The Journal of Systems & Software 202 (2023) 111699

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding[☆]

Dawei Yuan, Xiaohui Wang, Yao Li, Tao Zhang [*]

School of Computer Science and Engineering, Macau University of Science and Technology, Macao, China

ARTICLE INFO

Article history:
Received 2 September 2022
Received in revised form 20 January 2023
Accepted 5 April 2023
Available online 7 April 2023

Keywords:
Smart contract
Bug injection
Transfer learning
Vulnerability detection

ABSTRACT

Smart contracts have been widely used in the blockchain world these years, and simultaneously vulnerability detection has gained more and more attention due to the staggering economic losses caused by the attacker. Existing tools that analyze vulnerabilities for smart contracts heavily rely on rules predefined by experts, which are labour-intense and require domain knowledge. Moreover, predefined rules tend to be misconceptions and increase the risk of crafty potential back-doors in the future. Recently, researchers mainly used static and dynamic execution analysis to detect the vulnerabilities of smart contracts and have achieved acceptable results. However, the dynamic method cannot cover all the program inputs and execution paths, which leads to some vulnerabilities that are hard to detect. The static analysis method commonly includes symbolic execution and theorem proving, which requires using constraints to detect vulnerability. These shortcomings show that traditional methods are challenging to apply and expand on a large scale. This paper aims to detect vulnerabilities via the Bug Injection framework and transfer learning techniques. First, we train a Transformer encoder using multi-modality code, which contains source code, intermediate representation, and assembly code. The input code consists separately of Solidity source code, intermediate representation, and assembly code. Specifically, we translate source code into the intermediate representation and decompile the byte code into assembly code by the EVM compiler. Then, we propose a novel entropy embedding technique, which combines token embedding, segment embedding, and positional embedding of the Transformer encoder in our approach. After that, we utilize the Bug Injection framework to automatically generate specific types of buggy code for fine-tuning and evaluating the performance of vulnerability detection. The experimental results show that our proposed approach improves the performance in detecting reentrancy vulnerabilities and timestamp dependence. Moreover, our approach is more flexible and scalable than static and dynamic analysis approaches in detecting smart contract vulnerabilities. Our approach improves the baseline approaches by an average of 11.89% in term of F1 score.

© 2023 Elsevier Inc. All rights reserved.

1. Introduction

Blockchain is essentially a distributed ledger that shares transactions, maintained by the miners following a consensus protocol in the distributed network. When someone invokes functions of a smart contract, a transaction is submitted to the transaction pool and waits to be packaged by a miner. The consensus protocol enforces the ledger to record all transactions only once, and the ledger is immutable to be modified on the blockchain once recorded.

Smart contracts are programs that run on the blockchain. Many decentralized applications (DAPPs)[1] rely on smart contracts deployed on the Ethereum blockchain for backend functionality. Ethereum is one of the most popular blockchain platforms on which many smart contracts are deployed. Implementing smart contracts makes transferring cryptocurrency assets and depositing them into specific accounts easy and automatic. For example, users may transfer cryptocurrency assets, such as ETH (Ethereum coin) and BNB (Binance coin), from one account to another by simply invoking the inline `transfer` function rather than doing some extra validation. In addition, anyone can transfer cryptocurrency assets from the user authorized to others by calling the `transferFrom` function. By the end of June 2022, the block height of Ethereum has reached 15040265 block, which

☆ Editor: Laurence Duchien.
* Corresponding author.
E-mail addresses: 2009853SIA30001@student.must.edu.mo (D. Yuan), 2109853wim20002@student.must.edu.mo (X. Wang), 2109853gia30001@student.must.edu.mo (Y. Li), tazhang@must.edu.mo (T. Zhang).

1 https://ethereum.org/en/dapps/.

https://doi.org/10.1016/j.jss.2023.111699
0164-1212/© 2023 Elsevier Inc. All rights reserved.

means that a significant number of transactions have immutably been recorded on Ethereum. These enormous number of transactions mainly result from the prosperity of DAPP markets such as Decentralized finance (Defi)[2] and Decentralized game (GameFi).[3]

Smart contract vulnerabilities arise from poorly audited smart contracts. For example, a malicious smart contract address indicates that the smart contract has already been deployed on Ethereum and has a vulnerability, which is utilized by someone to perform a front-run attack. Its Ethereum coin has been transferred away by the attacker. Recently, many smart contracts have been deployed and run on Ethereum, with a transaction volume of `164M`, and Ethereum Market Cap keeps the current level of `187.67B`. The considerable profit attracts many malicious smart contracts deployed on Ethereum, trying to steal the cryptocurrency from them. Smart contract vulnerabilities have several characteristics that differ from vulnerabilities of traditional programming languages. The Solidity programming language is not popular enough and widely used. Moreover, the commonly used third-party libraries, such as OpenZeppelin,[4] are relatively simple, leading to most novice developers writing source code prone to vulnerabilities. Besides, the smart contract is not allowed to be modified once it is deployed on the blockchain, which prevents developers from patching the bugs and updating the smart contract when errors occur, resulting in a huge loss of cryptocurrency assets. Therefore, it is necessary to perform effective vulnerability detection on smart contracts before deployment.

Limitations of common methods. With the increase in the number of smart contracts, developers and auditors need to make more efforts to be concerned about smart contract vulnerability detection. The traditional way to detect vulnerability by manually defined rules is no longer applicable. The traditional static and dynamic analysis methods need to redefine rules to fit the requirement, which results in the upgrades of detection methods being hard to keep up with the speed of vulnerability upgrades. Moreover, traditional detection methods may fail to detect potential vulnerabilities if there exist unsuitable upgrades. Some methods (Yu et al., 2021; Mi et al., 2021) adopt deep learning techniques to detect smart contract vulnerabilities, such as utilizing the LSTM-based neural networks (Essaid et al., 2019) to train and interfere with the byte code. However, these deep learning methods only focus on uni-modality data without additional information, which fail to capture insufficient semantic features of Solidity source code and impact detection accuracy. For example, Mythril (Parizi et al., 2018) uses symbolic and taint analysis to detect possible vulnerabilities in smart contracts. Typical symbolic execution methods use consistent rules to transform all possible execution paths into a constraint programming solver problem, such as applying SMT (Luo et al., 2001) to solve spatial analysis ahead of time. Securify (Tsankov et al., 2018) proposes a way to convert source code into a domain-specific (DSL) domain language and utilize a pattern-matching algorithm to find vulnerabilities in Solidity source code. However, these methods are hard to deploy and only work on particular platforms. In addition, these methods are unsuitable for Solidity source code since the deployment of the Ethereum virtual machine (EVM)[5] platform is relatively elaborate, which is problematic for traditional static and dynamic analysis. For example, developers need to build a private blockchain or fork from the public blockchain by using the Geth toolkit,[6] which is hard to scalable. To solve this problem, it is better to abandon the manual construction of rules and apply the transfer learning technique to automatically learn the features from the multi-modality of Solidity source code, which is scalable on different platforms.

Why do we utilize multi-modality code and entropy embedding? Traditional deep learning methods usually provide uni-modality data into the model for training, such as source code or natural language text. For example, users tend to separately transmit uni-modality data into encoders and decoders with the Neural machine translation (NMT) deep learning model in the software engineerings tasks, such as code comments generation and code search. In contrast, using multi-modality code snippets of smart contracts, combining and transmitting them into the Transformer encoder for training may effectively improve the ability to capture the inherent features of the source code. When multi-modality data is transmitted into the Transformer encoder, the usual practice is evenly to distribute the weight of multi-modality data. However, the source code, intermediate representation, and assembly code contribute unequally to the transformer encoder, where a new algorithm or strategy is needed to assign different weights to multi-modality data. Since entropy theory quantifies the information of code or text, the resulting entropy value is calculated for multi-modality code. Therefore, a new entropy embedding algorithm combined with the Transformer encoder's segment embedding may improve the performance of multi-modality code embedding.

To fill the gap for these shortages, we scrape and collect the public smart contracts datasets on Ethereum and propose an intuitive and scalable approach based on the transfer learning technique for detecting vulnerabilities of smart contracts. In summary, the key contributions of this paper are as follows:

- We translate Solidity source code into an intermediate representation and assembly code, respectively, and regard them as a multi-modality sequence to train the Transformer encoder in the pre-training phase.
- To the best of our knowledge, we are the *first* to propose entropy embedding technology compatible with the Transformer encode, which assigns weights to different modalities input, to optimize the representation capability of the encoder.
- We utilize the Bug Injection framework (Fu et al., 2007) to automatically yield various typical buggy code snippets to fine-tune the classifier with the Transformer encoder. Moreover, our implementation[7] is released to help researchers reproduce the current work.

The remaining sections of this paper are organized as follows: Section 2 introduces the background. Section 3 describes a motivating example to explain the reason for introducing the intermediate code-based graph. Section 4 introduces the details of our proposed approach. Section 5 performs the experiments and analyzes the results. In Section 6, we discuss the results of performance evaluation and the limitations of the proposed approach. In Section 7, we introduce some related works. In Section 8, we conclude the paper and introduce future work.

2. Background

2.1. Smart contract vulnerability detection tools

Ethereum is a popular open-source platform that supports a Turing-complete virtual machine based on blockchain technology. On Ethereum, anyone can write a smart contract program and run it on the Ethereum virtual machine. After that, users send a call request to invoke functions of the smart contract, and a

2 https://uniswap.org/.
3 https://gamefi.org/.
4 https://www.openzeppelin.com/.
5 https://ethereum.org/en/developers/docs/evm/.
6 https://geth.ethereum.org/.
7 https://github.com/davidyuan666/SmartContractDetection.

transaction will be submitted in the transaction pool at this time. Then, the miner chooses the transaction that pays a higher fee and stores the selected transaction permanently in a new block. From the perspective of blockchain security, smart contract vulnerability detection plays a vital role in the blockchain ecosystem. Most of the current methods employ formal verification to detect vulnerabilities. For example, Bhargavan et al. (2016) translate the solidity code or byte code into a verification system to verify. The literature (Hirai, 2017) utilizes the Isabelle Framework (Wenzel et al., 2008) tool to suspect potential vulnerabilities. Although these methods provide robust validation, they are still not sufficiently automated to detect vulnerabilities. Another way to detect vulnerabilities relies on test cases and symbolic execution, such as Oyente (Luu et al., 2016), Maian (Nikolić et al., 2018), and Securify (Tsankov et al., 2018). These tools detect vulnerabilities by manually constructing various inputs to compare the actual and expected output. Unfortunately, these methods fail to cover all test cases to inspect the possible vulnerabilities. Besides, some methods use dynamic analysis to detect vulnerabilities. Jiang et al. (2018) used contractFuzzer to monitor the program running status and aim to identify vulnerabilities. Reguard (Liu et al., 2018) is another fuzzing-based analyzer that seeks to identify vulnerabilities. Moreover, the Dytan (Clause et al., 2007) method uses taint analysis to analyze runtime data flow during program execution. Through the above description, dynamic analysis methods examine test cases' output based on interacting with programs. Unfortunately, these methods based on dynamic analysis still need to construct test cases in advance manually. Recently, some methods utilized deep learning techniques to detect smart contract vulnerabilities. For example, Qian et al. (2020) apply Long short-term memory (LSTM) to detect reentrancy type vulnerability, but the accuracy of detection vulnerability results is not satisfactory. The main reason is that LSTM is not suitable for capturing long-distance information.

