Lightning Cat: A Deep Learning-based Solution for Smart Contracts Vulnerability Detection

Xueyan Tang (✉ mirror.tang@alumni.stanford.edu)
Salus Security

Yuying Du
Salus Security

Alan Lai
Salus Security

Ze Zhang
Salus Security

Lingzhi Shi
Salus Security

Article

Keywords:

Posted Date: July 3rd, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3104649/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.
Lightning Cat: A Deep Learning-based Solution for Smart Contracts Vulnerability Detection

Xueyan Tang*, Yuying Du, Alan Lai, Ze Zhang, and Lingzhi Shi

1Salus Security, Beijing, 100020, China
*mirror.tang@alumni.stanford.edu

ABSTRACT

This paper aims to explore the application of deep learning in smart contract vulnerabilities detection. Smart contracts are an essential part of blockchain technology and are crucial for developing decentralized applications. However, smart contract vulnerabilities can cause financial losses and system crashes. Static analysis tools are frequently used to detect vulnerabilities in smart contracts, but they often result in false positives and false negatives because of their high reliance on predefined rules and lack of semantic analysis capabilities. Furthermore, these predefined rules quickly become obsolete and fail to adapt or generalize to new data. In contrast, deep learning methods do not require predefined detection rules and can learn the features of vulnerabilities during the training process. In this paper, we introduce a solution called Lighting Cat which is based on deep learning techniques. We trained three deep learning models for detecting vulnerabilities in smart contract: Optimized-CodeBERT, Optimized-LSTM, and Optimized-CNN. To precisely extract vulnerability features, we acquired segments of vulnerable code functions to retain critical vulnerability features. Using the CodeBERT pre-training model for data preprocessing, we could capture the syntax and semantics of the code more accurately, thereby enhancing the performance of vulnerabilities detection. This is particularly significant in the inspection of Solidity Code. To demonstrate the feasibility of our proposed solution, we evaluated its performance using the SolidiFI-benchmark dataset, which consists of 9369 vulnerable contracts injected with vulnerabilities from seven different types. Experimental results showed that, among the Lighting Cat we proposed, Optimized-CodeBERT model surpassed other methods, achieving an f1-score of 93.53%.

Introduction

Blockchain is a new application pattern based on technologies such as point-to-point transmission, encryption algorithm, consensus mechanism and distributed data storage. Since the emergence of Bitcoin, the basic blockchain system has become widely known among professionals, resulting in the development of numerous blockchain applications. This is made possible by smart contracts, which are automated programs running in a trusted environment provided by the blockchain. If there are vulnerabilities in smart contracts publicly deployed on the blockchain, attackers can exploit these vulnerabilities to launch attacks. For example, on June 17, 2016, TheDAO, the largest crowdfunding project in the blockchain industry at the time, was attacked. The hacker exploited a reentrancy vulnerability and stole 3.6 million Ether worth around $60 million from TheDAO’s asset pool, which directly led to the Ethereum blockchain splitting into ETH (Ethereum) and ETC (Ethereum Classic). On April 22, 2018, hackers targeted the BEC smart contract, which was based on the ERC-20 standard, using an integer overflow vulnerability. They transferred a significant amount of BEC tokens to two external accounts and dumped them, causing the token price to rapidly plummet to zero, disrupting the market. On February 15, 2020, the bZx protocol, a set of smart contracts built on Ethereum, experienced its first attack. The attacker profited over three hundred thousand dollars, leading the project to temporarily suspend all functions except lending. On the 18th, the bZx protocol was targeted again, and the hacker exploited the manipulation of virtual asset prices through controlling the oracle, resulting in a profit of over 2,000 Ether. The above smart contract attack incidents demonstrate that due to the control of a substantial amount of cryptocurrency and financial assets, if smart contracts are targeted and attacked, it will result in unpredictable asset losses. Conducting vulnerability detection on smart contracts can help identify and fix potential vulnerabilities in contracts at an early stage, ensuring their security and protecting against asset theft or other security risks. Therefore, smart contract vulnerability detection is crucial for ensuring security, preventing financial losses, and maintaining user trust. It is an essential aspect of smart contract development and deployment processes.

Current methods for detecting smart contract vulnerabilities include human review, static analysis, fuzz testing, and formal verification. Well-known detection tools include Oyente, Mythril, Security, Slither, and Smartcheck. These tools
automatically analyze contract code and cover various common types of smart contract security vulnerabilities, such as reentrancy, incorrect tx.origin authorization, timestamp dependency, and unhandled exceptions. However, they may produce false positives or false negatives because they highly depend on predefined detection rules and lack the ability to accurately comprehend complex code logic. Additionally, predefined rules become outdated quickly and cannot adapt or generalize to new data, which is rapidly evolving in the smart contract domain. In contrast, deep learning approaches learn from data and can continuously update themselves to stay relevant. In recent years, there has been significant research on using deep learning for smart contract vulnerability detection. However, some methods tend to overlook critical vulnerability features in their data processing approaches, and certain models lack semantic analysis capabilities for vulnerability code, leading to potential false negatives.

This paper proposes a deep learning-based solution called Lightning Cat. The solution includes three deep learning models, namely optimized CodeBERT, Optimized-LSTM, and Optimized-CNN, which are trained to detect vulnerabilities in smart contracts. To better identify vulnerability features, code snippets of functions containing vulnerabilities were obtained to preserve key features. The CodeBERT pre-training model was employed to preprocess the data, enhancing the semantic analysis capabilities.

The main contributions of this paper can be summarized as follows:

1. This paper designs a smart contracts vulnerabilities detection solution called Lightning Cat using deep learning methods. The solution optimizes three deep learning models.
2. We introduce an effective data preprocessing method that captures the semantic features of smart contract vulnerabilities. During the data preprocessing stage, we retrieve code snippets of functions containing vulnerabilities to extract vulnerability features. We also use the CodeBERT pre-trained model for data preprocessing to enhance the model’s semantic analysis capabilities, with the primary goal of improving model performance.
3. Based on the experimental evaluation results, the Lightning Cat proposed in this paper shows better detection performance than other vulnerability detection tools. The optimized CodeBERT model in Lightning Cat outperforms Optimized-LSTM and Optimized-CNN models, achieving a recall rate of 93.55%, which is 11.85% higher than Slither, a precision rate of 96.77%, and an F1-score of 93.53%.

