The Journal of Systems and Software 209 (2024) 111919

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction ☆

Jie Cai [a,b], Bin Li [a,*], Tao Zhang [b], Jiale Zhang [a], Xiaobing Sun [a]

[a] School of Information Engineering, Yangzhou University, Yangzhou, China
[b] School of Computer Science and Engineering, Macau University of Science and Technology, Macao Special Administrative Region of China

ARTICLE INFO

Keywords:
Smart contract
Static analysis
Vulnerability detection
Graph neural network

ABSTRACT

Context: Recently, several deep learning based smart contract vulnerability detection approaches have been proposed. However, challenges still exist in applying deep learning for fine-grained vulnerability detection in smart contracts, including the lack of the dataset with sufficient statement-level labeled smart contract samples and neglect of heterogeneity between syntax and semantic features during code feature learning.
Objective: To utilize deep learning for fine-grained smart contract vulnerability detection, we propose a security best practices (SBP) based dataset construction approach to address the scarcity of datasets. Moreover, we propose a syntax-sensitive graph neural network to address the challenge of heterogeneous code feature learning.
Method: The dataset construction approach is motivated by the insight that smart contract code fragments guarded by security best practices may contain vulnerabilities in their original unguarded code form. Thus, we locate and strip security best practices from the smart contract code to recover its original vulnerable code form and perform sample labeling. Meanwhile, as the heterogeneity between tree-structured syntax features embodied inside the abstract syntax tree (AST) and graph-structured semantic features reflected by relations between statements, we propose a code graph whose nodes are each statement's AST subtree with a syntax-sensitive graph neural network that enhances the graph neural network by a child-sum tree-LSTM cell to learn these heterogeneous features for fine-grained smart contract vulnerability detection.
Results: We compare our approach with three state-of-the-art deep learning-based approaches that only support contract-level vulnerability detection and two popular static analysis-based approaches that support fine detection granularity. The experiment results show that our approach outperforms the baselines at both coarse and fine granularities.
Conclusion: In this paper, we propose utilizing security best practices inside the smart contract code to construct the dataset with statement-level labels. To learn both tree-structured syntax and graph-structured semantic code features, we propose a syntax-sensitive graph neural network. The experimental results show that our approach outperforms the baselines.

1. Introduction

The smart contract is a program written by high-level programming languages (e.g., Solidity Dannen, 2017 or Vyper Buterin, 2018) and runs on the blockchain platform. As the smart contract has various properties, such as decentralized, unchangeable, and anonymous execution, it has been utilized in several areas (e.g., IoT, data management, or finance) (Zhou et al., 2022). However, as a program, the smart contract still suffers from vulnerabilities. Furthermore, as smart contracts often manage large amounts of cryptocurrency assets or sensitive information on the blockchain, vulnerabilities in these smart contracts can lead to serious consequences. For example, in the infamous theDAO incident (Falkon, 2017), hackers exploit a vulnerability in the smart contract code to steal approximately $70 million. Thus, the vulnerabilities become a severe threat to the security of the blockchain ecosystem. For the security of the blockchain, vulnerability detection in smart contracts has become an urgent and vital task and has received widespread attention from researchers.

As Solidity (Dannen, 2017) is the dominant programming language for smart contracts, most researchers select the smart contracts written by Solidity as the research object. To facilitate comparison and analysis,

☆ Editor: Yan Cai.

* Corresponding author.
E-mail addresses: lb@yzu.edu.cn (B. Li), tazhang@must.edu.mo (T. Zhang).

https://doi.org/10.1016/j.jss.2023.111919
Received 6 May 2023; Received in revised form 24 September 2023; Accepted 4 December 2023
Available online 14 December 2023

we also select the smart contracts written by Solidity as the research object to do the vulnerability detection for our approach. Specifically, several approaches (Luu et al., 2016; Tikhomirov et al., 2018; Brent et al., 2020; Schneidewind et al., 2020; Mossberg et al., 2019; Tsankov et al., 2018; Zhuang et al., 2020; Liu et al., 2021b,a; Zhang and Liu, 2022; Nguyen et al., 2022; Wu et al., 2021; Zhang et al., 2022a) have been proposed to detect vulnerabilities for Solidity smart contract. These approaches can be divided into two categories: pattern-based and learning-based. The pattern-based static analysis approaches utilize some technology, such as symbolic execution (Luu et al., 2016) or taint analysis (Brent et al., 2020), to explore the potential execution space of a smart contract and check the existence of some predefined vulnerability patterns inside the code. However, some complex patterns and corner cases are non-trivial to be covered by the expert-defined patterns that make these approaches error-prone and high human effort cost. To Avoid relying on expert-defined patterns, researchers have proposed several learning-based approaches for Solidity smart contracts vulnerability detection. The general process of these approaches is converting the smart contract to a specific representation (e.g., sequence Wu et al., 2021 or graph Zhuang et al., 2020) and utilizing a deep-learning model to learn vulnerability-related code patterns from a well-labeled dataset. Compared with pattern-based approaches, learning-based ones can automatically extract vulnerability patterns from prior vulnerable code instead of requiring expert involvement. However, existing learning-based approaches face several challenges when applied to fine-grained smart contract vulnerability detection. These challenges include:

Challenge 1: Lacking the Dataset with Sufficient Statement-Level Labels for Fine-Grained Smart Contract Vulnerability Detection. To achieve fine-grained smart contract vulnerability detection using learning-based approaches, a dataset with sufficient statement-level labeled smart contract samples is required to train the detecting model. However, the current dataset construction approaches, including manual-based labeling (Zhuang et al., 2020; Liu et al., 2021b,a; Wu et al., 2021; Zhang et al., 2022a), existing detection tools-based labeling (Zhang et al., 2022b; Huang et al., 2022; Zhang et al., 2022a; Durieux et al., 2020), and vulnerability injection (Ghaleb and Pattabiraman, 2020; Feng, 2021), have limitations in generating such a dataset. For instance, manual-based labeling requires a high human effort with professional knowledge for manual inspection of many smart contracts. Furthermore, the detection tools-based labeling approach cannot generate statement-level labels due to the limitations of existing detection tools. The vulnerability injection-based approach only generates the samples by templates, which cannot fully cover the various forms of a given vulnerability type. Meanwhile, the injected vulnerable code is isolated from the original smart contract code context, which may not accurately reflect real-world smart contract scenarios. As a result, these three approaches cannot balance the effort required for labeling and the quality of statement-level labels.

Challenge 2: Neglecting the heterogeneity between code syntax and semantic features during code representation learning. Prior researchers (Yamaguchi et al., 2014; Zhou et al., 2019; Wang et al., 2020) have proposed using Code Property Graph (CPG) with Graph Neural Networks (GNN) to capture the syntax and semantic features inside the code. In CPG construction (Yamaguchi et al., 2014), it merges the Abstract Syntax Tree (AST) into the Program Dependence Graph (PDG) by converting each statement's AST subtree into a graph and attaching it under the corresponding node in the PDG. Then, these approaches utilize the GNN's message-passing paradigm to propagate information between nodes following edges in the CPG for feature learning. However, all these approaches neglect the heterogeneity between syntax and semantic features of the smart contract code. More specifically, syntax features are organized as a tree structure and embodied inside the AST, while semantic features are embodied inside the relations between statements and represented as a graph structure, such as the control flow graph (CFG). Furthermore, since the tree structure is hierarchical and organized from leaf to root, unifying these two heterogeneous features into a graph structure and learning features by aggregating information from each node's neighbors without following the original leaf-to-root order is inappropriate and breaks the original tree structure of the syntax features. Thus, the current common code feature learning approach cannot fully utilize the heterogeneous syntax and semantic features inside the smart contract code.

To solve the above challenges, we propose a novel approach utilizing the Security Best Practices (SBP) to construct the dataset with sufficient statement-level labels and a Syntax Sensitive Graph Neural Network (SS-GNN) model to address the heterogeneous syntax and semantic feature learning. **To address Challenge 1**, we propose to leverage the presence of security best practices in smart contract code for dataset construction. We have observed that the smart contract community has summarized several security best practices (SBP) for protecting smart contracts from vulnerabilities. These SBPs are widely accepted by the smart contract developer community and are commonly applied in current smart contract development. Our key insight is that the statements guarded by security best practices in a smart contract code reflect the knowledge of the developers about which statements may be vulnerable and require specific security measures to protect them. Based on this insight, we propose using the knowledge embodied in statements guarded by security best practices to construct the dataset. Specifically, this dataset construction approach involves locating code statements guarded by the security best practices in smart contract code and stripping the security best practices from them to restore their original, vulnerable code form and label them as vulnerable. In contrast, code fragments without any security best practices applied will be labeled as neutral. **To address Challenge 2**, we propose a novel approach that avoids simply merging the AST into the code graph during code representation construction. Instead, during code graph construction, we replace each node's content in the graph with each statement's corresponding AST subtree to maintain the original tree structure of the syntax feature. Then, during feature learning, we propose a syntax-sensitive GNN (SS-GNN) that incorporates an additional child-sum tree-LSTM cell (Tai et al., 2015) into the GNN model to enhance the model to capture syntax features from the tree structure before performing message passing between adjacent nodes to learn semantic features. Thus, compared with the common GNN models that only capture the relations between nodes, our model can leverage the tree-structured syntax features inside each statement's AST subtree with the graph-structured semantic features in its context for smart contract vulnerability detection and named as Syntax Sensitive Graph Neural Network (SS-GNN). Moreover, we formulate the fine-grained smart contract vulnerability detection as the node classification task to train the detecting model.

We construct a dataset with 21,183 lines of vulnerable statements and 103,585 lines of neutral statements for statement-level smart contract vulnerability detection. We compare our approach against three state-of-the-art learning-based detection approaches (Zhuang et al., 2020; Wu et al., 2021; Zhang et al., 2022a) and two static analysis-based detection approaches (Tikhomirov et al., 2018; Feist et al., 2019). The result shows that our approach outperforms all baselines. We also compare our proposed SS-GNN with four widely used GNN models (Li et al., 2015; Kipf and Welling, 2016; Veličković et al., 2017; Brody et al., 2021), and the result shows that our SS-GNN outperforms these models in smart contract vulnerability detection tasks.