2.2. Typical smart contract vulnerabilities

In this section, we define contract vulnerabilities, introduce the three types of vulnerabilities, and explain why we focus on these vulnerabilities.

Vulnerability Definition. Given a smart contract's solidity source code, developers try to determine whether the smart contract has vulnerability and point out the potential type of vulnerability. In other words, we estimate an identifier y for each function f in smart contract, where y=n represents that y belongs to the vulnerable type n. For example, y = 0 denotes that the function f has the *re − entrancy* vulnerability, which denotes *y* has a 0(vulnerability) type.

Reentrancy vulnerability refers to an error when the thread of execution re-enters a function before it has not yet finished. A program or subroutine can be regarded as reentrancy if it can be interrupted at any time. However, the reentrancy vulnerability that happens in EVM is different from traditional program scheduling. The reentrancy vulnerability mainly describes the invoke cycle between the caller and the callee, which leads to the transfer of the cryptocurrency assets to the caller. Reentrancy vulnerabilities are prone to occur and are mainly attributed to the following situations:

- Caller information leaks, i.e., the callee has an access to the caller's internal state and functions.
- Share the same context, i.e., the caller and the callee who do not trust each other share the same thread.
- Exception suppression, when a function invokes through a low-level operation such as call opcode, the callee ignores the exception and does not pass it to the caller.

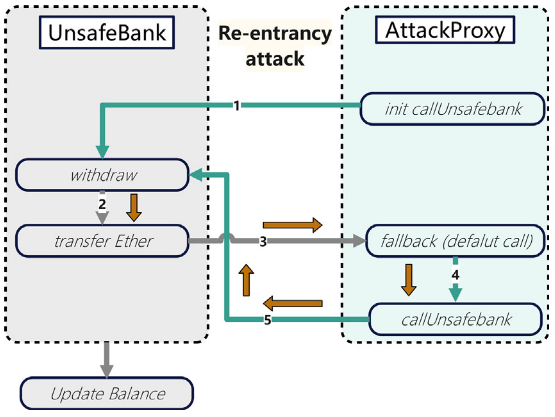


Fig. 1. Reentrancy vulnerability explanation.

Fig. 1 shows the details of the reentrancy attack. First, the `Attackproxy` contract invokes the `withdraw` function in the `Unsafebank` contract that aims to retrieve cryptocurrency assets like Ethereum coin.[8] Meanwhile, the `Unsafebank` contract is prepared to transfer Ethereum coin to the `Attackproxy` contract through the `withdraw` function. The `Attackproxy` contract deliberately ignores implementing the procedure of receiving Ethereum coin, which leads to the anonymous fallback function in the `Attackproxy` contract receiving Ethereum coin by default. The `Attackproxy` contract is eager to receive additional cryptocurrency assets. Hence, the `Attackproxy` contract overwrites the anonymous fallback function, which leads to a circular transfer function, so that all the Ethereum coins in the `Unsafebank` contract are transferred to the `Attackproxy` contract. Fig. 2 shows the source code of reentrancy, the `Unsafebank` contract transfers cryptocurrency assets through the `withdraw` function. When the `Attackproxy` contract prepares to retrieve Ethereum coin from the `Unsafebank` contract, the `Unsafebank` contract executes source code like `msg.sender.callvalue:` in `withdraw` function. Then, the `Unsafebank` contract is trapped in the `Attackproxy` contract and calls anonymous `fallback` functions in the `Attackproxy` contract. At this point, the `Attackproxy` contract deliberately implements logic like stealing Ethereum coin from the `Unsafebank` contract in the anonymous `fallback` function. Therefore, unless the balance of the `Unsafebank` contract is empty, like (`balances[msg.sender] = 0`), the `Attackproxy` contract can continuously obtain Ethereum coin from the `Unsafebank` contract.

Timestamp dependence vulnerability happens when smart contracts utilize block timestamps as a condition to perform a critical operation (such as sending an Ethereum coin) or as a seed to generate random numbers in an auction game. During package transactions in the transaction pool, miners can easily modify the timestamp of blocks in short intervals. For example, a smart contract transfers Ethereum coin according to a specific timestamp, such as regularly sending ether or generating random numbers. Right now, miners can exploit this vulnerability based on manipulating block timestamps. Fig. 3 shows the description of how `Timestamp dependency` vulnerability works. Smart contracts execute twice and submit two transactions into the transaction pool. Right now, the transaction pool contains `Transaction i` and `Transaction j`. Both transactions are waiting to be minted by a miner in the transaction pool. The mint order of two transactions into blocks determines the smart contract state. The reason is that the state change of the smart contract depends on the

[8] https://coinmarketcap.com/currencies/ethereum/.

```
contract UnsafeBank{
    function withdraw() public
    {
        bool result = msg.sender
                  .call.value(balance[msg.sender])();
        if(!result)
        {
            throw;
        }
        balances[msg.sender] = 0;
    }
}
contract AttackProxy{
    function callUnsafebank(address _address) public
    {
        UnsafeBank(_address).withdraw();
    }
    function  () public payable
    {
        if (address(this).balance < 10000 ether)
        {
            callUnsafebank(msg.sender);
        }
    }
}
```

Fig. 2. Reentrancy vulnerability source code example.

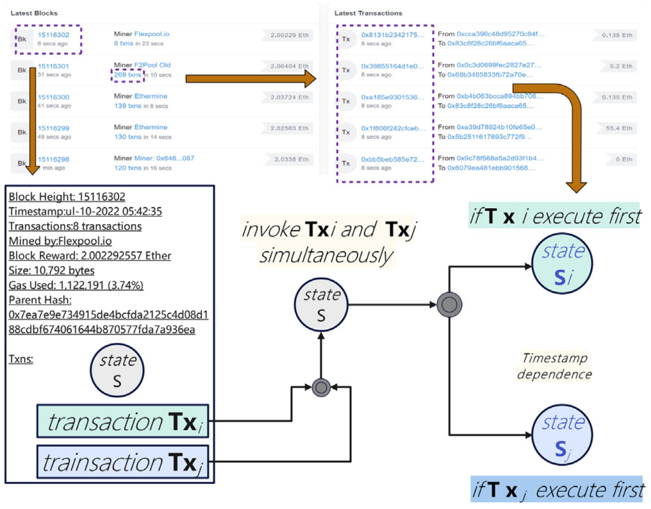


Fig. 3. Timestamp dependence vulnerability explanation.

```
contract TimestampeRace{
    uint private Last_payout = 0;
    uint256 salt = block.timestamp;
    function random_generate()
            returns (uint256 result) {
        uint256 y = salt * block.number/(salt%5);
        uint256 seed = block.number/3 + (salt%300)
                + Last_payout + y;
        uint256 h = uint256(block.blockhash(seed));
        return uint256(h%100) + 1;
    }
}
```

Fig. 4. Timestamp dependency vulnerability source code example.

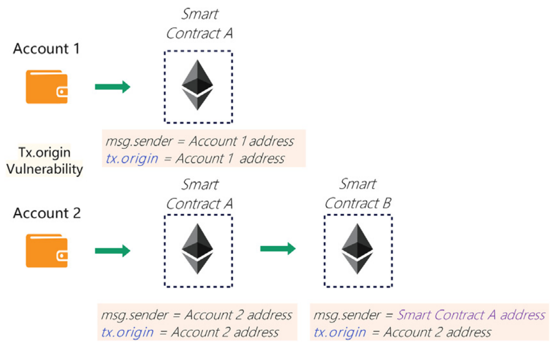

Account 1
Smart Contract A
msg.sender = Account 1 address
tx.origin = Account 1 address
Tx.origin Vulnerability
Account 2
Smart Contract A
Smart Contract B
msg.sender = Account 2 address
tx.origin = Account 2 address
msg.sender = Smart Contract A address
tx.origin = Account 2 address

Fig. 5. Tx.origin vulnerability explanation.

```
contract Victim {
    function () public payable {}
    function withdrawAll(address _recipient) public {
        require(tx.origin == owner);
        _recipient.transfer(this.balance);
    }
}
contract Exp {
    function attack() internal{
        address ph0wner = phInstance.owner();
        if (ph0wner == msg.sender){
            phInstance.withdrawAll(owner);
        }else{
            owner.transfer(address(this).balance);
        }
    }
}
```

Fig. 6. Tx origin vulnerability source code example.

timestamp. Interestingly, miners sort transactions based on the transaction's gas fee, so miners prefer mint transactions with a higher gas fee. Finally, The mint order of `Transaction i` and `Transaction j` tend to affect the smart contract state in this situation. Fig. 4 shows how the vulnerability occurs in the source code. The `random_generate` function generates random numbers in the `TimestampeRace` contract, where the seed decides the final result by the `block.hash` function. Therefore, miners can deliberately determine the seed value by manipulating the timestamp.

Tx Origin vulnerability is a transaction origin attack that is a form of phishing attack that can drain a contract of all funds. Moreover, `tx.origin` and `msg.sender` are global variables in the Solidity programming language. The `tx.origin` returns the address of the account that initially sent the transaction, and `msg.sender` represents the last sender. Fig. 5 shows how `origin.sender` exposes potential vulnerability. When `Account_1` calls contract A, the `tx.origin` address is the same as `msg.sender`. For comparison, when `Account_2` indirectly calls contract B through contract A. The `tx.origin` address in contract B is still the address of the initial `Account_2` address, but the `msg.sender` address becomes the contract A address. Therefore, there is an opportunity for `Account_2` to invoke smart contract B by accessing smart contract A, which performs equivalent to bypassing the authentication. Fig. 6 shows the details about how the source code works. `Exp` contract invokes the `attack()` function to call the `withdrawAll` function in the `Victim` contract. In the `withdrawAll` function, the `Victom` contract needs to check if the sender equals `tx.orgin`, which depicts a fact that only the initial user can access successfully. The `require` statement prevents the `Victim` contract from stealing Ethereum coins.

Why do we care about these vulnerabilities?. We mainly focus on these typical vulnerabilities because many attacks have taken advantage of these vulnerabilities to achieve huge profits that profoundly impact real life. For example, the DAO (Mehar et al., 2019) attack suffered a considerable loss of assets because of the Re-Entrancy vulnerability in the DAO smart contract. Due to this vulnerability, the DAO community has to fork a new ETC chain. Recently, many malicious smart contracts work on the Ethereum-compatible blockchain due to the popularity of the decentralized application (DAPP) market, which leads to the potential loss of funds. It is necessary to pay attention to these vulnerabilities and avoid fund loss by detecting them.

2.3. Assembly code of smart contract

Multi-modality code input relies on the assembly code, and intermediate representation converted from smart contracts. First, there are some significant differences between smart contracts and traditional x86 programming languages. The instruction addresses of smart contracts are all stored in the stack memory area because the Ethereum virtual machine is a stack-based Turing machine. Secondly, the smart contract applies `JUMP` instructions instead of the traditional `CALL` instructions. Each smart contract yields records according to the `JUMP` instructions in the smart contract. The records illustrate the destination addresses of the `JUMP` instruction. Meanwhile, the Ethereum virtual machine takes arguments from the stack, performs instructions, and puts the consequent value on the stack again. Specifically, the `PUSH` instruction pushes parameters directly onto the stack; `PUSH1` and `PUSH2` depict the Ethereum virtual machine that can push 1 or 2 bytes onto the stack. Moreover, the instructions between different contracts are described as `CREATE`, `CALL`, `CALLCODE`, and `SELFDESTRUCT` respectively. For example, the byte code `6005600301` can be parsed as `PUSH1 0x05 PUSH1 0x03 ADD`. That is to say, Ethereum virtual machine first pushes `0x05` onto the stack. Then, Ethereum virtual machine pushes an additional `0x03` onto the stack and adds the stack value. Later, the value `0x08` is on the top of the stack. In addition, the function records describe the boundary addresses of the instructions. The instructions are divided into multiple basic blocks in sequence according to boundary addresses. The basic blocks are regarded as multi-modality inputs.