In addition to smart contract vulnerability detection, the Lightning Cat can also be extended to other areas of code vulnerability detection. Modern software systems are prone to various types of vulnerabilities, such as buffer overflow, null pointer dereference, and logic errors. For instance, buffer overflow code vulnerabilities are characterized by the use of unsafe string manipulation functions like strcpy and strcat without proper input boundary checks. Null pointer dereference vulnerabilities involve the misuse of dangling pointers by failing to set a pointer to NULL after freeing the associated memory. Logic errors manifest when incorrect logical operators are used in conditional statements, such as using $=$ instead of $==$. By learning and training on a large number of code samples, Lightning Cat can extract and comprehend different types of vulnerabilities. It also employs the CodeBERT pre-trained model for data preprocessing, making it better suited for identifying code vulnerabilities. As a result, it can detect various kinds of code vulnerabilities, thereby enhancing the security and dependability of the code.

Related Work

In this section, we present related work on the detection of smart contract vulnerabilities, focusing primarily on static analysis techniques and deep learning methods.

Static Analysis Techniques

The Mythril security analysis method is designed to inspect bytecode executed in the Ethereum Virtual Machine (EVM). When defects are found in a program, it can help infer potential causes by analyzing input records. This assists in identifying existing vulnerabilities and reducing the likelihood of exploiting them. It utilizes taint analysis and symbolic execution techniques. However, when performing taint analysis, it faces limitations when crossing memory boundaries. This limitation becomes more severe when dealing with reference-style parameter invocations. Additionally, Mythril may encounter the state explosion problem when processing complex contracts. Symbolic execution is a powerful general method for detecting vulnerabilities, but it may not cover all execution paths, leading to false positives.

Slither is a static code analysis tool used for detecting security vulnerabilities and potential issues in Solidity smart contracts. It integrates numerous detectors capable of identifying different types of vulnerabilities. Compared to Mythril, Slither is much more efficient and performs fast detection. However, Slither lacks formal semantic analysis, which limits its ability to perform more detailed security analysis and accurately determine low-level information such as gas calculations.
SmartCheck uses static analysis techniques to detect common security vulnerabilities and coding issues in smart contracts. It offers numerous rule sets to identify different types of vulnerabilities and improve contract security. However, due to its heavy reliance on logical rules for vulnerability detection, it may generate false positives and false negatives. Furthermore, it may fail to detect severe programming errors, leading to overlooked vulnerabilities or incorrect reporting.

**Deep Learning for Smart Contract Vulnerabilities Detection**

Huang et al.\(^\text{18}\) proposed a vulnerabilities detection model for smart contracts using a convolutional neural network. This network converts the binary representation of vulnerable code into RGB images. However, converting binary files to image format makes it challenging to preserve the syntax and semantic information of the code. Although this approach improves accuracy to some extent\(^\text{19}\), it suffers from a high false negative rate.

Liao et al.\(^\text{20}\) used N-gram language modeling and tf-idf feature vectors to characterize smart contract source code. They trained traditional machine learning models to verify 13 types of vulnerabilities and employed a gray-box fuzz testing mechanism for real-time transaction validation. However, this method treats certain critical opcodes as stop words during the representation process, which can result in false negatives and missed detections.

Yu et al.\(^\text{21}\) developed the first systematic and modular framework for smart contract vulnerability detection based on deep learning. They introduced the concept of Vulnerability Candidate Slice (VCS), which focuses on analyzing the dependencies between diverse data and control program elements. Experimental results showed a significant improvement of 25.76% in the F1 score using this approach. However, the performance improvement is not substantial for vulnerability types with limited data and control flow dependencies.

From related work, it has been observed that some tools based on static analysis techniques suffer from false positives and false negatives, mainly due to their reliance on predefined rules. These tools lack the ability to perform syntax and semantic analysis, and the predefined rules can become outdated quickly and cannot adapt or generalize to new data. In contrast, deep learning methods do not require predefined detection rules and can learn vulnerability features during the training process.

We have found that some literature has utilized deep learning for smart contract vulnerabilities detection. These works provide various methods for data preprocessing aimed at enabling the deep learning models to extract vulnerability features more effectively. However, some methods may result in the deletion of important keywords or the ignoring of critical vulnerability features during data processing\(^\text{22}\). Additionally, some of the models used may have an insufficient understanding of the semantic characteristics of vulnerability code programs\(^\text{23}\), which can result in false negatives. To address these issues, we utilized the CodeBERT pre-training model for data preprocessing. CodeBERT is a Transformer-based pre-training model designed specifically for learning and processing source code. It demonstrates stronger semantic analysis abilities, providing significant advantages in smart contract vulnerability detection. Additionally, we introduced the concept of critical vulnerability code segments and removed code unrelated to vulnerabilities from the training samples. We retained only the function code of critical vulnerabilities for learning. This strategy eliminates training noise introduced by redundant code, reduces model complexity, and improves model performance. During the model training stage, we utilized three models - optimized CodeBERT, Optimized-LSTM, and Optimized-CNN - to capture vulnerability features more effectively.

Using the aforementioned methods, our proposed Lighting Cat tool extracts critical features from vulnerability code and has strong semantic analysis capabilities, which significantly improves model performance.

**Methodology**

Figure 1 illustrates the complete process of developing a vulnerability detection model called **Lighting Cat** for smart contracts, which consists of three stages. The first stage involves building and preprocessing the labeled dataset of vulnerable Solidity code. In the second stage, training three models (Optimized-CodeBERT, Optimized-LSTM, and Optimized-CNN) and comparing their performance to determine the best one. Finally, in the third stage, the selected model is evaluated using the Solidify-benchmark dataset to assess its effectiveness in detecting vulnerabilities.

**Data Preprocessing**

During the data preprocessing phase, we collected three datasets and subsequently performed data cleaning. Finally, we utilized the CodeBERT model to encode the data.