In summary, the main contributions of this article are as follows:

- We propose a dataset construction approach by utilizing the developers' knowledge embodied inside the security best practice to save human efforts.
- We formulate the fine-grained smart contract vulnerability detection task as a node classification task using a code graph, where each node represents a statement's AST subtree and propose a novel syntax-sensitive graph neutral network for heterogeneous code feature learning to identify suspicious vulnerable statements in smart contracts.

```
1. c = a * b
2. require(c / a == b) //SBP: multiple overflow check
3. c = a + b
4. require(c > b) //SBP: add overflow check
```

Fig. 1. Example of SBP for integer overflow.

```
1. (success, data) = Alice.call()
2. if (success) //SBP: check success or not
3.   if (data.length == 0)
4.     require(isContract(Alice)) //SBP: check contract or not
```

Fig. 2. Example of SBP for unchecked low-level call.

- We evaluate our approach, and the results show that our approach can effectively detect vulnerabilities over state-of-the-art vulnerability detection approaches at both statement and contract levels.

The rest of this article is organized as follows. In Section 2, we introduce the background of security best practices in smart contracts and explain the motivation of our approach. In Section 3, we explain the details of the dataset construction approach. Then, in Section 4, we explain our proposed vulnerability detection approach. In Section 5, we show the experiment setting and implementation of our approach. After that, we discuss the experimental results in Section 6. In Section 7, we discuss the performance and threats to the validity. Next, we discuss related studies in Section 8. Finally, Section 9 concludes this article.

2. Background and motivation

In this section, we will provide some background knowledge about smart contract vulnerabilities and their corresponding security best practices. We will also discuss the motivation behind our proposed smart contract vulnerability detection framework.

2.1. Vulnerabilities and corresponding security best practices

This section presents common smart contract vulnerabilities and the security best practices widely used to prevent them in smart contracts.

2.1.1. Integer overflow (IO)

Overflow can occur as the arithmetic operation exceeds specific length restrictions of the integer types. Meanwhile, Solidity (Dannen, 2017) is an immature program language that does not support auto overflow checking for arithmetic operations during compilation before the 0.8.0 version. Thus, it requires developers to be careful about the arithmetic operations during programming and makes inexperienced developers error-prone. As shown in Fig. 1, the common security best practice to prevent this vulnerability is adding the overflow checking after arithmetic operations.

2.1.2. Unchecked low-level call (ULC)

The low-level call API allows a smart contract to call another smart contract through the Blockchain, but this call type cannot throw the exception automatically when an error happens. Meanwhile, due to the anonymity of the blockchain network, the object of the low-level call is unknown. Thus, developers should always check the low-level call's return value and ensure the target of the low-level call is a smart contract. As shown in Fig. 2, the common security best practice checks both the return value at line 2 and the called object's type at line 4.

2.1.3. Access control (AC)

There are several risky operations in the smart contract: `delegatecall` allows a smart contract dynamically load code from another contract at run time, or `selfdestruct` allows a smart contract to destroy itself. Thus, developers should be cautious when using these risky operations. The most widely used security best practice for these risky operations is access control protection. As shown in Fig. 3, before `selfdestruct`, the contract must check the permission at line 2.

```
1. function FUN() {
2.   require(msg.sender == Owner) //SBP: access control
3.   selfdestruct(msg.sender);
4. }
```

Fig. 3. Example of SBP for access control.

```
1.function withdrawReward(address Alice) public {
2.   Lock(Alice); //SBP: mutex lock to prevent reenter
3.   require(Alice.call.value(credit[Alice])());  credit[Alice] = 0;
4.   unLock(Alice);
5.}
```

Fig. 4. Example of SBP for reentrancy.

```
1.function tokenApprove(address Bob,  uint256 addedValue) public {
2.   //SBP: check the real token value for approve
3.   uint256 realValue= ERC20(Alice).allowance(Bob);
4.   ERC20(Alice).approve(Bob, realValue + addedValue);
5.}
```

Fig. 5. Example of SBP for ERC20 transaction order dependence.

```
1.while(....) {
2.   Alice.call.value();
3.   if(gasleft() > max) //SBP: gas limit in loop
4.     {···;return;}
5.}
```

Fig. 6. Example of SBP for dos by gas limit.

2.1.4. Reentrancy (RE)

Reentrancy vulnerability refers to a situation where a malicious contract can call back into the calling contract before the first invocation is completed. This can occur when a smart contract invokes an external contract through the blockchain network. The community proposes a security best practice that adds an enter mutex lock to prevent this vulnerability. As shown in Fig. 4, this function must get the lock to prevent reentering.

2.1.5. ERC20 transaction order dependence (ERC20-TOD)

In the blockchain, users can invoke smart contracts by sending transactions. However, the order in which transactions are sent and the order in which they are executed may be different, as miners can adjust the execution order of each transaction according to their own needs. This uncertainty of execution order seriously threatens the security of ERC-20 contracts (Zhang et al., 2020). For the approve operation in ERC-20, the security best practice is to increase or decrease the allowance without directly changing it. As shown in 5, the security best practice is first to get the current allowance of Bob, then change the allowance by adding.

2.1.6. Dos by gas limit (DosGL)

Every operation executed by a smart contract requires a certain amount of computational resources named gas. The gas limit is the maximum amount of gas that can be used during the execution of a particular smart contract. If the gas limit is exceeded, the contract execution will fail, and the state of the blockchain network will be rolled back. For smart contracts with 'for' or 'while' loops, it is easy to exceed the gas limit during loop execution. As shown in Fig. 6, to prevent this vulnerability, the community recommends adding a `gasleft` check within the loop as a security best practice.

2.1.7. Unsafe recast (UR)

Downcasts are sometimes necessary when dealing with mixed integer types during arithmetic operations. Unfortunately, Solidity does not raise any errors on explicit downcasts where the destination type is not large enough to hold the value. To avoid this vulnerability, as shown in Fig. 7, the best security practice is to check the size before recasting.

```
1.function toUint128(uint256 value) returns (uint128) {
2.   require(value <= type(uint128).max);//SBP: check before under cast
3.   return uint128(value);
4.}
```

Fig. 7. Example of SBP for unsafe recast.

```
1.function withdraw(...)
2.nonReentrant {
3.   ...
4.   uint256 unusedAmount0 = _balance0().mul(shares);
5.   if (unusedAmount0 > 0) token0.safeTransfer(...);
6.   amount0 = baseAmount0.add(limitAmount0);
7.   ...
8.   _burn(msg.sender, shares);
9.  }
```

Fig. 8. The function with security best practice.

2.2. Motivation

To facilitate comprehension, we employ a representative smart contract code fragment in Fig. 8 to illustrate the motivation behind our approach. The `_deposit` function is extracted from a real-world smart contract, AlphaStrategy,[1] deployed on the Ethereum blockchain platform. The function applies three security best practices to prevent vulnerabilities, located at lines 2, 4-6. Line 2 uses the entering mutex lock security best practice `nonReentrant` to protect this function against Reentrancy vulnerability. In lines 4 and 6, the developers utilize a safe arithmetic API, `.mul()` and `.add()`, which implements overflow checking to prevent the integer overflow vulnerability. Meanwhile, in lines 5, to prevent unchecked low-level call vulnerability, the developers have applied a safe low-level call API, `.safeTransfer()`, which has implemented the security best practice of return value checking for the low-level call. From this example, we can draw the following observations:

Observation 1: Security best practices reflect developers' knowledge of vulnerabilities and can be employed as the basis for dataset Construction. The motivation example demonstrates that the developer has considered the potential vulnerabilities present in the function and employed various security best practices to safeguard against them. In line 2, the developer recognizes the potential for reentrancy vulnerability in the function and implements a precautionary measure to safeguard against it by utilizing the `nonReentrant` API. This API implements the security best practice of an entrance mutex lock, ensuring the protection of the function against any potential reentrancy attacks. In lines 6, the developer recognizes the potential integer overflow vulnerability caused by the add operation between the variable `baseAmount0` and the `unusedAmount0` and employs the `.add()` API, which incorporates the security best practice of overflow checks to ensure the safe execution of arithmetic operations. Furthermore, in line 5, the developer takes precautions to prevent the unchecked low-level call vulnerability by using the `.safeTransfer()` API, which includes built-in checks to ensure a safe low-level call. Thus, security best practices applied in the smart contract code reveal the developers' knowledge of understanding and recognition of the potential vulnerabilities in smart contracts. Without the protection of these security best practices, the original code format will suffer from vulnerabilities. Meanwhile, such as `nonReentrant` just do mutex lock checking or `.safeTransfer()` API just does return value checking. These security best practices only conduct additional checks to protect the code without changing its original semantics. Thus, by stripping these security best practices, we can construct a smart contract that contains vulnerabilities and has the same semantics as the original one. Utilizing the knowledge embodied in the security best practices can help us construct a more robust and realistic dataset by stripping the security best practices and restoring it to its original, vulnerable code format.

[1] https://etherscan.io/address/0x40C36799490042b31Efc4D3A7F8BDe5D3cB03526.

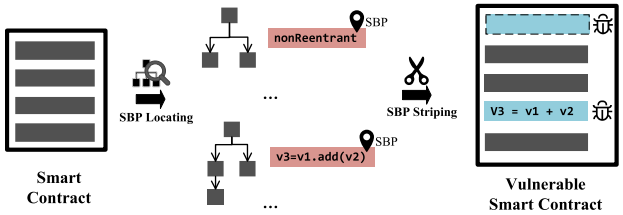

Fig. 9. The overview of security best practice based dataset construction approach.

Observation 2: Extracting various code features from different granularities helps refine the detection result. The integer overflow vulnerability introduced by the addition of `baseAmount0` and `unusedAmount0` in line 6, as seen in the motivating example, can be revealed via the local syntax features of that particular statement. Likewise, the unchecked low-level call vulnerability caused by the absence of return value checking of the external call in line 5 can be uncovered through the analysis of the contextual features of the statement, including its control and data dependencies. Thus, to detect these vulnerabilities, Extracting and learning each statement's syntax and dependency feature is necessary for the model. However, relying solely on the information embodied in individual statements and their contextual fragments may not suffice for understanding complex vulnerabilities, such as DoS by gas-limit (DosGL) or Reentrancy (RE) vulnerabilities. In the case of DoS by gas-limit vulnerability, the main reason is the excessive computational effort in a loop (e.g., for or while) that requires the detecting model to extract and learn the feature in the entire loop block. Furthermore, the Reentrancy vulnerability is caused by violating the check-effect-interaction code pattern, which requires the code to update state variables at line 6 before doing an external call at line 6. Thus, to detect this vulnerability, the model should summarize the entire function's code feature to identify the check-effect-interaction pattern. Therefore, for learning-based vulnerability detection in the smart contract, the model should support various code feature learning from different granularities, including an individual statement, loop block or entire function.