2.4. Transformer encoder

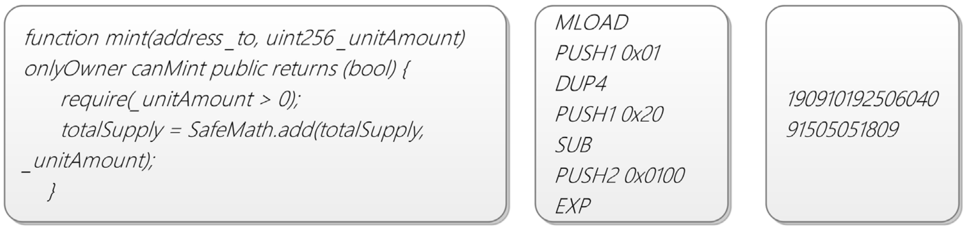
The Transformer Encoder is designed to learn the contextual information of the target word, which can effectively capture features. Traditionally, we train a language model to predict the next word in a sentence, such as the right-to-left context used in Generative Pre-trained Transformer (GPT Ethayarajh, 2019), or a left-to-right context used in LSTM (Yu et al., 2019). But these two architectures are prone to lose partial information due to unidirectionality. In contrast, the Transformer encoder adopts bidirectional self-attention network architecture and utilizes masked language (MLM Ghazvininejad et al., 2019) and next sentence prediction (NSP Li et al., 2021) strategy to train the encoder model aiming to make up for the above drawbacks. The model with the Transformer encoder predicts the missing words from the sequence itself during the pre-training phase. The model replaces each missing word as a `[MASK]` identifier and regards them as mask words. Next, The developer trains the model with the Transformer encoder to predict the mask word `[MASK]`, which aims to learn the features between words via MLM. In addition, developers tend to train the Transformer encoder via NSP, which seeks to capture the relationship between sentences. To prevent the model from focusing on a specific location or masked tokens, by default, 15% of the words are randomly masked by default. Meanwhile, in these 15% masked words, 80% of the words are replaced with `[MASK]` tokens, and 10% of the words are randomly replaced. Other words remain unchanged. For the NSP, 50% of the subsequent sentence is the next sentence of the first sentence, and the remaining 50% of the second sentence is a random sentence in the corpus. From a mathematical point of view, we input a sequence $\left[X = x_1, x_2, x_3 ..., x_n\right]$ into the model, and its output is $\left[Y = y_1, y_2, y_3 ..., y_m\right]$. The purpose of training the model is to translate `X` into `Y`, which aims to maximize the conditional probability `P(Y|X)` by continuously optimizing the parameter θ. Recently, transfer learning has been widely used in software engineering. Many types of research applying transfer learning in software engineering aim to learn code and natural language text features and utilize them in different studies. To achieve better performance, it is typical to train models with many source code snippets and natural language text in advance. Then, the learned pre-training models can be fine-tuned by specific tasks later. For example, CuBERT (Kanade et al., 2020), CodeBert (Feng et al., 2020), and GraphCodeBert (Guo et al., 2020) are based on the transfer learning used for different tasks, such as code comment generation, code search, and other software tasks.

3. Motivating example

In this section, we present an example to illustrate the motivation that uses multi-modality code and entropy embedding to improve the performance of detecting vulnerabilities of the smart contract. Note that this example only aims to exhibit our proposed approach. Using multi-modal codes for deep neural network training can improve the model's generalization ability, enabling it to learn more features. This is because using multiple types of data and features can provide more comprehensive information, which helps the model learn a broader range of regularities and relationships. For example, when training a code function classification model, using various code datasets, including code snippets and different code languages, can help the model learn a broader range of features and rules, thereby, it can improve the classification performance and the model's accuracy in detecting smart contract code vulnerabilities. Multi-modal code representation refers to using many different coding representations and techniques in the code to analyze the code's complexity. Using entropy embedding in codes refers to encoding information for different code types to assign appropriate weights.

Fig. 7 shows the example of multi-modality code. This smart contract example is a simple owner-transfer contract that allows the creator to transfer ownership. However, this code has a critical vulnerability. It has a "mint" function to directly change the owner without the owner's permission. This means that anyone can directly take over the contract, which is unsafe. This type of vulnerability is very common and serious, such as losing funds in the smart contract.

Additionally, Fig. 8 shows the example of the embedding space of multi-code smart contracts. We can see from the vector space in the above figure that if only a single data source is transmitted into the deep neural learning model, such as source code, then there is only 64 dimension in the vector space. In this situation, if supposing that we add multi-code input, such as assembly code and intermediate representation of smart contract. In that case, we then concatenate these code snippets and transmit them into models that aim to expand the small dimensional space to a larger dimension space so that it contains more features, and vector points with similar functions can be found in the vector space. If there are only small dimensions used to train deep learning neural models, it is difficult to distinguish them in large data sets and also difficult to distinguish the features of the multi-modality.

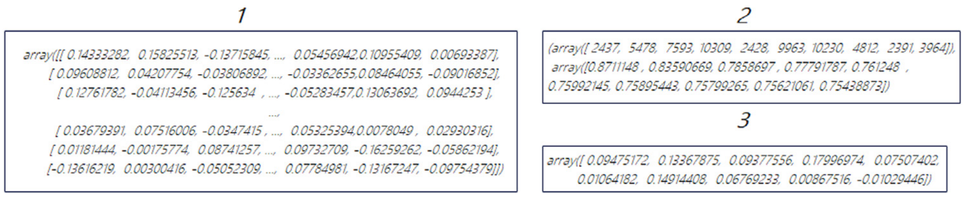


```
function mint(address _to, uint256 _unitAmount)
onlyOwner canMint public returns (bool) {
      require(_unitAmount > 0);
      totalSupply = SafeMath.add(totalSupply,
_unitAmount);
   }
```

```
MLOAD
PUSH1 0x01
DUP4
PUSH1 0x20
SUB
PUSH2 0x0100
EXP
```

```
190910192506040
91505051809
```

Fig. 7. An example of multi-modal smart contract.



1

```
array([[ 0.14333282,  0.15825513, -0.13715845, ...,  0.05456942,0.10955409,  0.00693387],
       [ 0.09608812,  0.04207754, -0.03806892, ..., -0.03362655,0.08464055, -0.09016852],
       [ 0.12761782, -0.04113456, -0.125634  , ..., -0.05283457,0.13063692,  0.0944253 ],
       ...,
       [ 0.03679391,  0.07516006, -0.0347415 , ...,  0.05325394,0.0078049 ,  0.02930316],
       [ 0.01181444, -0.00175774,  0.08741257, ...,  0.09732709, -0.16259262, -0.05862194],
       [-0.13616219,  0.00300416, -0.05052309, ...,  0.07784981, -0.13167247, -0.09754379]])
```

2

```
(array([ 2437,  5478,  7593, 10309,  2428,  9963, 10230,  4812,  2391, 3964]),
 array([0.8711148 , 0.83590669, 0.7858697 , 0.77791787, 0.761248  ,
0.75992145, 0.75895443, 0.75799265, 0.75621061, 0.75438873]))
```

3

```
array([ 0.09475172,  0.13367875,  0.09377556,  0.17996974,  0.07507402,
        0.01064182,  0.14914408,  0.06769233,  0.00867516, -0.01029446])
```

Fig. 8. An example of embedding value of the multi-modal smart contract.

4. Proposed approach

Our approach pre-trains the Transformer encoder using source code, intermediate representation, and assembly code. Before pre-training, we first translate the source and byte codes into an intermediate representation and assembly code, respectively. Then we concatenate these multi-modality code snippets into sequences and transmit them into the Transformer encoder for learning code features. Next, we fine-tune and evaluate our approach by applying the Bug Injection framework and utilizing a classifier to detect smart contract vulnerabilities. Fig. 9 shows an overview of the working procedure of our approach. The approach contains a Transformer encoder, which is a multi-layer encoder-based model. Besides, a Bug Injection framework and Softmax classifier aim to fine-tune the model and detect new smart contract vulnerabilities. Our approach works on three different modalities of code information: (1) source code of the smart contract, (2) intermediate representation translated from source code, (3) assembly code converted from the byte code. To better extract the features of the smart contract for our approach, we try to extract more information from the smart contract. In detail, we consider source code, intermediate representation, and assembly code as multi-modalities code snippets and transmit them into the Transformer encoder for pre-training. In general, our approach mainly consists of three steps. First, the pre-processing step combines and tokenizes these input code snippets. Then the Transformer encoder takes these code snippets as multi-modality code sequences and transmits them into the model for pre-training. Finally, we fine-tune and evaluate our approach through the Bug Injection framework and Softmax classifier.

4.1. Approach overview

We show the overall framework of the proposed approach in Fig. 9. The approach is composed of the following steps:

- **Data Collection**: We use a crawler bot developed by ourselves to retrieve Solidity source code and byte code by parsing the transactions on the Ethereum blockchain, then translating them into intermediate representation and assembly code, respectively.
- **Input Code Sequence**: The source code, intermediate representation, and assembly code are treated as multi-modality code sequences. Then, we calculate the entropy value of the multi-modality code sequence and use a technique we proposed called code embedding to be compatible with the default embedding of the Transformer encoder for training, representing the weight distribution of the multi-modality code sequence.
- **Pre-processing**: We apply the tokenizer technique to tokenize multi-modality code sequences into code tokens and distinguish them through a `<seq>` identifier.
- **Transformer Encoder**: The Transformer encoder treats the multi-modality data as the input data and is trained to learn the features of the smart contract.
- **Bug Injection**: Bug Injection frameworks generate different types of vulnerable smart contract code snippets and connect with the original code. Then, we fine-tune and evaluate our approach based on these buggy code snippets.
- **Vulnerabilities Detection**: The `softmax` classifier determines the vulnerability type of the smart contract by calculating the probability distribution of the vulnerability type of the smart contract. The vulnerability type with the highest probability is the final output of our approach.

Our approach regards code snippets as multi-modality code sequence and transmits them into the Transformer encoder in the pre-training phase, and calculate the distribution of weights according to the entropy value of the multi-modality code sequence. Then, we fine-tune the encoder by applying the Bug Injection framework (Ghaleb and Pattabiraman, 2020) to generate different types of vulnerability smart contract code snippets and utilize it to evaluate the performance of the classifier to detect smart contract vulnerabilities.

4.2. Pre-processing

Input Processing. In the pre-processing step, we use the multi-modality code sequence `X` from the multiple modalities, consisting of source code, intermediate representation, and assembly code, to train the Transformer encoder. The multi-modality code sequence `X` is separated by a unique `<s>` token. Then, we train the Transformer encoder by applying the multi-modality code sequence as input and transmitting them into the encoder model. For example, the multi-modality code sequence is described as: `if keccak256 ( abi . encode ( guess ) ) <sep> TMP_77 uint8 = vundflw - 10 vundflw uint8 := TMP_77 uint8 <sep> PUSH x DUP REVERT EXIT BLOCK <sep>`.