**Data Collection**

The training dataset is primarily composed of 10,000 contracts from the Slither Audited Smart Contracts Dataset\(^\text{24}\), 20,000 contracts from smartbugs-wild\(^\text{25}\), and contracts containing typical vulnerabilities identified during expert audits. To effectively compare our model’s results with those of other auditing tools, we use the Solidify-benchmark dataset\(^\text{26}\) as the testing dataset.
We select four types of vulnerabilities from the Solidify-benchmark dataset: Re-entrancy, Timestamp-Dependency, Unhandled-Exception, and tx.origin. These four types of vulnerabilities correspond to three auditing tools: Slither, Smartcheck, and Mythril. We utilized the Slither and a manually labeled training set consisting of 31,909 entries, along with a test set containing 5,434 entries. Table 1 displays the mapping between the four types of vulnerabilities and the three auditing tools.

**Table 1.** Mapping of Four Vulnerability Types

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Slither</th>
<th>Smartcheck</th>
<th>Mythril</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-entrancy</td>
<td>reentrancy-benign</td>
<td>SOLIDITY_ETRNANCY</td>
<td>External Call To User-Supplied Address</td>
</tr>
<tr>
<td></td>
<td>reentrancy-eth</td>
<td></td>
<td>External Call To Fixed Address State change</td>
</tr>
<tr>
<td></td>
<td>reentrancy-unlimited-gas</td>
<td></td>
<td>after external call</td>
</tr>
<tr>
<td></td>
<td>reentrancy-no-eth</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timestamp-Dependency</td>
<td>timestamp</td>
<td>SOLIDITY_EXACT_TIME</td>
<td>Dependence on predictable environment variable</td>
</tr>
<tr>
<td>Unhandled-Exceptions</td>
<td>unchecked-send</td>
<td>SOLIDITY_UNCHECKED_CALL</td>
<td>Unchecked Call Return Value</td>
</tr>
<tr>
<td></td>
<td>unchecked-lowlevel</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tx.origin</td>
<td>tx-origin</td>
<td>SOLIDITY_TX_ORIGIN</td>
<td>Use of tx.origin</td>
</tr>
</tbody>
</table>

**Data Cleaning**

The length of a smart contract typically depends on its functionality and complexity. Some complex contracts can exceed several thousand tokens. However, handling long text has been a long-standing challenge in deep learning. Transformer-based models can only handle a maximum of 512 tokens. Therefore, we attempted two methods to address the issue of text length exceeding 510 tokens.

**Direct Splitting** The data is split into chunks of 510 tokens each, and all the chunks are assigned the same label. For example, if we have a group of Re-entrancy vulnerability code with a length of 2000 tokens, it would be split into four chunks, each containing 510 tokens. If there are chunks with fewer than 510 tokens, we pad them with zeros. However, the training results show that the model’s loss does not converge. We speculate that this is due to the introduction of noise from unrelated chunks,
which negatively affects the model’s generalization capability.

**Vulnerability Function Code Extraction**  Audit experts extracted the function code of vulnerabilities from smart contracts and assigned corresponding vulnerability labels. If the extracted code exceeds 510 tokens, it is truncated, and if the code falls short of 510 tokens, it is padded with zeros. This approach ensures consistent input data length, addresses the length limitation of Transformer models, and preserves the characteristics of the vulnerabilities.

After comparing the two methods, we observed that training on vulnerability-based function code helped the model’s loss function converge better. Therefore, we chose to use this data processing method in subsequent experiments. Additionally, we removed unrelated characters such as comments and newline characters from the functions to enhance the model’s performance. As shown in Figure 2, we only extracted the function parts containing the vulnerability code, reducing the length of the training dataset while maintaining the vulnerability characteristics. This approach not only improves the model’s accuracy, but also enhances its generalization ability.

![Figure 2. Extraction of Vulnerable Function Code](image)

**Data Encoding**
CodeBERT is a pretraining model based on the Transformer architecture, specifically designed for learning and processing source code. By undergoing pretraining on extensive code corpora, CodeBERT acquires knowledge of the syntax and semantic relationships inherent in source code, as well as the interactive dynamics between different code segments.

During the data preprocessing stage, CodeBERT is employed due to its strong representation ability. The source code undergoes tokenization, where it is segmented into tokens that represent semantic units. Subsequently, the tokenized sequence is encoded into numerical representations, with each token mapped to a unique integer ID, forming the input token ID sequence. To meet the model’s input requirements, padding and truncation operations are applied, ensuring a fixed sequence length. Additionally, an attention mask is generated to distinguish relevant positions from padded positions containing invalid information. Thus, the processed data includes input IDs and attention masks, transforming the source code text into a numericalized format compatible with the model while indicating the relevant information through the attention mask.

For Optimized-LSTM and Optimized-CNN models, direct processing of input IDs and masks is not feasible. Therefore, CodeBERT is utilized to further process the data and convert it into tensor representations of embedding vectors. The input IDs and attention masks obtained from the preprocessing steps are passed to the CodeBERT model to obtain meaningful representations of the source code data. These embedding vectors can be used as inputs for Optimized-LSTM and Optimized-CNN models, facilitating their integration for subsequent vulnerability detection.

**Models**
In the current stage, our approach involves the utilization of three machine learning models: Optimized-CodeBERT, Optimized-LSTM, and Optimized-CNN. The CodeBERT model is specifically fine-tuned to enhance its compatibility with the target task by...
accepting preprocessed input IDs and attention masks as input. However, in the case of Optimized-LSTM and Optimized-CNN models, we do not conduct any fine-tuning on the CodeBERT model for data preprocessing.

**Model 1: Optimized-CodeBERT**

CodeBERT is a specialized application that utilizes the Transformer model for learning code representations in code-related tasks. In this paper, we focus on fine-tuning the CodeBERT model to specifically address the needs of smart contract vulnerability detection. The CodeBERT model is built upon the Transformer architecture, which comprises multiple encoder layers. Prior to entering the encoder layers of CodeBERT, the input data undergoes an embedding process. Following the encoding stage of CodeBERT, fully connected layers are added for classification purposes. The model architecture of our CodeBERT implementation is depicted in Figure 3.