Based on the above motivations, we first utilize the first motivation to construct a more realistic smart contract dataset by stripping security best practices from the smart contract code to obtain the original vulnerable code format as the vulnerable samples in our dataset. Subsequently, inspired by the second motivation, we propose a model named Syntax Sensitive Graph Neural Network (SS-GNN) that can learn the additional tree structured syntax feature with semantic features inside the smart contract code as the vulnerability detection model.

3. Security best practice based dataset construction

To overcome the limitations of existing dataset construction approaches, we propose utilizing security best practices within smart contract code to construct the dataset for smart contract vulnerability detection. Based on our observation, seasoned smart contract developers apply various security best practices to protect vulnerable code fragments. The presence of security best practices in a code fragment signifies the developer's knowledge of the potential vulnerability in the original, no best practices unprotected format of that specific code fragment. Thus, our dataset construction approach is based on the insight that the realistic smart contract vulnerable sample can be obtained by stripping the security best practices from the original smart contract samples.

As shown in Fig. 9, it contains two main steps: (1) security best practices locating and (2) security best practices stripping. We first generated the abstract syntax tree (AST) for each smart contract to identify the presence of security best practices in the smart contract code and get their specific locations. Then, we propose a set of rules to strip these security best practices from code to obtain the original vulnerable code format. The code fragments that have been stripped of security best practices will be labeled as "vulnerable", while the others will be labeled as "neutral".

In the smart contract development community, there are three common types of applying security best practices in a smart contract:

3.1. Security best practices API calling

The first type of applying security best practices in smart contracts is calling APIs provided by various Solidity security libraries. These libraries have pre-implemented security best practices wrapped as APIs. By calling these APIs, developers can effectively secure their potentially vulnerable code fragments.

3.1.1. Locating

To locate these security best practice APIs calling in a smart contract, we need to (1) collect all callable functions that implement security best practices in one smart contract and (2) locate where they are being called. For callable SBP function collection, it aims to find all functions that implement the security best practices and can be called as an API inside this smart contract. We first scan the Abstract Syntax Tree (AST) to find nodes that satisfy the conditions that: (a) its 'nodeType' field's value is 'FunctionDefinition', (b) its 'implemented' field's value is 'true'. These nodes that satisfy the above conditions represent all callable functions within the smart contract, and we get these functions' names from the value of the "name" field in these nodes. Next, we determine which of these callable functions can be used as security best practice APIs based on their names. We verify if this function's name matches a commonly used security best practice API. If a function satisfies the name-checking, we can treat it as the SBP API. For locating the statement that calls these SBP APIs, we extract the 'id' field's value of the node as the key to search in the AST for nodes whose 'referencedDeclaration' field value is the same as the key. The node satisfying this condition will be the statement that calls some SBP API.

3.1.2. Stripping

After locating the SBP API calling in smart contract code, we will perform SBP stripping. To strip the security best practice API calling, we replace these SBP APIs with their original operations, which are not guarded by the security best practices. Thus, we can recover the original vulnerable code format without changing its original semantics. The rules of the SBP APIs with their corresponding stripped format can be seen in Table 1.

3.2. Security modifier

The second approach to applying security best practices in smart contracts is using security modifiers. A modifier is a mechanism that enables the addition of conditions or prerequisites to a function before its execution. The smart contract development community implements several security best practices based on this mechanism, such as the `onlyOwner` modifier, to protect the function from improper access control (AC) or reentrancy (RE) vulnerabilities. Therefore, by stripping these security modifiers, the non-guarded function may be susceptible to these vulnerabilities and can serve as the vulnerable sample for our dataset.

Table 1
The SBP APIs and after stripping formats.

Vulnerability	SBP	After stripping
Integer overflow	a.add(b), a.sub(b), a.mul(b)	a + b, a − b, a ∗ b
Unchecked low-level call	Bob.functionCall(), Bob.functionCallWithValue(), Bob.functionStaticCall(), Bob.safeTransfer() Token.safeTransferFrom(Bob)	Bob.call()
ERC20 transaction order dependency	Token.safeApprove(), Token.safeIncreaseAllowance(), Token.safeDecreaseAllowance()	Token.approve()
Unsafe recast	toUint64(var), toInt64(var)	uint64(var), int64(var)

Table 2
The SBP Modifiers and after stripping formats.

Vulnerability	SBP	After stripping
Reentrancy	function FUN noReentrant{ ... }	function FUN { ... }
Access control	function FUN onlyOwner { ... }	function FUN { ... }

3.2.1. Locating

Similarly, to locate these security modifiers calling in a smart contract, we need to (1) collect all callable security modifiers in one smart contract and (2) locate where they are being called. We first scan the AST for all nodes whose 'nodeType' field's value is 'ModifierDefinition' with the 'name' field's value matching one of the widely utilized security modifiers. If some nodes satisfy the above conditions, it is a security modifier. Then, we will utilize its 'id' field's value as the key to search in the AST for nodes whose 'nodeType' value is 'ModifierInvocation' and the "referencedDeclaration" field's value is the same as the key. This allows us to identify locations in the smart contract code where the security modifier is being called.

3.2.2. Stripping

After getting the location of the called security modifiers, we just directly remove it from where they are called to strip this type of security best practice. The rules of the SBP modifier with their corresponding stripped format can be seen in Table 2.

3.3. Gas check

The third type of security best practice applied in smart contracts is the gas limit check within loop blocks through the `gasleft` API. This approach ensures that the gas consumption remains within a specified limit during loop execution and halts the loop if the limit is exceeded, thus preventing the vulnerability of Dos by gas-limit (DosGL). Therefore, these code fragments incorporating this security best practice may be prone to gas exhaustion caused Dos vulnerabilities if this gas limit check is stripped, and they can be utilized as vulnerable samples to enrich the dataset.

3.3.1. Locating

To locate the gas checking inside a loop block, we first scan the AST for all nodes whose 'nodeType' field's value is 'WhileStatement', 'ForStatement', or 'DoWhileStatement'. These nodes are the root of one loop block. Subsequently, to identify and locate the gas check within the loop block, we evaluate all sub-nodes of the loop root using the following criteria: (1) its 'nodeType' field's value should be 'FunctionCall', and (2) its first child node must have a 'nodeType' of 'Identifier', and its 'name' field's value must be 'gasleft'. If any node satisfies these two criteria, it should be the location that applies this security best practice.

Table 3
The gas check SBP and after stripping format.

Vulnerability	SBP	After stripping

Dos by gas-limit	for(;;){
	if(gasleft())break;}
	}
		for(;;){
		...
		}

```
1.function withdraw(...)
2.          {
3.   ...
4.   uint256 unusedAmount0 = _balance0()*(shares);
5.   if (unusedAmount0 > 0) token0.call(...);
6.   amount0 = baseAmount0 + (limitAmount0);
7.   ...
8.   _burn(msg.sender, shares);
9. }
```

Fig. 10. The example of security best practice stripping of Fig. 8.

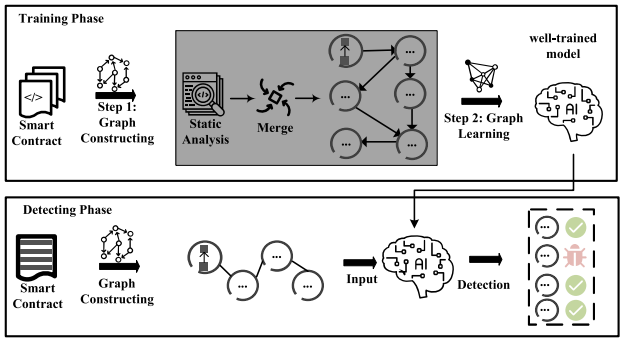

Fig. 11. The overview of our vulnerability detecting approach.

3.3.2. Stripping

Similarly, to strip this gas check SBP in code, we just remove the statement which contains the `gasleft()` API inside the loop as shown in Table 3.

We utilize the example in Fig. 10 to illustrate our proposed security best practice based dataset construction approach. The function `withdraw` is the after security best practice stripping format of the example in Fig. 8. Its original code format guarded by security best practices is shown in Fig. 8. In line 2, the security modifier `nonReentran` is removed. In line 4 and 6, the safe math API calling is replaced with the original arithmetic operation as "$*$" and "+" in the code. Similarly, the safe external API calling at line 5 is replaced by the external call, which means this statement contains the unchecked low-level call (ULC) vulnerability.

4. Vulnerability detection approach

Fig. 11 shows the framework of our proposed vulnerability detection approach. Our approach takes the smart contract source code as input and proceeds to construct code graph representations for each function within the smart contract. The detection model conducts predictions for each node within the code graph, determining whether each node is vulnerable or not as the final output. More specifically, our approach consists of two phases: the training phase and the detection phase. The training phase contains two steps: graph construction and graph feature learning. In the graph construction step, we utilize static analysis to extract the CFG, AST, and data flow of the code and also get the type information for each statement. To preserve the original tree structure of each statement's syntax feature, we merge each statement's AST subtree into its corresponding node in CFG to construct the code graph.

During the graph feature learning step, we formulate fine-grained vulnerability detection as the node classification task and propose a model named Syntax-Sensitive Graph Neural Network (SS-GNN) for feature learning. This SS-GNN model incorporates an additional child-sum tree-LSTM cell (Tai et al., 2015) and statement type-based attention mechanism with the GNN to effectively learn heterogeneous features, including tree-structured syntax features and graph-structured semantic features, from the graph. In the detection phase, we transform the smart contract code to graph with the same steps in the training phase. Then, we use the graph as the input to the well-trained detection model for vulnerability detection. For each node in the graph, if the model predicts it as "1", its corresponding statement will be the vulnerable statement.

4.1. Code graph construction

Fig. 12 is the overview of the code graph construction approach, which utilizes static analysis techniques to extract various information from the smart contract code and merge them to construct the code graph.