Tokenization. We develop a code tokenizer suitable for processing multi-modality code sequences and use this tokenizer to

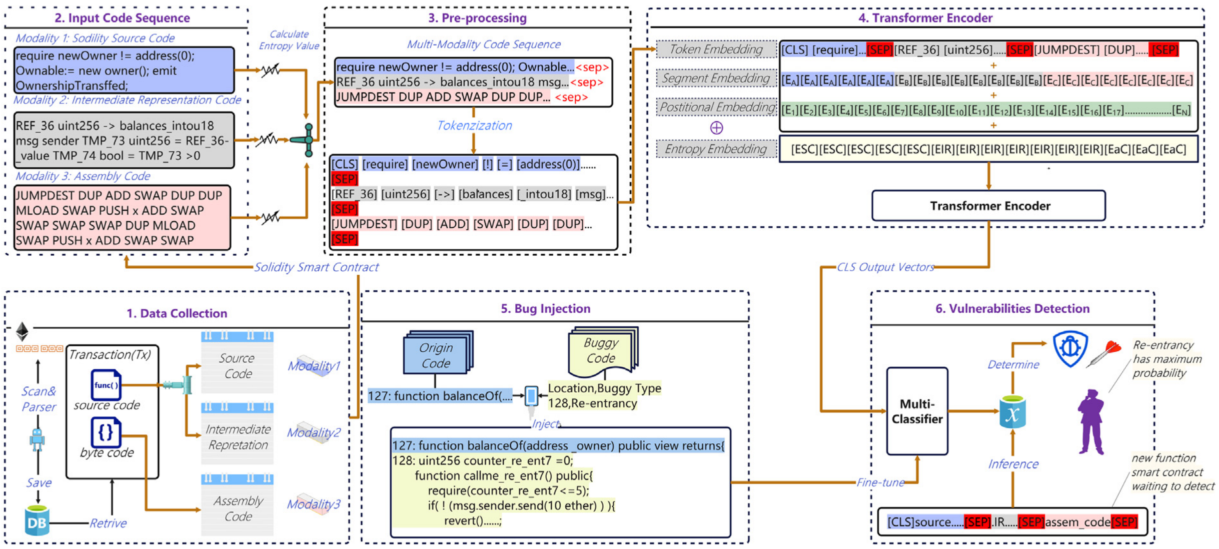

2. Input Code Sequence
Modality 1: Sodility Source Code
require newOwner != address(0); Ownable:= new owner(); emit OwnershipTransffed;
Modality 2: Intermediate Representation Code
REF_36 uint256 -> balances_intou18 msg sender TMP_73 uint256 = REF_36-_value TMP_74 bool = TMP_73 >0
Modality 3: Assembly Code
JUMPDEST DUP ADD SWAP DUP DUP MLOAD SWAP PUSH x ADD SWAP SWAP SWAP SWAP DUP MLOAD SWAP PUSH x ADD SWAP SWAP
Calculate Entropy Value
3. Pre-processing
Multi-Modality Code Sequence
Tokenzization
4. Transformer Encoder
Token Embedding
Segment Embedding
Positional Embedding
Entropy Embedding
Transformer Encoder
Solidity Smart Contract
CLS Output Vectors
1. Data Collection
Scan& Parser
Save
DB
Retrive
Transaction(Tx)
func()
source code
byte code
Source Code
Intermediate Repretation
Assembly Code
Modality1
Modality2
Modality3
5. Bug Injection
Origin Code
Buggy Code
127: function balanceOf(....
Location,Buggy Type 128,Re-entrancy
Inject
127: function balanceOf(address _owner) public view returns(128: uint256 counter_re_ent7 =0; function callme_re_ent7() public{ require(counter_re_ent7<=5); if(! (msg.sender.send(10 ether))){ revert()......;
6. Vulnerabilities Detection
Multi-Classifier
Determine
Inference
Fine-tune
Re-entrancy has maximum probability
new function smart contract waiting to detect
[CLS]source.....[SEP].IR.....[SEP]assem_code[SEP]

Fig. 9. An overview of the proposed approach.

divide the multi-model code sentence into tokens. The tokenizer works similarly to WordPiece[9] and NLTK,[10] which are adopted for various software engineering tasks.

4.3. Multi-modality code sequence

Transformer Encoder. Given a code input sequence X, the Transformer encoder uses a self-attention mechanism to learn the representation of code tokens at each layer. From a mathematical point of view, the Transformer encoder learns the features by calculating the self-attention score matrix for input code sequences. The code sequences in a batch need to be padded to the same length, and the maximum length is 512 tokens in default. We assume that the dimension of the tensor of the input data is $[B, L, D * h]$, where B represents the batch size, L represents sequence length after padding operation, D represents the dimension of each head in the multi-head, and h is the number of heads. Then, for each head, after the input data across each layer, the dimensions of Q, K and V are all $[B, L, D]$. Specifically, we define the weighted matrix as:

$$Q^T K \in \mathbb{R}^{B \times L \times L} \tag{1}$$

and the Scaled Dot-Product attention is:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2}$$

where $\sqrt{d_k}$ is the dimension of the key vector k and query vector q . Moreover, the Multi-head attention is:

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W^O \tag{3}$$

where

$$h_i(head_i) = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{4}$$

The self-attention layer is a tensor which is only related to batch size and sequence length. Then batch size and sequence length multiply V on the right which is equivalent to the operation that a tensor in $[B, L, L]$. Another tensor in $[B, L, D]$ does matrix multiplication in the batch dimension, and the obtained tensor dimension is still $[B, L, D]$. After that, the subsequent feed-forward neural network (Bebis and Georgiopoulos, 1994) (FFN) layer does not change the tensor dimension of the output. Therefore, when the input length changes, for example, it becomes $[B, L_i, H]$, the weight matrix dimension of self-attention becomes $[B, L_i, H_i]$, then the output tensor dimension becomes $[B, L_i, H]$. At the same time, the Transformer encoder utilizes position encoding to retain the position information of the code sequence. Each position is initialized randomly with a vector and is concatenated to the code token vector. Finally, we obtain an embedding containing the position information.

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{dmodel}}}\right) \tag{5}$$

and

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{dmodel}}}\right) \tag{6}$$

$$H_{(o,c)} = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \tag{7}$$

We define the loss function to train our approach, which is described as H(o,c), the predicted probability observation o is of class c. The objective function is used to minimize the sum of the loss functions and the model parameters are updated via the gradient descent algorithm. If $M > 2$, the smart contract vulnerability type is more than two, where the number of classes M represents the type of vulnerabilities.

4.4. Code entropy embedding

The entropy value of the code measures the effective information of the code. According to the following formula, the lower the code entropy value, the higher the code information. Significantly, the higher the probability of occurrence, the lower the information entropy it carries.

$$CodeEntropy\,(x) = -\sum_{x=1}^{X} p_x \log(p_x) \tag{8}$$

To assign different weights to multi-modality code sequences, we calculate the entropy value of the code sequence, which consists of source code, intermediate representation, and assembly code. Then we add the entropy embedding together with the

[9] https://huggingface.co/course/chapter6/6?fw=pt.
[10] https://www.nltk.org/.

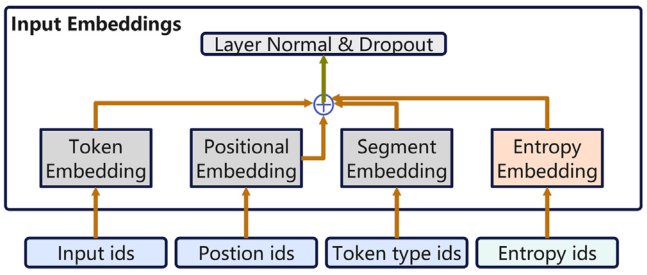
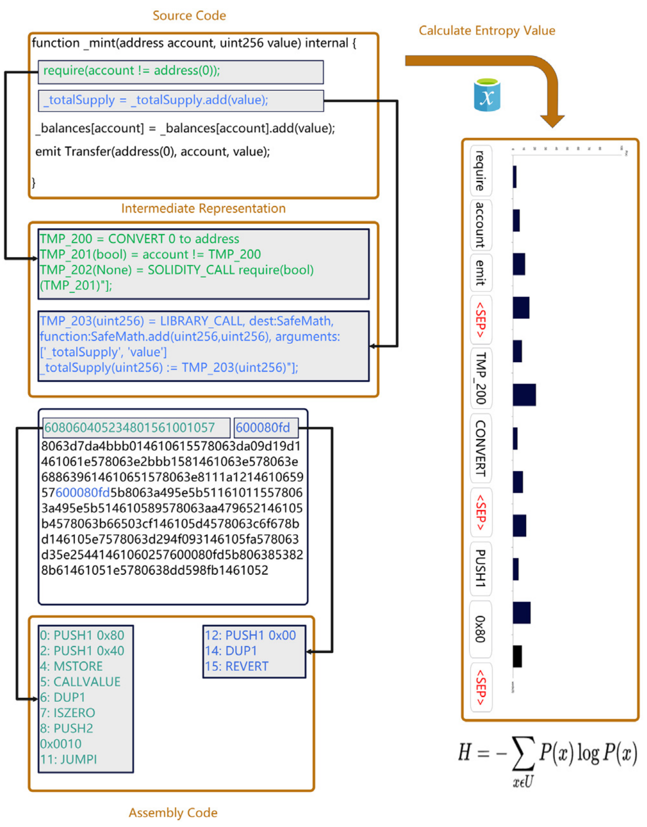


Fig. 10. Entropy embedding.

default embedding of the Transformer encoder, which includes token embedding, segment embedding, and position embedding. For example, if an input code sequence is `source code <s> intermediate representation <s> assembly code`, the embeddings of the Transformer encoder are `token embedding + segment embedding + positional embedding + code entropy embedding`. Fig. 10 shows the details of the default embeddings of the Transformer encoder and the entropy embedding we proposed. The default embedding of the Transformer encoder consists of three parts: token embedding, segment embedding, and position embedding. Therefore, the default embedding technique ignores capturing the input importance of the input sequence. To optimize the performance of the default embedding, we propose to apply the entropy embedding to enhance the performance of the embedding of the Transformer encoder. In the entropy embedding, we use the entropy value to determine the weights of the code sequence, which influence the weight distribution of the multi-modality code sequence. Fig. 11 shows the procedure for calculating the code's entropy value and the entropy embedding we proposed derived from the entropy value of the code. First, we translate the Solidity source code into an intermediate representation using the EVM compiler. Meanwhile, the assembly code is converted from byte code, which comes from paring the transactions data by EVM decompiler (Brent et al., 2018). Then, we count the code token frequency of the multi-modality code sequence according to the vocabulary we provided. After that, we regard the multi-modality code sequence as a random variable `X`, where the frequency of `X` is `p`, and then we calculate the entropy value H_x, which is used to generate entropy embedding.