![Figure 3. Our Optimized-CodeBERT Model Architecture](image)

**Word Embedding and Position Encoding** In the data preprocessing stage, we have utilized a specialized CodeBERT tokenizer to process each word into the input information. In this model training stage, the tokenizer employs embedding methods, which are used to convert text or symbol data into vector representations. This processing transforms each word into a 512-dimensional word embedding. In addition, we introduce position embedding, which is a technique introduced to assist the model in understanding the positional information within the sequence. It associates each position with a specific vector representation to express the relative positions of tokens in the sequence. For a given position \( i \) and dimension \( k \), the Position Encoding \( PE(i, k) \) is computed as follows:

\[
PE(i, k) = \begin{cases} 
\sin \left( \frac{i}{10000^{k/d}} \right) & \text{if } k \text{ is even} \\
\cos \left( \frac{i}{10000^{k/d}} \right) & \text{if } k \text{ is odd}
\end{cases}
\]

Here, \( d \) represents the dimension of the input sequence. The formula utilizes sine and cosine functions to generate position vectors, injecting positional information into the embeddings. The exponential term \( \frac{i}{10000^{k/d}} \) controls the rate of change of the position encoding, ensuring differentiation among positions. By adding the Position Encoding to the Word Embedding, positional information is integrated into the embedded representation of the input sequence. This enables CodeBERT to better comprehend the semantics and contextual relationships of different positions in the code. The processing steps are illustrated in Figure 4.
Encoder Layers  The CodeBERT model performs deep representation learning by stacking multiple encoder layers. Each encoder layer comprises two sub-layers: multi-head self-attention and feed-forward neural network. The self-attention mechanism helps encode the relationships and dependencies between different positions in the input sequence. The feed-forward neural network is responsible for independently transforming and mapping the features at each position.

The multi-head self-attention mechanism calculates attention weights, denoted as $w_{ij}$, for each position $i$ in the input code sequence. The attention weights are computed using the following equation:

$$w_{ij} = \text{Softmax} \left( \frac{q_i \cdot k_j}{\sqrt{d}} \right)$$

Here, $q_i$ represents the query at position $i$, $k_j$ denotes the key at position $j$, and $d$ is the dimension of the queries and keys. The output of the self-attention mechanism at position $i$, denoted as $o_i$, is obtained by multiplying the attention weights $w_{ij}$ with their corresponding values $v_j$ and summing them up:

$$o_i = \sum_{j=1}^{n} w_{ij} \cdot v_j$$

where $n$ is the length of the input sequence.

Each encoder layer also contains a feed-forward neural network sub-layer, which processes the output of the self-attention sub-layer using the following equation:
Here, \( x \) represents the output of the self-attention sub-layer, and \( W_1, b_1 \) and \( W_2, b_2 \) are the parameters of the feed-forward neural network.

**Fully Connected Layers** To output the classification labels, we added fully connected layers. Firstly, we added a new linear layer with 100 features on top of the existing linear layer. To avoid the limited capacity of a single linear layer, we utilized the ReLU activation function. Additionally, to prevent overfitting, we introduced a dropout layer with a dropout rate of 0.1 after the activation layer. Lastly, we used a linear layer with four features for the output. During the fine-tuning process, the parameters of these new layers were updated.

**Model 2: Optimized-LSTM**

The Optimized-LSTM model is specifically designed for processing sequential data, capable of capturing temporal dependencies and syntactic-semantic information\(^{28}\). For the task of smart contract vulnerability detection, our constructed Optimized-LSTM model provides a serialization-based representation of Solidity source code, taking into account the order of statements and function calls. The Optimized-LSTM model captures the syntax, semantics, and dependencies within the code, enabling an understanding of the logical structure and execution flow. Compared to traditional RNNs, the Optimized-LSTM model we constructed addresses the issue of vanishing or exploding gradients when handling long sequences\(^{29}\). This is accomplished through the key mechanism of gated cells, which enable selective retention or forgetting of previous states. The model consists of shared components across time steps, including the cell, input gate, output gate, and forget gate. In the Optimized-LSTM model, we have defined an LSTM layer and a fully connected layer, with the LSTM layer being the core component. Within the LSTM layer, the input \( x^{(t)} \), the output from the previous time step \( h^{(t-1)} \), and the cell state from the previous time step \( c^{(t-1)} \) are fed into an LSTM unit. This unit contains a forget gate \( f^{(t)} \), an input gate \( i^{(t)} \), and an output gate \( o^{(t)} \), as shown in Figure 5.

![Figure 5. The Architecture of Optimized-LSTM](image)

In the model, we utilize a bidirectional Optimized-LSTM, where the forward Optimized-LSTM and backward Optimized-LSTM are independent and concatenated at the final step. This allows for better capture of long-term dependencies and local correlations within the sequence. During the forward propagation of the model, the input \( x \) is first passed through the Optimized-LSTM layer to obtain the output \( h \) and the final cell state \( c \). Since the lengths of the data instances may vary, we calculate the average output by averaging the outputs at each time step in \( h \). Then, the average output is fed into a fully connected layer to obtain the final prediction output \( y \). We used the cross-entropy loss function \( L \) for training, which is defined as:

\[
L_i = - \sum_{j=1}^{N} y_{i,j} \log \hat{y}_{i,j}
\]
Here, $N$ represents the number of classes, $y_{i,j}$ denotes the probability of the $j$th class in the true label of sample $i$, and $\hat{y}_{i,j}$ represents the probability of sample $i$ being predicted as the $j$th class by the model.

**Model 3: Optimized-CNN**

The Convolutional Neural Network (CNN) is a feedforward neural network that exhibits remarkable advantages when processing two-dimensional data, such as the two-dimensional structures represented by code. In our model design, we transform the code token sequence into a matrix, and CNN efficiently extracts local features of the code and captures the spatial structure, effectively capturing the syntax structure, relationships between code blocks, and important patterns within the code.