As shown in Fig. 12(a)–(d), during static analysis, four different pieces of information are extracted from the code. First, we use the Solidity static analysis tool, Slither (Feist et al., 2019), to generate the control flow graph (CFG), which serves as the foundation for the code graph. In the CFG, each node represents a statement, and each edge represents the control flow between statements. Then, based on the CFG, we extract the data flow between statements through each variable's define-use chain. Then, the abstract syntax tree (AST) subtree for each statement is extracted to reflect the syntax features of the code. This AST subtree is achieved using the Solidity syntax analysis tool, solc-typed-ast (ConsenSys, 2022). Finally, we obtain each statement's type information from the 'nodeType' field of its AST subtree root node. This statement's type can reflect some semantics, such as the 'ForStatement' type indicating the current statement will start a loop, and the 'VariableDeclaration' type means this statement will define some new variable. However, for statements with the type of 'Assignment' or 'FunctionCall', we will perform further analysis to identify a more exact statement type. The reason is that both 'Assignment' or 'FunctionCall' type statements may trigger risky operations in smart contracts, such as state variable assignment or calling an external contract, that may introduce vulnerabilities. Accurately identifying statement types can help the code graph better reflect these risky operations. More specifically, if a statement has the type of 'Assignment', it means it will assign a value to a variable. To determine the exact assignment type of this statement, we analyze the type of the assigned variable. If the assigned variable is a memory variable, a temporary variable cleared when the function completes, the statement type will be 'Memory Assignment'. Similarly, if the assigned variable is a state variable, which is persistent on the blockchain, the statement type will be 'State Assignment'. Meanwhile, if a statement is the type of 'FunctionCall', it means this statement will call a function. We will utilize interprocedural analysis to determine whether the called function interacts with external contracts. If the called function can interact with external contracts, the type of the statement calling this function will be 'External FunctionCall'. Otherwise, it will be 'Internal FunctionCall'.

After obtaining all four pieces of code information through static analysis, we combine them along with two types of additional virtual nodes ('v_fun' and 'v_loop') to construct the code graph. Initially, we utilize the CFG as the foundation of the code graph and add the data flow information into it. Subsequently, to preserve the tree structure of the syntax feature, we use each statement's AST subtree to replace the corresponding node's content in the CFG. Then, we incorporate each statement's type into the corresponding node in the graph. Upon merging the above four types of code information, nodes in this code graph can reflect the syntax information of statements in a tree-structured

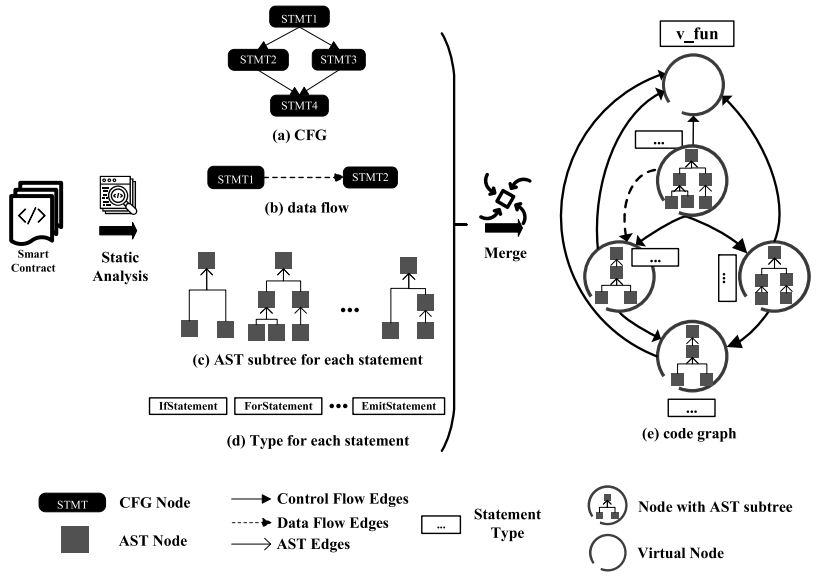

Fig. 12. The code graph construction approach.

form with the directed edges indicating the control flows and data flows between statements as graph-structured.

Based on this constructed code graph, we introduce two types of virtual nodes, v_fun and v_loop, to connect various scopes of nodes inside the graph. Notably, all the v_fun and v_loop type nodes have no contents. Specifically, the v_fun node is linked to all nodes inside the CFG through directed edges from all CFG nodes to this v_fun node. Similarly, the v_loop type node connects all nodes within one loop block via directed edges from nodes inside the loop block to this v_loop node. The primary motivation for adding these virtual nodes is to facilitate the GNN model to collect code features from the scopes of the entire function or a loop block based on the GNN's message-passing paradigm. As our proposed approach aims to get a fine-grained detecting result by formulating vulnerability detection as a node classification task, the model should gather vulnerability-related features inside one node. While the nodes in this graph only capture the syntax and semantic information of individual statements, it is insufficient to identify vulnerabilities such as Dos or Reentrancy vulnerabilities, which require code features from a larger scope, such as a whole loop block's feature or an entire function's feature to detect them. By adding these virtual nodes, different scopes of nodes inside the graph can be connected to make the model gather features from a more extensive scope, enhancing its ability to detect vulnerabilities that require features from the entire function or loop block.

As shown in Fig. 12(e), each node in the code graph has the content of a tree with the additional statement type, and the edges represent the control flow or data flow between statements. Additionally, there is a v_fun type virtual node that connects with all other nodes of the graph with no content inside.

4.2. Graph feature learning

After constructing the code graph, we formulate fine-grained vulnerability as the node classification task and utilize the SS-GNN model to learn the tree-structured syntax features and graph-structured semantic features from the graph to identify potentially vulnerable nodes.

Our proposed graph defines each node's content as a combination of one statement's corresponding AST subtree and its statement type. Specifically, we denote our graph as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of all nodes in the graph and $\mathcal{E}$ represents the set of all edges between statements. Meanwhile, in our graph, the content of each node includes the current node's statement type and its corresponding AST subtree. Thus, the ith node in the graph $v_i \in \mathcal{V}$ denote as $v_i = (T_i, type_i)$, where T_i represents the AST subtree for the ith node and $type_i$ represent ith node's statement type. The jth tree node in the AST subtree T_i is denoted as u_j^i where $u_j^i \in T_i$.

4.2.1. Node embedding initialize

As the acceptable input form for a DL model is low-dimensional vectors, we first initialize node embedding as the input to our model. For our graph, we utilize the pre-trained program language model InferCode (Bui et al., 2021) to obtain the initial embedding vector for all nodes in statements' AST subtrees and also get the embedding for each statement's type. For one graph node $v_i = (T_i, type_i)$ that $v_i \in G$, the embedding vector of the statement type is denoted as h_i^{type}. Meanwhile, for the jth node in AST subtree T_i as $u_j^i \in T_i$, its initial embedding vector is denoted as $z_j^{(0)}$.

4.2.2. Feature learning

After obtaining initial embeddings for all nodes in each statement's AST subtree and the statement type, we employ a GCN with a statement type-based attention layer and a child-sum tree-LSTM cell to learn features from our proposed code graph. The child-sum tree-LSTM cell conducts syntax feature learning and serves as the shared layer for each AST subtree. Then, the GCN with the statement type-based attention will conduct semantic feature learning between each statement. As shown in Fig. 13, our proposed model contains three components as: (1) a child-sum tree-LSTM cell (Tai et al., 2015) for syntax feature learning, (2) the statement type-based attention layer for attention score calculation, and (3) the GCN layer for semantic feature learning.

For each node $v_i \in V$, its hidden feature h_i contains two parts including syntax feature h_i^{syn} and semantic feature h_i^{sem} as $h_i = [h_i^{syn}, h_i^{sem}]$. Specifically, during the graph feature learning, the features h_i^{syn} and h_i^{sem} of a statement node i can be updated as follows:

$$h_i^{sem} = \sum_{j \in \mathbb{N}(i)} \left[\delta \left(\varphi \left(h_i^{type}, h_j^{type} \right) W h_j^{syn} \right) \right] \tag{1}$$

where the $\mathbb{N}(i)$ denotes the set of neighbors of node i. The $\delta(\cdot)$ is the activation function. The h_i^{type} and h_j^{type} are the statement type embedding vectors for nodes i and j, which are obtained during node embedding initialization. The $\varphi(\cdot)$ is the attention function, which calculates the attention score based on the statements' type. The W is the learnable weight of the GCN layer. And h_j^{syn} represent the syntax

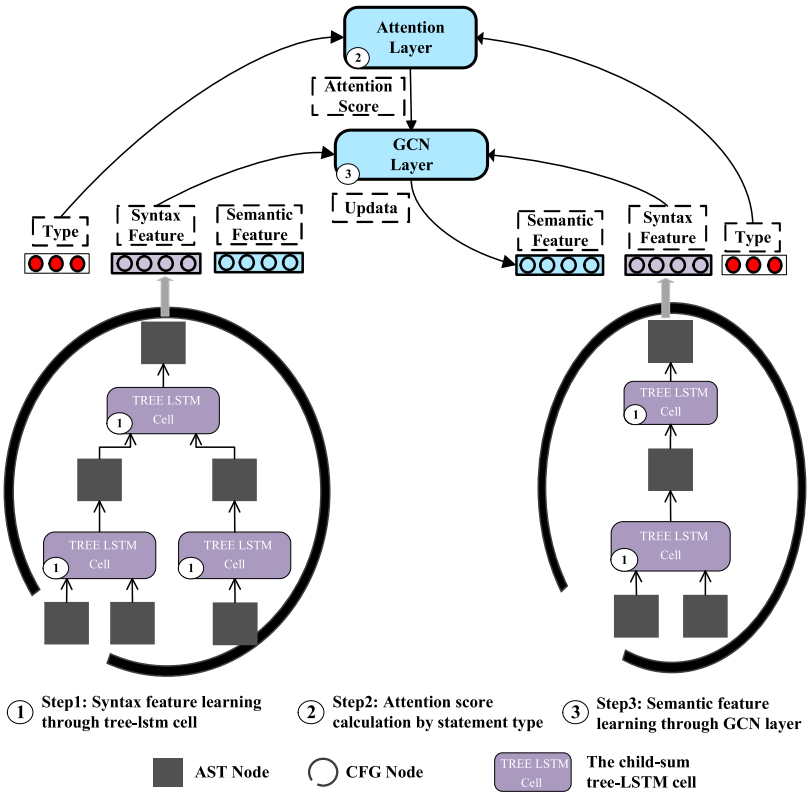

Fig. 13. The example of message-passing paradigm based graph feature learning between two nodes.

feature of node i, which is learned by a child-sum tree-LSTM cell, which is denoted as:

$$h_j^{syn} = \phi\left(T_j, r_j\right) \tag{2}$$

where, the $\phi(\cdot)$ is the child-sum tree-lstm cell, T_j is the AST subtree for node j and r_j is the root node of the tree T_j.