4.5. Bug injection framework

Bug injection as a testing framework has been widely explored in the field of traditional programming (Bonett et al., 2018; Dolan-Gavitt et al., 2016; Pewny and Holz, 2016). However, many smart contract vulnerability analysis tools evaluate their performance on customized datasets and require manual testing. Therefore, most evaluation tools work based on a limited number of customized datasets (Durieux et al., 2020; Parizi et al., 2018). The Bug Injection framework can generate plenty of buggy code snippets with different vulnerable types, which can provide vulnerability analysis with more datasets. The original code comes from the millions of compiled smart contracts collected by the public datasets. Our approach applies the framework and automatically inserts the buggy code into the original code, which works similarly to attackers inserting a backdoor in the original code. We incorporate the Bug Injection framework and configure the number of buggy code insertions in advance, and then we fine-tune our approach based on these buggy code snippets in predefined smart contract vulnerable types. Therefore, it is convenient for us to abandon collecting public vulnerability



```
function _mint(address account, uint256 value) internal {
 require(account != address(0));
 _totalSupply = _totalSupply.add(value);
 _balances[account] = _balances[account].add(value);
 emit Transfer(address(0), account, value);

}
```

```
TMP_200 = CONVERT 0 to address
TMP_201(bool) = account != TMP_200
TMP_202(None) = SOLIDITY_CALL require(bool)
(TMP_201)"];

TMP_203(uint256) = LIBRARY_CALL, dest:SafeMath,
function:SafeMath.add(uint256,uint256), arguments:
['_totalSupply', 'value']
_totalSupply(uint256) := TMP_203(uint256)"];
```

```
608060405234801561001057 600080fd
8063d7da4bbb014610615578063da09d19d1
461061e578063e2bbb1581461063e578063e
688639614610651578063e8111a1214610659
57600080fd5b8063a495e5b51161011557806
3a495e5b514610589578063aa479652146105
b4578063b66503cf146105d4578063c6f678b
d146105e7578063d294f093146105fa578063
d35e25441461060257600080fd5b806385382
8b61461051e5780638dd598fb1461052
```

```
0: PUSH1 0x80
2: PUSH1 0x40
4: MSTORE
5: CALLVALUE
6: DUP1
7: ISZERO
8: PUSH2
0x0010
11: JUMPI

12: PUSH1 0x00
14: DUP1
15: REVERT
```

Fig. 11. The calculations of multi-modality code entropy.

smart contracts datasets before fine-tuning and evaluating them. Moreover, the generated buggy code automatically labels the vulnerable type simultaneously, which improves the efficiency for us to fine-tune and evaluate the performance of our approach automatically.

4.6. Softmax classifier for detecting vulnerabilities

We utilize the `softmax` classifier to predict the vulnerability type of smart contract. Usually, the `softmax` classifier connects to the `[CLS]` vector, which represents the output of the Transformer encoder. Therefore, we regard `X` as the input of the Transformer encoder, and the component between `X` and `Y` has multiple hidden layers, then the code sequences are passed from `X` to all layers and received by `Y` at the end. When we need to detect the vulnerability types of smart contracts, we should predict the vulnerability type of a given code sequence. Therefore, We consider the output with the highest probability as having this vulnerability.

Algorithm 1 shows the algorithm of our approach: (1) We define two embedding functions: F_t represents the default embedding of the Transformer encoder, and F_e represents the entropy embedding. C_s, C_r, and C_a represent the source code, the intermediate representation, and the assembly code, respectively. C represents the combination of C_s, C_r, and C_a. (2) We calculate the corresponding default and entropy embedding of C through F_t and F_e functions, respectively. (3) We pre-train the Transformer encoder model by utilizing MLM and NSP techniques to learn the intrinsic features of Solidity source code. The resulting output is then transmitted into a softmax classifier for fine-tuning and evaluation.

Algorithm 1 Smart Contract Vulnerability Detection

```
assign $F_t$ = $Embedding()$
assign $F_e$ = $EntropyEmbedding()$
assign $C_s \in [SourceCode]$
assign $C_r \in [IntermediateRepresentation]$
assign $C_a \in [AssemblyCode]$
assign $C \in [C_s, C_r, C_a]$
for i =1,....3 do
    $E_i$ = $F_t(C_i)$+$F_e(C_i)$
    assign $E_m = E_m \cup \{E_i\}$
end for
$Output = MLM(E_m) + NSP(E_m)$ //Pre-training
assign $F_s$ = $SoftmaxClassifier()$
assign $Ret$ = $F_s(Output)$
Return $Ret$
```

5. Experimental setup

We first retrieve and parse the source code and byte code from the transaction on the Ethereum blockchain. Next, we convert them into intermediate representation and assembly code using the EVM compiler and decompiler. Then, we regard them as multi-modality code sequences and transmit them into the Transformer encoder to train for learning to capture the features of the smart contract. In addition, Table 1 shows the details of the dataset we provide. Next, we apply the Bug Injection framework to generate buggy code to fine-tune the classifier and evaluate our approach based on these buggy code snippets. To evaluate our approach more deeply, we investigate the following research questions.

5.1. Research questions

In this subsection, we propose three RQs to discuss our proposed approach. The details are shown below:

RQ 1:

What is the difference between different modality code snippets used to capture smart contract features in the pre-training stage?

Many researchers do not consider taking multi-modality code sequences as input to train deep learning models, such as LSTM and CNN, which limits the ability to capture the features of source code. Hence, we consider taking multi-modality code sequences as input and transmitting them into the Transformer encoder, which aims to learn the internal features of smart contracts. Besides, the Transformer encoder supports multi-modality data for training, which satisfies our expectations for combining source code, intermediate representation, and assembly code as a whole multi-modality code sequence. Therefore, we first need to analyze the impact of uni-modality data separately and decide the influence of each modality data. To achieve such a goal, we explore the pre-training performance of each modality code snippet and aim to improve the performance of the Transformer encoder for capturing the features of the smart contract.

RQ 2:

What is the performance of the Transformer encoder with different modality code sequences on smart contract vulnerability detection in terms of Accuracy, Precision, Recall, and F1 score?

This question illustrates the performance of the different modality code sequences on the Transformer encoder from the specific indicators, which include precision, recall, and F1 score.

RQ 3:

Can entropy embedding and multi-modality code sequence improve the performance of the Transformer encoder in detecting smart contract vulnerabilities?

This question focuses on analyzing the multi-modality and entropy distribution of the multi-modality code sequence, which gives the weight of the input code sequence. We introduce entropy embedding into the Transformer encoder, which can optimize the performance of our approach to capture more features of the smart contract compared to the uni-modality code sequence.

5.2. Data collection

5.2.1. Data collection and process

In order to train our proposed approach, we examine the open-source smart contract dataset and process them to meet our requirements.

- We download the transaction from the Ethereum blockchain and parse it to extract Solidity source code and byte code.
- We translate source code and byte code into intermediate representation and assembly code using EVM compiler and decompiler, respectively.
- To capture intrinsic characteristics of smart contracts, we treat source code, intermediate representation, and assembly code as multi-modality data and concatenate them into code sequences, which are separated by `<SEP>`.

5.2.2. Smart contract collection

We obtain numerous open-source and successful compilation smart contracts by downloading and parsing Ethereum transactions from the public restful API service. Anyone can browse and access the blockchain details information in Etherscan. Each Ethereum transaction consists of byte code and relevant information, including deployed contact name, address, and transaction hash. Besides, each smart contract has a unique address after deployment. Once deployed, each smart contract's source code is immutable, enabling us to generate the corresponding intermediate language and assembly code successfully. Table 1 shows the detailed information of the datasets we collected, which contains `Smart Contract Wild` published by other investigators and `Raw Smart Contract` published by us, both of which are adopted for pre-training the Transformers encoder. `Smart Contract Curated` belongs to public contracts with vulnerabilities, and SolidiFI (Ghaleb and Pattabiraman, 2020) Benchmark comes from the Bug Injection framework. Both are used for fine-tuning and evaluating our proposed approach.

5.2.3. Vulnerability collection

Table 2 shows several common types of vulnerabilities. We can find that `Reentrancy` is the most common vulnerability type. Meanwhile, miners tend to take control of Front Running and Time Manipulation vulnerabilities by sorting the transaction order according to the gas fee. Besides, Arithmetic and Short Addresses vulnerabilities are mainly attributed to the insecure design of the Solidity programming language. We utilize the Bug Injection framework to generate source code with these vulnerabilities and generate a dataset called `SolidiFI Benchmark`. Then, we fine-tune and evaluate our approach based on this dataset which contains buggy code snippets containing these common vulnerabilities.

Table 1
Dataset for pre-trained and evaluated.

Dataset	Description
Raw Smart Contract	*Self collected dataset used for pre-trained model*
Smart Contract Curated	*A curated dataset that contains 143 annotated contracts with 208 tagged vulnerabilities*
Smart Contract Wild	*A dataset with 47,398 unique contracts from the Ethereum network*
SolidiFI Benchmark	*A remote dataset of contracts injected with 9369 bugs of 7 different types*

Table 2
Smart contract vulnerability collection.

Vulnerability	Description
Reentrancy	*Reentrant function calls make a contract to behave in an unexpected way*
Access control	*Failure to use function modifiers or use of tx.origin*
Arithmetic	*Integer over/underflows*
Unchecked low level calls	*call(), callcode(), delegatecall() or send() fails and it is not checked*
Denial of service	*The contract is overwhelmed with time-consuming computations*
Bad randomness	*Malicious miner biases the outcome*
Front running	*Two dependent transactions that invoke the same contract are included in one block*
Time manipulation	*The timestamp of the block is manipulated by the miner*
Short addresses	*EVM itself accepts incorrectly padded arguments*
Unknown	*Vulnerabilities not identified in DASP*

Table 3
Supported tools for detecting smart contract vulnerabilities.

Tools	Tool URLs
HoneyBadger	https://github.com/christoftorres/HoneyBadger
Maian	https://github.com/ivicanikolicsg/MAIAN
Manticore	https://github.com/trailofbits/manticore
Mythril	https://github.com/ConsenSys/mythril
Osiris	https://github.com/christoftorres/Osiris
Oyente	https://github.com/enzymefinance/oyente
Securify	https://github.com/eth-sri/securify
Slither	https://github.com/crytic/slither
Smartcheck	https://github.com/smartdec/smartcheck
HoneyBadger	https://github.com/christoftorres/HoneyBadger

In addition, we use the Bug Injection framework because we need sufficient labeled data to fine-tune our approach. There are inadequate datasets for us to fine-tune and evaluate our approach, which results in using the Bug Injection framework to generate sufficient smart contract code with particular vulnerabilities. The `SolidiFi` Benchmark has buggy source code in seven types, such as `reentrancy`, `timestamp dependency`, `unhandled exceptions`, `unchecked send`, and `underflow-overflow`. These buggy code snippets are enough to fine-tune and evaluate our approach.

5.3. Vulnerability detection tools

To compare with the current popular smart contract vulnerability detection tools, we have collected several tools that can detect at least seven vulnerability types. As shown in Table 3, not all approaches are suited for our study, and we only consider the tools that met the requirements of our study. The main criterion is that the tools can analyze smart contracts that are publicly available. After inspecting public analysis tools, we found several tools that met our requirements, such as Oyente, Securify, Mythril, and SmartCheck. According to these research papers (Luu et al., 2016; Tikhomirov et al., 2018; Tsankov et al., 2018), we found that Oyente, Mythril, and SmartCheck are suitable for detecting the timestamp dependency bug (TOD), while other tools support detecting other vulnerabilities. Table 3 presents the details of tools suitable for detecting vulnerabilities.

5.4. Evaluation methods

We compare our approach with some vulnerability detection approaches in terms of precision (`P`), recall (`R`), and `F1` value. Except for the overall detection performance measured by precision, recall, and F1 value, we also have an interest in knowing the performance across various vulnerability types. The calculation formula for accuracy is:

$$\texttt{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{9}$$

where `TN` is the true negative vulnerability type detection, and `FN` refers to the false negative vulnerability type. The calculation formula for precision is:

$$\texttt{Precision} = \frac{TP}{TP + FP} \tag{10}$$

where `TP` is the true positive vulnerability type detection, and `FP` refers to the wrong vulnerability type we detected. For recall, it is defined as:

$$\texttt{Recall} = \frac{TP}{TP + FN} \tag{11}$$

where `FN` refers to the true vulnerability type we missed. For `F1` which is the harmonic mean of the Precision and Recall, it is defined as:

$$\texttt{F1} = \frac{2 * \texttt{Precision} * \texttt{Recall}}{\texttt{Precision} + \texttt{Recall}} \tag{12}$$

6. Evaluation result

This section presents our experimental results and the answers to our research questions.