The Optimized-CNN primarily consists of convolutional layers, pooling layers, fully connected layers, and activation functions. Its core idea is to extract features from input data through convolution operations, reduce the dimensionality of feature maps through pooling layers, and ultimately perform classification or regression tasks through fully connected layers. The key module of the Optimized-CNN is the convolutional layer, which is computed as follows:

$$y_{i,j} = \sigma \left( \sum_{k=1}^{K} \sum_{l=1}^{L} \sum_{m=1}^{M} w_{k,l,m} x_{i+l-1,j+m-1,k} + b \right)$$

Here, $x_{i,j,k}$ represents the element value of the input data at the $i$-th row, $j$-th column, and $k$-th channel, $w_{k,l,m}$ represents the weight value of the $k$-th channel, $l$-th row, and $m$-th column of the convolutional kernel, and $b$ represents the bias term. $\sigma$ denotes the activation function, and in this case, we use the Rectified Linear Unit (ReLU).

The output of the convolutional layer is passed to the pooling layer for further processing. The commonly used pooling methods are Max Pooling and Average Pooling. In this case, we employ Max Pooling, and the calculation formula is as follows:

$$y_{i,j} = \max_{m=1}^{M} \max_{n=1}^{N} x_{i+m-1,j+n-1}$$

Pooling operations can reduce the dimensionality of feature maps, model parameters, and to some extent alleviate overfitting issues. Finally, a fully connected layer is used to compute the model, which is expressed as:

$$y = \sigma(Wx + b)$$

Here, $x$ represents the output of the previous layer, $W$ and $b$ denote the weights and bias terms, and $\sigma$ is the activation function. By stacking multiple convolutional layers, pooling layers, and fully connected layers, we construct an Optimized-CNN model as shown in Figure 6, which has powerful feature extraction and classification capabilities for smart contract classification.

**Experiments**

To ensure a fair evaluation of different methods, we trained and tested them in identical environments. All experiments were performed on a computer featuring an Intel Xeon(R) Silver 4210R CPU clocked at 2.4GHz, dual RTX A5000 GPUs, and 128GB of RAM, running on the Windows operating system, utilizing the PyCharm software, the PyTorch framework, and the Python programming language.
Parameter Settings
Then, we do the tuning process with respect to each hyperparameter. For Optimized-CodeBERT model, we employed the AdamW optimizer\(^3\) and conducted a grid search to find the optimal settings for hyperparameters. The hyperparameters and their corresponding search ranges were as follows: learning rate: \((3e-5, 1e-4, 3e-4)\), batch size: \((32, 64, 128, 256)\), dropout rate: \((0.1, 0.2, 0.3, 0.4, 0.5)\), L2 regularization: \((1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1)\), learning rate decay gamma: \((0.98, 0.99)\), number of fully connected layers: \((1, 2, 3)\), and number of epochs: \((10, 20, 30, 40, 50, 60)\). The cross-entropy loss was calculated using the BCEWithLogitsLoss method. The best parameter settings corresponding to the final results are shown in Table 2.

<table>
<thead>
<tr>
<th>Model Parameters</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer of MLP</td>
<td>2</td>
</tr>
<tr>
<td>Epoch</td>
<td>60</td>
</tr>
<tr>
<td>Batch size</td>
<td>128</td>
</tr>
<tr>
<td>Learning rate</td>
<td>1e-3</td>
</tr>
<tr>
<td>Dropout</td>
<td>0.1</td>
</tr>
<tr>
<td>Optimizer</td>
<td>Adam</td>
</tr>
<tr>
<td>Loss function</td>
<td>BCEWithLogitsLoss</td>
</tr>
<tr>
<td>learning rate decay gamma</td>
<td>0.98</td>
</tr>
<tr>
<td>L2 regularization</td>
<td>1e-4</td>
</tr>
</tbody>
</table>

For Optimized-LSTM Model, the best parameter settings are shown in Table 3.

<table>
<thead>
<tr>
<th>Layer Parameters</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>0.0001</td>
</tr>
<tr>
<td>Input Dimension</td>
<td>768</td>
</tr>
<tr>
<td>Hidden Dimension</td>
<td>128</td>
</tr>
<tr>
<td>Number of Layers</td>
<td>2</td>
</tr>
<tr>
<td>Bidirectional</td>
<td>True</td>
</tr>
<tr>
<td>Batch First</td>
<td>True</td>
</tr>
<tr>
<td>Dropout</td>
<td>0.5</td>
</tr>
<tr>
<td>Output Dimension</td>
<td>4</td>
</tr>
</tbody>
</table>

For Optimized-CNN Model, the best parameter settings are shown in Table 4.

<table>
<thead>
<tr>
<th>Layer Parameters</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning rate</td>
<td>0.0001</td>
</tr>
<tr>
<td>Conv1d_1</td>
<td>in_channels=768, out_channels=256, kernel_size=3</td>
</tr>
<tr>
<td>Conv1d_2</td>
<td>in_channels=256, out_channels=128, kernel_size=3</td>
</tr>
<tr>
<td>Conv1d_3</td>
<td>in_channels=128, out_channels=64, kernel_size=3</td>
</tr>
<tr>
<td>MaxPool1d</td>
<td>kernel_size=2</td>
</tr>
<tr>
<td>Linear_1</td>
<td>in_channels=64, out_channels=32</td>
</tr>
<tr>
<td>Linear_2</td>
<td>in_channels=32, out_channels=6</td>
</tr>
</tbody>
</table>

Metrics
To evaluate our methods, we use various performance metrics, including accuracy, F1 score, recall, and precision. Accuracy is indeed the ratio of correctly predicted instances (both true positives and true negatives) to the total number of instances. It provides a general measure of overall correctness.

\[
\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives}}
\]
F1 score is another important metric that combines both precision and recall. It considers the trade-off between them and provides a balance between the two. F1 score is particularly useful when the dataset is imbalanced or when both precision and recall are important.

\[
F1 \text{ score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}
\]

Precision, also known as positive predictive value, measures the proportion of correctly predicted positive instances (true positives) out of all instances predicted as positive. It focuses on the accuracy of positive predictions.

\[
\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}
\]

Recall, also known as sensitivity or true positive rate, calculates the proportion of correctly predicted positive instances (true positives) out of all actual positive instances. It focuses on the ability of the model to identify all positive instances.