Compared to the standard LSTM, which updates its input gate, output gate, and memory cell based on the current input and previous hidden state, the child-sum tree-LSTM updates these gates and memory based on the hidden features of all child nodes inside the tree and learns each node's feature by summing all its sub-nodes hidden features following the leaf-to-root order (Tai et al., 2015). Therefore, the hidden feature of the root node contains the hidden features from all its sub-nodes and can reflect the syntax feature of the current statement. Specifically, for the kth node in the AST subtree T_i, the child-sum tree-LSTM cell based syntax feature learning $\phi(\cdot)$ is defined as:

$$\begin{aligned}
i_k &= \sigma(W_i[z_{k-1}, \frac{1}{|C_k|}\sum_{j\in C_k} c_j] + b_i)\\
f_{kj} &= \sigma(W_f[z_{k-1}, c_j] + b_f)\\
o_k &= \sigma(W_o[z_{k-1}, \frac{1}{|C_k|}\sum_{j\in C_k} c_j] + b_o)\\
u_k &= \tanh(W_u[z_{k-1}, \frac{1}{|C_k|}\sum_{j\in C_k} c_j] + b_u)\\
c_k &= i_k u_k + \sum_{j\in C_k} f_{kj} c_j\\
z_k &= o_k \tanh(c_k)
\end{aligned} \tag{3}$$

where k is the index of a node in the AST subtree, C_k is the set of child nodes of node k in the AST subtree. The z_k is the hidden state of the kth node in the AST subtree. The c_k is the cell state of node k. The i_k, f_{kj}, and o_k are the input, forget, and output gates, respectively, and u_k is the new candidate cell state. The W_i, W_f, W_o, W_u and b are learnable weight and bias parameters of the child-sum tree-LSTM cell, and σ and tanh are the sigmoid and hyperbolic tangent activation functions, respectively. Finally, the syntax feature of the AST subtree T_j can be extracted from its root node r_i and denoted as $h_j^{syn} = h_{r_j}$.

After getting each node's syntax feature, we utilize the statement type between each node to calculate the attention weight. For one node i and its neighboring node $j \in \mathbb{N}(i)$, the attention weight calculation $\varphi(\cdot)$ is defined as:

$$\varphi\left(h_i^{type}, h_j^{type}\right) = \frac{1}{K}\sum_{k=1}^{K} \alpha_{i,j}^k \tag{4}$$

where K is the number of attention heads, and $\alpha_{i,j}^k$ is the attention weight of the kth head for nodes i, j when computing the representation of node i. The attention weight is computed using a shared attention mechanism, which can be defined as:

$$\alpha_{i,j}^k = \frac{\exp\left(\text{LeakyReLU}\left(\vec{a}^{k\top}[W h_i^{type} | W h_j^{type}]\right)\right)}{\sum_{j\in\mathbb{N}(i)} \exp\left(\text{LeakyReLU}\left(\vec{a}^{k\top}[W h_i^{type} | W h_j^{type}]\right)\right)} \tag{5}$$

where $\vec{a}^k$ is a learnable attention parameter for the kth head, | denotes the concatenation operation, and LeakyReLU is the activation function.

4.2.3. Node classification

As shown in Fig. 14, after successfully extracting all node embeddings in the graph, we utilize a multilayer perceptron (MLP) as the classifier, concatenating each node's syntax and semantic features as input to identify whether the statement is vulnerable or not. For the node i, it final predicted label $\widetilde{y}_i$ denote as:

$$\widetilde{y}_i = MLP\left(h_i^{syn} | h_i^{sem}\right) \tag{6}$$

If the value of y_i is '1', it indicates that the corresponding statement for node i is vulnerable. In contrast, if the value of y_i is '0', the corresponding statement for node i is considered non-vulnerable.

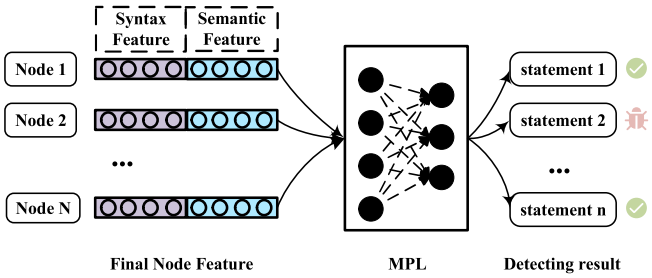

Fig. 14. The statement-level vulnerability detection.

4.3. Vulnerability detection

In the detection phase, we apply the well-trained model to detect potentially vulnerable statements in the smart contract code. Specifically, similar to the training phase, we first construct the code graph with the same steps in the training phase as static analysis and feature merging. After constructing the code graph for the target smart contract, we will use these graphs as the input to feed into the well-trained detection model. For all graphs that come from one smart contract code, if one node of these graphs is identified by the well-trained model as containing a potential vulnerability, this statement will be identified as a vulnerable one.

5. Experimental

5.1. Research questions

To evaluate our approach, we design experiments to answer the following research questions:

RQ1: How effective is our approach for detecting vulnerabilities in Solidity smart contracts compared with existing state-of-art smart contract vulnerability detection approaches?

The studies most relevant to our approach are deep learning-based approaches. However, most existing learning-based approaches only evaluate their performance in reentrancy vulnerability detection tasks for Solidity smart contracts at the coarse-grained level in their research papers. Thus, for fairness, in this research question, we aim to evaluate the performance of our approach compared to state-of-the-art approaches for reentrancy vulnerability detection at the contract level, which is the same as the deep learning-based baselines in their paper.

RQ2: How effective is our approach for detecting vulnerabilities at a fine-grained level for Solidity smart contracts?

As our approach aims to get a fine-grained vulnerability detection result for smart contracts, this research question aims to evaluate the ability of fine-grained level vulnerability detection compared to other tools that support fine-grained vulnerability detection for smart contracts.

RQ3: How effective is our proposed detecting model in fine-grained vulnerability detection compared with other common GNN models?

One of the key contributions of our approach is our proposed model, which utilizes a child sum tree-LSTM cell to enhance the GNN model for learning heterogeneous code features. Through this RQ, we investigate whether our proposed model is more effective for fine-grained vulnerability detection in smart contracts in comparison with other popular GNN models.

5.2. Experiment setup

5.2.1. Dataset

We construct a Solidity smart contract dataset with 21,183 lines of statements labeled vulnerable and 103,585 lines of statements labeled neutral. The Solidity (Dannen, 2017) is a rapidly evolving programming language and has evolved from 0.1 version to 0.8 version. Meanwhile, there is an incompatibility between these versions. Thus, the older versions of the Solidity language are abandoned by the community, and most of the current smart contracts are implemented by Solidity later than version 0.5. Thus, to construct a dataset that can reflect the real situation of current smart contract development, we utilize the API provided by the Etherscan platform to collect 6000 open-source smart contracts from the Ethereum blockchain, which are implemented with the Solidity version from 0.5 to 0.8. Then, we use our proposed security best practice based dataset construction approach to construct a dataset for smart contract vulnerability detection. The statement which applied the security best practice will be conducted SBP stripping and labeled 'vulnerable' and others will be labeled 'neutral'. The smart contracts of our dataset are available at https://github.com/smartcontract-detect-yzu/sbp_dataset.

5.2.2. Baseline

To answer the RQ1, we selected three state-of-the-art Depp learning-based smart contract vulnerability detection approaches as TMP (Zhuang et al., 2020), Peculiar (Wu et al., 2021) and CBGRU (Zhang et al., 2022a) with two widely used pattern-based tools Smartcheck (Tikhomirov et al., 2018) and Slither (Feist et al., 2019) as the baselines. TMP and Peculiar are both code graph representation learning based approaches that transfer the smart contract code to one graph with a GNN model for vulnerability detection. In TMP (Zhuang et al., 2020), it converts a contract to a simplified control flow graph and then utilizes a deep learning model named temporal message propagation network (TMP) for vulnerability detection. For Peculiar (Wu et al., 2021), it converts a contract to a crucial data flow graph and puts it into the GraphCodeBERT (Guo et al., 2020) for vulnerability detection. CBGRU (Zhang et al., 2022a) treat the contract code as plain text and utilize a combination of LSTM, GRU, and CNN for vulnerability detection. Both Smartcheck (Tikhomirov et al., 2018) and Slither (Feist et al., 2019) are pattern-based static analysis tools for smart contract vulnerability detection. The Smartcheck (Tikhomirov et al., 2018) converts AST to an XML parse tree as an intermediate representation and then checks pre-defined vulnerability patterns on this intermediate representation. The Slither (Feist et al., 2019) represents a contract code as the CFG based on a specific intermediate representation language named SlithIR and checks pre-defined vulnerability patterns in the CFG.

To answer the RQ2, we utilize the Smartcheck (Tikhomirov et al., 2018) and Slither (Feist et al., 2019), which both of these tools support smart contract vulnerability detection at the statement level, as the baselines to compare with us in fine-grained vulnerability detection.

To answer the RQ3, we chose four popular GNN models, including GGNN (Li et al., 2015), GCN (Kipf and Welling, 2016), GAT (Veličković et al., 2017), GATv2 (Brody et al., 2021), as the baselines to compare with our proposed model.

5.2.3. Implementation

We construct and analyze the smart contract abstract syntax tree on the solc-typed-ast (ConsenSys, 2022) with 1042 lines of typescript code. Then, we implemented the smart contract code graph constructor by Slither (Feist et al., 2019) and Networkx (Hagberg et al., 2008) with 3162 lines of Python code. Our model is implemented by PyTorch and DGL (Wang, 2019). Our experiments were performed with the Nvidia Tesla T40 GPU, installed with Windows 10 and CUDA 11.3. The embedding size for each node is 100; the batch size is 1024; the optimizer is ADAMAX; the size of the hidden feature vector size of each node is 64; the learning rate is initialized by the linear warmup strategy; the dropout is 0.1.

Table 4
Capability of the different approaches in Reentrancy vulnerability detection at contract-level.

Reentrancy				
Approach	A (%)	R (%)	P (%)	F1 (%)
Slither	41.24	54.71	37.58	44.55
Smartcheck	40.11	72.51	34.81	47.03
TMP	75.25	66.29	47.59	55.41
Peculiar	74.46	50.82	63.47	56.45
CBGRU	65.95	67.07	69.59	68.31
Ours	***95.47***	***89.77***	***91.40***	***90.57***

5.2.4. Evaluation metrics

Evaluation metrics. We apply the following four widely used evaluation metrics to measure the effectiveness of our approach and the other competitors.