6.1. Answer to RQ1

To better train the learning ability of the Transformer encoder, we introduce two learning techniques in the training process: MLM and NSP. For the MLM learning technique, the strategy is to mask the code token randomly, like replacing the original code token with [MASK], then predict the output value corresponding to the mask token. In particular, we mask the tokens of the buggy code in the input sequence and enforce the Transformer encoder to predict the mask token from buggy code, which improves the ability to recognize the characteristics of the smart contract vulnerabilities. Moreover, we use the NSP learning technique to obtain the relationship between different modality code snippets within the same input sequence. Generally, NSP and MLM are two standard training methods in the Transformer encoder. NSP determines whether there is a correlation between

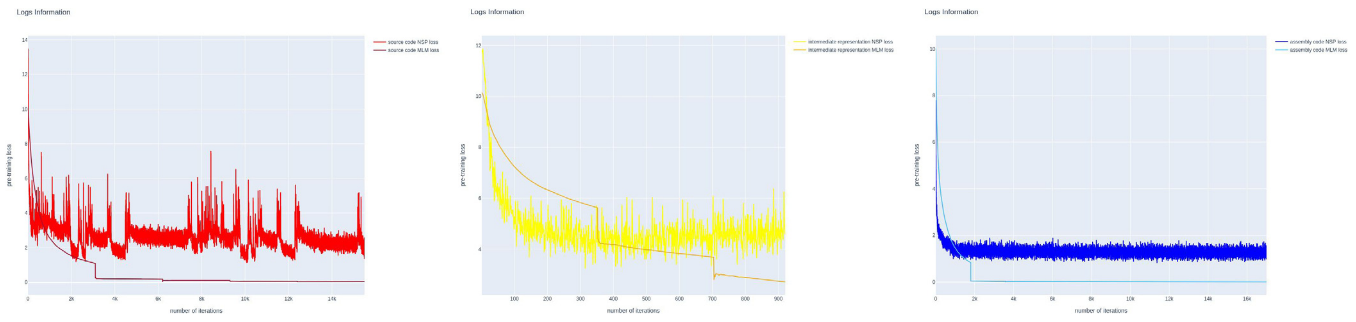


Fig. 12. Multi-modality code training loss comparison between NSP & MLM.

different modality data. MLM considers whether the code sequence is missing tokens and predicts the unknown token in the code sequence. Both affect our approach's performance because the ability of these two training methods is different. The first question mainly focuses on the performance of how modality data affect the Transformer encoder. We translate the raw smart contracts into intermediate representation and assembly code using the EVM compiler and decompiler. Then we transmit a uni-modality code sequence into the Transformer encoder and train it using MLM and NSP, respectively. Among them, Fig. 12 represents the loss change curve of the source code, intermediate representation, and assembly code respectively. They all represent the experimental results of the pre-training loss curve for multi-modality code sequences. After that, we combine them together and transmit them into the Transformer encoder for training and use the NSP and MLM techniques separately. From the experiment result, we can figure out that the iterative process of MLM works very stably and is easy to converge. It may be explained that MLM does not need to consider the relationship between the multi-modality code sequence but only needs to consider the masked token. In contrast, the NSP technique needs to consider the relationship between multi-modality code sequences, resulting in an unstable iterative process, and the correlation between multi-modality code sequences is not apparent. It may demonstrate why the code generation task is more challenging to be implemented than the code complement task. In addition, Fig. 13 shows the loss change curve obtained only with the NSP and MLM training methods simultaneously. Moreover, we employ a multi-modality code sequence, including source code, intermediate representation, and assembly code under the same NSP and MLM techniques to train the Transformer encoder. According to the loss change curve shown, the convergence speed of assembly code by using NSP and MLM is faster than that of source code and intermediate representation. Moreover, the intermediate representation is more stable than the source code by using the NSP technique, and the loss value is higher than the source code. Meanwhile, the source code is more stable than the assembly code and the intermediate representation using the MLM technique, but the convergence speed of the source code is slower than the intermediate representation. The loss value of the intermediate representation is higher than the source and assembly code snippets under the same number of iterations.

Summary. To answer the `RQ1`, we analyze the pre-trained loss iteration curve because we can infer the impact of different modalities on the Transformer encoder model by discovering the difference in loss changes. When we apply MLM and NSP techniques to the two modalities of source code and intermediate representation, the performance of the Transformer encoder works unstably. Meanwhile, the loss performance of the assembly code works very stably. When we apply the MLM technique to the Transformer encoder alone, the intermediate representation works stably more than the source code in loss iteration, which shows that the intermediate representation is more steady than the source code. The major reason is that the source code written by humans is not as concise and regular as the intermediate representation translated by the EVM compiler.

6.2. Answer to RQ2

Table 4 shows the performance of our approach for detecting smart contract vulnerabilities using multi-modality data to fine-tune, and we evaluate our approach in terms of precious, recall, and f1 scores. The experimental results show that when we only use source code as a modality input sequence. In addition, support indicates the number of detected vulnerabilities. Our approach tends to be more likely to detect the vulnerabilities of `Re-entrancy` and `Timestamp-Dependency` and the accuracy rates are 0.89 and 0.65, respectively. However, it performs poorly detecting `Uncheck-Send` and `Unhandle-Exception` vulnerabilities. Moreover, the experimental result shows that when we fine-tune our approach only on intermediate representation modality code sequence, our approach tends to detect `Tx.origin` and `Re-entrancy` type vulnerabilities. The detection accuracy rates are 0.91 and 0.76, respectively, and the detection accuracy of `Uncheck-Send` type vulnerabilities is also `0.75`. Meanwhile, the performance in detecting Timestamp-Dependency vulnerability is not as well as in detecting `Tx.origin` and `Re-entrancy` vulnerabilities. The accuracy rates of detecting `Tx.origin` and `Re-entrancy` types are improved by 36% and 13% than the average, and the F1-scores are 24% and 37% which are higher than the average. Finally, our approach with assembly code is inefficient in detecting the common smart contract vulnerabilities but works better in detecting `Uncheck-Send` vulnerability. The major reason is that the vocabulary size of the assembly code is too small. The vocabulary we use has only `125` words, making the assembly code modality data easy to converge, but performs unsatisfactorily in general compared to source code and intermediate representation.

Summary. To answer RQ2, we use the different modalities, each of the different modalities i.e., source code, intermediate representation, and assembly code, to fine-tune the Transformer encoder. Then, we apply the Bug Injection framework to automatically yield labeled data in different vulnerability types of smart contracts. Experimental results show that when we fine-tune the classifier with the Transformer encoder using source code or intermediate representation, respectively, our approach has a high precious and recall value to detect common type vulnerabilities, namely `reentrancy`, `Overflow`, `Underflow`, and `tx-origin` vulnerabilities, and these types of vulnerabilities are the most common and threatening ones to users. In contrast, when we fine-tune the classifier using assembly code, which has a much smaller vocabulary than source code and intermediate representations, the assembly code works worse in detecting common type vulnerabilities but has a higher recall value in detecting `Uncheck-Send` vulnerability.

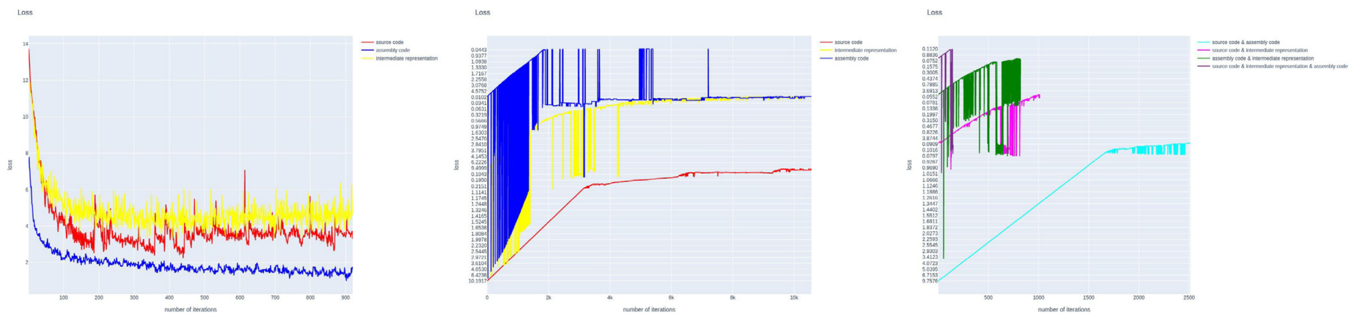


Fig. 13. Multi-modality code training loss.

Table 4
Precision, Recall, and F1 score of **Source Code**, **Intermediate Representation**, and **Assembly Code** for vulnerabilities detection, respectively.

Vulnerability types	Source code				Intermediate representation				Assembly code			
	Precision	Recall	F1-score	Support	Precision	Recall	F1-score	Support	Precision	Recall	F1-score	Support
TOD	0.5000	0.7000	0.5833	30	0.6923	0.7500	0.7200	12	0.1667	0.1667	0.1667	12
Overflow, Underflow	0.6429	0.6545	0.6486	55	0.4737	0.6923	0.5625	13	0.0909	0.2308	0.1304	13
Re-entrancy	0.8947	0.5965	0.7158	57	0.7619	1.0000	0.8649	16	0.1333	0.1429	0.1379	14
Timestamp-Dependency	0.6596	0.7561	0.7045	41	0.4286	0.4286	0.4286	7	0.2308	0.2143	0.2222	14
Tx.origin	0.5714	0.6250	0.5970	32	0.9167	0.6875	0.7857	16	0.1667	0.0500	0.0769	20
Uncheck-Send	0.1667	0.1000	0.1250	20	0.7500	0.2500	0.3750	12	0.1111	0.3279	0.1659	19
Unhandled-Exception	0.3182	0.4118	0.3590	17	0.6667	0.6667	0.6667	12	0.0000	0.0000	0.0000	17
Accuracy	–	–	0.5992	252	–	–	0.6705	88	–	–	0.1284	109
Macro Avg	0.5362	0.5491	0.5333	252	0.6700	0.6393	0.6290	88	0.1285	0.1375	0.1235	109
Weighted Avg	0.6168	0.5992	0.5975	252	0.6968	0.6705	0.6575	88	0.1259	0.1284	0.1170	109

6.3. Answer to RQ3

To answer this research question, we analyze the effect of the entropy embedding of the multi-modality data on the Transformer encoder. Before that, we calculate the entropy value of the multi-modality code in advance. Fig. 14 shows entropy, total and unit value distribution in different modality data from the raw smart contract dataset. The above experimental results show that the distribution of entropy values among different modality code snippets is different, so we need to assign different weight parameters to different modality code snippets during training. We apply the entropy embedding technique to handle the weight distribution of multi-modality code, which is beneficial for the Transformer encoder to capture important modality code information and ignore unimportant modality code information. To seek the relation between the code entropy and multi-modality code, we calculate the entropy value distribution of multi-modality code, including source code, intermediate representation, and assembly code. Fig. 15 shows the entropy, total and unit distribution of the multi-modality code. We can figure out that the distribution of intermediate presentation is relatively balanced compared to source and assembly code, but the scope is larger than assembly code. The experimental results show that the lower the entropy value, the easier the loss curve is to converge during the training. Furthermore, we compare the performance of our approach with other popular vulnerability detection tools, where multi-modality data without entropy embedding denotes $\mathtt{NV}_m$ and with entropy embedding denotes $\mathtt{NV}_m(ent)$.