\[
\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}
\]

**Results and Discussion**

We used Solidify-benchmark as the testing dataset, and the comparison results of the metrics of the three models in Lighting Cat are shown in Table 5.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>F1</th>
<th>Accuracy</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized-CodeBERT</td>
<td>93.53%</td>
<td>96.77%</td>
<td>96.77%</td>
<td>93.55%</td>
</tr>
<tr>
<td>Optimized-LSTM</td>
<td>63.05%</td>
<td>81.96%</td>
<td>73.61%</td>
<td>64.06%</td>
</tr>
<tr>
<td>Optimized-CNN</td>
<td>70.62%</td>
<td>85.54%</td>
<td>71.61%</td>
<td>71.36%</td>
</tr>
</tbody>
</table>

Clearly, the performance of Optimized-CodeBERT is the best among all models (as shown in Table 5), with the highest scores in all metrics. Its F1-score is 30.48% higher than that of Optimized-LSTM and 22.91% higher than that of Optimized-CNN.

In addition, we obtained the metrics results of the three models for four types of vulnerabilities, as shown in Table 6.

<table>
<thead>
<tr>
<th>Vulnerability</th>
<th>Method</th>
<th>Accuracy(%)</th>
<th>Precision(%)</th>
<th>Recall(%)</th>
<th>F1(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re-entrancy</td>
<td>Optimized-CodeBERT</td>
<td>93.58</td>
<td>85.45</td>
<td>89.20</td>
<td>87.29</td>
</tr>
<tr>
<td></td>
<td>Optimized-LSTM</td>
<td>75.08</td>
<td>49.80</td>
<td>100</td>
<td>66.49</td>
</tr>
<tr>
<td></td>
<td>Optimized-CNN</td>
<td>91</td>
<td>73.31</td>
<td>100</td>
<td>84.60</td>
</tr>
<tr>
<td>Timestamp-Dependency</td>
<td>Optimized-CodeBERT</td>
<td>97.29</td>
<td>90.43</td>
<td>99.92</td>
<td>94.94</td>
</tr>
<tr>
<td></td>
<td>Optimized-LSTM</td>
<td>85.08</td>
<td>75.49</td>
<td>61.12</td>
<td>67.55</td>
</tr>
<tr>
<td></td>
<td>Optimized-CNN</td>
<td>75.27</td>
<td>51.62</td>
<td>42.79</td>
<td>46.79</td>
</tr>
<tr>
<td>Unhandled-Exceptions</td>
<td>Optimized-CodeBERT</td>
<td>96.23</td>
<td>100</td>
<td>100</td>
<td>99.96</td>
</tr>
<tr>
<td></td>
<td>Optimized-LSTM</td>
<td>84.63</td>
<td>100</td>
<td>39.23</td>
<td>56.35</td>
</tr>
<tr>
<td></td>
<td>Optimized-CNN</td>
<td>80.53</td>
<td>61.52</td>
<td>61.43</td>
<td>61.47</td>
</tr>
<tr>
<td>tx.origin</td>
<td>Optimized-CodeBERT</td>
<td>99.98</td>
<td>99.93</td>
<td>85.08</td>
<td>91.94</td>
</tr>
<tr>
<td></td>
<td>Optimized-LSTM</td>
<td>83.03</td>
<td>69.17</td>
<td>55.91</td>
<td>61.84</td>
</tr>
<tr>
<td></td>
<td>Optimized-CNN</td>
<td>95.38</td>
<td>100</td>
<td>81.21</td>
<td>89.63</td>
</tr>
</tbody>
</table>

From Table 6, it can be seen that among the four types of vulnerabilities, Optimized-CodeBERT has the best detection performance for Timestamp-Dependency and Unhandled-Exceptions. However, for the Re-entrancy vulnerability, the Recall of Optimized-CodeBERT is lower than that of Optimized-LSTM and Optimized-CNN. When detecting the tx.origin vulnerability, Optimized-CodeBERT has higher Accuracy, Recall, and F1 than the other two models, while its Precision is 0.07% lower than
that of Optimized-CNN. The three models use the same training set, but their detection performance differs because they have different modeling and generalization capabilities. Overall, Optimized-CodeBERT has better detection performance.

We obtained the vulnerability detection results of different methods in the Solidifi-benchmark testing dataset, and the true positive results detected by each method are shown in Table 7. "NA" indicates vulnerabilities that cannot be identified by the corresponding method.

**Table 7. True Positive Detection Results of Different Methods**

<table>
<thead>
<tr>
<th>Security bug</th>
<th>Re-entrancy</th>
<th>Timestamp dep</th>
<th>Unhandle exp</th>
<th>tx.origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Injected bugs</td>
<td>1343</td>
<td>1381</td>
<td>1374</td>
<td>1336</td>
</tr>
<tr>
<td>Manticore</td>
<td>93</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Mythril</td>
<td>258</td>
<td>571</td>
<td>618</td>
<td>891</td>
</tr>
<tr>
<td>Oyente</td>
<td>335</td>
<td>0</td>
<td>322</td>
<td>NA</td>
</tr>
<tr>
<td>Security</td>
<td>1111</td>
<td>NA</td>
<td>701</td>
<td>NA</td>
</tr>
<tr>
<td>Smartcheck</td>
<td>0</td>
<td>479</td>
<td>49</td>
<td>97</td>
</tr>
<tr>
<td>Slither</td>
<td>1343</td>
<td>844</td>
<td>917</td>
<td>1336</td>
</tr>
<tr>
<td>Optimized-CodeBERT</td>
<td>1198</td>
<td>1380</td>
<td>1169</td>
<td>1336</td>
</tr>
<tr>
<td>Optimized-LSTM</td>
<td>1343</td>
<td>844</td>
<td>539</td>
<td>747</td>
</tr>
<tr>
<td>Optimized-CNN</td>
<td>1343</td>
<td>591</td>
<td>844</td>
<td>1085</td>
</tr>
</tbody>
</table>

Table 7 displays the vulnerability detection results for different types of vulnerabilities. "Injected bugs" represents the actual quantity of vulnerabilities, and "tp" (true positive) represents the number of detected vulnerabilities. From Table 7, it can be seen that among these auditing tools (Manticore, Mythril, Oyente, Security, Slither), Slither detected the most actual vulnerabilities. In terms of the detection results of all methods, Slither, Optimized-LSTM, and Optimized-CNN detected the most true positive for the Re-entrancy vulnerability, while Optimized-CodeBERT detected the most vulnerabilities for the three types of vulnerabilities (Timestamp_Dependency, Unhandle_Exceptions, tx.origin).