True Positive (TP). The number of cases correctly identified as positive by a model out of all the actual positive cases.

False Negative (FN). The number of actual positive instances that are incorrectly classified as negative by the model.

Accuracy (A). The proportion of all samples that are detected correctly. It is calculated as $A = \frac{TP+TN}{TP+TN+FP+FN}$.

Recall (R). The proportion of correctly detected vulnerable samples to all vulnerable samples. It is calculated as $R = \frac{TP}{TP+FN}$.

Precision(P). The proportion of correctly detected vulnerable samples to those detected to be vulnerable. It is calculated as $P = \frac{TP}{TP+FP}$.

F1-Score (F1). A score that measures the overall effect by considering precision and recall. It is calculated as $F1 = 2 * \frac{R*P}{R+P}$.

6. Experimental result

6.1. Performance comparison in coarse-grained vulnerability detection: Answer to RQ1

Table 4 shows the results of each vulnerability detection approach in terms of evaluating the metrics mentioned above for Reentrancy vulnerability detection at the contract-level. In general, our approach outperforms all other learning-based baseline approaches and pattern based baseline approaches. More specifically, our approach achieves a recall of 89.77%, improving the state-of-the-art by 23.80%, and a precision of 91.40%, improving the state-of-the-art by 31.34%. To the F1, ours achieves 90.57% and improves the state-of-the-art by 32.58%.

Compared to the pattern-based detection approaches (Slither and Smartcheck), our proposed approach outperforms them. The reason is that Slither and Smartcheck rely on pre-defined code patterns to detect reentrancy vulnerability, which may be simplistic or inflexible to cover all possible vulnerability scenarios. For detecting reentrancy vulnerabilities, both Slither and Smartcheck first locate external calls within a function and then check for state change operations after the external call. However, due to the complexity of external calls in smart contracts, the rules utilized by both Slither and Smartcheck cannot cover all possible situations. As the example, which is missed by Slither and Smartcheck, demonstrated in Fig. 15, the function `bid`, which comes from the Sales721 contract,[2] performs state variable changing (effect) at line 31-32 after invoking the external call interface `payToken.safeTransfer` (interaction) at line 23, violating the recommended pattern of effect before interaction and may potentially lead to reentrancy vulnerability.[3] However, the `payToken.safeTransferFrom` is just declared in this Sales721 contract, and its implementation is in the external contract at the address of

```
1. function bid(uint _saleID, uint _bidPrice) external {
2.   ...
17.  if (auction.highestBidPrice != 0) {
18.     ...
23.     payToken.safeTransfer(auction.highestBidder, auction.highestBidPrice);
24.  }
25.  ...
31.  auctions[_saleID].highestBidPrice = _bidPrice;
32.  auctions[_saleID].highestBidder = msg.sender;
33.  ...
39.}
```

Fig. 15. The reentrancy vulnerability sample missed by baselines.

`payToken`. Thus, the unavailable implementation of this interface restricts the static analysis tools from identifying the external call and makes them fail to detect the reentrancy vulnerability in this function. However, as `safeTransfer` is a common API name for the external call, our approach utilizes the deep learning model to learn this potential external call semantics from a large corpus of smart contract samples, avoiding the limitations of pattern-based approaches.

Compared to the learning-based detection approaches (Peculiar, TMP, and CBGRU), ours still outperforms these. The main reason is that all these approaches will miss syntax or partial semantics information inside the smart contract code during code representation learning. For CBGRU, it converts the entire smart contract into a token sequence with an LSTM and a CNN model for feature learning. This token sequence representation learning based approach may omit syntax information within each statement and relations between statements. However, syntax information is essential for identifying state variable assignments (effect) or external calls (interaction), and relations between statements can reflect the order of state variable assignment and external call, which are both critical for reentrancy vulnerability detection. Thus, missing this information can restrict the reentrancy vulnerability detection ability of CBGRU. For the Peculiar and TMP, they both convert the smart contract code to a graph, which can preserve the relations between statements. However, their proposed code graphs still lose some critical information inside the code. In Peculiar, the code graph is constructed based on the data flow information, and each node inside its graph only represents one variable in the code. Thus, this approach can only capture partially the state variable changing related features inside the code but missing the interaction operation with the relation between interaction and state variable in their code graph, which means the statement at line 23 in example 15 is absent in their graph. Thus, this data flow related code graph makes Peculiar hard to learn all necessary features from such a one-side code representation approach. For TMP, its proposed normalized contract graph is constructed based on the CFG of smart contracts, with several rules to simplify it. During CFG simplification, TMP only preserves the nodes containing external calls or state variables in the CFG and merges these nodes to normalize the CFG, which means the statements at lines 23, 31-32 in Fig. 15 will be contained in a single node in the normalized CFG. Although node merging can reduce the graph size, it also eliminates the control flow information between statements, which is crucial for reentrancy vulnerability detection. Meanwhile, similar to the limitation of static analysis tools, TMP also uses a pre-defined rule to identify nodes with external calls in the CFG, which may not cover all possible situations and could erroneously delete the external call node in CFG. As a result, this oversimplified graph may limit TMP's ability to detect reentrancy vulnerabilities. For our approach, it represents the code of the smart contract by merging each statement's AST subtree into CFG, which can comprehensively capture both syntax embodied in each statement' AST and relations between statements and make our approach outperform these learning-based approaches.

> **Answer to RQ1**: *Compared with other popular learning-based and pattern-based approaches, our proposed heterogeneous code features learning-based approach achieves better results in Reentrancy vulnerability at the contract level.*

[2] https://etherscan.io/address/0x9f96a3cf403b116de4e44737b823fa119e023d27.

[3] https://docs.soliditylang.org/en/v0.4.21/security-considerations.html#re-entrancy.

Table 5
Capability of the different approaches in vulnerabilities detection at statement-level.

Approach	A (%)	R (%)	P (%)	F1 (%)
Slither	90.83	44.48	47.21	45.83
Smartcheck	92.63	25.90	71.17	37.97
Ours	***96.95***	***90.31***	***91.22***	***90.76***

Table 6
Vulnerability detection capability of the different approach for various vulnerabilities at statement-level.

Vulnerability name	Ours *R (%) (TP/FN)*	Slither *R (%) (TP/FN)*	Smartcheck *R (%) (TP/FN)*
IO	89.62 (570/66)	NA	NA
ERC2O-TOD	91.13 (370/36)	NA	NA
RE	89.77 (404/46)	21.77 (98/352)	15.77 (71/379)
UR	97.42 (190/5)	NA	NA
ULC	96.42 (297/11)	91.23 (281/27)	52.27 (161/147)
DosGL	95.61 (87/4)	30.76 (28/63)	2.19 (2/89)
AC	75.75 (50/16)	0 (0/66)	4.54 (3/63)

```
1.function timeLockERC721(address recipient, ...) public{
8.    bytes32 key = keccak256(abi.encodePacked(...));
15.   timelockERC721s[key] = TimelockERC721({
16.     recipient: recipient,
19.    ...
20.   });
22.   timelockERC721Keys[nftContract].push(key);
26.}
```

(a) Example of time lock setting lack of Access Control

```
1.function setVaultConfig(address _vault, ...) external {
8.    vaultConfig[_vault] = VaultConfig(...);
9.     IERC20 crvToken = IERC20(IYearnVaultV2(_vault).token());
10.   _checkApprove(USDC.approve(...);
11.   _checkApprove(crvToken.approve(...));
12.   ...
14.}
```

(b) Example of token approve checking lack of Access Control

Fig. 16. The examples of access control vulnerability missed by baselines.

6.2. Performance comparison in fine-grained vulnerability detection: Answer to RQ2

Table 5 shows the evaluation results of our proposed approach and baselines for detecting vulnerabilities at the statement level based on the aforementioned metrics. Overall, our proposed approach demonstrates superior performance compared to the baselines in terms of fine-grained vulnerability detection at the statement level based on the evaluation metrics. Additionally, Table 6 illustrates the recall performance of our approach and baselines for different vulnerabilities. The result shows that our approach can detect more vulnerable statements with different vulnerability types than baselines.

Firstly, regarding vulnerabilities not supported by the baselines, such as integer overflow (IO), ERC20 transaction order dependency (ERC20-TOD), and unsafe recast (UR), our approach achieves satisfactory results. Specifically, our approach detects 570 out of 636 lines of statements containing the integer overflow vulnerability, 370 out of 406 lines of statements containing the ERC20 transaction order dependency vulnerability, and 190 out of 195 lines of statements containing the unsafe recast vulnerability.

Secondly, we observed that the baseline methods did not achieve satisfactory results in detecting access control vulnerability (AC) compared to our proposed approach. The main reason is that these two baselines do fine-grained vulnerability detection based on pre-defined code patterns. However, due to the complexity of this vulnerability, fixed pre-defined rules are insufficient for accurate detection. For example, Fig. 16 depicts two vulnerable functions, namely `timeLockERC721` from the Visor contract[4] and `setVaultConfig` from YearnVaultInstantRebindStrategy.[5] Both of these functions utilize the 'onlyOwner' security best practice to enforce access control in their original implementation, and we have stripped this security best practice and included them as vulnerable samples in our dataset. Specifically, the `timeLockERC721` function is implemented to set a time lock to limit the frequency of external calls from other users, which must be only configured by the owner of the contract based on its available processing power. And `setVaultConfig` function is implemented to set the amount of certain tokens in the vault for future token transactions, which needs to be strictly controlled and can only be performed by the owner of the vault. From this example, we can observe that the operations that require access control in these two functions are diverse. The timeLockERC721 function performs state variable assignments, whereas the setVaultConfig function makes ERC20 API calls. This indicates that the operations that require access control are diverse and strongly dependent on the specific functionality of the contract. As a result, it is challenging to define a set of fixed rules that can cover all such scenarios. However, our proposed approach constructs the smart contract as a graph with additional virtual nodes and utilizes a deep learning model to learn potential code scenarios requiring access control through big data to avoid relying on pre-defined fixed code patterns.

For reentrancy (RE) and dos by gas limit vulnerabilities, our proposed approach still detects more than baselines. The main reason is that these baselines do not fully utilize the semantic features within the smart contract code during vulnerability detection. For example, the function `process` in Fig. 17 from the BULLFIGHT contract[6] contains the dos by gas limit (DosGL) vulnerability but was not detected by the baselines. This vulnerability arises due to the external call inside the loop at line 27 in function `process`. Furthermore, as the actual external call operation `.transfer()` is at line 5 inside the `_withdrawDividendOfUser` function, to identify this external call should extract the interprocedural semantic feature from the code which is neglected by baselines and make the baselines fail to detect this sample. However, in our proposed approach, we conduct interprocedural analysis during code graph construction to determine the type for each statement that makes our approach identify the statement at line 27 in function `process` as the 'External FunctionCall'. Thus, our proposed code graph construction approach can better utilize the syntax and semantic features inside the smart contract code than baselines for vulnerability detection.