As shown in Table 5, we select several popular analysis tools to detect smart contract vulnerabilities. The data set contains `9153` unique smart contracts, and each smart contract contains multiple function implementations. Then, we utilize the SmartBug framework, which integrates these analysis tools in advance to run these analysis tools and evaluate performance. We investigate the $\mathtt{NV}_m$ experiment results and find that `Timestamp-Dependency` and `Re-entrancy` are the two most common types of vulnerabilities in smart contracts, with a probability of `4.0%` and `1.7%`, respectively. Generally, `8.8%` of the collected smart contracts have vulnerabilities in our selected smart contracts. The experimental results show that our approach performs better than Manticore, Mythril, Osiris, Oyente, Security, Slither, and Smartcheck in detecting common vulnerabilities, such as `Re-entrancy` and `Time-Manipulation`. Meanwhile, the $\mathtt{NV}_m(\mathtt{ent})$ results show that if we apply the entropy embedding with the default embedding of the Transformer encoder, our approach is prone to detect more common types of vulnerabilities than $\mathtt{NV}_m$.

Summary. To answer `RQ3`, we detect the vulnerabilities using a multi-modality code sequence and compared it with other popular vulnerability detection tools. The experimental results show that our approach performs better than other popular vulnerability detection tools, such as Maian and HoneyBadger when detecting the most common vulnerabilities. In detail, Maian and HoneyBadger fail to detect Overflow, Underflow, Reentrancy, Timestamp-Dependence, and Tx.origin vulnerabilities. Our approach performs better than Mythril, Osiris, Oyente, and Securify methods in detecting reentrancy, overflow, and tx-origin vulnerabilities. Table 5 shows that about `7%` of the vulnerability types in the `9153` smart contracts belong to reentrancy and overflow vulnerabilities. Moreover, $\mathtt{NV}_m(\mathtt{ent})$ experimental result shows that our approach with entropy embedding tends to detect reentrancy and overflow vulnerabilities more than without using entropy embedding. In this situation, our approach with entropy embedding detects about `4.2%` vulnerabilities belonging to reentrancy among the `9642` smart contracts.

6.4. Methodology and result analysis

By analyzing the above experimental results, we found the following conclusions.

The Effective of Different Code Representation When Pre-training the model: We use the results of ablation experiments to analyze the effectiveness of different code representations in detecting smart contract vulnerabilities. By comparing the loss iterations of the Transformer model using MLM and NSP pre-training tasks, we find that assembly code and intermediate language representations are easier to converge than human-written

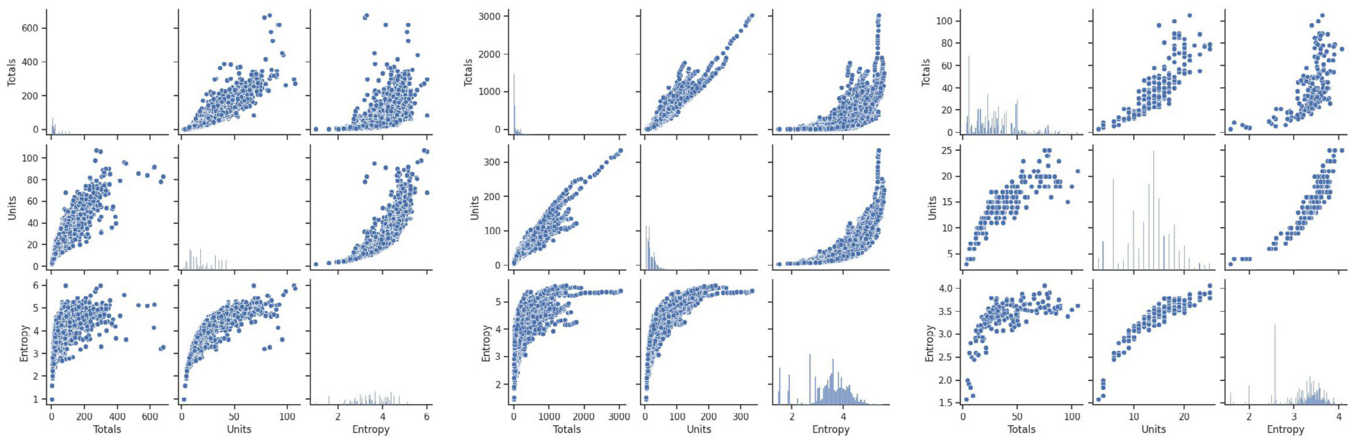


Fig. 14. Totals, units and entropy distribution of multi-modality code.

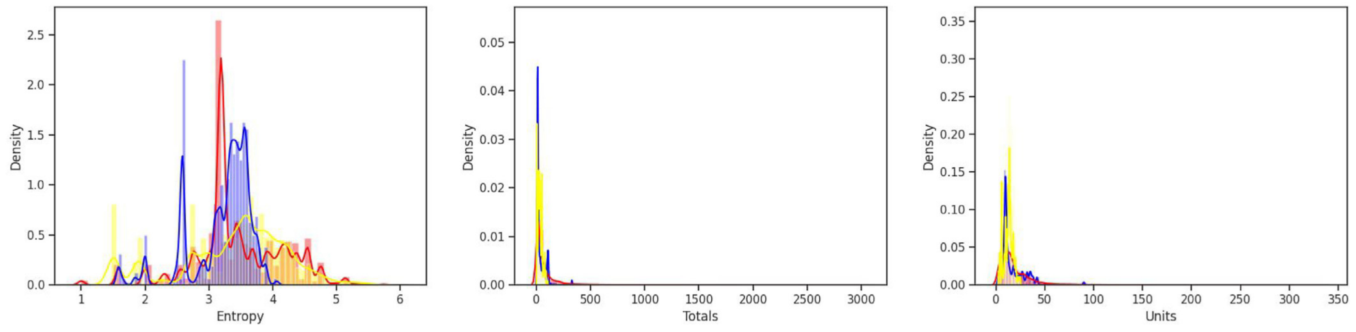


Fig. 15. Entropy, totals and units distribution of multi-modality code.

Table 5
The total number of smart contracts that have at least one vulnerability (Analysis of 9153 Smart Contracts).

Category	HoneyBadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	NV$_m$	NV$_m$(ent)
Access control	0/19	0/19	4/19	4/19	0/19	0/19	0/19	–	–
Denial of service	0/7	0/7	0/7	0/7	0/7	0/7	0/7	–	–
Front running	0/7	0/7	0/7	2/7	0/7	0/7	2/7	–	–
TOD	–	–	–	–	–	–	0 (0%)	0	0
Overflow, Underflow	0/22	0/22	–	15/22	11/22	12/22	0/22	40 (0.4%)	52 (0.5%)
Reentrancy	0/8	0/8	2/8	5/8	5/8	5/8	5/8	162 (1.7%)	409 (4.2%)
Timestamp-Dependency	0/5	0/5	1/5	0/5	0/5	0/5	0/5	374 (4.0%)	143 (0.14%)
Tx.origin	–	–	–	–	–	–	–	126 (1.3%)	17 (0.1%)
Uncheck-Send	0/12	0/12	2/12	5/12	0/12	0/12	3/12	59 (0.6%)	11 (0.1%)
Unhandled-Exception	–	–	–	–	–	–	–	45 (0.4%)	163 (1.6%)
Unknown	2/3	0/3	0/3	0/3	0/3	0/3	0/3	–	–
Total	2/115	0/115	13/115	31/115	16/115	17/115	10/115	9153 (8.8%)	9642 (8.2%)

code fragments because of their single grammatical structure. It is easy to learn the grammatical structure information of the code. This is because the intermediate code actually generates the language through a syntax analyzer, and the assembly language needs to run on the EVM virtual machine. Therefore the repeated statements in the source code are simplified into basic operation instructions. Overall, the ablation studies show that the two pre-training tasks are somewhat practical on different code representations.

The Effect of Multi-code Representation: Our experimental results show that multi-modal data can enhance the model's ability to detect contract vulnerabilities. If we suppose that the source code of the contract is simply input, the model can only detect some conventional types of vulnerabilities, such as *reentrancy*, *overflow* and *underflow*, which can discover the characteristics of crucial information by analyzing the source code. Like *uncheck — send*, this type of vulnerability needs to analyze the underlying assembly code to discover the characteristic information.

Advantages of Entropy Embedding Algorithms: When we input multi-modal data, we need to assign different weight coefficients according to the entropy value of the modal data to optimize the training effect of the model. Here we use the technology of entropy representation to achieve this goal. The experimental results show that after adding entropy representation coding, the overall input modal data achieves a better performance detecting contract loopholes than the original input to train the model.

7. Threats to validity

Within this study, we investigate two types of validity: internal and external.

7.1. Internal validity

For the internal threats, we only inject seven vulnerability types using the bug injection framework. However, these vulnerability types have been detected by most vulnerability detection tools. Moreover, hack attacks have evaluated and exploited these vulnerabilities in the past years. Therefore, these types of smart contract vulnerabilities are representative of smart contracts. In order to mitigate this validity, in the future, we need to collect smart contract data sets that contain more vulnerabilities, which will allow our tools to detect more vulnerabilities on various smart contract codes and test our tools on the same datasets and runtime environment.

7.2. External validity

For the external threat, we collect the limited number of smart contracts. The smart contract code snippets we collected do not involve other public blockchain platforms because other platforms may have other types of smart contract vulnerabilities. To mitigate external validity in the future, we need to collect smart contracts from different platforms and verify them in different test environments. The experimental results also need to be verified in other similar environments.

7.3. Construct validity

The construct threat concerns about our measurement of false negatives and positives. For false negatives, it is possible that the vulnerabilities cannot be exploited in practice. To mitigate this validity, our vulnerability detection method detects all important vulnerability types by sampling the set of false negatives and attempting to exploit them.

7.4. Conclusion validity

The conclusion threat concerns about the relationship between treatment and outcome. Appropriate statistical procedures have been adopted to draw our conclusions. In the future, we will utilize the more effective test method to investigate the statistically significant differences in the values of readability metrics obtained on code snippets from smart contracts.

8. Discussion

8.1. Vulnerability analysis and verification

Although our approach in this paper can detect common vulnerability types like other analysis tools, various vulnerabilities cannot be detected since our approach uses source code information. In more detail, these undetected vulnerabilities do not correlate with the source code of the smart contract, such as a front-run attack, which takes advantage of the order of transactions by a miner. Moreover, some vulnerabilities tend to be taken advantage of by malicious execution by attackers, which have nothing to do with the smart contract code itself. These vulnerabilities include `Fake Recharge`, `Fake Notification`, and `Fake Token`. It is necessary to collect non-code data to detect these vulnerabilities by analyzing transaction records, traffic, etc. After that, we can use the open source platform, such as Truffle (Bogner et al., 2016) to verify our detected vulnerabilities in reality.