To better compare the detection performance of different methods, we will only compare the methods that can detect the four types of vulnerabilities. The comparison results of the Recall of different methods are shown in Figure 7.

![Figure 7. Comparison of the Recall Results of Different Methods](image)

Figure 7 illustrates the comparison of Recall results among different methods, measuring the classification models' capability to accurately identify true positive samples. The figure compares the recall rates of six methods, namely Mythril, Smartcheck, Slither, Optimized-CodeBERT, Optimized-LSTM, and Optimized-CNN. The findings indicate that the Optimized-CodeBERT model exhibits the highest recall at 93.55%, surpassing Slither by 11.85%. This highlights the Optimized-CodeBERT model's
exceptional accuracy and reliability in detecting and identifying true positive samples. Conversely, the Optimized-LSTM and Optimized-CNN models demonstrate relatively lower recall rates of 64.06% and 71.36%, respectively, suggesting potential challenges or limitations in recognizing true positive samples.

Based on the insights gained from Figure 7, it is evident that the Optimized-CodeBERT method excels in recall, displaying superior proficiency in identifying true positive samples. These findings offer valuable guidance for model selection and practical applications.

**Conclusion**

This paper introduces Lighting Cat, which uses deep learning methods to detect vulnerabilities in smart contracts, including three models: Optimized-CodeBERT, Optimized-LSTM, and Optimized-CNN. Based on experimental results, the Optimized-CodeBERT model achieved the best overall performance. We optimized and compared three models, and found that Optimized-CodeBERT achieved the best results in evaluation metrics such as Accuracy, Precision, and F1-score. This research utilized the CodeBERT pre-trained model for data preprocessing, which improved the ability of code semantic analysis. In data preprocessing, we extracted problem code segments functions, which not only considered the key features of smart contract vulnerability code but also solved the length limitation problem of deep learning for processing long texts. This approach avoids issues such as unclear features due to excessively long texts or overfitting due to excessively short texts, thereby improving the model’s performance. The results show that the proposed method has more reasonable data preprocessing and model optimization, resulting in better detection performance.

This paper analyzed the detection performance of each type of vulnerability and found that the Optimized-CodeBERT model outperformed Slither, Optimized-LSTM, and Optimized-CNN in detecting three types of vulnerabilities, but was inferior in one type. This is because different models have different structures, parameters, and learning algorithms, which affect their modeling and generalization abilities. Therefore, in future work, we aim to improve the performance of the three models in LightingCat and extend the application of our proposed Lighting Cat to more code security fields beyond smart contract vulnerabilities detection.

**References**

4. The dao smart contract. https://etherscan.io/address/0xbb9bc244d798123fde783fcec1c72d3bb8c189413 (2016).


Supplementary Information

We provide four examples of representative smart contract vulnerabilities, namely Reentrancy Vulnerabilities, Incorrect tx.origin Authorization, Timestamp Dependency, and Unhandled Exceptions.
(1) Reentrancy Vulnerabilities

Reentrancy vulnerability has always been the most harmful and frequently occurring type of vulnerability. It is related to external call and callback mechanism. The key is that when the victim contract calls the malicious contract fallback function for transfer, it will call back the victim contract itself, resulting in recursive calls, causing a loop transfer and illegal outflow of funds. Although the frequency of basic reentrancy vulnerabilities in smart contract code has decreased due to the increased security awareness of developers, they still cannot escape from different types of variant reentrancy attacks, such as Token-callback Reentrancy, Cross-function Reentrancy, Basic Reentrancy with a Twist, Cross-contract Reentrancy, and Read Only Reentrancy. Taking Read Only Reentrancy as an example, we will explain how the attack occurs.

The location where the Read Only Reentrancy vulnerability occurs is the function in the smart contract that is marked as “view”. This type of function does not change the state variables in the contract and is generally not decorated with mutex locks. When the victim contract calls the function marked as “view” in the vulnerable contract, it may cause abnormal conditions due to obtaining the state that has not been updated in time, affecting subsequent operations.

```solidity
1 function addLiquidity(uint256 eth_amount) external payable nonReentrant() returns (uint256)
2 {
3     //Some necessary checks
4     uint256 eth_reserve = address(this).balance - eth_amount;
5     uint256 totalLiquidity = lpToken.totalSupply();
6     uint256 lp_amount = (eth_amount * totalLiquidity) / eth_reserve;
7     require(lp_amount > 0, "No Liquidity Shares Minted");
8     lpToken.mint(msg.sender, lp_amount);
9     return eth_amount;
10 }
11
12 function removeLiquidity(uint256 lp_amount) external nonReentrant() returns (uint256)
13 {
14     //Some necessary checks
15     uint256 eth = address(this).balance;
16     uint256 totalLiquidity = lpToken.totalSupply();
17     uint256 eth_amount = (lp_amount * eth) / totalLiquidity;
18     (bool success, ) = msg.sender.call(value: eth_amount(""));
19     require(success, "Transfer failed");
20     lpToken.burn(msg.sender, lp_amount);
21     return eth_amount;
22 }
23
24 function getSpotPriceEth(uint256 amount) public view returns (uint256)
25 {
26     return amount * address(this).balance / lpToken.totalSupply();
27 }
```

Supplementary Figure 1. Example of Read Only Reentrancy

As shown in the Supplement Fig 1, users can provide liquidity assets (using ETH as an example) to the pool to obtain the corresponding LP tokens, or remove their liquidity assets from the pool and get back the corresponding ETH. The calculation method for the LP tokens and ETH exchange ratio can be found in line 8 and line 21. There are three functions, addLiquidity, removeLiquidity, and getSpotPriceEth. The addLiquidity function exchanges ETH for LP tokens, and the removeLiquidity function exchanges LP tokens for ETH. Both of these functions have external visibility and inherit the nonreentrant function modifier from the OpenZeppelin official library. The getSpotPriceEth function is used to calculate the virtual value of LP tokens, i.e., the ETH share corresponding to a certain amount of LP tokens. This function is declared as a view type and does not require the use of a mutex lock.

