proceedings of the ASE, pp. 1349–1352 (2021)
14. Tikhomirov, S., Voskresenskaya, E.: Smartcheck: static analysis of ethereum smart contracts. In: Proceedings of the ICSE, pp. 9–16 (2018)
15. Feist, J., Grieco, G., Groce, A.: Slither: a static analysis framework for smart contracts. In: Proceedings of the WETSEB, pp. 8–15 (2019)
16. Luu, L., Chu, D.H., Olickel, H.: Making smart contracts smarter. In: Proceedings of the CCS, pp. 254–269 (2016)
17. Jiang, B., Liu, Y., Chan, W.K.: Contractfuzzer: fuzzing smart contracts for vulnerability detection. In: Proceedings of the ASE, pp. 259–269 (2018)
18. Zhuang, Y., Liu, Z., Qian, P.: Smart contract vulnerability detection using graph neural network. In: Proceedings of the IJCAI, pp. 3283–3290 (2020)
19. Sendner, C., Chen, H., Fereidooni, H.: Smarter contracts: detecting vulnerabilities in smart contracts with deep transfer learning. In: Proceedings of the NDSS, pp. 1–18 (2023)
20. Sun, X., Liangqiong, T., Zhang, J., Cai, J., Li, B., Wang, Yu.: ASSBert: active and semi-supervised bert for smart contract vulnerability detection. J. Inf. Secur. Appl. **73**, 103423 (2023)