8.2. Performance deterioration

The approach we proposed relies on updates and maintenance in the future. When the EVM compiler upgrades, the intermediate representation, and assembly code also change, which requires new multi-modality code sequences to transmit into the Transformer encoder to learn new features of smart contracts. For example, we try to train the Transformer encoder using an intermediate representation; the identifier may not exist in the old vocabulary. When our approach receives new multi-modality code sequences, the `UNK` identifier replaces the new code token, which will fail to capture the new features of the smart contract.

9. Related work

In this section, we mainly focus on techniques related to smart contract analysis methods.

9.1. Static analysis

The static analysis technique simulates the execution of the program by inserting dummy variables in the program code and tracking the value and state of the variables during execution. This makes it possible to detect bugs and vulnerabilities in source code without running the actual code. Symbolic execution is practical in automated testing, code review, and static analysis. For example, Oyente (Luu et al., 2016) detects vulnerabilities by traversing all possible symbolic output results. Maian (Nikolić et al., 2018) detects potential vulnerabilities by traversing all execution paths. Osiris focuses on detecting arithmetic operation vulnerabilities through symbolic execution and taints analysis. teEther (Krupp and Rossow, 2018) and sCompile (Chang et al., 2019) also utilize symbolic execution to focus on critical operations to detect vulnerabilities. In addition, Manticore (Mossberg et al., 2019) and Mythril (Parizi et al., 2018) also apply symbolic execution to detect vulnerabilities. These tools utilize symbolic execution to detect vulnerabilities according to predefined rules, which leads to failure to detect all vulnerabilities.

9.2. Symbolic execution program analysis

The Symbolic Execution tools are also used in detecting smart contract vulnerabilities. For example, Securify (Tsankov et al., 2018), which extracts the semantic features of byte code and checks the violation of the security rules. Vandal (Brent et al., 2018) converts byte code to intermediate representation and checks whether it violates the predefined rules. Zeuse (Chalupa, 2020) converts source code to LLVM bytecode and leverages the framework for interpretation and symbolic model checking. These smart contract vulnerability detection tools. Securify (Tsankov et al., 2018) uses symbolic execution program analysis technology, simulates program execution by inserting dummy variables in smart contract code, and tracks the value and state of variables during execution to detect vulnerabilities and risks in smart contracts. Vandal (Brent et al., 2018) also uses symbolic executive analysis, which looks for vulnerabilities by analyzing smart contracts' data flow and control flow. Moreover, VeriSmart (So et al., 2020) and VerX (Permenev et al., 2020) use formal verification to check security boundaries.

9.3. Fuzz testing

ContractFuzzer (Jiang et al., 2018) uses fuzzing methods to detect vulnerabilities. Fuzz testing is a software testing technique that simulates unreasonable or extreme usage situations by generating random input data to find bugs and vulnerabilities in software. ContractFuzzer uses fuzzing techniques to detect vulnerabilities in smart contracts. It tests smart contracts by generating random input data and tracks the values and states of variables during execution to detect vulnerabilities and risks in smart contracts. ReGuard (Liu et al., 2018) converts smart contracts into equivalent traditional programming languages and uses fuzzing methods to detect vulnerabilities. These methods detect smart contract vulnerabilities according to formulating predetermined rules. In contrast, our approach does not need to formulate rules in advance. The vulnerability pattern can be automatically obtained and evaluated by our approach.

10. Furture work

In the future, we plan to develop automated tools for detecting vulnerabilities in smart contracts. These tools can use deep learning algorithms to detect common patterns in code that could indicate potential vulnerabilities. In addition, we will develop more sophisticated tools to detect non-code-level smart contract vulnerabilities, such as double spending and sandwich attacks. Additionally, we plan to develop best practices and standards for smart contract vulnerability detection, which can ensure the consistency and reliability of the process.

11. Conclusion

This paper proposes a novel approach to detecting smart contract vulnerabilities. To capture the intrinsic features of the smart contract, we utilize multi-modality code and entropy embedding techniques to enhance the ability of the Transformer encoder. First, we collect the Solidity source and byte code from the Ethereum blockchain. Then we use the EVM compiler to translate the source code into an intermediate representation and the EVM decompiler to translate byte code into assembly code, respectively. After that, we concatenate these code snippets, regard them as multi-modality code sequences, and transmit them into the Transformer encoder for pre-training. Besides, we utilize the Bug Injection framework to automatically generate the buggy code, which results in fine-tuning the classifier to detect the vulnerabilities of the smart contract. The experimental result shows that our proposed approach improves vulnerability detection accuracy by using multi-modality and entropy embedding techniques compared to uni-modality code sequences.

CRediT authorship contribution statement

Dawei Yuan: Conceptualization, Methodology, Code development, Writing – original draft. **Xiaohui Wang:** Writing – original draft, Investigation, Writing – review & editing. **Yao Li:** Validation, Writing – review & editing. **Tao Zhang:** Conceptualization, Resources, Project administration, Supervision, Funding acquisition, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

This work was supported in part by the Macao Science and Technology Development Fund, Macao, China under Grant 0047/2020/A1 and 0014/2022/A.

References

Bebis, G., Georgiopoulos, M., 1994. Feed-forward neural networks. IEEE Potentials 13 (4), 27–31.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al., 2016. Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 91–96.

Bogner, A., Chanson, M., Meeuw, A., 2016. A decentralised sharing app running a smart contract on the ethereum blockchain. In: Proceedings of the 6th International Conference on the Internet of Things. pp. 177–178.

Bonett, R., Kafle, K., Moran, K., Nadkarni, A., Poshyvanyk, D., 2018. Discovering flaws in {security-focused} static analysis tools for android using systematic mutation. In: 27th USENIX Security Symposium. pp. 1263–1280.

Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B., 2018. Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981.

Chalupa, M., 2020. DG: analysis and slicing of LLVM bitcode. In: International Symposium on Automated Technology for Verification and Analysis. pp. 557–563.

Chang, J., Gao, B., Xiao, H., Sun, J., Cai, Y., Yang, Z., 2019. Scompile: Critical path identification and analysis for smart contracts. In: International Conference on Formal Engineering Methods. pp. 286–304.

Clause, J., Li, W., Orso, A., 2007. Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis. pp. 196–206.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R., 2016. Lava: Large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy. IEEE, pp. 110–121.

Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P., 2020. Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering. pp. 530–541.

Essaid, M., Kim, D., Maeng, S.H., Park, S., Ju, H.T., 2019. A collaborative ddos mitigation solution based on ethereum smart contract and RNN-LSTM. In: 2019 20th Asia-Pacific Network Operations and Management Symposium. pp. 1–6.

Ethayarajh, K., 2019. How contextual are contextualized word representations? comparing the geometry of BERT, ELMo, and GPT-2 embeddings. arXiv preprint arXiv:1909.00512.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L., 2007. A static analysis framework for detecting SQL injection vulnerabilities. In: Proceedings of the 2017 Annual International Computer Software and Applications Conference, Vol. 1. pp. 87–96.

Ghaleb, A., Pattabiraman, K., 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 2020 ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 415–427.

Ghazvininejad, M., Levy, O., Liu, Y., Zettlemoyer, L., 2019. Mask-predict: Parallel decoding of conditional masked language models. arXiv preprint arXiv:1904.09324.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Hirai, Y., 2017. Defining the ethereum virtual machine for interactive theorem provers. In: Proceedings of the 2017 International Conference on Financial Cryptography and Data Security. pp. 520–535.

Jiang, B., Liu, Y., Chan, W.K., 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In: Proceedings of the 2018 IEEE/ACM International Conference on Automated Software Engineering. pp. 259–269.

Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K., 2020. Learning and evaluating contextual embedding of source code. In: Proceedings of the 2020 International Conference on Machine Learning. pp. 5110–5121.

Krupp, J., Rossow, C., 2018. {Teether}: Gnawing at ethereum to automatically exploit smart contracts. In: 27th USENIX Security Symposium. pp. 1317–1333.

Li, Y., Cui, B., Zhang, Z.M., 2021. Efficient relational sentence ordering network. IEEE Trans. Pattern Anal. Mach. Intell..

Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B., 2018. Reguard: finding reentrancy bugs in smart contracts. In: Proceedings of the 2018 IEEE/ACM International Conference on Software Engineering: Companion. IEEE, pp. 65–68.

Luo, K., Gummaraju, J., Franklin, M., 2001. Balancing thoughput and fairness in SMT processors. In: 2001 IEEE International Symposium on Performance Analysis of Systems and Software. pp. 164–165.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., Hobor, A., 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Mehar, M.I., Shier, C.L., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H.M., Laskowski, M., 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. J. Cases Inf. Technol. 21 (1), 19–32.

Mi, F., Wang, Z., Zhao, C., Guo, J., Ahmed, F., Khan, L., 2021. VSCL: Automating vulnerability detection in smart contracts with deep learning. In: Proceedings of the 2021 IEEE International Conference on Blockchain and Cryptocurrency. pp. 1–9.
Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: Proceedings of the 2019 IEEE/ACM International Conference on Automated Software Engineering. pp. 1186–1189.
Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A., 2018. Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 2018 Annual Computer Security Applications Conference. pp. 653–663.
Parizi, R.M., Dehghantanha, A., Choo, K.-K.R., Singh, A., 2018. Empirical vulnerability analysis of automated smart contracts security testing on blockchains. arXiv preprint arXiv:1809.02702.
Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M., 2020. Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy. IEEE, pp. 1661–1677.
Pewny, J., Holz, T., 2016. EvilCoder: automated bug insertion. In: Proceedings of the 2016 Annual Conference on Computer Security Applications. pp. 214–225.
Qian, P., Liu, Z., He, Q., Zimmermann, R., Wang, X., 2020. Towards automated reentrancy detection for smart contracts based on sequential models. IEEE Access 8, 19685–19695.
So, S., Lee, M., Park, J., Lee, H., Oh, H., 2020. VeriSmart: A highly precise safety verifier for ethereum smart contracts. In: 2020 IEEE Symposium on Security and Privacy. pp. 1678–1694.
Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y., 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 2018 International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 9–16.
Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M., 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.
Wenzel, M., Paulson, L.C., Nipkow, T., 2008. The isabelle framework. In: Proceedings of the 2008 International Conference on Theorem Proving in Higher Order Logics. pp. 33–38.
Yu, Y., Si, X., Hu, C., Zhang, J., 2019. A review of recurrent neural networks: LSTM cells and network architectures. Neural Comput. 31 (7), 1235–1270.
Yu, X., Zhao, H., Hou, B., Ying, Z., Wu, B., 2021. DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection. In: Proceedings of the 2021 International Joint Conference on Neural Networks. pp. 1–8.

Dawei Yuan is a Ph.D. student in the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. Before joining MUST, he worked as a software engineer at United Imaging Healthcare Corporation (UIH), Shanghai. He also got an M.Sc. in Software Engineering from the University of Science and Technology of China and an BS degree in Network Engineering from Nanjing University of Posts and Telecommunications. His research interests mainly focus on using artificial intelligence techniques to solve problems in software engineering, such as bug localization and code analysis.

Xiaohui Wang is a postgraduate student in the School of Computer Science and Engineering, Macau University of Science and Technology (MUST), under the supervision of Prof. Tao Zhang. His research interests include NLP, software report analysis and text processing.

Yao Li is a Ph.D. student in the School of Computer Science and Engineering at Macau University of Science and Technology (MUST) under the supervision of Prof. Zhang Tao. Prior to joining MUST, he received his Master's degree in Computer Science from Shantou University in 2020. His research interests lie in software engineering and android malware detection.

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the Ph.D. degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He is a senior member of IEEE and ACM.