The attack point of the Read Only Reentrancy vulnerability is in line 22 of the removeLiquidity function and the getSpotPriceEth function. When a user removes their liquidity assets, the function calls the call function to send the calculated amount of ETH to the msg.sender address. If the address corresponds to a contract address, its fallback function will be called to transfer the funds. At this time, the LP tokens have not been burned yet (line 24), the total liquidity of LP tokens in the pool has not changed, and the total ETH balance of the contract has decreased. The getSpotPriceEth function, which uses these two variables, will be affected. If the getSpotPriceEth function is called during a reentry attack in the fallback function of the msg.sender contract, the
final result obtained will be smaller than the normal value, which means that the price of LP tokens has decreased.

There have been multiple cases of attacks on the Read Only Reentrancy vulnerability. For example, on February 9th, 2023, a hacker attacked the DeFi protocol dForcenet and made a profit of 1236 ETH and 710,000 USX tokens. On April 4th, 2023, the lending project Sentiment on Arbitrum was attacked, resulting in a loss of $1 million.

(2) Incorrect tx.origin Authorization

The difference between tx.origin and msg.sender is often overlooked in smart contract development. “tx.origin” is a global variable in the smart contract that traverses the entire call stack and returns the address that initially sent the call, which must be an external account address rather than a contract address. “msg.sender” returns the direct caller of the function, which can be either an external account address or a contract address. Using tx.origin for identity verification in a smart contract can make the contract vulnerable to phishing attacks. When a contract uses tx.origin as a condition for transfer, a malicious attacker can steal ether or tokens from the contract by constructing a designed call chain.

Supplementary Figure 2. Example of Incorrect tx.origin Authorization

As shown in the Supplement Fig 2, there are two contracts, VulnerableContract and AttackContract. The withdrawAll function in VulnerableContract is used to withdraw all balances in the contract, and owner check is required before withdrawal (line 7). Assuming that the owner of the VulnerableContract contract calls the withdrawAll function to transfer funds to the AttackContract address (line 8), its fallback function will be called (lines 19-21). The fallback function is injected with malicious code, which calls the withdrawAll function of VulnerableContract again. In the withdrawAll function, the tx.origin obtained is the external account address that deployed VulnerableContract, so the check passes (line 7), and all balances in VulnerableContract are transferred to AttackContract. If authorization is checked using msg.sender in line 7, it will get the AttackContract contract address instead of the external account address that deployed VulnerableContract. In this case, the check does not pass, and the transfer to AttackContract will not be executed.

(3) Timestamp Dependency

Unlike traditional solutions, the execution environment of smart contracts is on the miner’s terminal. Smart contract code that includes weak randomness based on chain properties can be exploited by malicious miners. In Ethereum, timestamp is a common chain property. The timestamp of a block is the local system time of the miner who mined the block, but Ethereum allows miners to modify the timestamp of a block within a certain range (30 seconds after block validation). When certain logic in a contract depends on the current time, a miner can control the execution result by manipulating the current time, achieving a certain expectation or even gaining illegal benefits. Therefore, chain property values obtained based on “block.timestamp” have weak randomness. In addition, other fields such as “block.difficulty”, “block.coinbase”, and “block.number” are also insecure.

As shown in the Supplement Fig 3, it is a lottery contract. A “lucky number” is calculated based on the current block’s timestamp and other variables that can be known in advance. The participant with the same encoded “lucky number” wins the prize. Miners can try different variables (such as “block.timestamp”, “block.number”, and “blockhash()”) during mining to calculate this “lucky number” in advance, thereby controlling who can become the winner.
(4) Unhandled Exceptions
When calling low-level but important functions such as “send”, “call”, and “delegatecall”, if an exception occurs during the call, the exception will not be propagated (reasons for exceptions in low-level calls include actively calling “revert()”, insufficient gas, and call stack overflow), and only “true” or “false” will be returned, while continuing to execute the next contract instruction. If the contract code does not handle exceptions or return values for these three functions, it may cause logical errors in the code.

```
contract lottery {
  uint256 private Last_Payout = 0;
  uint256 weakRandVal1 = block.timestamp;
  uint256 weakRandVal2 = block.number;
  function random() public view returns (uint256 result) {
    uint256 y = weakRandVal1 * weakRandVal2 / (weakRandVal1 % 5);
    uint256 seed = weakRandVal2 / 3 + (weakRandVal1 % 300) + Last_Payout + y;
    uint256 h = uint256(blockhash(seed));
    return uint256(h % 100) + 1;
  }
}
```

Supplementary Figure 3. Example of Timestamp Dependency

As shown in the Supplement Fig 4, the “withdraw” function is used for withdrawal operations. The code in line 3 calls the “send” function for transferring funds, but does not check the return value of the transfer. If the transfer fails, the code will still continue to execute, as shown in line 4 where the amount is deducted. This leads to the abnormal situation where the transfer fails but the balance is still deducted.

Supplementary Figure 4. Example of Unhandled Exceptions

Author information
Salus Security, Beijing Chaoyang District Chuangfu Port (Yuandadu Branch) 3002, 100020, Beijing, China
Xueyan Tang, Yuying Du, Alan Lai, Ze Zhang, and Lingzhi Shi

Corresponding author
Correspondence to Xueyan Tang.

Author contributions statement
X.T. conceived the design and proposed questions. Y.D. was responsible for model design and overall planning. A.L. was responsible for overall planning and revised the manuscript. Z.Z. conducted model reproduction and optimization. L.S. wrote the introduction and conducted literature review. All authors reviewed the manuscript.

Data availability
The datasets generated and/or analyzed during the current study are available from the corresponding author upon reasonable request.

Competing interests
The authors declare no competing interests.