> **Answer to RQ2**: *Compared with baselines that support fine-grained vulnerability detection, our proposed approach achieves better results for statement-level vulnerability detection.*

6.3. Comparison with other graph neural networks: Answer to RQ3

Table 7 presents the statement-level vulnerability detection result of different common GNN models and our proposed model. In general, our proposed model outperforms all other common baseline GNN models. Specifically, our approach achieves a precision of 90.31%, improving the state-of-the-art by 4.41%, and a recall of 91.22%, improving the

[4] https://etherscan.io/address/0x17cc4e5267e1482df05e23888f5b02d05acd2f79.

[5] https://etherscan.io/address/0x53edc5519464559b1c1aad384f188e149c3e36d4.

[6] https://etherscan.io/address/0xdd267a0eeafa48b796403705b75dab614d44216.

```
1.function process(uint256 gas) public ...{
2.   ...
17.   while(iterations < numberOfTokenHolders) {
18.     ...
27.     if(processAccount(payable(account), true))
28.     {...}
41.   }
42.   ...
46.}

1.function processAccount(...) {
2.  uint256 amount = _withdrawDividendofUser(account);
3.  ...
4.}

1.function _withdrawDividendofUser(...) {
2.  ...
5.  bool success = IERC20(...).transfer(...);
6.}
```

Fig. 17. The dos by gas limit vulnerability sample missed by Smartcheck and Slither.

state-of-the-art by 5.51%. Additionally, our approach achieves an F1 score of 90.76%, improving the state-of-the-art by 4.94%.

There are mainly two reasons for this. First, our model utilizes a child-sum tree-LSTM cell to learn each statement's syntax feature through its corresponding AST subtree, which is essential for statement-level vulnerability detection. To detect vulnerabilities such as integer overflow (IO) or unsafe recast (UR), the most relevant features are the arithmetic operations or variable recasting type inside each statement, which are both syntax features and can be reflected in the content of the AST node. All of these popular baselines GNN models lack the ability to learn both syntactic and semantic features of heterogeneous structures such as trees and graphs. These baseline models treat each node's content as the token sequence of each statement and do feature learning just by gathering features from its neighboring nodes, which may neglect the syntax feature of each statement embodied inside the AST. In contrast, our proposed model effectively captures syntax features using a child-sum tree-LSTM cell from each statement's AST subtree, resulting in improved vulnerability detection performance. The second reason why our proposed approach outperforms other baseline GNN models is due to our attention score calculation method, which focuses on each statement's type rather than the content of the entire node. Specifically, both gate updating in GGNN (Li et al., 2015) and attention score calculation in GAT (Veličković et al., 2017) or GATv2 (Brody et al., 2021) are based on the contents of two neighboring nodes, which can be influenced by irrelevant content such as variable names or function names inside statements' content. As shown in Fig. 16(a), the line 15 conducts the state variable assignment, which should be carefully protected and may cause access control vulnerability. Solely from the content of this statement, it is hard for the model to identify this statement as conducting state variable assignment. However, our approach avoids this issue by focusing on the statement type, resulting in more accurate vulnerability detection. In our approach, we extract the type information for its AST subtree through static analysis for each statement as the basis for attention that facilitates the model to correctly identify the type of current statement to improve vulnerability detection performance at the statement level.

> **Answer to RQ3**: *Our proposed model, which enhances GNN with tree-LSTM for heterogeneous feature learning, achieves better results for statement-level vulnerability detection than the popular GNN models in baselines.*

7. Discussion

7.1. Performance analysis

In this paper, for deep learning-based fine-grained smart contract vulnerability detection, our approach can outperform baselines as ours can preserve the heterogeneity between tree-structured syntax features and graph-structured semantic features inside the code during feature learning, which is neglected by most related studies. For constructing a dataset with statement-level labeled smart contract samples, we propose utilizing the security best practices inside the smart contract code as the basis for sample labeling, which avoids relying on high human efforts and can reflect real-world smart contract scenarios.

Table 7
Capability of the different GNN models in vulnerability detection at statement-level.

Approach	A (%)	P (%)	R (%)	F1 (%)
GCN	84.26	55.04	22.60	32.46
GGNN	94.98	85.61	83.46	84.52
GAT	93.07	80.01	77.10	75.52
GATv2	95.56	86.49	86.45	86.48
Ours	***96.95***	***90.31***	***91.22***	***90.76***

7.2. Threats to validity

External Threats: The main external threat to our approach is the generalizability of our proposed dataset construction approach. As our proposed dataset construction approach utilizes the security best practices (SBP), these vulnerabilities whose corresponding security best practices are still not summarized by the smart contract development community will not be identified by our proposed dataset construction approach and make our approach hardly detect these types of vulnerabilities. Specifically, we summarize seven security best practices with corresponding vulnerabilities in this work and may miss some corner cases. This limitation restricts our approach to only collecting samples with these seven types of vulnerabilities in our dataset. Our dataset construction approach relies on the security best practices for dataset construction. However, not all vulnerabilities in the current Solidity smart contract ecosystem have mature and universally accepted corresponding security best practices for protection, such as the new coming vulnerabilities or function-specific vulnerabilities. For these vulnerabilities, without mature and universally accepted corresponding security best practices, our dataset construction approaches cannot be suitable. However, since our proposed dataset construction approach is based on the abstract syntax tree (AST), which is a generic representation of smart contract code, it is convenient to extend our approach to support new security best practice related vulnerabilities by providing APIs.

Internal Threats: The main internal threat to our approach is our imperfect security best practices summarized. In this work, we manually summarize the security best practices from the community, which may not cover all situations. Unperfect security best practice collection for some vulnerabilities may make our approach mislabel some samples that contain unidentified security best practices.

Construct Threats: Threats to construct validity in this paper mainly relate to the selection bias of the baselines and the performance measures. In our approach, we combine AST with CFG for code graph representation construction. Although the AST and CFG are the most commonly used code representation approaches, there are still some other code representations, such as the value-flow graph or the program dependence graph, that can reflect partial features inside the smart contract code. Meanwhile, we utilize four performance measures, accuracy, recall, precision, and F1, to evaluate the vulnerability detection performance. Although these four measures have been widely used in previous studies, there are other occasionally used measures, such as G-mean, also have been used in several previous studies. Therefore, these threats exist in this study.

Conclusion Threats: The conclusion threat to our approach relates to the generalization of our dataset. Our dataset is constructed from 6000 smart contracts implemented in the Solidity language, covering versions 0.5 to 0.8. While we have included more recent smart contracts to our dataset for model training, it is still quite possible

that the selected smart contract samples are imperfect and may miss some corner cases. Furthermore, the ecology of Solidity smart contracts is rapidly evolving. Syntax updates and the emergence of new vulnerabilities are common occurrences. Our dataset may miss some new syntax features and not introduce the new vulnerability types. Moreover, our approach heavily relies on security best practices. For those vulnerability types with no established and widely used security best practices, the detection results of our approach may not be so satisfactory.

8. Related work

Smart contract vulnerability poses a serious threat to the security of the blockchain ecosystem. Existing Smart contract vulnerability detection approaches can be divided into the following categories:

8.1. Static analysis

Static analysis is a widely used technology that aims at checking for pre-defined vulnerable code patterns in the bytecode or high-level code of a smart contract. Smartcheck (Tikhomirov et al., 2018) proposes to utilize the XML parse tree to represent the smart contract code with pre-defined rules for vulnerability detection. Slither (Feist et al., 2019) utilizes an intermediate language named SlithIR to construct the CFG of the smart contract code and search pre-defined vulnerable code patterns in CFG. Sereum (Rodler et al., 2018) proposes to utilize the taint analysis to monitor data flows between each state variable based on the bytecode to detect reentrancy vulnerability. SoliDetector (Hu et al., 2023) transforms the smart contract code into the knowledge graph and makes vulnerable code patterns searching by SPARQL query.

Since all these approaches depend on pre-defined vulnerable code patterns, they may not be flexible enough to cover complex vulnerability patterns. In contrast, ours utilizes deep learning to extract potentially vulnerable code features from big data without pre-defined patterns.

8.2. Symbolic execution

Symbolic execution is a widely used approach in smart contract vulnerability detection that involves creating a virtual execution environment for the target contract and abstracting its inputs into symbolic values. The symbolic execution engine then uses an SMT solver to solve path constraints and explore program branches to the fullest extent possible. Oyente (Luu et al., 2016) is the first symbolic execution tool with Z3 SMT to detect four types of vulnerabilities: transaction ordering dependency, reentrancy, and unchecked exceptions. Mythril (ConsenSys, 2021) combines symbolic execution, SMT solving, and taint analysis to detect various security vulnerabilities in a smart contract. Securify (Tsankov et al., 2018) is a security analysis tool that utilizes symbolic execution to analyze the contract's dependency graph to extract precise semantic information from the code with compliance and violation patterns checking for vulnerability detection.

Since symbolic execution-based approaches detect vulnerabilities following the control flow, they may neglect the syntax feature inside the code. In contrast, our vulnerability detection approach relies on syntax and semantic code features inside the code graph.

8.3. Deep learning

Deep learning is a data-driven technology. With the rapid development of deep learning, many researchers propose to utilize deep learning models to learn vulnerability-related features from many smart contract samples for vulnerability detection. SmartEmbed (Gao et al., 2020) proposes utilizing the FastText model to get the embedding for each smart contract and do vulnerability detection by calculating the similarity between the target sample's embedding and already known vulnerable smart contract samples' embedding. DeeSCVHunter (Yu et al., 2021) proposes representing smart contract code as a token sequence with the LSTM model for vulnerability detection. TMP (Zhuang et al., 2020) and Peculiar (Wu et al., 2021) represent a smart contract as a graph with various graph neural networks for vulnerability detection.

Most current learning-based approaches do not capture both syntax and semantic features during code representation learning. In contrast, ours merges each statement's AST subtree with CFG for representing and learning both syntax and semantic features inside the code.

9. Conclusion

This paper proposes a security best practices based dataset construction approach with a code graph representation learning based approach for fine grained vulnerability detection in smart contracts. We propose to locate and strip the code fragments guarded by security best practices as the vulnerable statement samples to construct our dataset. To learn the heterogeneous syntax and semantic features of the smart contract code, we have introduced a code graph where each node represents an AST subtree. Furthermore, we have formulated vulnerability detection as a node classification task and proposed a syntax-sensitive graph neural network combining a child-sum tree-LSTM cell and graph convolution to learn the tree-structured syntax and graph-structured semantic features. The experimental results show the effectiveness of our approach by comparing our approach with three state-of-the-art deep learning-based approaches and two popular static analysis-based detectors in both fine and coarse granularity smart contract vulnerability detection.

In future work, we will improve our approach in three aspects. First, we manually summarize the common security best practices from the development community. In the future, we will propose a security best practices mining approach for auto-collecting common security best practices from the community. Second, we evaluate our approach by smart contracts written in Solidity language. In the future, we will develop a general approach for different programming languages for smart contracts. Third, we perform vulnerability detection within a function. We will extend our approach to support cross-contract vulnerability detection.

CRediT authorship contribution statement

Jie Cai: Methodology, Software, Validation, Writing – original draft. **Bin Li:** Supervision. **Tao Zhang:** Writing – review & editing, Resources, Supervision. **Jiale Zhang:** Writing – review & editing, Conceptualization. **Xiaobing Sun:** Formal analysis, Validation, Project administration.

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 is supported by the National Natural Science Foundation of China (No. 62206238, No. 61972335), Natural Science Foundation of Jiangsu Higher Education Institutions of China (Grant No. 22KJB520010), Natural Science Foundation of Jiangsu Province (Grant No. BK20220562), the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project, the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project (YZ202 1157, YZ2021158), the Natural Science Research Project of Universities in Jiangsu Province (No. 20KJB520024), and Yangzhou University Top-level Talents Support Program (2019).

References

Brent, Lexi, Grech, Neville, Lagouvardos, Sifis, Scholz, Bernhard, Smaragdakis, Yannis, 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 454–469.

Brody, Shaked, Alon, Uri, Yahav, Eran, 2021. How attentive are graph attention networks? arXiv preprint arXiv:2105.14491.

Bui, Nghi D.Q., Yu, Yijun, Jiang, Lingxiao, 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 1186–1197.

Buterin, Vitalik, 2018. Vyper documentation. Retrieved Oct 30, 2018.

ConsenSys, 2021. Mythril. https://github.com/ConsenSys/mythril.

ConsenSys, 2022. Solc-typed-ast. https://github.com/ConsenSys/solc-typed-ast.

Dannen, Chris, 2017. Introducing Ethereum and Solidity, Vol. 1. Springer.

Durieux, Thomas, Ferreira, João F, Abreu, Rui, Cruz, Pedro, 2020. Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 530–541.

Falkon, S., 2017. The story of the DAO—Its history and consequences. https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee.

Feist, Josselin, Grieco, Gustavo, Groce, Alex, 2019. Slither: a static analysis framework for smart contracts. In: 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, pp. 8–15.

Feng, Xiao, 2021. HuangGai. https://github.com/xf97/HuangGai.

Gao, Zhipeng, Jiang, Lingxiao, Xia, Xin, Lo, David, Grundy, John, 2020. Checking smart contracts with structural code embedding. IEEE Trans. Softw. Eng. 47 (12), 2874–2891.

Ghaleb, Asem, Pattabiraman, Karthik, 2020. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 415–427.

Guo, Daya, Ren, Shuo, Lu, Shuai, Feng, Zhangyin, Tang, Duyu, Liu, Shujie, Zhou, Long, Duan, Nan, Svyatkovskiy, Alexey, Fu, Shengyu, et al., 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Hagberg, Aric, Swart, Pieter, S. Chult, Daniel, 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. Technical Report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Hu, Tianyuan, Li, Bixin, Pan, Zhenyu, Qian, Chen, 2023. Detect defects of solidity smart contract based on the knowledge graph. IEEE Trans. Reliab..

Huang, Jing, Zhou, Kuo, Xiong, Ao, Li, Dongmeng, 2022. Smart contract vulnerability detection model based on multi-task learning. Sensors 22 (5), 1829.

Kipf, Thomas N., Welling, Max, 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

Li, Yujia, Tarlow, Daniel, Brockschmidt, Marc, Zemel, Richard, 2015. Gated graph sequence neural networks. arXiv preprint arXiv:1511.05493.

Liu, Zhenguang, Qian, Peng, Wang, Xiang, Zhu, Lei, He, Qinming, Ji, Shouling, 2021a. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In: Zhou, Zhi-Hua (Ed.), Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021. ijcai.org, pp. 2751–2759.

Liu, Zhenguang, Qian, Peng, Wang, Xiaoyang, Zhuang, Yuan, Qiu, Lin, Wang, Xun, 2021b. Combining graph neural networks with expert knowledge for smart contract vulnerability detection. IEEE Trans. Knowl. Data Eng..

Luu, Loi, Chu, Duc-Hiep, Olickel, Hrishi, Saxena, Prateek, Hobor, Aquinas, 2016. Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269.

Mossberg, Mark, Manzano, Felipe, Hennenfent, Eric, Groce, Alex, Grieco, Gustavo, Feist, Josselin, Brunson, Trent, Dinaburg, Artem, 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 1186–1189.

Nguyen, Hoang H, Nguyen, Nhat-Minh, Doan, Hong-Phuc, Ahmadi, Zahra, Doan, Thanh-Nam, Jiang, Lingxiao, 2022. MANDO-GURU: vulnerability detection for smart contract source code by heterogeneous graph embeddings. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1736–1740.

Rodler, Michael, Li, Wenting, Karame, Ghassan O, Davi, Lucas, 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv preprint arXiv:1812.05934.

Schneidewind, Clara, Grishchenko, Ilya, Scherer, Markus, Maffei, Matteo, 2020. Ethor: Practical and provably sound static analysis of ethereum smart contracts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 621–640.

Tai, Kai Sheng, Socher, Richard, Manning, Christopher D., 2015. Improved semantic representations from tree-structured long short-term memory networks. arXiv preprint arXiv:1503.00075.

Tikhomirov, Sergei, Voskresenskaya, Ekaterina, Ivanitskiy, Ivan, Takhaviev, Ramil, Marchenko, Evgeny, Alexandrov, Yaroslav, 2018. Smartcheck: Static analysis of ethereum smart contracts. In: Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. pp. 9–16.

Tsankov, Petar, Dan, Andrei, Drachsler-Cohen, Dana, Gervais, Arthur, Buenzli, Florian, Vechev, Martin, 2018. Securify: Practical security analysis of smart contracts. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 67–82.

Veličković, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Lio, Pietro, Bengio, Yoshua, 2017. Graph attention networks. arXiv preprint arXiv:1710.10903.

Wang, Minjie Yu, 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In: ICLR Workshop on Representation Learning on Graphs and Manifolds.

Wang, Huanting, Ye, Guixin, Tang, Zhanyong, Tan, Shin Hwei, Huang, Songfang, Fang, Dingyi, Feng, Yansong, Bian, Lizhong, Wang, Zheng, 2020. Combining graph-based learning with automated data collection for code vulnerability detection. IEEE Trans. Inf. Forensics Secur. 16, 1943–1958.

Wu, Hongjun, Zhang, Zhuo, Wang, Shangwen, Lei, Yan, Lin, Bo, Qin, Yihao, Zhang, Haoyu, Mao, Xiaoguang, 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In: Jin, Zhi, Li, Xuandong, Xiang, Jianwen, Mariani, Leonardo, Liu, Ting, Yu, Xiao, Ivaki, Nahgmeh (Eds.), 32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021. IEEE, pp. 378–389.

Yamaguchi, Fabian, Golde, Nico, Arp, Daniel, Rieck, Konrad, 2014. Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy. IEEE, pp. 590–604.

Yu, Xingxin, Zhao, Haoyue, Hou, Botao, Ying, Zonghao, Wu, Bin, 2021. DeeSCVHunter: A deep learning-based framework for smart contract vulnerability detection. In: International Joint Conference on Neural Networks, IJCNN 2021, Shenzhen, China, July 18-22, 2021. IEEE, pp. 1–8.

Zhang, Lejun, Chen, Weijie, Wang, Weizheng, Jin, Zilong, Zhao, Chunhui, Cai, Zhennao, Chen, Huiling, 2022a. CBGRU: A detection method of smart contract vulnerability based on a hybrid model. Sensors 22 (9), 3577.

Zhang, Yujian, Liu, Daifu, 2022. Toward vulnerability detection for ethereum smart contracts using graph-matching network. Future Internet 14 (11), 326.

Zhang, Lejun, Wang, Jinlong, Wang, Weizheng, Jin, Zilong, Su, Yansen, Chen, Huiling, 2022b. Smart contract vulnerability detection combined with multi-objective detection. Comput. Netw. 217, 109289.

Zhang, Pengcheng, Xiao, Feng, Luo, Xiapu, 2020. A framework and dataset for bugs in ethereum smart contracts. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 139–150.

Zhou, Yaqin, Liu, Shangqing, Siow, Jingkai, Du, Xiaoning, Liu, Yang, 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Zhou, Haozhe, Milani Fard, Amin, Makanju, Adetokunbo, 2022. The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support. J. Cybersecur. Priv. 2 (2), 358–378.

Zhuang, Yuan, Liu, Zhenguang, Qian, Peng, Liu, Qi, Wang, Xiang, He, Qinming, 2020. Smart contract vulnerability detection using graph neural network. In: IJCAI. pp. 3283–3290.

Jie Cai received the Master's degree from Nanjing University of Post and Telecommunications, Nanjing, China, in 2016. He is working toward the Ph.D. in software engineering at Yangzhou University, Yangzhou, China. His research interests include blockchain security, software safety and security, etc.

Bin Li is a professor in School of Information Engineering, Yangzhou University, China. His current research interests include software engineering, artificial intelligence.

Tao Zhang received the B.S. degree in automation, the M.Eng. 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.

Jiale Zhang received the Ph.D. degree in computer science and technology the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 2021. He is currently the Associate Professor with the School of Information Engineering, Yangzhou University, Yangzhou, China. His research interests are mainly federated learning, blockchain security, and privacy-preserving.

Xiaobing Sun is currently a professor in School of Information Engineering, Yangzhou University, China. He received the Ph.D. degree in School of computer science and engineering, Southeast University in 2012. His research interests include intelligent software engineering and software data analytics.